Mordechai Ben-Ari

Mathematical Logic for Computer Science

Third Edition

Springer

Mathematical Logic for Computer Science

Mordechai Ben-Ari

Mathematical Logic for Computer Science

Third Edition

Springer

Prof. Mordechai (Moti) Ben-Ari
Department of Science Teaching
Weizmann Institute of Science
Rehovot, Israel

ISBN 978-1-4471-4128-0 ISBN 978-1-4471-4129-7 (eBook)
DOI 10.1007/978-1-4471-4129-7
Springer London Heidelberg New York Dordrecht

Library of Congress Control Number: 2012941863

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

For Anita

Preface

Students of science and engineering are required to study mathematics during their first years at a university. Traditionally, they concentrate on calculus, linear algebra and differential equations, but in computer science and engineering, logic, combinatorics and discrete mathematics are more appropriate. Logic is particularly important because it is the mathematical basis of software: it is used to formalize the semantics of programming languages and the specification of programs, and to verify the correctness of programs.

Mathematical Logic for Computer Science is a *mathematics* textbook, just as a first-year calculus text is a mathematics textbook. A scientist or engineer needs more than just a facility for manipulating formulas and a firm foundation in mathematics is an excellent defense against technological obsolescence. Tempering this requirement for mathematical competence is the realization that applications use only a fraction of the theoretical results. Just as the theory of calculus can be taught to students of engineering without the full generality of measure theory, students of computer science need not be taught the full generality of uncountable structures. Fortunately (as shown by Raymond M. Smullyan), tableaux provide an elegant way to teach mathematical logic that is both theoretically sound and yet sufficiently elementary for the undergraduate.

Audience

The book is intended for undergraduate computer science students. No specific mathematical knowledge is assumed aside from informal set theory which is summarized in an appendix, but elementary knowledge of concepts from computer science (graphs, languages, programs) is used.

Organization

The book can be divided into four parts. Within each part the chapters should be read sequentially; the prerequisites between the parts are described here.

Propositional Logic: Chapter 2 is on the syntax and semantics of propositional logic. It introduces the method of semantic tableaux as a decision procedure for the logic. This chapter is an essential prerequisite for reading the rest of the book. Chapter 3 introduces deductive systems (axiomatic proof systems). The next three chapters present techniques that are used in practice for tasks such as automatic theorem proving and program verification: Chap. 4 on resolution, Chap. 5 on binary decision diagrams and Chap. 6 on SAT solvers.

First-Order Logic: The same progression is followed for first-order logic. There are two chapters on the basic theory of the logic: Chap. 7 on syntax, semantics and semantic tableaux, followed by Chap. 8 on deductive systems. Important application of first-order logic are automatic theorem proving using resolution (Chap. 10) and logic programming (Chap. 11). These are preceded by Chap. 9 which introduces an essential extension of the logic to terms and functions. Chapter 12 surveys fundamental theoretical results in first-order logic. The chapters on first-order logic assume as prerequisites the corresponding chapters on propositional logic; for example, you should read Chap. 4 on resolution in the propositional logic before the corresponding Chap. 10 in first-order logic.

Temporal Logic: Again, the same progression is followed: Chap. 13 on syntax, semantics and semantic tableaux, followed by Chap. 14 on deductive systems. The prerequisites are the corresponding chapters on propositional logic since first-order temporal logic is not discussed.

Program Verification: One of the most important applications of mathematical logic in computer science is in the field of program verification. Chapter 15 presents a deductive system for the verification of sequential programs; the reader should have mastered Chap. 3 on deductive systems in propositional logic before reading this chapter. Chapter 16 is highly dependent on earlier chapters: it includes deductive proofs, the use of temporal logic, and implementations using binary decision diagrams and satisfiability solvers.

Supplementary Materials

Slides of the diagrams and tables in the book (in both PDF and LaTeX) can be downloaded from http://www.springer.com/978-1-4471-4128-0, which also contains instructions for obtaining the answers to the exercises (qualified instructors only). The source code and documentation of Prolog programs for most of the algorithms in the book can be downloaded from http://code.google.com/p/mlcs/.

Third Edition

The third edition has been totally rewritten for clarity and accuracy. In addition, the following major changes have been made to the content:

- The discussion of logic programming has been shortened somewhat and the Prolog programs and their documentation have been removed to a freely available archive.
- The chapter on the $\mathscr{Z}$ notation has been removed because it was difficult to do justice to this important topic in a single chapter.
- The discussion of model checking in Chap. 16 has been significantly expanded since model checking has become a widely used technique for program verification.
- Chapter 6 has been added to reflect the growing importance of SAT solvers in all areas of computer science.

Notation

If and only if is abbreviated *iff*. Definitions by convention use *iff* to emphasize that the definition is restrictive. For example: A natural number is even iff it can be expressed as $2k$ for some natural number k. In the definition, *iff* means that numbers expressed as $2k$ are even and these are the only even numbers.

Definitions, theorems and examples are consecutively numbered within each chapter to make them easy to locate. The end of a definition, example or proof is denoted by ■.

Advanced topics and exercises, as well as topics outside the mainstream of the book, are marked with an asterisk.

Acknowledgments

I am indebted to Jørgen Villadsen for his extensive comments on the second edition which materially improved the text. I would like to thank Joost-Pieter Katoen and Doron Peled for reviewing parts of the manuscript. I would also like to thank Helen Desmond, Ben Bishop and Beverley Ford of Springer for facilitating the publication of the book.

Rehovot, Israel Mordechai (Moti) Ben-Ari

Contents

Chapter 1
Introduction

1.1 The Origins of Mathematical Logic

Logic formalizes valid methods of reasoning. The study of logic was begun by the ancient Greeks whose educational system stressed competence in reasoning and in the use of language. Along with rhetoric and grammar, logic formed part of the *trivium*, the first subjects taught to young people. Rules of logic were classified and named. The most widely known set of rules are the *syllogisms*; here is an example of one form of syllogism:

Premise All rabbits have fur.
Premise Some pets are rabbits.
Conclusion Some pets have fur.

If both premises are true, the rules ensure that the conclusion is true.

Logic must be formalized because reasoning expressed in informal natural language can be flawed. A clever example is the following 'syllogism' given by Smullyan (1978, p. 183):

Premise Some cars rattle.
Premise My car is some car.
Conclusion My car rattles.

The formalization of logic began in the nineteenth century as mathematicians attempted to clarify the foundations of mathematics. One trigger was the discovery of non-Euclidean geometries: replacing Euclid's parallel axiom with another axiom resulted in a different theory of geometry that was just as consistent as that of Euclid. Logical systems—axioms and rules of inference—were developed with the understanding that different sets of axioms would lead to different theorems. The questions investigated included:

Consistency A logical system is consistent if it is impossible to prove both a formula and its negation.

Independence The axioms of a logical system are independent if no axiom can be proved from the others.

M. Ben-Ari, *Mathematical Logic for Computer Science*,
DOI 10.1007/978-1-4471-4129-7_1, © Springer-Verlag London 2012

Soundness All theorems that can be proved in the logical system are true.
Completeness All true statements can be proved in the logical system.

Clearly, these questions will only make sense once we have formally defined the central concepts of *truth* and *proof*.

During the first half of the twentieth century, logic became a full-fledged topic of modern mathematics. The framework for research into the foundations of mathematics was called *Hilbert's program*, (named after the great mathematician David Hilbert). His central goal was to prove that mathematics, starting with arithmetic, could be axiomatized in a system that was both consistent and complete. In 1931, Kurt Gödel showed that this goal cannot be achieved: any consistent axiomatic system for arithmetic is incomplete since it contains true statements that cannot be proved within the system.

In the second half of the twentieth century, mathematical logic was applied in computer science and has become one of its most important theoretical foundations. Problems in computer science have led to the development of many new systems of logic that did not exist before or that existed only at the margins of the classical systems. In the remainder of this chapter, we will give an overview of systems of logic relevant to computer science and sketch their applications.

1.2 Propositional Logic

Our first task is to formalize the concept of the *truth* of a statement. Every statement is assigned one of two values, conventionally called *true* and *false* or T and F. These should be considered as arbitrary symbols that could easily be replaced by any other pair of symbols like 1 and 0 or even ♣ and ♠.

Our study of logic commences with the study of *propositional logic* (also called the *propositional calculus*). The *formulas* of the logic are built from *atomic propositions*, which are statements that have no internal structure. Formulas can be combined using *Boolean operators*. These operators have conventional names derived from natural language (*and*, *or*, *implies*), but they are given a formal meaning in the logic. For example, the Boolean operator *and* is defined as the operator that gives the value *true* if and only if applied to two formulas whose values are *true*.

Example 1.1 The statements 'one plus one equals two' and 'Earth is farther from the sun than Venus' are both true statements; therefore, by definition, so is the following statement:

'one plus one equals two' *and* 'Earth is farther from the sun than Venus'.

Since 'Earth is farther from the sun than Mars' is a false statement, so is:

'one plus one equals two' *and* 'Earth is farther from the sun than Mars'. ■

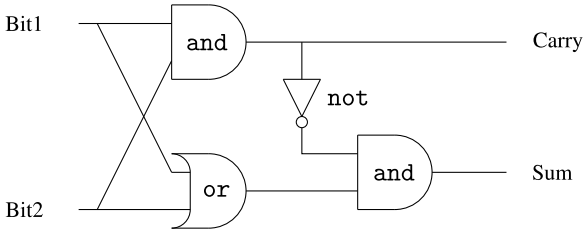
Rules of *syntax* define the legal structure of formulas in propositional logic. The *semantics*—the meaning of formulas—is defined by *interpretations*, which assign

one of the *(truth) values* T or F to every atomic proposition. For every legal way that a formula can be constructed, a semantical rule specifies the truth value of the formula based upon the values of its constituents.

Proof is another syntactical concept. A proof is a deduction of a formula from a set of formulas called *axioms* using *rules of inference*. The central theoretical result that we prove is the soundness and completeness of the axiom system: the set of provable formulas is the same as the set of formulas which are always true.

Propositional logic is central to the design of computer hardware because hardware is usually designed with components having two voltage levels that are arbitrarily assigned the symbols 0 and 1. Circuits are described by idealized elements called *logic gates*; for example, an `and`-gate produces the voltage level associated with 1 if and only if both its input terminals are held at this same voltage level.

Example 1.2 Here is a *half-adder* constructed from `and`, `or`- and `not`-gates.



The half-adder adds two one-bit binary numbers and by joining several half-adders we can add binary numbers composed of many bits. ■

Propositional logic is widely used in software, too. The reason is that any program is a finite entity. Mathematicians may consider the natural numbers to be infinite $(0, 1, 2, \ldots)$, but a word of a computer's memory can only store numbers in a finite range. By using an atomic proposition for each bit of a program's state, the meaning of a computation can be expressed as a (very large) formula. Algorithms have been developed to study properties of computations by evaluating properties of formulas in propositional logic.

1.3 First-Order Logic

Propositional logic is not sufficiently expressive for formalizing mathematical theories such as arithmetic. An arithmetic expression such as $x + 2 > y - 1$ is neither true nor false: (a) its truth depends on the values of the *variables* x and y; (b) we need to formalize the meaning of the operators $+$ and $-$ as *functions* that map a pair of numbers to a number; (c) *relational* operators like $>$ must be formalized as mapping pairs of numbers into truth values. The system of logic that can be interpreted by values, functions and relations is called *first-order logic* (also called *predicate logic* or the *predicate calculus*).

The study of the foundations of mathematics emphasized first-order logic, but it has also found applications in computer science, in particular, in the fields of automated theorem proving and logic programming. Can a computer carry out the work of a mathematician? That is, given a set of axioms for, say, number theory, can we write software that will find proofs of known theorems, as well as statements and proofs of new ones? With luck, the computer might even discover a proof of Goldbach's Conjecture, which states that every even number greater than two is the sum of two prime numbers:

$$4 = 2 + 2, \quad 6 = 3 + 3, \quad \ldots,$$
$$100 = 3 + 97, \quad 102 = 5 + 97, \quad 104 = 3 + 101, \quad \ldots.$$

Goldbach's Conjecture has not been proved, though no counterexample has been found even with an extensive computerized search.

Research into automated theorem proving led to a new and efficient method of proving formulas in first-order logic called *resolution*. Certain restrictions of resolution have proved to be so efficient they are the basis of a new type of programming language. Suppose that a theorem prover is capable of proving the following formula:

> Let A be an array of integers. Then there exists an array A' such that the elements of A' are a permutation of those of A, and such that A' is ordered: $A'(i) \leq A'(j)$ for $i < j$.

Suppose, further, that given any specific array A, the theorem prover constructs the array A' which the required properties. Then the formula is a *program* for sorting, and the proof of the formula generates the *result*. The use of theorem provers for computation is called *logic programming*. Logic programming is attractive because it is *declarative*—you just write what you *want* from the computation—as opposed to classical programming languages, where you have to specify in detail *how* the computation is to be carried out.

1.4 Modal and Temporal Logics

A statement need not be absolutely true or false. The statement 'it is raining' is sometimes true and sometimes false. *Modal logics* are used to formalize statements where finer distinctions need to be made than just 'true' or 'false'. Classically, modal logic distinguished between statements that are *necessarily* true and those that are *possibly* true. For example, $1 + 1 = 2$, as a statement about the natural numbers, is necessarily true because of the way the concepts are defined. But any historical statement like 'Napoleon lost the battle of Waterloo' is only possibly true; if circumstances had been different, the outcome of Waterloo might have been different.

Modal logics have turned out to be extremely useful in computer science. We will study a form of modal logic called *temporal logic*, where 'necessarily' is interpreted as *always* and 'possibly' is interpreted as *eventually*. Temporal logic has turned out to be the preferred logic for program verification as described in the following section.

1.5 Program Verification

One of the major applications of logic to computer science is in *program verification*. Software now controls our most critical systems in transportation, medicine, communications and finance, so that it is hard to think of an area in which we are not dependent on the correct functioning of a computerized system. Testing a program can be an ineffective method of verifying the correctness of a program because we test the scenarios that we think will happen and not those that arise unexpectedly. Since a computer program is simply a formal description of a calculation, it can be verified in the same way that a mathematical theorem can be verified using logic.

First, we need to express a *correctness specification* as a formal statement in logic. Temporal logic is widely used for this purpose because it can express the dynamic behavior of program, especially of *reactive* programs like operating systems and real-time systems, which do not compute an result but instead are intended to run indefinitely.

Example 1.3 The property '*always* not deadlocked' is an important correctness specification for operating systems, as is 'if you request to print a document, *eventually* the document will be printed'. ■

Next, we need to formalize the semantics (the meaning) of a program, and, finally, we need a formal system for deducing that the program fulfills a correctness specification. An axiomatic system for temporal logic can be used to prove concurrent programs correct.

For sequential programs, verification is performed using an axiomatic system called *Hoare logic* after its inventor C.A.R. Hoare. Hoare logic assumes that we know the truth of statements of the program's domain like arithmetic; for example, $-(1-x) = (x-1)$ is considered to be an axiom of the logic. There are axioms and rules of inference that concern the structure of the program: assignment statements, loops, and so on. These are used to create a proof that a program fulfills a correctness specification.

Rather than deductively prove the correctness of a program relative to a specification, a *model checker* verifies the truth of a correctness specification in every possible state that can appear during the computation of a program. On a physical computer, there are only a finite number of different states, so this is always possible. The challenge is to make model checking feasible by developing methods and algorithms to deal with the very large number of possible states. Ingenious algorithms and data structures, together with the increasing CPU power and memory of modern computers, have made model checkers into viable tools for program verification.

1.6 Summary

Mathematical logic formalizes reasoning. There are many different systems of logic: propositional logic, first-order logic and modal logic are really families of logic with

many variants. Although systems of logic are very different, we approach each logic in a similar manner: We start with their syntax (what constitutes a formula in the logic) and their semantics (how truth values are attributed to a formula). Then we describe the method of *semantic tableaux* for deciding the validity of a formula. This is followed by the description of an axiomatic system for the logic. Along the way, we will look at the applications of the various logics in computer science with emphasis on theorem proving and program verification.

1.7 Further Reading

This book was originally inspired by Raymond M. Smullyan's presentation of logic using semantic tableaux. It is still worthwhile studying Smullyan (1968). A more advanced logic textbook for computer science students is Nerode and Shore (1997); its approach to propositional and first-order logic is similar to ours but it includes chapters on modal and intuitionistic logics and on set theory. It has a useful appendix that provides an overview of the history of logic as well as a comprehensive bibliography. Mendelson (2009) is a classic textbook that is more mathematical in its approach.

Smullyan's books such as Smullyan (1978) will exercise your abilities to think logically! The final section of that book contains an informal presentation of Gödel's incompleteness theorem.

1.8 Exercise

1.1 What is wrong with Smullyan's 'syllogism'?

References

E. Mendelson. *Introduction to Mathematical Logic* (*Fifth Edition*). Chapman & Hall/CRC, 2009.
A. Nerode and R.A. Shore. *Logic for Applications* (*Second Edition*). Springer, 1997.
R.M. Smullyan. *First-Order Logic*. Springer-Verlag, 1968. Reprinted by Dover, 1995.
R.M. Smullyan. *What Is the Name of This Book?—The Riddle of Dracula and Other Logical Puzzles*. Prentice-Hall, 1978.

Chapter 2
Propositional Logic: Formulas, Models, Tableaux

Propositional logic is a simple logical system that is the basis for all others. Propositions are claims like 'one plus one equals two' and 'one plus two equals two' that cannot be further decomposed and that can be assigned a truth value of *true* or *false*. From these *atomic propositions*, we will build complex *formulas* using *Boolean operators*:

'one plus one equals two' *and* 'Earth is farther from the sun than Venus'.

Logical systems formalize reasoning and are similar to programming languages that formalize computations. In both cases, we need to define the syntax and the semantics. The *syntax* defines what strings of symbols constitute legal formulas (legal programs, in the case of languages), while the *semantics* defines what legal formulas mean (what legal programs compute). Once the syntax and semantics of propositional logic have been defined, we will show how to construct *semantic tableaux*, which provide an efficient *decision procedure* for checking when a formula is true.

2.1 Propositional Formulas

In computer science, an *expression* denoted the computation of a value from other values; for example, $2 * 9 + 5$. In propositional logic, the term *formula* is used instead. The formal definition will be in terms of trees, because our the main proof technique called structural induction is easy to understand when applied to trees. Optional subsections will expand on different approaches to syntax.

M. Ben-Ari, *Mathematical Logic for Computer Science*,
DOI 10.1007/978-1-4471-4129-7_2, © Springer-Verlag London 2012

2.1.1 Formulas as Trees

Definition 2.1 The symbols used to construct formulas in propositional logic are:

- An unbounded set of symbols $\mathscr{P}$ called *atomic propositions* (often shortened to *atoms*). Atoms will be denoted by lower case letters in the set $\{p, q, r, \ldots\}$, possibly with subscripts.
- *Boolean operators*. Their names and the symbols used to denote them are:

negation	$\neg$
disjunction	$\vee$
conjunction	$\wedge$
implication	$\rightarrow$
equivalence	$\leftrightarrow$
exclusive or	$\oplus$
nor	$\downarrow$
nand	$\uparrow$

The negation operator is a *unary operator* that takes one operand, while the other operators are *binary operators* taking two operands. ■

Definition 2.2 A *formula* in propositional logic is a tree defined recursively:

- A formula is a leaf labeled by an atomic proposition.
- A formula is a node labeled by $\neg$ with a single child that is a formula.
- A formula is a node labeled by one of the binary operators with two children both of which are formulas. ■

Example 2.3 Figure 2.1 shows two formulas. ■

2.1.2 Formulas as Strings

Just as we write expressions as strings (linear sequences of symbols), we can write formulas as strings. The string associated with a formula is obtained by an *inorder traversal* of the tree:

Algorithm 2.4 (Represent a formula by a string)
Input: A formula A of propositional logic.
Output: A string representation of A.

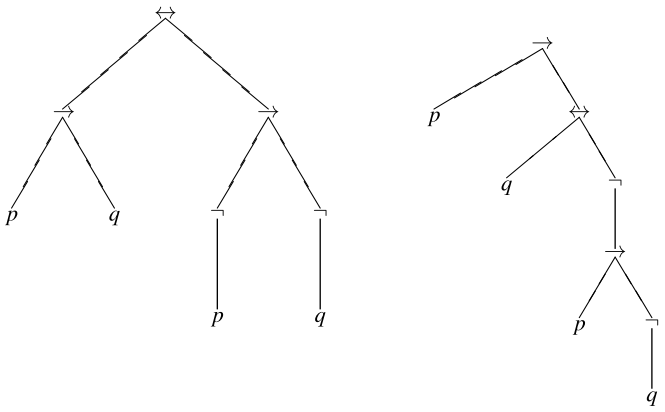


Fig. 2.1 Two formulas

Call the recursive procedure `Inorder(A)`:

```
Inorder(F)
  if F is a leaf
    write its label
    return
  let F1 and F2 be the left and right subtrees of F
  Inorder(F1)
  write the label of the root of F
  Inorder(F2)
```

If the root of `F` is labeled by negation, the left subtree is considered to be empty and the step `Inorder(F1)` is skipped. ■

Definition 2.5 The term *formula* will also be used for the string with the understanding that it refers to the underlying tree. ■

Example 2.6 Consider the left formula in Fig. 2.1. The inorder traversal gives: write the leftmost leaf labeled p, followed by its root labeled $\rightarrow$, followed by the right leaf of the implication labeled q, followed by the root of the tree labeled $\leftrightarrow$, and so on. The result is the string:

$$p \rightarrow q \leftrightarrow \neg\, p \rightarrow \neg\, q.$$

Consider now the right formula in Fig. 2.1. Performing the traversal results in the string:

$$p \rightarrow q \leftrightarrow \neg\, p \rightarrow \neg\, q,$$

which is precisely the same as that associated with the left formula. ■

Although the formulas are *not* ambiguous—the trees have entirely different structures—their representations as strings are ambiguous. Since we prefer to deal with strings, we need some way to resolve such ambiguities. There are three ways of doing this.

2.1.3 Resolving Ambiguity in the String Representation

Parentheses

The simplest way to avoid ambiguity is to use parentheses to maintain the structure of the tree when the string is constructed.

Algorithm 2.7 (Represent a formula by a string with parentheses)
Input: A formula A of propositional logic.
Output: A string representation of A.

Call the recursive procedure `Inorder(A)`:

```
Inorder(F)
  if F is a leaf
    write its label
    return
  let F1 and F2 be the left and right subtrees of F
  write a left parenthesis '('
  Inorder(F1)
  write the label of the root of F
  Inorder(F2)
  write a right parenthesis ')'
```

If the root of `F` is labeled by negation, the left subtree is considered to be empty and the step `Inorder(F1)` is skipped. ■

The two formulas in Fig. 2.1 are now associated with two different strings and there is no ambiguity:

$$((p \to q) \leftrightarrow ((\neg q) \to (\neg p))),$$
$$(p \to (q \leftrightarrow (\neg (p \to (\neg q))))).$$

The problem with parentheses is that they make formulas verbose and hard to read and write.

Precedence

The second way of resolving ambiguous formulas is to define *precedence* and *associativity* conventions among the operators as is done in arithmetic, so that we

immediately recognize $a * b * c + d * e$ as $(((a * b) * c) + (d * e))$. For formulas the order of precedence from high to low is as follows:

$$\begin{array}{c} \neg \\ \wedge, \uparrow \\ \vee, \downarrow \\ \rightarrow \\ \leftrightarrow, \oplus \end{array}$$

Operators are assumed to associate to the right, that is, $a \vee b \vee c$ means $(a \vee (b \vee c))$.

Parentheses are used only if needed to indicate an order different from that imposed by the rules for precedence and associativity, as in arithmetic where $a * (b + c)$ needs parentheses to denote that the addition is done before the multiplication. With minimal use of parentheses, the formulas above can be written:

$$p \rightarrow q \leftrightarrow \neg q \rightarrow \neg p,$$
$$p \rightarrow (q \leftrightarrow \neg (p \rightarrow \neg q)).$$

Additional parentheses may always be used to clarify a formula: $(p \vee q) \wedge (q \vee r)$.

The Boolean operators $\wedge$, $\vee$, $\leftrightarrow$, $\oplus$ are associative so we will often omit parentheses in formulas that have repeated occurrences of these operators: $p \vee q \vee r \vee s$. Note that $\rightarrow$, $\downarrow$, $\uparrow$ are *not* associative, so parentheses must be used to avoid confusion. Although the implication operator is assumed to be right associative, so that $p \rightarrow q \rightarrow r$ unambiguously means $p \rightarrow (q \rightarrow r)$, we will write the formula with parentheses to avoid confusion with $(p \rightarrow q) \rightarrow r$.

Polish Notation *

There will be no ambiguity if the string representing a formula is created by a *preorder traversal* of the tree:

Algorithm 2.8 (Represent a formula by a string in Polish notation)
Input: A formula A of propositional logic.
Output: A string representation of A.

Call the recursive procedure `Preorder(A)`:

```
Preorder(F)
  write the label of the root of F
  if F is a leaf
    return
  let F1 and F2 be the left and right subtrees of F
  Preorder(F1)
  Preorder(F2)
```

If the root of `F` is labeled by negation, the left subtree is considered to be empty and the step `Preorder(F1)` is skipped. ■

Example 2.9 The strings associated with the two formulas in Fig. 2.1 are:

$$\leftrightarrow \rightarrow p\, q \rightarrow \neg\, p \neg\, q,$$
$$\rightarrow p \leftrightarrow q \neg \rightarrow p \neg\, q$$

and there is no longer any ambiguity. ■

The formulas are said to be in *Polish notation*, named after a group of Polish logicians led by Jan Łukasiewicz.

We find infix notation easier to read because it is familiar from arithmetic, so Polish notation is normally used only in the internal representation of arithmetic and logical expressions in a computer. The advantage of Polish notation is that the expression can be evaluated in the linear order that the symbols appear using a stack. If we rewrite the first formula backwards (*reverse Polish notation*):

$$q \neg\, p \neg \rightarrow\, qp \rightarrow \leftrightarrow,$$

it can be directly compiled to the following sequence of instructions of an assembly language:

```
Push q
Negate
Push p
Negate
Imply
Push q
Push p
Imply
Equiv
```

The operators are applied to the top operands on the stack which are then popped and the result pushed.

2.1.4 Structural Induction

Given an arithmetic expression like $a * b + b * c$, it is immediately clear that the expression is composed of two terms that are added together. In turn, each term is composed of two factors that are multiplied together. In the same way, any propositional formula can be classified by its top-level operator.

Definition 2.10 Let $A \in \mathscr{F}$. If A is not an atom, the operator labeling the root of the formula A is the *principal operator* of the A. ■

Example 2.11 The principal operator of the left formula in Fig. 2.1 is $\leftrightarrow$, while the principal operator of the right formulas is $\rightarrow$. ■

Structural induction is used to prove that a property holds for *all* formulas. This form of induction is similar to the familiar numerical induction that is used to prove that a property holds for all natural numbers (Appendix A.6). In numerical induction, the *base case* is to prove the property for 0 and then to prove the *inductive step*: assume that the property holds for arbitrary n and then show that it holds for $n + 1$. By Definition 2.10, a formula is either a leaf labeled by an atom or it is a tree with a principal operator and one or two subtrees. The base case of structural induction is to prove the property for a leaf and the inductive step is to prove the property for the formula obtained by applying the principal operator to the subtrees, assuming that the property holds for the subtrees.

Theorem 2.12 (Structural induction) *To show that a property holds for all formulas* $A \in \mathscr{F}$:

1. *Prove that the property holds all atoms* p.
2. *Assume that the property holds for a formula* A *and prove that the property holds for* $\neg A$.
3. *Assume that the property holds for formulas* A_1 *and* A_2 *and prove that the property holds for* A_1 *op* A_2, *for each of the binary operators*.

Proof Let A be an arbitrary formula and suppose that (1), (2), (3) have been shown for some property. We show that the property holds for A by numerical induction on n, the height of the tree for A. For $n = 0$, the tree is a leaf and A is an atom p, so the property holds by (1). Let $n > 0$. The subtrees A are of height $n - 1$, so by numerical induction, the property holds for these formulas. The principal operator of A is either negation or one of the binary operators, so by (2) or (3), the property holds for A. ■

We will later show that all the binary operators can be defined in terms negation and either disjunction or conjunction, so a proof that a property holds for all formulas can be done using structural induction with the base case and only two inductive steps.

2.1.5 Notation

Unfortunately, books on mathematical logic use widely varying notation for the Boolean operators; furthermore, the operators appear in programming languages with a different notation from that used in mathematics textbooks. The following table shows some of these alternate notations.

Operator	Alternates	Java language
$\neg$	$\sim$	!
$\wedge$	&	&, &&
$\vee$		|, ||
$\rightarrow$	$\supset$, $\Rightarrow$	
$\leftrightarrow$	$\equiv$, $\Leftrightarrow$	
$\oplus$	$\not\equiv$	^
$\uparrow$	|	

*2.1.6 A Formal Grammar for Formulas **

This subsection assumes familiarity with formal grammars.

Instead of defining formulas as trees, they can be defined as strings generated by a context-free formal grammar.

Definition 2.13 Formula in propositional logic are derived from the context-free grammar whose terminals are:

- An unbounded set of symbols $\mathscr{P}$ called *atomic propositions*.
- The *Boolean operators* given in Definition 2.1.

The productions of the grammar are:

$$\begin{array}{lcll} \mathit{fml} & ::= & p & \text{for any } p \in \mathscr{P} \\ \mathit{fml} & ::= & \neg\, \mathit{fml} & \\ \mathit{fml} & ::= & \mathit{fml}\ \mathit{op}\ \mathit{fml} & \\ \mathit{op} & ::= & \vee \mid \wedge \mid \rightarrow \mid \leftrightarrow \mid \oplus \mid \uparrow \mid \downarrow & \end{array}$$

A formula is a word that can be derived from the nonterminal *fml*. The set of all formulas that can be derived from the grammar is denoted $\mathscr{F}$. ■

Derivations of strings (words) in a formal grammar can be represented as trees (Hopcroft et al., 2006, Sect. 4.3). The word generated by a derivation can be read off the leaves from left to right.

Example 2.14 Here is a derivation of the formula $p \rightarrow q \leftrightarrow \neg\, p \rightarrow \neg\, q$ in propositional logic; the tree representing its derivation is shown in Fig. 2.2.

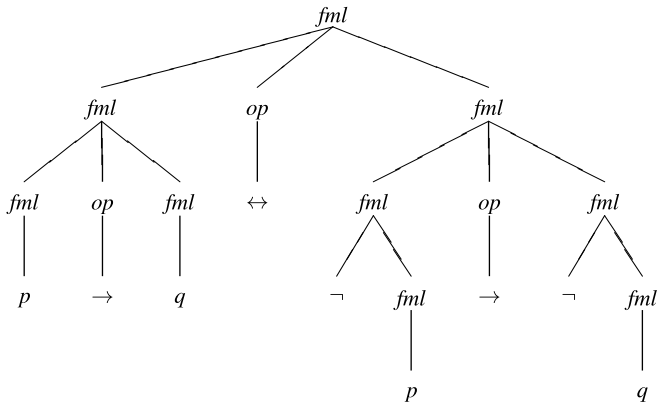


Fig. 2.2 Derivation tree for $p \to q \leftrightarrow \neg\, p \to \neg\, q$

1. *fml*
2. *fml op fml*
3. $fml \leftrightarrow fml$
4. *fml op fml* $\leftrightarrow$ *fml*
5. $fml \to fml \leftrightarrow fml$
6. $p \to fml \leftrightarrow fml$
7. $p \to q \leftrightarrow fml$
8. $p \to q \leftrightarrow fml\ op\ fml$
9. $p \to q \leftrightarrow fml \to fml$
10. $p \to q \leftrightarrow \neg fml \to fml$
11. $p \to q \leftrightarrow \neg\, p \to fml$
12. $p \to q \leftrightarrow \neg\, p \to \neg fml$
13. $p \to q \leftrightarrow \neg\, p \to \neg\, q$

■

The methods discussed in Sect. 2.1.2 can be used to resolve ambiguity. We can change the grammar to introduce parentheses:

$$\begin{aligned} fml &::= (\neg fml) \\ fml &::= (fml\ \ op\ \ fml) \end{aligned}$$

and then use precedence to reduce their number.

$v_{\mathscr{I}}(A) = \mathscr{I}_A(A)$	if A is an atom
$v_{\mathscr{I}}(\neg A) = T$	if $v_{\mathscr{I}}(A) = F$
$v_{\mathscr{I}}(\neg A) = F$	if $v_{\mathscr{I}}(A) = T$
$v_{\mathscr{I}}(A_1 \vee A_2) = F$	if $v_{\mathscr{I}}(A_1) = F$ and $v_{\mathscr{I}}(A_2) = F$
$v_{\mathscr{I}}(A_1 \vee A_2) = T$	otherwise
$v_{\mathscr{I}}(A_1 \wedge A_2) = T$	if $v_{\mathscr{I}}(A_1) = T$ and $v_{\mathscr{I}}(A_2) = T$
$v_{\mathscr{I}}(A_1 \wedge A_2) = F$	otherwise
$v_{\mathscr{I}}(A_1 \rightarrow A_2) = F$	if $v_{\mathscr{I}}(A_1) = T$ and $v_{\mathscr{I}}(A_2) = F$
$v_{\mathscr{I}}(A_1 \rightarrow A_2) = T$	otherwise
$v_{\mathscr{I}}(A_1 \uparrow A_2) = F$	if $v_{\mathscr{I}}(A_1) = T$ and $v_{\mathscr{I}}(A_2) = T$
$v_{\mathscr{I}}(A_1 \uparrow A_2) = T$	otherwise
$v_{\mathscr{I}}(A_1 \downarrow A_2) = T$	if $v_{\mathscr{I}}(A_1) = F$ and $v_{\mathscr{I}}(A_2) = F$
$v_{\mathscr{I}}(A_1 \downarrow A_2) = F$	otherwise
$v_{\mathscr{I}}(A_1 \leftrightarrow A_2) = T$	if $v_{\mathscr{I}}(A_1) = v_{\mathscr{I}}(A_2)$
$v_{\mathscr{I}}(A_1 \leftrightarrow A_2) = F$	if $v_{\mathscr{I}}(A_1) \neq v_{\mathscr{I}}(A_2)$
$v_{\mathscr{I}}(A_1 \oplus A_2) = T$	if $v_{\mathscr{I}}(A_1) \neq v_{\mathscr{I}}(A_2)$
$v_{\mathscr{I}}(A_1 \oplus A_2) = F$	if $v_{\mathscr{I}}(A_1) = v_{\mathscr{I}}(A_2)$

Fig. 2.3 Truth values of formulas

2.2 Interpretations

We now define the semantics—the meaning—of formulas. Consider again arithmetic expressions. Given an expression E such as $a * b + 2$, we can *assign* values to a and b and then *evaluate* the expression. For example, if $a = 2$ and $b = 3$ then E evaluates to 8. In propositional logic, truth values are assigned to the atoms of a formula in order to evaluate the truth value of the formula.

2.2.1 The Definition of an Interpretation

Definition 2.15 Let $A \in \mathscr{F}$ be a formula and let $\mathscr{P}_A$ be the set of atoms appearing in A. An *interpretation* for A is a total function $\mathscr{I}_A : \mathscr{P}_A \mapsto \{T, F\}$ that assigns one of the *truth values* T or F to *every* atom in $\mathscr{P}_A$. ■

Definition 2.16 Let $\mathscr{I}_A$ be an interpretation for $A \in \mathscr{F}$. $v_{\mathscr{I}_A}(A)$, the *truth value of* A *under* $\mathscr{I}_A$ is defined inductively on the structure of A as shown in Fig. 2.3. ■

In Fig. 2.3, we have abbreviated $v_{\mathscr{I}_A}(A)$ by $v_{\mathscr{I}}(A)$. The abbreviation $\mathscr{I}$ for $\mathscr{I}_A$ will be used whenever the formula is clear from the context.

Example 2.17 Let $A = (p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$ and let $\mathscr{I}_A$ be the interpretation:

$$\mathscr{I}_A(p) = F, \qquad \mathscr{I}_A(q) = T.$$

The truth value of A can be evaluated inductively using Fig. 2.3:

$$\begin{aligned} v_{\mathscr{I}}(p) &= \mathscr{I}_A(p) = F \\ v_{\mathscr{I}}(q) &= \mathscr{I}_A(q) = T \\ v_{\mathscr{I}}(p \to q) &= T \\ v_{\mathscr{I}}(\neg q) &= F \\ v_{\mathscr{I}}(\neg p) &= T \\ v_{\mathscr{I}}(\neg q \to \neg p) &= T \\ v_{\mathscr{I}}((p \to q) \leftrightarrow (\neg q \to \neg p)) &= T. \end{aligned}$$

■

Partial Interpretations *

We will later need the following definition, but you can skip it for now:

Definition 2.18 Let $A \in \mathscr{F}$. A *partial interpretation* for A is a partial function $\mathscr{I}_A : \mathscr{P}_A \mapsto \{T, F\}$ that assigns one of the *truth values* T or F to *some* of the atoms in $\mathscr{P}_A$. ■

It is possible that the truth value of a formula can be determined in a partial interpretation.

Example 2.19 Consider the formula $A = p \wedge q$ and the partial interpretation that assigns F to p. Clearly, the truth value of A is F. If the partial interpretation assigned T to p, we cannot compute the truth value of A. ■

2.2.2 Truth Tables

A truth table is a convenient format for displaying the semantics of a formula by showing its truth value for every possible interpretation of the formula.

Definition 2.20 Let $A \in \mathscr{F}$ and supposed that there are n atoms in $\mathscr{P}_A$. A *truth table* is a table with $n + 1$ columns and 2^n rows. There is a column for each atom in $\mathscr{P}_A$, plus a column for the formula A. The first n columns specify the interpretation $\mathscr{I}$ that maps atoms in $\mathscr{P}_A$ to $\{T, F\}$. The last column shows $v_{\mathscr{I}}(A)$, the truth value of A for the interpretation $\mathscr{I}$. ■

Since each of the n atoms can be assigned T or F independently, there are 2^n interpretations and thus 2^n rows in a truth table.

Example 2.21 Here is the truth table for the formula $p \to q$:

p	q	$p \to q$
T	T	T
T	F	F
F	T	T
F	F	T

■

When the formula A is complex, it is easier to build a truth table by adding columns that show the truth value for subformulas of A.

Example 2.22 Here is a truth table for the formula $(p \to q) \leftrightarrow (\neg q \to \neg p)$ from Example 2.17:

p	q	$p \to q$	$\neg p$	$\neg q$	$\neg q \to \neg p$	$(p \to q) \leftrightarrow (\neg q \to \neg p)$
T	T	T	F	F	T	T
T	F	F	F	T	F	T
F	T	T	T	F	T	T
F	F	T	T	T	T	T

■

A convenient way of computing the truth value of a formula for a specific interpretation $\mathscr{I}$ is to write the value T or F of $\mathscr{I}(p_i)$ under each atom p_i and then to write down the truth values incrementally under each operator as you perform the computation. Each step of the computation consists of choosing an innermost subformula and evaluating it.

Example 2.23 The computation of the truth value of $(p \to q) \leftrightarrow (\neg q \to \neg p)$ for the interpretation $\mathscr{I}(p) = T$ and $\mathscr{I}(q) = F$ is:

$(p$	$\to$	$q)$	$\leftrightarrow$	$(\neg$	q	$\to$	$\neg$	$p)$
T		F			F			T
T		F		T	F			T
T		F		T	F		F	T
T		F		T	F	F	F	T
T	F	F		T	F	F	F	T
T	F	F	T	T	F	F	F	T

If the computations for all subformulas are written on the same line, the truth table from Example 2.22 can be written as follows:

p	q	$(p$	$\rightarrow$	$q)$	$\leftrightarrow$	$(\neg$	q	$\rightarrow$	$\neg$	$p)$
T	T	T	T	T	T	F	T	T	F	T
T	F	T	F	F	T	T	F	F	F	T
F	T	F	T	T	T	F	T	T	T	F
F	F	F	T	F	T	T	F	T	T	F

■

2.2.3 *Understanding the Boolean Operators*

The natural reading of the Boolean operators $\neg$ and $\wedge$ correspond with their formal semantics as defined in Fig. 2.3. The operators $\uparrow$ and $\downarrow$ are simply negations of $\wedge$ and $\vee$. Here we comment on the operators $\vee$, $\oplus$ and $\rightarrow$, whose formal semantics can be the source of confusion.

Inclusive or *vs.* Exclusive or

Disjunction $\vee$ is *inclusive or* and is a distinct operator from $\oplus$ which is *exclusive or*. Consider the compound statement:

> At eight o'clock 'I will go to the movies' *or* 'I will go to the theater'.

The intended meaning is 'movies' $\oplus$ 'theater', because I can't be in both places at the same time. This contrasts with the disjunctive operator $\vee$ which evaluates to true when either or both of the statements are true:

> Do you want 'popcorn' or 'candy'?

This can be denoted by 'popcorn' $\vee$ 'candy', because it is possible to want both of them at the same time.

For $\vee$, it is sufficient for one statement to be true for the compound statement to be true. Thus, the following strange statement is true because the truth of the first statement by itself is sufficient to ensure the truth of the compound statement:

> 'Earth is farther from the sun than Venus' $\vee$ '$1 + 1 = 3$'.

The difference between $\vee$ and $\oplus$ is seen when both subformulas are true:

> 'Earth is farther from the sun than Venus' $\vee$ '$1 + 1 = 2$'.
> 'Earth is farther from the sun than Venus' $\oplus$ '$1 + 1 = 2$'.

The first statement is true but the second is false.

Inclusive or *vs.* Exclusive or in Programming Languages

When *or* is used in the context of programming languages, the intention is usually inclusive or:

```
if (index < min || index > max) /* There is an error */
```

The truth of one of the two subexpressions causes the following statements to be executed. The operator || is not really a Boolean operator because it uses *short-circuit evaluation*: if the first subexpression is true, the second subexpression is not evaluated, because its truth value cannot change the decision to execute the following statements. There is an operator | that performs true Boolean evaluation; it is usually used when the operands are bit vectors:

```
mask1 = 0xA0;
mask2 = 0x0A;
mask  = mask1 | mask2;
```

Exclusive or ^ is used to implement encoding and decoding in error-correction and cryptography. The reason is that when used twice, the original value can be recovered. Suppose that we encode bit of data with a secret key:

```
codedMessage = data ^ key;
```

The recipient of the message can decode it by computing:

```
clearMessage = codedMessage ^ key;
```

as shown by the following computation:

```
clearMessage == codedMessage ^ key
             == (data ^ key) ^ key
             == data ^ (key ^ key)
             == data ^ false
             == data
```

Implication

The operator of $p \rightarrow q$ is called *material implication*; p is the *antecedent* and q is the *consequent*. Material implication does not claim causation; that is, it does not assert there the antecedent *causes* the consequent (or is even related to the consequent in any way). A material implication merely states that if the antecedent is true the consequent must be true (see Fig. 2.3), so it can be falsified only if the antecedent is true and the consequent is false. Consider the following two compound statements:

'Earth is farther from the sun than Venus' $\rightarrow$ '$1 + 1 = 3$'.

is false since the antecedent is true and the consequent is false, but:

'Earth is farther from the sun than Mars' $\rightarrow$ '$1+1=3$'.

is true! The falsity of the antecedent by itself is sufficient to ensure the truth of the implication.

2.2.4 An Interpretation for a Set of Formulas

Definition 2.24 Let $S = \{A_1, \ldots\}$ be a set of formulas and let $\mathscr{P}_S = \bigcup_i \mathscr{P}_{A_i}$, that is, $\mathscr{P}_S$ is the set of all the atoms that appear in the formulas of S. An *interpretation* for S is a function $\mathscr{I}_S : \mathscr{P}_S \mapsto \{T, F\}$. For any $A_i \in S$, $v_{\mathscr{I}_S}(A_i)$, the *truth value of* A_i *under* $\mathscr{I}_S$, is defined as in Definition 2.16. ■

The definition of $\mathscr{P}_S$ as the union of the sets of atoms in the formulas of S ensures that each atom is assigned exactly one truth value.

Example 2.25 Let $S = \{p \rightarrow q,\ p,\ q \wedge r,\ p \vee s \leftrightarrow s \wedge q\}$ and let $\mathscr{I}_S$ be the interpretation:

$$\mathscr{I}_S(p) = T, \qquad \mathscr{I}_S(q) = F, \qquad \mathscr{I}_S(r) = T, \qquad \mathscr{I}_S(s) = T.$$

The truth values of the elements of S can be evaluated as:

$$\begin{aligned} v_{\mathscr{I}}(p \rightarrow q) &= F \\ v_{\mathscr{I}}(p) &= \mathscr{I}_S(p) = T \\ v_{\mathscr{I}}(q \wedge r) &= F \\ v_{\mathscr{I}}(p \vee s) &= T \\ v_{\mathscr{I}}(s \wedge q) &= F \\ v_{\mathscr{I}}(p \vee s \leftrightarrow s \wedge q) &= F. \end{aligned}$$

■

2.3 Logical Equivalence

Definition 2.26 Let $A_1,\ A_2 \in \mathscr{F}$. If $v_{\mathscr{I}}(A_1) = v_{\mathscr{I}}(A_2)$ for all interpretations $\mathscr{I}$, then A_1 is *logically equivalent* to A_2, denoted $A_1 \equiv A_2$. ■

Example 2.27 Is the formula $p \vee q$ logically equivalent to $q \vee p$? There are four distinct interpretations that assign to the atoms p and q:

$\mathscr{I}(p)$	$\mathscr{I}(q)$	$v_{\mathscr{I}}(p \vee q)$	$v_{\mathscr{I}}(q \vee p)$
T	T	T	T
T	F	T	T
F	T	T	T
F	F	F	F

Since $p \vee q$ and $q \vee p$ agree on all the interpretations, $p \vee q \equiv q \vee p$. ■

This example can be generalized to arbitrary *formulas*:

Theorem 2.28 *Let* $A_1, A_2 \in \mathscr{F}$. *Then* $A_1 \vee A_2 \equiv A_2 \vee A_1$.

Proof Let $\mathscr{I}$ be an arbitrary interpretation for $A_1 \vee A_2$. Obviously, $\mathscr{I}$ is also an interpretation for $A_2 \vee A_1$ since $\mathscr{P}_{A_1} \cup \mathscr{P}_{A_2} = \mathscr{P}_{A_2} \cup \mathscr{P}_{A_1}$.

Since $\mathscr{P}_{A_1} \subseteq \mathscr{P}_{A_1} \cup \mathscr{P}_{A_2}$, $\mathscr{I}$ assigns truth values to all atoms in A_1 and can be considered to be an interpretation for A_1. Similarly, $\mathscr{I}$ can be considered to be an interpretation for A_2.

Now $v_{\mathscr{I}}(A_1 \vee A_2) = T$ if and only if either $v_{\mathscr{I}}(A_1) = T$ or $v_{\mathscr{I}}(A_2) = T$, and $v_{\mathscr{I}}(A_2 \vee A_1) = T$ if and only if either $v_{\mathscr{I}}(A_2) = T$ or $v_{\mathscr{I}}(A_1) = T$. If $v_{\mathscr{I}}(A_1) = T$, then:

$$v_{\mathscr{I}}(A_1 \vee A_2) = T = v_{\mathscr{I}}(A_2 \vee A_1),$$

and similarly if $v_{\mathscr{I}}(A_2) = T$. Since $\mathscr{I}$ was arbitrary, $A_1 \vee A_2 \equiv A_2 \vee A_1$. ■

This type of argument will be used frequently. In order to prove that something is true of *all* interpretations, we let $\mathscr{I}$ be an *arbitrary* interpretation and then write a proof without using any property that distinguishes one interpretation from another.

2.3.1 The Relationship Between $\leftrightarrow$ and $\equiv$

Equivalence, $\leftrightarrow$, is a Boolean operator in propositional logic and can appear in formulas of the logic. Logical equivalence, $\equiv$, is not a Boolean operator; instead, is a notation for a *property* of pairs of formulas in propositional logic. There is potential for confusion because we are using a similar vocabulary both for the *object language*, in this case the language of propositional logic, and for the *metalanguage* that we use reason about the object language.

Equivalence and logical equivalence are, nevertheless, closely related as shown by the following theorem:

Theorem 2.29 $A_1 \equiv A_2$ *if and only if* $A_1 \leftrightarrow A_2$ *is true in every interpretation.*

Proof Suppose that $A_1 \equiv A_2$ and let $\mathscr{I}$ be an arbitrary interpretation; then $v_{\mathscr{I}}(A_1) = v_{\mathscr{I}}(A_2)$ by definition of logical equivalence. From Fig. 2.3, $v_{\mathscr{I}}(A_1 \leftrightarrow A_2) = T$. Since $\mathscr{I}$ was arbitrary, $v_{\mathscr{I}}(A_1 \leftrightarrow A_2) = T$ in all interpretations. The proof of the converse is similar. ■

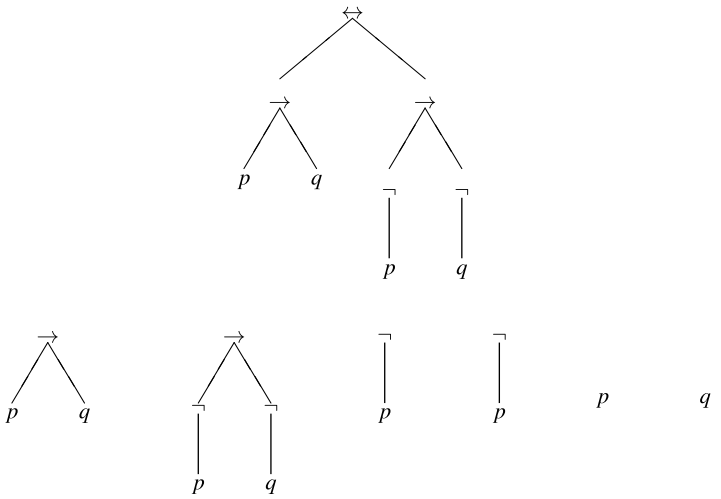


Fig. 2.4 Subformulas

2.3.2 Substitution

Logical equivalence justifies substitution of one formula for another.

Definition 2.30 A is a *subformula* of B if A is a subtree of B. If A is not the same as B, it is a *proper subformula* of B. ■

Example 2.31 Figure 2.4 shows a formula (the left formula from Fig. 2.1) and its proper subformulas. Represented as strings, $(p \to q) \leftrightarrow (\neg p \to \neg q)$ contains the proper subformulas: $p \to q$, $\neg p \to \neg q$, $\neg p$, $\neg q$, p, q. ■

Definition 2.32 Let A be a subformula of B and let A' be any formula. $B\{A \leftarrow A'\}$, the *substitution of A' for A in B*, is the formula obtained by replacing all occurrences of the subtree for A in B by A'. ■

Example 2.33 Let $B = (p \to q) \leftrightarrow (\neg p \to \neg q)$, $A = p \to q$ and $A' = \neg p \vee q$.

$$B\{A \leftarrow A'\} = (\neg p \vee q) \leftrightarrow (\neg q \to \neg p).$$

■

Given a formula A, substitution of a logically equivalent formula for a subformula of A does not change its truth value under any interpretation.

Theorem 2.34 *Let A be a subformula of B and let A' be a formula such that $A \equiv A'$. Then $B \equiv B\{A \leftarrow A'\}$.*

Proof Let $\mathscr{I}$ be an arbitrary interpretation. Then $v_{\mathscr{I}}(A) = v_{\mathscr{I}}(A')$ and we must show that $v_{\mathscr{I}}(B) = v_{\mathscr{I}}(B')$. The proof is by induction on the depth d of the highest occurrence of the subtree A in B.

If $d = 0$, there is only one occurrence of A, namely B itself. Obviously, $v_{\mathscr{I}}(B) = v_{\mathscr{I}}(A) = v_{\mathscr{I}}(A') = v_{\mathscr{I}}(B')$.

If $d \neq 0$, then B is $\neg B_1$ or $B_1 \, op \, B_2$ for some formulas B_1, B_2 and operator op. In B_1, the depth of A is less than d. By the inductive hypothesis, $v_{\mathscr{I}}(B_1) = v_{\mathscr{I}}(B_1') = v_{\mathscr{I}}(B_1\{A \leftarrow A'\})$, and similarly $v_{\mathscr{I}}(B_2) = v_{\mathscr{I}}(B_2') = v_{\mathscr{I}}(B_2\{A \leftarrow A'\})$. By the definition of v on the Boolean operators, $v_{\mathscr{I}}(B) = v_{\mathscr{I}}(B')$. ■

2.3.3 Logically Equivalent Formulas

Substitution of logically equivalence formulas is frequently done, for example, to simplify a formula, and it is essential to become familiar with the common equivalences that are listed in this subsection. Their proofs are elementary from the definitions and are left as exercises.

Absorption of Constants

Let us extend the syntax of Boolean formulas to include the two constant atomic propositions *true* and *false*. (Another notation is $\top$ for *true* and $\bot$ for *false*.) Their semantics are defined by $\mathscr{I}(true) = T$ and $\mathscr{I}(false) = F$ for any interpretation. Do not confuse these symbols in the object language of propositional logic with the truth values T and F used to define interpretations. Alternatively, it is possible to regard *true* and *false* as abbreviations for the formulas $p \vee \neg p$ and $p \wedge \neg p$, respectively.

The appearance of a constant in a formula can collapse the formula so that the binary operator is no longer needed; it can even make a formula become a constant whose truth value no longer depends on the non-constant subformula.

$$
\begin{array}{lcllcl}
A \vee true & \equiv & true & A \wedge true & \equiv & A \\
A \vee false & \equiv & A & A \wedge false & \equiv & false \\
A \rightarrow true & \equiv & true & true \rightarrow A & \equiv & A \\
A \rightarrow false & \equiv & \neg A & false \rightarrow A & \equiv & true \\
A \leftrightarrow true & \equiv & A & A \oplus true & \equiv & \neg A \\
A \leftrightarrow false & \equiv & \neg A & A \oplus false & \equiv & A
\end{array}
$$

Identical Operands

Collapsing can also occur when both operands of an operator are the same or one is the negation of another.

$$
\begin{array}{lcllcl}
A & \equiv & \neg\neg A & & & \\
A & \equiv & A \wedge A & A & \equiv & A \vee A \\
A \vee \neg A & \equiv & true & A \wedge \neg A & \equiv & false \\
A \rightarrow A & \equiv & true & & & \\
A \leftrightarrow A & \equiv & true & A \oplus A & \equiv & false \\
\neg A & \equiv & A \uparrow A\neg A & \equiv & A \downarrow A &
\end{array}
$$

Commutativity, Associativity and Distributivity

The binary Boolean operators are commutative, except for implication.

$$
\begin{array}{lcllcl}
A \vee B & \equiv & B \vee A & A \wedge B & \equiv & B \wedge A \\
A \leftrightarrow B & \equiv & B \leftrightarrow A & A \oplus B & \equiv & B \oplus A \\
A \uparrow B & \equiv & B \uparrow A & A \downarrow B & \equiv & B \downarrow A
\end{array}
$$

If negations are added, the direction of an implication can be reversed:

$$A \rightarrow B \equiv \neg B \rightarrow \neg A$$

The formula $\neg B \rightarrow \neg A$ is the *contrapositive* of $A \rightarrow B$.

Disjunction, conjunction, equivalence and non-equivalence are associative.

$$
\begin{array}{lcllcl}
A \vee (B \vee C) & \equiv & (A \vee B) \vee C & A \wedge (B \wedge C) & \equiv & (A \wedge B) \wedge C \\
A \leftrightarrow (B \leftrightarrow C) & \equiv & (A \leftrightarrow B) \leftrightarrow C & A \oplus (B \oplus C) & \equiv & (A \oplus B) \oplus C
\end{array}
$$

Implication, *nor* and *nand* are not associative.

Disjunction and conjunction distribute over each other.

$$
\begin{array}{lcl}
A \vee (B \wedge C) & \equiv & (A \vee B) \wedge (A \vee C) \\
A \wedge (B \vee C) & \equiv & (A \wedge B) \vee (A \wedge C)
\end{array}
$$

Defining One Operator in Terms of Another

When proving theorems *about* propositional logic using structural induction, we have to prove the inductive step for each of the binary operators. It will simplify proofs if we can eliminate some of the operators by replacing subformulas with formulas that use another operator. For example, equivalence can be eliminated be-

cause it can be defined in terms of conjunction and implication. Another reason for eliminating operators is that many algorithms on propositional formulas require that the formulas be in a *normal form*, using a specified subset of the Boolean operators. Here is a list of logical equivalences that can be used to eliminate operators.

$$
\begin{array}{lcllcl}
A \leftrightarrow B & \equiv & (A \rightarrow B) \wedge (B \rightarrow A) & A \oplus B & \equiv & \neg (A \rightarrow B) \vee \neg (B \rightarrow A) \\
A \rightarrow B & \equiv & \neg A \vee B & A \rightarrow B & \equiv & \neg (A \wedge \neg B) \\
A \vee B & \equiv & \neg (\neg A \wedge \neg B) & A \wedge B & \equiv & \neg (\neg A \vee \neg B) \\
A \vee B & \equiv & \neg A \rightarrow B & A \wedge B & \equiv & \neg (A \rightarrow \neg B)
\end{array}
$$

The definition of conjunction in terms of disjunction and negation, and the definition of disjunction in terms of conjunction and negation are called *De Morgan's laws*.

2.4 Sets of Boolean Operators *

From our earliest days in school, we are taught that there are four basic operators in arithmetic: addition, subtraction, multiplication and division. Later on, we learn about additional operators like modulo and absolute value. On the other hand, multiplication and division are theoretically redundant because they can be defined in terms of addition and subtraction.

In this section, we will look at two issues: What Boolean operators are there? What sets of operators are adequate, meaning that all other operators can be defined using just the operators in the set?

2.4.1 Unary and Binary Boolean Operators

Since there are only two Boolean values T and F, the number of possible n-place operators is 2^{2^n}, because for each of the n arguments we can choose either of the two values T and F and for each of these 2^n n-tuples of arguments we can choose the value of the operator to be either T or F. We will restrict ourselves to one- and two-place operators.

The following table shows the $2^{2^1} = 4$ possible one-place operators, where the first column gives the value of the operand x and the other columns give the value of the nth operator $\circ_n(x)$:

x	$\circ_1$	$\circ_2$	$\circ_3$	$\circ_4$
T	T	T	F	F
F	T	F	T	F

x_1	x_2	$\circ_1$	$\circ_2$	$\circ_3$	$\circ_4$	$\circ_5$	$\circ_6$	$\circ_7$	$\circ_8$
T	T	T	T	T	T	T	T	T	T
T	F	T	T	T	T	F	F	F	F
F	T	T	T	F	F	T	T	F	F
F	F	T	F	T	F	T	F	T	F

x_1	x_2	$\circ_9$	$\circ_{10}$	$\circ_{11}$	$\circ_{12}$	$\circ_{13}$	$\circ_{14}$	$\circ_{15}$	$\circ_{16}$
T	T	F	F	F	F	F	F	F	F
T	F	T	T	T	T	F	F	F	F
F	T	T	T	F	F	T	T	F	F
F	F	T	F	T	F	T	F	T	F

Fig. 2.5 Two-place Boolean operators

Of the four one-place operators, three are trivial: $\circ_1$ and $\circ_4$ are the constant operators, and $\circ_2$ is the identity operator which simply maps the operand to itself. The only non-trivial one-place operator is $\circ_3$ which is negation.

There are $2^{2^2} = 16$ two-place operators (Fig. 2.5). Several of the operators are trivial: $\circ_1$ and $\circ_{16}$ are constant; $\circ_4$ and $\circ_6$ are projection operators, that is, their value is determined by the value of only one of operands; $\circ_{11}$ and $\circ_{13}$ are the negations of the projection operators.

The correspondence between the operators in the table and those we defined in Definition 2.1 are shown in the following table, where the operators in the right-hand column are the negations of those in the left-hand column.

op	name	symbol	op	name	symbol
$\circ_2$	disjunction	$\vee$	$\circ_{15}$	nor	$\downarrow$
$\circ_8$	conjunction	$\wedge$	$\circ_9$	nand	$\uparrow$
$\circ_5$	implication	$\rightarrow$			
$\circ_7$	equivalence	$\leftrightarrow$	$\circ_{10}$	exclusive or	$\oplus$

The operator $\circ_{12}$ is the negation of implication and is not used. Reverse implication, $\circ_3$, is used in logic programming (Chap. 11); its negation, $\circ_{14}$, is not used.

2.4.2 Adequate Sets of Operators

Definition 2.35 A binary operator $\circ$ *is defined from* a set of operators $\{\circ_1, \ldots, \circ_n\}$ iff there is a logical equivalence $A_1 \circ A_2 \equiv A$, where A is a formula constructed from occurrences of A_1 and A_2 using the operators $\{\circ_1, \ldots, \circ_n\}$. The unary operator $\neg$

is defined by a formula $\neg A_1 \equiv A$, where A is constructed from occurrences of A_1 and the operators in the set. ■

Theorem 2.36 *The Boolean operators* $\vee, \wedge, \rightarrow, \leftrightarrow, \oplus, \uparrow, \downarrow$ *can be defined from negation and* one *of* $\vee, \wedge, \rightarrow$.

Proof The theorem follows by using the logical equivalences in Sect. 2.3.3. The *nand* and *nor* operators are the negations of conjunction and disjunction, respectively. Equivalence can be defined from implication and conjunction and non-equivalence can be defined using these operators and negation. Therefore, we need only $\rightarrow, \vee, \wedge$, but each of these operators can be defined by one of the others and negation as shown by the equivalences on page 26. ■

It may come as a surprise that it is possible to define all Boolean operators from either *nand* or *nor* alone. The equivalence $\neg A \equiv A \uparrow A$ is used to define negation from *nand* and the following sequence of equivalences shows how conjunction can be defined:

$$\begin{array}{lcl} (A \uparrow B) \uparrow (A \uparrow B) & \equiv & \text{by the definition of } \uparrow \\ \neg((A \uparrow B) \wedge (A \uparrow B)) & \equiv & \text{by idempotence} \\ \neg(A \uparrow B) & \equiv & \text{by the definition of } \uparrow \\ \neg\neg(A \wedge B) & \equiv & \text{by double negation} \\ A \wedge B. & & \end{array}$$

From the formulas for negation and conjunction, all other operators can be defined. Similarly definitions are possible using *nor*.

In fact it can be proved that only *nand* and *nor* have this property.

Theorem 2.37 *Let* $\circ$ *be a binary operator that can define negation and all other binary operators by itself. Then* $\circ$ *is either nand or nor.*

Proof We give an outline of the proof and leave the details as an exercise.

Suppose that $\circ$ is an operator that can define all the other operators. Negation must be defined by an equivalence of the form:

$$\neg A \equiv A \circ \cdots \circ A.$$

Any binary operator *op* must be defined by an equivalence:

$$A_1 \; op \; A_2 \equiv B_1 \circ \cdots \circ B_n,$$

where each B_i is either A_1 or A_2. (If $\circ$ is not associative, add parentheses as necessary.) We will show that these requirements impose restrictions on $\circ$ so that it must be *nand* or *nor*.

Let $\mathscr{I}$ be any interpretation such that $v_{\mathscr{I}}(A) = T$; then

$$F = v_{\mathscr{I}}(\neg A) = v_{\mathscr{I}}(A \circ \cdots \circ A).$$

Prove by induction on the number of occurrences of $\circ$ that $v_{\mathscr{I}}(A_1 \circ A_2) = F$ when $v_{\mathscr{I}}(A_1) = T$ and $v_{\mathscr{I}}(A_2) = T$. Similarly, if $\mathscr{I}$ is an interpretation such that $v_{\mathscr{I}}(A) = F$, prove that $v_{\mathscr{I}}(A_1 \circ A_2) = T$.

Thus the only freedom we have in defining $\circ$ is in the case where the two operands are assigned different truth values:

A_1	A_2	$A_1 \circ A_2$
T	T	F
T	F	T or F
F	T	T or F
F	F	T

If $\circ$ is defined to give the same truth value T for these two lines then $\circ$ is *nand*, and if $\circ$ is defined to give the same truth value F then $\circ$ is *nor*.

The remaining possibility is that $\circ$ is defined to give different truth values for these two lines. Prove by induction that only projection and negated projection are definable in the sense that:

$$B_1 \circ \cdots \circ B_n \equiv \neg \cdots \neg B_i$$

for some i and zero or more negations. ■

2.5 Satisfiability, Validity and Consequence

We now define the fundamental concepts of the semantics of formulas:

Definition 2.38 Let $A \in \mathscr{F}$.

- A is *satisfiable* iff $v_{\mathscr{I}}(A) = T$ for *some* interpretation $\mathscr{I}$.
  A satisfying interpretation is a *model* for A.
- A is *valid*, denoted $\models A$, iff $v_{\mathscr{I}}(A) = T$ for *all* interpretations $\mathscr{I}$.
  A valid propositional formula is also called a *tautology*.
- A is *unsatisfiable* iff it is not satisfiable, that is, if $v_{\mathscr{I}}(A) = F$ for *all* interpretations $\mathscr{I}$.
- A is *falsifiable*, denoted $\not\models A$, iff it is not valid, that is, if $v_{\mathscr{I}}(A) = F$ for *some* interpretation v. ■

These concepts are illustrated in Fig. 2.6.
The four semantical concepts are closely related.

Theorem 2.39 *Let $A \in \mathscr{F}$. A is valid if and only if $\neg A$ is unsatisfiable. A is satisfiable if and only if $\neg A$ is falsifiable.*

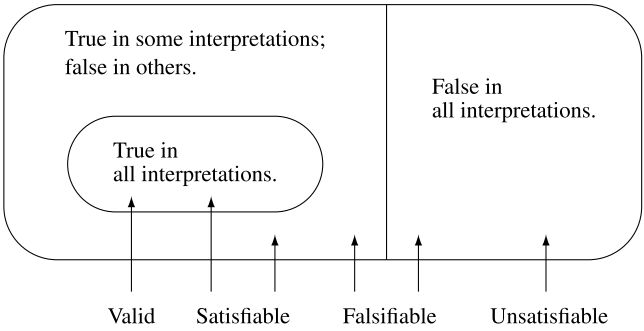


Fig. 2.6 Satisfiability and validity of formulas

Proof Let $\mathscr{I}$ be an arbitrary interpretation. $v_{\mathscr{I}}(A) = T$ if and only if $v_{\mathscr{I}}(\neg A) = F$ by the definition of the truth value of a negation. Since $\mathscr{I}$ was arbitrary, A is true in all interpretations if and only if $\neg A$ is false in all interpretations, that is, iff $\neg A$ is unsatisfiable.

If A is satisfiable then for *some* interpretation $\mathscr{I}$, $v_{\mathscr{I}}(A) = T$. By definition of the truth value of a negation, $v_{\mathscr{I}}(\neg A) = F$ so that $\neg A$ is falsifiable. Conversely, if $v_{\mathscr{I}}(\neg A) = F$ then $v_{\mathscr{I}}(A) = T$. ■

2.5.1 Decision Procedures in Propositional Logic

Definition 2.40 Let $\mathscr{U} \subseteq \mathscr{F}$ be a set of formulas. An algorithm is a *decision procedure* for $\mathscr{U}$ if given an arbitrary formula $A \in \mathscr{F}$, it terminates and returns the answer *yes* if $A \in \mathscr{U}$ and the answer *no* if $A \notin \mathscr{U}$. ■

If $\mathscr{U}$ is the set of satisfiable formulas, a decision procedure for $\mathscr{U}$ is called a decision procedure for satisfiability, and similarly for validity.

By Theorem 2.39, a decision procedure for satisfiability can be used as a decision procedure for validity. To decide if A is valid, apply the decision procedure for satisfiability to $\neg A$. If it reports that $\neg A$ is satisfiable, then A is not valid; if it reports that $\neg A$ is not satisfiable, then A is valid. Such an decision procedure is called a *refutation procedure*, because we prove the validity of a formula by refuting its negation. Refutation procedures can be efficient algorithms for deciding validity, because instead of checking that the formula is always true, we need only search for a falsifying counterexample.

The existence of a decision procedure for satisfiability in propositional logic is trivial, because we can build a truth table for any formula. The truth table in Example 2.21 shows that $p \to q$ is satisfiable, but not valid; Example 2.22 shows that $(p \to q) \leftrightarrow (\neg q \to \neg p)$ is valid. The following example shows an unsatisfiable formula.

Example 2.41 The formula $(p \vee q) \wedge \neg p \wedge \neg q$ is unsatisfiable because all lines of its truth table evaluate to F.

p	q	$p \vee q$	$\neg p$	$\neg q$	$(p \vee q) \wedge \neg p \wedge \neg q$
T	T	T	F	F	F
T	F	T	F	T	F
F	T	T	T	F	F
F	F	F	T	T	F

■

The method of truth tables is a very inefficient decision procedure because we need to evaluate a formula for each of 2^n possible interpretations, where n is the number of distinct atoms in the formula. In later chapters we will discuss more efficient decision procedures for satisfiability, though it is extremely unlikely that there is a decision procedure that is efficient for all formulas (see Sect. 6.7).

2.5.2 Satisfiability of a Set of Formulas

The concept of satisfiability can be extended to a set of formulas.

Definition 2.42 A set of formulas $U = \{A_1, \ldots\}$ is *(simultaneously) satisfiable* iff there exists an interpretation $\mathscr{I}$ such that $v_{\mathscr{I}}(A_i) = T$ for all i. The satisfying interpretation is a *model* of U. U is *unsatisfiable* iff for every interpretation $\mathscr{I}$, there exists an i such that $v_{\mathscr{I}}(A_i) = F$. ■

Example 2.43 The set $U_1 = \{p, \neg p \vee q, q \wedge r\}$ is simultaneously satisfiable by the interpretation which assigns T to each atom, while the set $U_2 = \{p, \neg p \vee q, \neg p\}$ is unsatisfiable. Each formula in U_2 is satisfiable by itself, but the set is not simultaneously satisfiable. ■

The proofs of the following elementary theorems are left as exercises.

Theorem 2.44 *If U is satisfiable, then so is $U - \{A_i\}$ for all i.*

Theorem 2.45 *If U is satisfiable and B is valid, then $U \cup \{B\}$ is satisfiable.*

Theorem 2.46 *If U is unsatisfiable, then for any formula B, $U \cup \{B\}$ is unsatisfiable.*

Theorem 2.47 *If U is unsatisfiable and for some i, A_i is valid, then $U - \{A_i\}$ is unsatisfiable.*

2.5.3 Logical Consequence

Definition 2.48 Let U be a set of formulas and A a formula. A is a *logical consequence* of U, denoted $U \models A$, iff every model of U is a model of A. ■

The formula A need not be true in every possible interpretation, only in those interpretations which satisfy U, that is, those interpretations which satisfy every formula in U. If U is empty, logical consequence is the same as validity.

Example 2.49 Let $A = (p \vee r) \wedge (\neg q \vee \neg r)$. Then A is a logical consequence of $\{p, \neg q\}$, denoted $\{p, \neg q\} \models A$, since A is true in all interpretations $\mathscr{I}$ such that $\mathscr{I}(p) = T$ and $\mathscr{I}(q) = F$. However, A is not valid, since it is not true in the interpretation $\mathscr{I}'$ where $\mathscr{I}'(p) = F$, $\mathscr{I}'(q) = T$, $\mathscr{I}'(r) = T$. ■

The caveat concerning $\leftrightarrow$ and $\equiv$ also applies to $\rightarrow$ and $\models$. Implication, $\rightarrow$, is an operator in the object language, while $\models$ is a symbol for a concept in the metalanguage. However, as with equivalence, the two concepts are related:

Theorem 2.50 *$U \models A$ if and only if $\models \bigwedge_i A_i \rightarrow A$.*

Definition 2.51 $\bigwedge_{i=1}^{i=n} A_i$ is an abbreviation for $A_1 \wedge \cdots \wedge A_n$. The notation $\bigwedge_i$ is used if the bounds are obvious from the context or if the set of formulas is infinite. A similar notation $\bigvee$ is used for disjunction. ■

Example 2.52 From Example 2.49, $\{p, \neg q\} \models (p \vee r) \wedge (\neg q \vee \neg r)$, so by Theorem 2.50, $\models (p \wedge \neg q) \rightarrow (p \vee r) \wedge (\neg q \vee \neg r)$. ■

The proof of Theorem 2.50, as well as the proofs of the following two theorems are left as exercises.

Theorem 2.53 *If $U \models A$ then $U \cup \{B\} \models A$ for any formula B.*

Theorem 2.54 *If $U \models A$ and B is valid then $U - \{B\} \models A$.*

2.5.4 Theories *

Logical consequence is the central concept in the foundations of mathematics. Valid logical formulas such as $p \vee q \leftrightarrow q \vee p$ are of little mathematical interest. It is much more interesting to assume that a set of formulas is true and then to investigate the consequences of these assumptions. For example, Euclid assumed five formulas about geometry and deduced an extensive set of logical consequences. The formal definition of a mathematical theory is as follows.

Definition 2.55 Let $\mathscr{T}$ be a set of formulas. $\mathscr{T}$ is *closed under logical consequence* iff for all formulas A, if $\mathscr{T} \models A$ then $A \in \mathscr{T}$. A set of formulas that is closed under logical consequence is a *theory*. The elements of $\mathscr{T}$ are *theorems*. ■

Theories are constructed by selecting a set of formulas called *axioms* and deducing their logical consequences.

Definition 2.56 Let $\mathscr{T}$ be a theory. $\mathscr{T}$ is said to be *axiomatizable* iff there exists a set of formulas U such that $\mathscr{T} = \{A \mid U \models A\}$. The set of formulas U are the *axioms* of $\mathscr{T}$. If U is finite, $\mathscr{T}$ is said to be *finitely axiomatizable*. ■

Arithmetic is axiomatizable: There is a set of axioms developed by Peano whose logical consequences are theorems of arithmetic. Arithmetic is not finitely axiomatizable, because the induction axiom is not by a single axiom but an axiom scheme with an instance for each property in arithmetic.

2.6 Semantic Tableaux

The method of *semantic tableaux* is an efficient decision procedure for satisfiability (and by duality validity) in propositional logic. We will use semantic tableaux extensively in the next chapter to prove important theorems about deductive systems. The principle behind semantic tableaux is very simple: search for a model (satisfying interpretation) by decomposing the formula into sets of atoms and negations of atoms. It is easy to check if there is an interpretation for each set: a set of atoms and negations of atoms is satisfiable iff the set does not contain an atom p and its negation $\neg p$. The formula is satisfiable iff one of these sets is satisfiable.

We begin with some definitions and then analyze the satisfiability of two formulas to motivate the construction of semantic tableaux.

2.6.1 Decomposing Formulas into Sets of Literals

Definition 2.57 A *literal* is an atom or the negation of an atom. An atom is a *positive literal* and the negation of an atom is a *negative literal*. For any atom p, $\{p, \neg p\}$ is a *complementary pair of literals*.

For any formula A, $\{A, \neg A\}$ is a *complementary pair of formulas*. A is the *complement* of $\neg A$ and $\neg A$ is the *complement* of A. ■

Example 2.58 In the set of literals $\{\neg p, q, r, \neg r\}$, q and r are positive literals, while $\neg p$ and $\neg r$ are negative literals. The set contains the complementary pair of literals $\{r, \neg r\}$. ■

Example 2.59 Let us analyze the satisfiability of the formula:

$$A = p \wedge (\neg q \vee \neg p)$$

in an arbitrary interpretation $\mathscr{I}$, using the inductive rules for the evaluation of the truth value of a formula.

- The principal operator of A is conjunction, so $v_{\mathscr{I}}(A) = T$ if and only if both $v_{\mathscr{I}}(p) = T$ and $v_{\mathscr{I}}(\neg q \vee \neg p) = T$.
- The principal operator of $\neg q \vee \neg p$ is disjunction, so $v_{\mathscr{I}}(\neg q \vee \neg p) = T$ if and only if either $v_{\mathscr{I}}(\neg q) = T$ or $v_{\mathscr{I}}(\neg p) = T$.
- Integrating the information we have obtained from this analysis, we conclude that $v_{\mathscr{I}}(A) = T$ if and only if either:

  1. $v_{\mathscr{I}}(p) = T$ and $v_{\mathscr{I}}(\neg q) = T$, or
  2. $v_{\mathscr{I}}(p) = T$ and $v_{\mathscr{I}}(\neg p) = T$.

A is satisfiable if and only if there is an interpretation such that (1) holds or an interpretation such that (2) holds. ■

We have reduced the question of the satisfiability of A to a question about the satisfiability of sets of literals.

Theorem 2.60 *A set of literals is satisfiable if and only if it does* not *contain a complementary pair of literals.*

Proof Let L be a set of literals that does not contain a complementary pair. Define the interpretation $\mathscr{I}$ by:

$$\begin{aligned} \mathscr{I}(p) &= T \quad \text{if } p \in L, \\ \mathscr{I}(p) &= F \quad \text{if } \neg p \in L. \end{aligned}$$

The interpretation is well-defined—there is only one value assigned to each atom in L—since there is no complementary pair of literals in L. Each literal in L evaluates to T so L is satisfiable.

Conversely, if $\{p, \neg p\} \subseteq L$, then for any interpretation $\mathscr{I}$ for the atoms in L, either $v_{\mathscr{I}}(p) = F$ or $v_{\mathscr{I}}(\neg p) = F$, so L is not satisfiable. ■

Example 2.61 Continuing the analysis of the formula $A = p \wedge (\neg q \vee \neg p)$ from Example 2.59, A is satisfiable if and only at least one of the sets $\{p, \neg p\}$ and $\{p, \neg q\}$ does not contain a complementary pair of literals. Clearly, only the second set does not contain a complementary pair of literals. Using the method described in Theorem 2.60, we obtain the interpretation:

$$\mathscr{I}(p) = T, \qquad \mathscr{I}(q) = F.$$

We leave it to the reader to check that for this interpretation, $v_{\mathscr{I}}(A) = T$. ■

The following example shows what happens if a formula is unsatisfiable.

Example 2.62 Consider the formula:

$$B = (p \vee q) \wedge (\neg p \wedge \neg q).$$

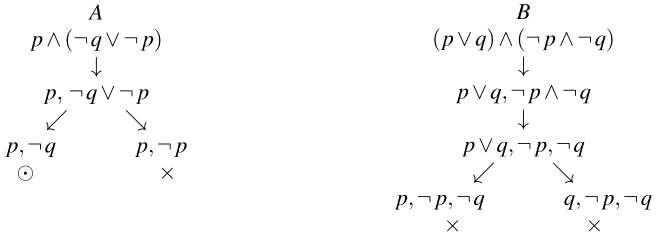


Fig. 2.7 Semantic tableaux

The analysis of the formula proceeds as follows:

- $v_{\mathscr{I}}(B) = T$ if and only if $v_{\mathscr{I}}(p \vee q) = T$ and $v_{\mathscr{I}}(\neg p \wedge \neg q) = T$.
- Decomposing the conjunction, $v_{\mathscr{I}}(B)$=T if and only if $v_{\mathscr{I}}(p \vee q) = T$ and $v_{\mathscr{I}}(\neg p) = v_{\mathscr{I}}(\neg q) = T$.
- Decomposing the disjunction, $v_{\mathscr{I}}(B) = T$ if and only if either:
  1. $v_{\mathscr{I}}(p) = v_{\mathscr{I}}(\neg p) = v_{\mathscr{I}}(\neg q) = T$, or
  2. $v_{\mathscr{I}}(q) = v_{\mathscr{I}}(\neg p) = v_{\mathscr{I}}(\neg q) = T$.

Both sets of literals $\{p, \neg p, \neg q\}$ and $\{q, \neg p, \neg q\}$ contain complementary pairs, so by Theorem 2.60, both set of literals are unsatisfiable. We conclude that it is impossible to find a model for B; in other words, B is unsatisfiable. ■

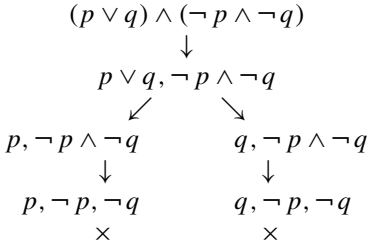
2.6.2 Construction of Semantic Tableaux

The decomposition of a formula into sets of literals is rather difficult to follow when expressed textually, as we did in Examples 2.59 and 2.62. In the method of semantic tableaux, sets of formulas label nodes of a tree, where each path in the tree represents the formulas that must be satisfied in one possible interpretation.

The initial formula labels the root of the tree; each node has one or two child nodes depending on how a formula labeling the node is decomposed. The leaves are labeled by the sets of literals. A leaf labeled by a set of literals containing a complementary pair of literals is marked $\times$, while a leaf labeled by a set not containing a complementary pair is marked $\odot$.

Figure 2.7 shows semantic tableaux for the formulas from the examples.

The tableau construction is not unique; here is another tableau for B:



α	α_1	α_2	β	β_1	β_2
$\neg\neg A_1$	A_1				
$A_1 \wedge A_2$	A_1	A_2	$\neg(B_1 \wedge B_2)$	$\neg B_1$	$\neg B_2$
$\neg(A_1 \vee A_2)$	$\neg A_1$	$\neg A_2$	$B_1 \vee B_2$	B_1	B_2
$\neg(A_1 \rightarrow A_2)$	A_1	$\neg A_2$	$B_1 \rightarrow B_2$	$\neg B_1$	B_2
$\neg(A_1 \uparrow A_2)$	A_1	A_2	$B_1 \uparrow B_2$	$\neg B_1$	$\neg B_2$
$A_1 \downarrow A_2$	$\neg A_1$	$\neg A_2$	$\neg(B_1 \downarrow B_2)$	B_1	B_2
$A_1 \leftrightarrow A_2$	$A_1 \rightarrow A_2$	$A_2 \rightarrow A_1$	$\neg(B_1 \leftrightarrow B_2)$	$\neg(B_1 \rightarrow B_2)$	$\neg(B_2 \rightarrow B_1)$
$\neg(A_1 \oplus A_2)$	$A_1 \rightarrow A_2$	$A_2 \rightarrow A_1$	$B_1 \oplus B_2$	$\neg(B_1 \rightarrow B_2)$	$\neg(B_2 \rightarrow B_1)$

Fig. 2.8 Classification of α- and β-formulas

It is constructed by branching to search for a satisfying interpretation for $p \vee q$ before searching for one for $\neg p \wedge \neg q$. The first tableau contains fewer nodes, showing that it is preferable to decompose conjunctions before disjunctions.

A concise presentation of the rules for creating a semantic tableau can be given if formulas are classified according to their principal operator (Fig. 2.8). If the formula is a negation, the classification takes into account both the negation and the principal operator. α-formulas are conjunctive and are satisfiable only if both subformulas α_1 and α_2 are satisfied, while β-formulas are disjunctive and are satisfied even if only one of the subformulas β_1 or β_2 is satisfiable.

Example 2.63 The formula $p \wedge q$ is classified as an α-formula because it is true if and only if both p and q are true. The formula $\neg(p \wedge q)$ is classified as a β-formula. It is logically equivalent to $\neg p \vee \neg q$ and is true if and only if either $\neg p$ is true or $\neg q$ is true. ■

We now give the algorithm for the construction of a semantic tableau for a formula in propositional logic.

Algorithm 2.64 (Construction of a semantic tableau)
Input: A formula ϕ of propositional logic.
Output: A semantic tableau $\mathcal{T}$ for ϕ all of whose leaves are marked.

Initially, $\mathcal{T}$ is a tree consisting of a single root node labeled with the singleton set $\{\phi\}$. This node is not marked.

Repeat the following step as long as possible: *Choose* an unmarked leaf l labeled with a set of formulas $U(l)$ and apply one of the following rules.

- $U(l)$ is a set of literals. Mark the leaf *closed* $\times$ if it contains a complementary pair of literals. If not, mark the leaf *open* $\odot$.
- $U(l)$ is not a set of literals. *Choose* a formula in $U(l)$ which is not a literal. Classify the formula as an α-formula A or as a β-formula B and perform one of the following steps according to the classification:

– A is an α-formula. Create a new node l' as a child of l and label l' with:

$$U(l') = (U(l) - \{A\}) \cup \{A_1, A_2\}.$$

(In the case that A is $\neg\neg A_1$, there is no A_2.)

– B is a β-formula. Create two new nodes l' and l'' as children of l. Label l' with:

$$U(l') = (U(l) - \{B\}) \cup \{B_1\},$$

and label l'' with:

$$U(l'') = (U(l) - \{B\}) \cup \{B_2\}.$$

■

Definition 2.65 A tableau whose construction has terminated is a *completed tableau*. A completed tableau is *closed* if all its leaves are marked closed. Otherwise (if some leaf is marked open), it is *open*. ■

2.6.3 Termination of the Tableau Construction

Since each step of the algorithm decomposes one formula into one or two simpler formulas, it is clear that the construction of the tableau for any formula terminates, but it is worth proving this claim.

Theorem 2.66 *The construction of a tableau for any formula ϕ terminates. When the construction terminates, all the leaves are marked $\times$ or $\odot$.*

Proof Let us assume that $\leftrightarrow$ and $\oplus$ do not occur in the formula ϕ; the extension of the proof for these cases is left as an exercise.

Consider an unmarked leaf l that is chosen to be expanded during the construction of the tableau. Let $b(l)$ be the total number of binary operators in all formulas in $U(l)$ and let $n(l)$ be the total number of negations in $U(l)$. Define:

$$W(l) = 3 \cdot b(l) + n(l).$$

For example, if $U(l) = \{p \vee q, \neg p \wedge \neg q\}$, then $W(l) = 3 \cdot 2 + 2 = 8$.

Each step of the algorithm adds either a new node l' or a pair of new nodes l', l'' as children of l. We claim that $W(l') < W(l)$ and, if there is a second node, $W(l'') < W(l)$.

Suppose that $A = \neg(A_1 \vee A_2)$ and that the rule for this α-formula is applied at l to obtain a new leaf l' labeled:

$$U(l') = (U(l) - \{\neg(A_1 \vee A_2)\}) \cup \{\neg A_1, \neg A_2\}.$$

Then:

$$W(l') = W(l) - (3 \cdot 1 + 1) + 2 = W(l) - 2 < W(l),$$

because one binary operator and one negation are removed, while two negations are added.

Suppose now that $B = B_1 \vee B_2$ and that the rule for this β-formula is applied at l to obtain two new leaves l', l'' labeled:

$$\begin{aligned} U(l') &= (U(l) - \{B_1 \vee B_2\}) \cup \{B_1\}, \\ U(l'') &= (U(l) - \{B_1 \vee B_2\}) \cup \{B_2\}. \end{aligned}$$

Then:

$$W(l') \leq W(l) - (3 \cdot 1) < W(l), \qquad W(l'') \leq W(l) - (3 \cdot 1) < W(l).$$

We leave it to the reader to prove that $W(l)$ decreases for the other α- and β-formulas.

The value of $W(l)$ decreases as each branch in the tableau is extended. Since, obviously, $W(l) \geq 0$, no branch can be extended indefinitely and the construction of the tableau must eventually terminate.

A branch can always be extended if its leaf is labeled with a set of formulas that is not a set of literals. Therefore, when the construction of the tableau terminates, all leaves are labeled with sets of literals and each is marked open or closed by the first rule of the algorithm. ■

*2.6.4 Improving the Efficiency of the Algorithm **

The algorithm for constructing a tableau is not deterministic: at most steps, there is a choice of which leaf to extend and if the leaf contains more than one formula which is not a literal, there is a choice of which formula to decompose. This opens the possibility of applying heuristics in order to cause the tableau to be completed quickly. We saw in Sect. 2.6.2 that it is better to decompose α-formulas before β-formulas to avoid duplication.

Tableaux can be shortened by closing a branch if it contains a formula and its negation and not just a pair of complementary literals. Clearly, there is no reason to continue expanding a node containing:

$$(p \wedge (q \vee r)), \quad \neg(p \wedge (q \vee r)).$$

We leave it as an exercise to prove that this modification preserves the correctness of the algorithm.

There is a lot of redundancy in copying formulas from one node to another:

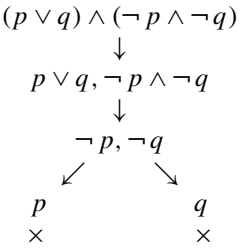
$$U(l') = (U(l) - \{A\}) \cup \{A_1, A_2\}.$$

In a variant of semantic tableaux called *analytic tableaux* (Smullyan, 1968), when a new node is created, it is labeled only with the new formulas:

$$U(l') = \{A_1, A_2\}.$$

The algorithm is changed so that the formula to be decomposed is selected from the set of formulas labeling the nodes on the branch from the root to a leaf (provided, of course, that the formula has not already been selected). A leaf is marked closed if two complementary literals (or formulas) appear in the labels of one or two nodes on a branch, and a leaf is marked open if is not closed but there are no more formulas to decompose.

Here is an analytic tableau for the formula B from Example 2.62, where the formula $p \vee q$ is not copied from the second node to the third when $p \wedge q$ is decomposed:



We prefer to use semantic tableaux because it is easy to see which formulas are candidates for decomposition and how to mark leaves.

2.7 Soundness and Completeness

The construction of a semantic tableau is a purely formal. The decomposition of a formula depends solely on its syntactical properties: its principal operator and—if it is a negation—the principal operator of the formula that is negated. We gave several examples to motivate semantic tableau, but we have not yet proven that the algorithm is correct. We have not connected the syntactical outcome of the algorithm (Is the tableau closed or not?) with the semantical concept of truth value. In this section, we prove that the algorithm is correct in the sense that it reports that a formula is satisfiable or unsatisfiable if and only if there exists or does not exist a model for the formula.

The proof techniques of this section should be studied carefully because they will be used again and again in other logical systems.

Theorem 2.67 Soundness and completeness *Let $\mathcal{T}$ be a completed tableau for a formula A. A is unsatisfiable if and only if $\mathcal{T}$ is closed.*

Here are some corollaries that follow from the theorem.

Corollary 2.68 *A is satisfiable if and only if $\mathcal{T}$ is open.*

Proof A is satisfiable iff (by definition) A is not unsatisfiable iff (by Theorem 2.67) $\mathcal{T}$ is not closed iff (by definition) $\mathcal{T}$ is open. ∎

Corollary 2.69 *A is valid if and only if the tableau for* $\neg A$ *closes.*

Proof A is valid iff $\neg A$ is unsatisfiable iff the tableau for $\neg A$ closes. ■

Corollary 2.70 *The method of semantic tableaux is a decision procedure for validity in propositional logic.*

Proof Let A be a formula of propositional logic. By Theorem 2.66, the construction of the semantic tableau for $\neg A$ terminates in a completed tableau. By the previous corollary, A is valid if and only if the completed tableau is closed. ■

The forward direction of Corollary 2.69 is called *completeness*: if A is valid, we can discover this fact by constructing a tableau for $\neg A$ and the tableau will close. The converse direction is called *soundness*: any formula A that the tableau construction claims valid (because the tableau for $\neg A$ closes) actually is valid. Invariably in logic, soundness is easier to show than completeness. The reason is that while we only include in a formal system rules that are obviously sound, it is hard to be sure that we haven't forgotten some rule that may be needed for completeness. At the extreme, the following vacuous algorithm is sound but far from complete!

Algorithm 2.71 (Incomplete decision procedure for validity)
Input: A formula A of propositional logic.
Output: A is not valid. ■

Example 2.72 If the rule for $\neg(A_1 \vee A_2)$ is omitted, the construction of the tableau is still sound, but it is not complete, because it is impossible to construct a closed tableau for the obviously valid formula $A = \neg p \vee p$. Label the root of the tableau with the negation $\neg A = \neg(\neg p \vee p)$; there is now no rule that can be used to decompose the formula. ■

2.7.1 Proof of Soundness

The theorem to be proved is: if the tableau $\mathscr{T}$ for a formula A closes, then A is unsatisfiable. We will prove a more general theorem: if $\mathscr{T}_n$, the subtree rooted at node n of $\mathscr{T}$, closes then the set of formulas $U(n)$ labeling n is unsatisfiable. Soundness is the special case for the root.

To make the proof easier to follow, we will use $A_1 \wedge A_2$ and $B_1 \vee B_2$ as representatives of the classes of α- and β-formulas, respectively.

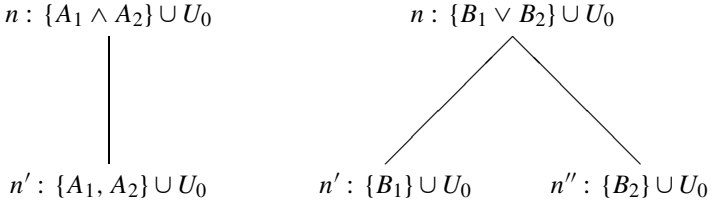
Proof of Soundness The proof is by induction on the height h_n of the node n in $\mathscr{T}_n$. Clearly, a closed leaf is labeled by an unsatisfiable set of formulas. Recall (Definition 2.42) that a set of formulas is unsatisfiable iff for any interpretation the truth value of at least one formula is false. In the inductive step, if the children of a node n

are labeled by an unsatisfiable set of formulas, then: (a) either the unsatisfiable formula also appears in the label of n, or (b) the unsatisfiable formulas in the labels of the children were used to construct an unsatisfiable formula in the label of n. Let us write out the formal proof.

For the base case, $h_n = 0$, assume that $\mathscr{T}_n$ closes. Since $h_n = 0$ means that n is a leaf, $U(n)$ must contain a complementary set of literals so it is unsatisfiable.

For the inductive step, let n be a node such that $h_n > 0$ in $\mathscr{T}_n$. We need to show that $\mathscr{T}_n$ is closed implies that $U(n)$ is unsatisfiable. By the inductive hypothesis, we can assume that for any node m of height $h_m < h_n$, if $\mathscr{T}_m$ closes, then $U(m)$ is unsatisfiable.

Since $h_n > 0$, the rule for some α- or β-formula was used to create the children of n:



Case 1: $U(n) = \{A_1 \wedge A_2\} \cup U_0$ and $U(n') = \{A_1, A_2\} \cup U_0$ for some (possibly empty) set of formulas U_0.
Clearly, $\mathscr{T}_{n'}$ is also a closed tableau and since $h_{n'} = h_n - 1$, by the inductive hypothesis $U(n')$ is unsatisfiable. Let $\mathscr{I}$ be an arbitrary interpretation. There are two possibilities:

- $v_{\mathscr{I}}(A_0) = F$ for some formula $A_0 \in U_0$. But $U_0 \subset U(n)$ so $U(n)$ is also unsatisfiable.
- Otherwise, $v_{\mathscr{I}}(A_0) = T$ for all $A_0 \in U_0$, so $v_{\mathscr{I}}(A_1) = F$ *or* $v_{\mathscr{I}}(A_2) = F$. Suppose that $v_{\mathscr{I}}(A_1) = F$. By the definition of the semantics of $\wedge$, this implies that $v_{\mathscr{I}}(A_1 \wedge A_2) = F$. Since $A_1 \wedge A_2 \in U(n)$, $U(n)$ is unsatisfiable. A similar argument holds if $v_{\mathscr{I}}(A_2) = F$.

Case 2: $U(n) = \{B_1 \vee B_2\} \cup U_0$, $U(n') = \{B_1\} \cup U_0$, and $U(n'') = \{B_2\} \cup U_0$ for some (possibly empty) set of formulas U_0.
Clearly, $\mathscr{T}_{n'}$ and $\mathscr{T}_{n''}$ are also closed tableaux and since $h_{n'} \leq h_n - 1$ and $h_{n''} \leq h_n - 1$, by the inductive hypothesis $U(n')$ and $U(n'')$ are both unsatisfiable. Let $\mathscr{I}$ be an arbitrary interpretation. There are two possibilities:

- $v_{\mathscr{I}}(B_0) = F$ for some formula $B_0 \in U_0$. But $U_0 \subset U(n)$ so $U(n)$ is also unsatisfiable.
- Otherwise, $v_{\mathscr{I}}(B_0) = T$ for all $B_0 \in U_0$, so $v_{\mathscr{I}}(B_1) = F$ (since $U(n')$ is unsatisfiable) *and* $v_{\mathscr{I}}(B_2) = F$ (since $U(n'')$ is unsatisfiable). By the definition of the semantics of $\vee$, this implies that $v_{\mathscr{I}}(B_1 \vee B_2) = F$. Since $B_1 \vee B_2 \in U(n)$, $U(n)$ is unsatisfiable. ■

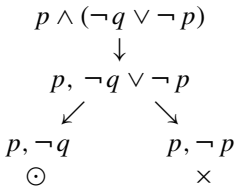
2.7.2 Proof of Completeness

The theorem to be proved is: if A is unsatisfiable then *every* tableau for A closes. Completeness is much more difficult to prove than soundness. For soundness, we had a single (though arbitrary) closed tableau for a formula A and we proved that A is unsatisfiable by induction on the structure of a tableau. Here we need to prove that no matter how the tableau for A is constructed, it must close.

Rather than prove that every tableau must close, we prove the contrapositive (Corollary 2.68): if some tableau for A is open (has an open branch), then A is satisfiable. Clearly, there is a model for the set of literals labeling the leaf of an open branch. We extend this to an interpretation for A and then prove by induction on the length of the branch that the interpretation is a model of the sets of formulas labeling the nodes on the branch, including the singleton set $\{A\}$ that labels the root.

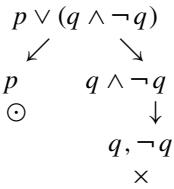
Let us look at some examples.

Example 2.73 Let $A = p \wedge (\neg q \vee \neg p)$. We have already constructed the tableau for A which is reproduced here:



The interpretation $\mathscr{I}(p) = T$, $\mathscr{I}(q) = F$ defined by assigning T to the literals labeling the leaf of the open branch is clearly a model for A. ■

Example 2.74 Now let $A = p \vee (q \wedge \neg q)$; here is a tableau for A:



The open branch of the tableau terminates in a leaf labeled with the singleton set of literals $\{p\}$. We can conclude that any model for A must define $\mathscr{I}(p) = T$. However, an interpretation for A must also define an assignment to q and the leaf gives us no guidance as to which value to choose for $\mathscr{I}(q)$. But it is obvious that it doesn't matter what value is assigned to q; in either case, the interpretation will be a model of A. ■

To prove completeness we need to show that the assignment of T to the literals labeling the leaf of an open branch can be extended to a model of the formula labeling the root. There are four steps in the proof:

1. Define a property of sets of formulas;
2. Show that the union of the formulas labeling nodes in an open branch has this property;
3. Prove that any set having this property is satisfiable;
4. Note that the formula labeling the root is in the set.

Definition 2.75 Let U be a set of formulas. U is a *Hintikka set* iff:

1. For all atoms p appearing in a formula of U, either $p \notin U$ or $\neg p \notin U$.
2. If $A \in U$ is an α-formula, then $A_1 \in U$ *and* $A_2 \in U$.
3. If $B \in U$ is a β-formula, then $B_1 \in U$ *or* $B_2 \in U$. ■

Example 2.76 U, the union of the set of formulas labeling the nodes in the open branch of Example 2.74, is $\{p,\ p \vee (q \wedge \neg q)\}$. We claim that U is a Hintikka set. Condition (1) obviously holds since there is only one literal p in U and $\neg p \notin U$. Condition (2) is vacuous. For Condition (3), $B = p \vee (q \wedge \neg q) \in U$ is a β-formula and $B_1 = p \in U$. ■

Condition (1) requires that a Hintikka set not contain a complementary pair of literals, which to be expected on an open branch of a tableau. Conditions (2) and (3) ensure that U is *downward saturated*, that is, U contains sufficient subformulas so that the decomposition of the formula to be satisfied will not take us out of U. In turn, this ensures that an interpretation defined by the set of literals in U will make all formulas in U true.

The second step of the proof of completeness is to show that the set of formulas labeling the nodes in an open branch is a Hintikka set.

Theorem 2.77 *Let l be an open leaf in a completed tableau $\mathscr{T}$. Let $U = \bigcup_i U(i)$, where i runs over the set of nodes on the branch from the root to l. Then U is a Hintikka set.*

Proof In the construction of the semantic tableau, there are no rules for decomposing a literal p or $\neg p$. Thus if a literal p or $\neg p$ appears for the first time in $U(n)$ for some n, the literal will be copied into $U(k)$ for all nodes k on the branch from n to l, in particular, $p \in U(l)$ or $\neg p \in U(l)$. This means that all literals in U appear in $U(l)$. Since the branch is open, no complementary pair of literals appears in $U(l)$, so Condition (1) holds for U.

Suppose that $A \in U$ is an α-formula. Since the tableau is completed, A was the formula selected for decomposing at some node n in the branch from the root to l. Then $\{A_1, A_2\} \subseteq U(n') \subseteq U$, so Condition (2) holds.

Suppose that $B \in U$ is an β-formula. Since the tableau is completed, B was the formula selected for decomposing at some node n in the branch from the root to l. Then either $B_1 \in U(n') \subseteq U$ or $B_2 \in U(n') \subseteq U$, so Condition (3) holds. ■

The third step of the proof is to show that a Hintikka set is satisfiable.

Theorem 2.78 (Hintikka's Lemma) *Let U be a Hintikka set. Then U is satisfiable.*

Proof We define an interpretation and then show that the interpretation is a model of U. Let $\mathscr{P}_U$ be set of all atoms appearing in all formulas of U. Define an interpretation $\mathscr{I} : \mathscr{P}_U \mapsto \{T, F\}$ as follows:

$$\begin{aligned} \mathscr{I}(p) &= T \qquad \text{if } p \in U, \\ \mathscr{I}(p) &= F \qquad \text{if } \neg p \in U, \\ \mathscr{I}(p) &= T \qquad \text{if } p \notin U \text{ and } \neg p \notin U. \end{aligned}$$

Since U is a Hintikka set, by Condition (1) $\mathscr{I}$ is well-defined, that is, every atom in $\mathscr{P}_U$ is given exactly one value. Example 2.74 demonstrates the third case: the atom q appears in a *formula* of U so $q \in \mathscr{P}_U$, but neither the literal q nor its complement $\neg q$ appear in U. The atom is arbitrarily mapped to the truth value T.

We show by structural induction that for any $A \in U$, $v_{\mathscr{I}}(A) = T$.

- If A is an atom p, then $v_{\mathscr{I}}(A) = v_{\mathscr{I}}(p) = \mathscr{I}(p) = T$ since $p \in U$.
- If A is a negated atom $\neg p$, then since $\neg p \in U$, $\mathscr{I}(p) = F$, so $v_{\mathscr{I}}(A) = v_{\mathscr{I}}(\neg p) = T$.
- If A is an α-formula, by Condition (2) $A_1 \in U$ and $A_2 \in U$. By the inductive hypothesis, $v_{\mathscr{I}}(A_1) = v_{\mathscr{I}}(A_2) = T$, so $v_{\mathscr{I}}(A) = T$ by definition of the conjunctive operators.
- If A is β-formula B, by Condition (3) $B_1 \in U$ or $B_2 \in U$. By the inductive hypothesis, either $v_{\mathscr{I}}(B_1) = T$ or $v_{\mathscr{I}}(B_2) = T$, so $v_{\mathscr{I}}(A) = v_{\mathscr{I}}(B) = T$ by definition of the disjunctive operators. ■

Proof of Completeness Let $\mathscr{T}$ be a completed open tableau for A. Then U, the union of the labels of the nodes on an open branch, is a Hintikka set by Theorem 2.77. Theorem 2.78 shows an interpretation $\mathscr{I}$ can be found such that U is simultaneously satisfiable in $\mathscr{I}$. A, the formula labeling the root, is an element of U so $\mathscr{I}$ is a model of A. ■

2.8 Summary

The presentation of propositional logic was carried out in a manner that we will use for all systems of logic. First, the syntax of formulas is given. The formulas are defined as trees, which avoids ambiguity and simplifies the description of structural induction.

The second step is to define the semantics of formulas. An interpretation is a mapping of atomic propositions to the values $\{T, F\}$. An interpretation is used to give a truth value to any formula by induction on the structure of the formula, starting from atoms and proceeding to more complex formulas using the definitions of the Boolean operators.

A formula is satisfiable iff it is true in *some* interpretation and it is valid iff is true in *all* interpretations. Two formulas whose values are the same in all interpretations are logically equivalent and can be substituted for each other. This can be used to show that for any formula, there exists a logically equivalent formula that uses only negation and either conjunction or disjunction.

While truth tables can be used as a decision procedure for the satisfiability or validity of formulas of propositional logic, semantic tableaux are usually much more efficient. In a semantic tableau, a tree is constructed during a search for a model of a formula; the construction is based upon the structure of the formula. A semantic tableau is closed if the formula is unsatisfiable and open if it is satisfiable.

We proved that the algorithm for semantic tableaux is sound and complete as a decision procedure for satisfiability. This theorem connects the syntactical aspect of a formula that guides the construction of the tableau with its meaning. The central concept in the proof is that of a Hintikka set, which gives conditions that ensure that a model can be found for a set of formulas.

2.9 Further Reading

The presentation of semantic tableaux follows that of Smullyan (1968) although he uses analytic tableaux. Advanced textbooks that also use tableaux are Nerode and Shore (1997) and Fitting (1996).

2.10 Exercises

2.1 Draw formation trees and construct truth tables for

$$(p \to (q \to r)) \to ((p \to q) \to (p \to r)),$$
$$(p \to q) \to p,$$
$$((p \to q) \to p) \to p.$$

2.2 Prove that there is a unique formation tree for every derivation tree.

2.3 Prove the following logical equivalences:

$$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C),$$
$$A \vee B \equiv \neg(\neg A \wedge \neg B),$$
$$A \wedge B \equiv \neg(\neg A \vee \neg B),$$
$$A \to B \equiv \neg A \vee B,$$
$$A \to B \equiv \neg(A \wedge \neg B).$$

2.4 Prove $((A \oplus B) \oplus B) \equiv A$ and $((A \leftrightarrow B) \leftrightarrow B) \equiv A$.

2.5 Simplify $A \wedge (A \vee B)$ and $A \vee (A \wedge B)$.

2.6 Prove the following logical equivalences using truth tables, semantic tableaux or Venn diagrams:

$$\begin{aligned} A \to B &\equiv A \leftrightarrow (A \wedge B), \\ A \to B &\equiv B \leftrightarrow (A \vee B), \\ A \wedge B &\equiv (A \leftrightarrow B) \leftrightarrow (A \vee B), \\ A \leftrightarrow B &\equiv (A \vee B) \to (A \wedge B). \end{aligned}$$

2.7 Prove $\models (A \to B) \vee (B \to C)$.

2.8 Prove or disprove:

$$\models ((A \to B) \to B) \to B,$$

$$\models (A \leftrightarrow B) \leftrightarrow (A \leftrightarrow (B \leftrightarrow A)).$$

2.9 Prove:

$$\models ((A \wedge B) \to C) \to ((A \to C) \vee (B \to C)).$$

This formula may seem strange since it could be misinterpreted as saying that if C follows from $A \wedge B$, then it follows from one or the other of A or B. To clarify this, show that:

$$\{A \wedge B \to C\} \models (A \to C) \vee (B \to C),$$

but:

$$\{A \wedge B \to C\} \not\models A \to C,$$

$$\{A \wedge B \to C\} \not\models B \to C.$$

2.10 Complete the proof that $\uparrow$ and $\downarrow$ can each define all unary and binary Boolean operators (Theorem 2.37).

2.11 Prove that $\wedge$ and $\vee$ cannot define all Boolean operators.

2.12 Prove that $\{\neg, \leftrightarrow\}$ *cannot* define all Boolean operators.

2.13 Prove that $\uparrow$ and $\downarrow$ are not associative.

2.14 Prove that if U is satisfiable then $U \cup \{B\}$ is not necessarily satisfiable.

2.15 Prove Theorems 2.44–2.47 on the satisfiability of sets of formulas.

2.16 Prove Theorems 2.50–2.54 on logical consequence.

2.17 Prove that for a set of axioms U, $\mathscr{T}(U)$ is closed under logical consequence (see Definition 2.55).

2.18 Complete the proof that the construction of a semantic tableau terminates (Theorem 2.66).

2.19 Prove that the method of semantic tableaux remains sound and complete if a tableau can be closed non-atomically.

2.20 Manna (1974) Let *ifte* be a *tertiary* (3-place) operator defined by:

A	B	C	$ifte(A, B, C)$
T	T	T	T
T	T	F	T
T	F	T	F
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	T
F	F	F	F

The operator can be defined using infix notation as:

$$if\ A\ then\ B\ else\ C.$$

1. Prove that *if then else* by itself forms an adequate sets of operators if the use of the constant formulas *true* and *false* is allowed.
2. Prove: $\models if\ A\ then\ B\ else\ C \equiv (A \rightarrow B) \wedge (\neg A \rightarrow C)$.
3. Add a rule for the operator *if then else* to the algorithm for semantic tableaux.

References

M. Fitting. *First-Order Logic and Automated Theorem Proving* (*Second Edition*). Springer, 1996.
J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation* (*Third Edition*). Addison-Wesley, 2006.
Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, New York, NY, 1974. Reprinted by Dover, 2003.
A. Nerode and R.A. Shore. *Logic for Applications* (*Second Edition*). Springer, 1997.
R.M. Smullyan. *First-Order Logic*. Springer-Verlag, 1968. Reprinted by Dover, 1995.

Chapter 3
Propositional Logic: Deductive Systems

The concept of deducing theorems from a set of axioms and rules of inference is very old and is familiar to every high-school student who has studied Euclidean geometry. Modern mathematics is expressed in a style of reasoning that is not far removed from the reasoning used by Greek mathematicians. This style can be characterized as 'formalized informal reasoning', meaning that while the proofs are expressed in natural language rather than in a formal system, there are conventions among mathematicians as to the forms of reasoning that are allowed. The deductive systems studied in this chapter were developed in an attempt to formalize mathematical reasoning.

We present two deductive systems for propositional logic. The second one $\mathscr{H}$ will be familiar because it is a formalization of step-by-step proofs in mathematics: It contains a set of three axioms and one rule of inference; proofs are constructed as a sequence of formulas, each of which is either an axiom (or a formula that has been previously proved) or a derivation of a formula from previous formulas in the sequence using the rule of inference. The system $\mathscr{G}$ will be less familiar because it has one axiom and many rules of inference, but we present it first because it is almost trivial to prove the soundness and completeness of $\mathscr{G}$ from its relationship with semantic tableaux. The proof of the soundness and completeness of $\mathscr{H}$ is then relatively easy to show by using $\mathscr{G}$. The chapter concludes with three short sections: the definition of an important property called *consistency*, a generalization to infinite sets of formulas, and a survey of other deductive systems for propositional logic.

3.1 Why Deductive Proofs?

Let $U = \{A_1, \ldots, A_n\}$. Theorem 2.50 showed that $U \models A$ if and only if $\models A_1 \wedge \cdots \wedge A_n \rightarrow A$. Therefore, if U is a set of axioms, we can use the completeness of the method of semantic tableaux to determine if A follows from U (see Sect. 2.5.4 for precise definitions). Why would we want to go through the trouble of searching for a mathematical proof when we can easily compute if a formula is valid?

M. Ben-Ari, *Mathematical Logic for Computer Science*,
DOI 10.1007/978-1-4471-4129-7_3, © Springer-Verlag London 2012

There are several problems with a purely semantical approach:

- The set of axioms may be infinite. For example, the axiom of induction in arithmetic is really an infinite set of axioms, one for each property to be proved. For semantic tableaux in propositional logic, the only formulas that appear in the tableaux are subformulas of the formula being checked or their negations, and there are only a finite number of such formulas.
- Very few logics have decision procedures like propositional logic.
- A decision procedure may not give insight into the relationship between the axioms and the theorem. For example, in proofs of theorems about prime numbers, we would want to know exactly where primality is used (Velleman, 2006, Sect. 3.7). This understanding can also help us propose other formulas that might be theorems.
- A decision procedure produces a 'yes/no' answer, so it is difficult to recognize intermediate results (lemmas). Clearly, the millions of mathematical theorems in existence could not have been inferred directly from axioms.

Definition 3.1 A *deductive system* is a set of formulas called *axioms* and a set of *rules of inference*. A *proof* in a deductive system is a sequence of formulas $S = \{A_1, \ldots, A_n\}$ such that each formula A_i is either an axiom or it can be inferred from previous formulas of the sequence $A_{j_1}, \ldots, A_{j_k}$, where $j_1 < \cdots < j_k < i$, using a rule of inference. For A_n, the last formula in the sequence, we say that A_n is a *theorem*, the sequence S is a *proof* of A_n, and A_n is *provable*, denoted $\vdash A_n$. If $\vdash A$, then A may be used like an axiom in a subsequent proof. ■

The deductive approach can overcome the problems described above:

- There may be an infinite number of axioms, but only a finite number will appear in any proof.
- Although a proof is not a decision procedure, it can be mechanically *checked*; that is, given a sequence of formulas, an syntax-based algorithm can easily check whether the sequence is a proof as defined above.
- The proof of a formula clearly shows which axioms, theorems and rules are used and for what purposes.
- Once a theorem has been proved, it can be used in proofs like an axiom.

Deductive proofs are not generated by decision procedures because the formulas that appear in a proof are not limited to subformulas of the theorem and because there is no algorithm telling us how to generate the next formula in the sequence forming a proof. Nevertheless, algorithms and heuristics can be used to build software systems called *automatic theorem provers* which search for proofs. In Chap. 4, we will study a deductive system that has been successfully used in automatic theorem provers. Another promising approach is to use a *proof assistant* which performs administrative tasks such as proof checking, bookkeeping and cataloging previously proved theorems, but a person guides the search by suggesting lemmas that are likely to lead to a proof.

α	α_1	α_2	β	β_1	β_2
$\neg\neg A$	A				
$\neg(A_1 \wedge A_2)$	$\neg A_1$	$\neg A_2$	$B_1 \wedge B_2$	B_1	B_2
$A_1 \vee A_2$	A_1	A_2	$\neg(B_1 \vee B_2)$	$\neg B_1$	$\neg B_2$
$A_1 \rightarrow A_2$	$\neg A_1$	A_2	$\neg(B_1 \rightarrow B_2)$	B_1	$\neg B_2$
$A_1 \uparrow A_2$	$\neg A_1$	$\neg A_2$	$\neg(B_1 \uparrow B_2)$	B_1	B_2
$\neg(A_1 \downarrow A_2)$	A_1	A_2	$B_1 \downarrow B_2$	$\neg B_1$	$\neg B_2$
$\neg(A_1 \leftrightarrow A_2)$	$\neg(A_1 \rightarrow A_2)$	$\neg(A_2 \rightarrow A_1)$	$B_1 \leftrightarrow B_2$	$B_1 \rightarrow B_2$	$B_2 \rightarrow B_1$
$A_1 \oplus A_2$	$\neg(A_1 \rightarrow A_2)$	$\neg(A_2 \rightarrow A_1)$	$\neg(B_1 \oplus B_2)$	$B_1 \rightarrow B_2$	$B_2 \rightarrow B_1$

Fig. 3.1 Classification of α- and β-formulas

3.2 Gentzen System $\mathscr{G}$

The first deductive system that we study is based on a system proposed by Gerhard Gentzen in the 1930s. The system itself will seem unfamiliar because it has one type of axiom and many rules of inference, unlike familiar mathematical theories which have multiple axioms and only a few rules of inference. Furthermore, deductions in the system can be naturally represented as trees rather in the linear format characteristic of mathematical proofs. However, it is this property that makes it easy to relate Gentzen systems to semantic tableaux.

Definition 3.2 (Gentzen system $\mathscr{G}$) An *axiom* of $\mathscr{G}$ is a set of literals U containing a complementary pair. *Rule of inference* are used to infer a set of formulas U from one or two other sets of formulas U_1 and U_2; there are two types of rules, defined with reference to Fig. 3.1:

- Let $\{\alpha_1, \alpha_2\} \subseteq U_1$ and let $U_1' = U_1 - \{\alpha_1, \alpha_2\}$. Then $U = U_1' \cup \{\alpha\}$ can be inferred.
- Let $\{\beta_1\} \subseteq U_1$, $\{\beta_2\} \subseteq U_2$ and let $U_1' = U_1 - \{\beta_1\}$, $U_2' = U_2 - \{\beta_2\}$. Then $U = U_1' \cup U_2' \cup \{\beta\}$ can be inferred.

The set or sets of formulas U_1, U_2 are the *premises* and set of formulas U that is inferred is the *conclusion*. A set of formulas U that is an axiom or a conclusion is said to be *proved*, denoted $\vdash U$. The following notation is used for rules of inference:

$$\frac{\vdash U_1' \cup \{\alpha_1, \alpha_2\}}{\vdash U_1' \cup \{\alpha\}} \qquad \frac{\vdash U_1' \cup \{\beta_1\} \qquad \vdash U_2' \cup \{\beta_2\}}{\vdash U_1' \cup U_2' \cup \{\beta\}}.$$

Braces can be omitted with the understanding that a sequence of formulas is to be interpreted as a set (with no duplicates). ■

Example 3.3 The following set of formulas is an axiom because it contains the complementary pair $\{r, \neg r\}$:

$$\vdash p \wedge q, q, r, \neg r, q \vee \neg r.$$

The disjunction rule for $A_1 = q$, $A_2 = \neg r$ can be used to deduce:

$$\frac{\vdash p \wedge q, q, r, \neg r, q \vee \neg r}{\vdash p \wedge q, r, q \vee \neg r, q \vee \neg r}.$$

Removing the duplicate formula $q \vee \neg r$ gives:

$$\frac{\vdash p \wedge q, q, r, \neg r, q \vee \neg r}{\vdash p \wedge q, r, q \vee \neg r}.$$

Note that the premises $\{q, \neg r\}$ are no longer elements of the conclusion. ■

A proof can be written as a sequence of sets of formulas, which are numbered for convenient reference. On the right of each line is its *justification*: either the set of formulas is an axiom, or it is the conclusion of a rule of inference applied to a set or sets of formulas earlier in the sequence. A rule of inference is identified by the rule used for the α- or β-formula on the principal operator of the conclusion and by the number or numbers of the lines containing the premises.

Example 3.4 Prove $\vdash (p \vee q) \rightarrow (q \vee p)$ in $\mathscr{G}$.

Proof

1.	$\vdash \neg p, q, p$	Axiom
2.	$\vdash \neg q, q, p$	Axiom
3.	$\vdash \neg (p \vee q), q, p$	$\beta\vee$, 1, 2
4.	$\vdash \neg (p \vee q), (q \vee p)$	$\alpha\vee$, 3
5.	$\vdash (p \vee q) \rightarrow (q \vee p)$	$\alpha\rightarrow$, 4

■

Example 3.5 Prove $\vdash p \vee (q \wedge r) \rightarrow (p \vee q) \wedge (p \vee r)$ in $\mathscr{G}$.

Proof

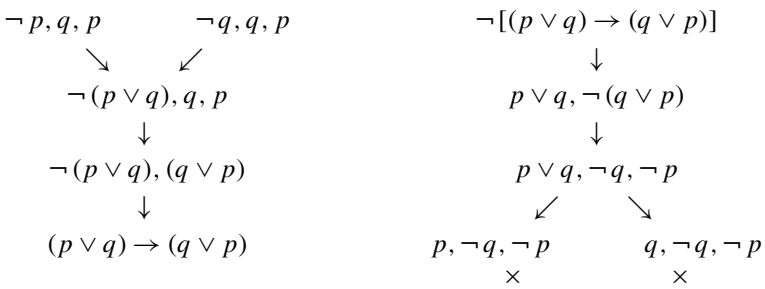
1.	$\vdash \neg p, p, q$	Axiom
2.	$\vdash \neg p, (p \vee q)$	$\alpha\vee$, 1
3.	$\vdash \neg p, p, r$	Axiom
4.	$\vdash \neg p, (p \vee r)$	$\alpha\vee$, 3
5.	$\vdash \neg p, (p \vee q) \wedge (p \vee r)$	$\beta\wedge$, 2, 4
6.	$\vdash \neg q, \neg r, p, q$	Axiom
7.	$\vdash \neg q, \neg r, (p \vee q)$	$\alpha\vee$, 6
8.	$\vdash \neg q, \neg r, p, r$	Axiom
9.	$\vdash \neg q, \neg r, (p \vee r)$	$\alpha\vee$, 8
10.	$\vdash \neg q, \neg r, (p \vee q) \wedge (p \vee r)$	$\beta\wedge$, 7, 9
11.	$\vdash \neg (q \wedge r), (p \vee q) \wedge (p \vee r)$	$\alpha\wedge$, 10
12.	$\vdash \neg (p \vee (q \wedge r)), (p \vee q) \wedge (p \vee r)$	$\beta\vee$, 5, 11
13.	$\vdash p \vee (q \wedge r) \rightarrow (p \vee q) \wedge (p \vee r)$	$\alpha\rightarrow$, 12

■

3.2.1 The Relationship Between $\mathscr{G}$ and Semantic Tableaux

It might seem that we have been rather clever to arrange all the inferences in these proofs so that everything comes out exactly right in the end. In fact, no cleverness was required. Let us rearrange the Gentzen proof into a tree format rather than a linear sequence of sets of formulas. Let the axioms be the leaves of the tree, and let the inference rules define the interior nodes. The root at the bottom will be labeled with the formula that is proved.

The proof from Example 3.4 is displayed in tree form on the left below:



If this looks familiar, it should. The semantic tableau on the right results from turning the derivation in $\mathscr{G}$ upside down and replacing each formula in the labels on the nodes by its complement (Definition 2.57).

A set of formulas labeling a node in a semantic tableau is an *implicit conjunction*, that is, all the formulas in the set must evaluate to true for the set to be true. By taking complements, a set of formulas labeling a node in a derivation in $\mathscr{G}$ is an *implicit disjunction*.

An axiom in $\mathscr{G}$ is valid: Since it contains a complementary pair of literals, as a disjunction it is:

$$\cdots \vee p \vee \cdots \vee \neg p \vee \cdots,$$

which is valid.

Consider a rule applied to obtain an α-formula, for example, $A_1 \vee A_2$; when the rule is written using disjunctions it becomes:

$$\frac{\vdash \bigvee U_1' \vee A_1 \vee A_2}{\vdash \bigvee U_1' \vee (A_1 \vee A_2)},$$

and this is a valid inference in propositional logic that follows immediately from associativity.

Similarly, when a rule is applied to obtain a β-formula, we have:

$$\frac{\vdash \bigvee U_1' \vee B_1 \qquad\qquad \bigvee U_2' \vee B_2}{\vdash \bigvee U_1' \vee \bigvee U_2' \vee (B_1 \wedge B_2)},$$

which follows by the distribution of disjunction over conjunction. This inference simply says that if we can prove both B_1 and B_2 then we can prove $B_1 \wedge B_2$.

The relationship between semantic tableaux and Gentzen systems is formalized in the following theorem.

Theorem 3.6 *Let A be a formula in propositional logic. Then $\vdash A$ in $\mathscr{G}$ if and only if there is a closed semantic tableau for $\neg A$.*

This follows immediately from a more general theorem on sets of formulas.

Theorem 3.7 *Let U be a set of formulas and let $\bar{U}$ be the set of complements of formulas in U. Then $\vdash U$ in $\mathscr{G}$ if and only if there is a closed semantic tableau for $\bar{U}$.*

Proof Let $\mathscr{T}$ be a closed semantic tableau for $\bar{U}$. We prove $\vdash U$ by induction on h, the height of $\mathscr{T}$. The other direction is left as an exercise.

If $h = 0$, then $\mathscr{T}$ consists of a single node labeled by $\bar{U}$. By assumption, $\mathscr{T}$ is closed, so it contains a complementary pair of literals $\{p, \neg p\}$, that is, $\bar{U} = \bar{U}' \cup \{p, \neg p\}$. Obviously, $U = U' \cup \{\neg p, p\}$ is an axiom in $\mathscr{G}$, hence $\vdash U$.

If $h > 0$, then some tableau rule was used on an α- or β-formula at the root of $\mathscr{T}$ on a formula $\bar{\phi} \in \bar{U}$, that is, $\bar{U} = \bar{U}' \cup \{\bar{\phi}\}$. The proof proceeds by cases, where you must be careful to distinguish between applications of the tableau rules and applications of the Gentzen rules of the same name.

Case 1: $\bar{\phi}$ is an α-formula (such as) $\neg(A_1 \vee A_2)$. The tableau rule created a child node labeled by the set of formulas $\bar{U}' \cup \{\neg A_1, \neg A_2\}$. By assumption, the subtree rooted at this node is a closed tableau, so by the inductive hypothesis, $\vdash U' \cup \{A_1, A_2\}$. Using the appropriate rule of inference from $\mathscr{G}$, we obtain $\vdash U' \cup \{A_1 \vee A_2\}$, that is, $\vdash U' \cup \{\phi\}$, which is $\vdash U$.

Case 2: $\bar{\phi}$ is a β-formula (such as) $\neg(B_1 \wedge B_2)$. The tableau rule created two child nodes labeled by the sets of formulas $\bar{U}' \cup \{\neg B_1\}$ and $\bar{U}' \cup \{\neg B_2\}$. By assumption, the subtrees rooted at this node are closed, so by the inductive hypothesis $\vdash U' \cup \{B_1\}$ and $\vdash U' \cup \{B_2\}$. Using the appropriate rule of inference from $\mathscr{G}$, we obtain $\vdash U' \cup \{B_1 \wedge B_2\}$, that is, $\vdash U' \cup \{\phi\}$, which is $\vdash U$. ∎

Theorem 3.8 (Soundness and completeness of $\mathscr{G}$)
$\models A$ *if and only if* $\vdash A$ *in* $\mathscr{G}$.

Proof A is valid iff $\neg A$ is unsatisfiable iff there is a closed semantic tableau for $\neg A$ iff there is a proof of A in $\mathscr{G}$. ∎

The proof is very simple because we did all the hard work in the proof of the soundness and completeness of tableaux.

The Gentzen system $\mathscr{G}$ described in this section is not very useful; other versions (surveyed in Sect. 3.9) are more convenient for proving theorems and are closer to Gentzen's original formulation. We introduced $\mathscr{G}$ as a theoretical stepping stone to Hilbert systems which we now describe.

3.3 Hilbert System $\mathscr{H}$

In Gentzen systems there is one axiom and many rules of inference, while in a Hilbert system there are several axioms but only one rule of inference. In this section, we define the deductive system $\mathscr{H}$ and use it to prove many theorems. Actually, only one theorem (Theorem 3.10) will be proved directly from the axioms and the rule of inference; practical use of the system depends on the use of derived rules, especially the deduction rule.

Notation: Capital letters $A, B, C, \ldots$ represent arbitrary formulas in propositional logic. For example, the notation $\vdash A \to A$ means: for *any* formula A of propositional logic, the formula $A \to A$ can be proved.

Definition 3.9 (Deductive system $\mathscr{H}$) The *axioms* of $\mathscr{H}$ are:

Axiom 1 $\vdash (A \to (B \to A))$,

Axiom 2 $\vdash (A \to (B \to C)) \to ((A \to B) \to (A \to C))$,

Axiom 3 $\vdash (\neg B \to \neg A) \to (A \to B)$.

The *rule of inference* is ***modus ponens*** (*MP* for short):

$$\frac{\vdash A \qquad \vdash A \to B}{\vdash B}.$$

In words: the formula B can be inferred from A and $A \to B$.

The terminology used for $\mathscr{G}$—*premises*, *conclusion*, *theorem*, *proved*— carries over to $\mathscr{H}$, as does the symbol $\vdash$ meaning that a formula is proved. ■

Theorem 3.10 $\vdash A \to A$.

Proof

1.	$\vdash (A \to ((A \to A) \to A)) \to ((A \to (A \to A)) \to (A \to A))$	Axiom 2
2.	$\vdash A \to ((A \to A) \to A)$	Axiom 1
3.	$\vdash (A \to (A \to A)) \to (A \to A)$	MP 1, 2
4.	$\vdash A \to (A \to A)$	Axiom 1
5.	$\vdash A \to A$	MP 3, 4

■

When an axiom is given as the justification, identify which formulas are substituted for the formulas A, B, C in the definition of the axioms above.

*3.3.1 Axiom Schemes and Theorem Schemes **

As we noted above, a capital letter can be replaced by any formula of propositional logic, so, strictly speaking, $\vdash A \to (B \to A)$ is not an axiom, and similarly, $\vdash A \to A$

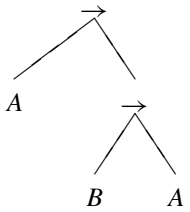
is not a theorem. A more precise terminology would be to say that $\vdash A \rightarrow (B \rightarrow A)$ is an *axiom scheme* that is a shorthand for an infinite number of axioms obtained by replacing the 'variables' A and B with actual formulas, for example:

$$\overbrace{((p \vee \neg q) \leftrightarrow r)}^{A} \rightarrow (\overbrace{\neg (q \wedge \neg r)}^{B} \rightarrow \overbrace{((p \vee \neg q) \leftrightarrow r)}^{A}).$$

Similarly, $\vdash A \rightarrow A$ is a *theorem scheme* that is a shorthand for an infinite number of theorems that can be proved in $\mathscr{H}$, including, for example:

$$\vdash ((p \vee \neg q) \leftrightarrow r) \rightarrow ((p \vee \neg q) \leftrightarrow r).$$

We will not retain this precision in our presentation because it will always clear if a given formula is an instance of a particular axiom scheme or theorem scheme. For example, a formula ϕ is an instance of Axiom 1 if it is of the form:



where there are subtrees for the formulas represented by A and B. There is a simple and efficient algorithm that checks if ϕ is of this form and if the two subtrees A are identical.

3.3.2 The Deduction Rule

The proof of Theorem 3.10 is rather complicated for such a trivial formula. In order to formalize the powerful methods of inference used in mathematics, we introduce new rules of inference called *derived rules*. The most important derived rule is the *deduction rule*. Suppose that you want to prove $A \rightarrow B$. *Assume* that A has already been proved and use it in the proof of B. This is not a proof of B unless A is an axiom or theorem that has been previously proved, in which case it can be used directly in the proof. However, we claim that the proof can be mechanically transformed into a proof of $A \rightarrow B$.

Example 3.11 The deduction rule is used frequently in mathematics. Suppose that you want to prove that *the sum of any two odd integer numbers is even*, expressed formally as:

$$odd(x) \wedge odd(y) \rightarrow even(x + y),$$

for every x and y. To prove this formula, let us *assume* the formula $odd(x) \wedge odd(y)$ as if it were an additional axiom. We have available all the theorems we have already

deduced about odd numbers, in particular, the theorem that any odd number can be expressed as $2k + 1$. Computing:

$$x + y = 2k_1 + 1 + 2k_2 + 1 = 2(k_1 + k_2 + 1),$$

we obtain that $x + y$ is a multiple of 2, that is, $even(x + y)$. The theorem now follows from the deduction rule which *discharges* the assumption. ∎

To express the deduction rule, we extend the definition of *proof*.

Definition 3.12 Let U be a set of formulas and A a formula. The notation $U \vdash A$ means that the formulas in U are *assumptions* in the proof of A. A *proof* is a sequence of lines $U_i \vdash \phi_i$, such that for each i, $U_i \subseteq U$, and ϕ_i is an axiom, a previously proved theorem, a member of U_i or can be derived by *MP* from previous lines $U_{i'} \vdash \phi_{i'}$, $U_{i''} \vdash \phi_{i''}$, where $i', i'' < i$. ∎

Rule 3.13 (Deduction rule)

$$\frac{U \cup \{A\} \vdash B}{U \vdash A \to B}.$$

We must show that this derived rule is *sound*, that is, that the use of the derived rule does not increase the set of provable theorems in $\mathscr{H}$. This is done by showing how to transform any proof using the rule into one that does not use the rule. Therefore, in principle, any proof that uses the derived rule could be transformed to one that uses only the three axioms and *MP*.

Theorem 3.14 (Deduction theorem) *The deduction rule is a sound derived rule.*

Proof We show by induction on the length n of the proof of $U \cup \{A\} \vdash B$ how to obtain a proof of $U \vdash A \to B$ that does not use the deduction rule.

For $n = 1$, B is proved in one step, so B must be either an element of $U \cup \{A\}$ or an axiom of $\mathscr{H}$ or a previously proved theorem:

- If B is A, then $\vdash A \to A$ by Theorem 3.10, so certainly $U \vdash A \to A$.
- Otherwise (B is an axiom or a previously proved theorem), here is a proof of $U \vdash A \to B$ that does not use the deduction rule or the assumption A:

1.	$U \vdash B$	Axiom or theorem
2.	$U \vdash B \to (A \to B)$	Axiom 1
3.	$U \vdash A \to B$	MP 1, 2

If $n > 1$, the last step in the proof of $U \cup \{A\} \vdash B$ is either a one-step inference of B or an inference of B using *MP*. In the first case, the result holds by the proof for $n = 1$. Otherwise, *MP* was used, so there is a formula C and lines $i, j < n$ in the proof such that line i in the proof is $U \cup \{A\} \vdash C$ and line j is $U \cup \{A\} \vdash C \to B$. By the inductive hypothesis, $U \vdash A \to C$ and $U \vdash A \to (C \to B)$. A proof of $U \vdash A \to B$ is given by:

1.	$U \vdash A \to C$	Inductive hypothesis
2.	$U \vdash A \to (C \to B)$	Inductive hypothesis
3.	$U \vdash (A \to (C \to B)) \to ((A \to C) \to (A \to B))$	Axiom 2
4.	$U \vdash (A \to C) \to (A \to B)$	MP 2, 3
5.	$U \vdash A \to B$	MP 1, 4

■

3.4 Derived Rules in $\mathscr{H}$

The general form of a derived rule will be one of:

$$\frac{U \vdash \phi_1}{U \vdash \phi}, \qquad \frac{U \vdash \phi_1 \qquad U \vdash \phi_2}{U \vdash \phi}.$$

The first form is justified by proving the formula $U \vdash \phi_1 \to \phi$ and the second by $U \vdash \phi_1 \to (\phi_2 \to \phi)$; the formula $U \vdash \phi$ that is the conclusion of the rule follows immediately by one or two applications of *MP*. For example, from Axiom 3 we immediately have the following rule:

Rule 3.15 (Contrapositive rule)

$$\frac{U \vdash \neg B \to \neg A}{U \vdash A \to B}.$$

The contrapositive is used extensively in mathematics. We showed the completeness of the method of semantic tableaux by proving: *If a tableau is open, the formula is satisfiable*, which is the contrapositive of the theorem that we wanted to prove: *If a formula is unsatisfiable (not satisfiable), the tableau is closed (not open).*

Theorem 3.16 $\vdash (A \to B) \to [(B \to C) \to (A \to C)]$.

Proof

1.	$\{A \to B, B \to C, A\} \vdash A$	Assumption
2.	$\{A \to B, B \to C, A\} \vdash A \to B$	Assumption
3.	$\{A \to B, B \to C, A\} \vdash B$	MP 1, 2
4.	$\{A \to B, B \to C, A\} \vdash B \to C$	Assumption
5.	$\{A \to B, B \to C, A\} \vdash C$	MP 3, 4
6.	$\{A \to B, B \to C\} \vdash A \to C$	Deduction 5
7.	$\{A \to B\} \vdash [(B \to C) \to (A \to C)]$	Deduction 6
8.	$\vdash (A \to B) \to [(B \to C) \to (A \to C)]$	Deduction 7

■

Rule 3.17 (Transitivity rule)

$$\frac{U \vdash A \to B \qquad U \vdash B \to C}{U \vdash A \to C}.$$

The transitivity rule justifies the step-by-step development of a mathematical theorem $\vdash A \to C$ through a series of *lemmas*. The antecedent A of the theorem is used to prove a lemma $\vdash A \to B_1$ whose consequent is used to prove the next lemma $\vdash B_1 \to B_2$ and so on until the consequent of the theorem appears as $\vdash B_n \to C$. Repeated use of the transitivity rule enables us to deduce $\vdash A \to C$.

Theorem 3.18 $\vdash [A \to (B \to C)] \to [B \to (A \to C)]$.

Proof

1.	$\{A \to (B \to C), B, A\} \vdash A$	Assumption
2.	$\{A \to (B \to C), B, A\} \vdash A \to (B \to C)$	Assumption
3.	$\{A \to (B \to C), B, A\} \vdash B \to C$	MP 1, 2
4.	$\{A \to (B \to C), B, A\} \vdash B$	Assumption
5.	$\{A \to (B \to C), B, A\} \vdash C$	MP 3, 4
6.	$\{A \to (B \to C), B\} \vdash A \to C$	Deduction 5
7.	$\{A \to (B \to C)\} \vdash B \to (A \to C)$	Deduction 6
8.	$\vdash [A \to (B \to C)] \to [B \to (A \to C)]$	Deduction 7

■

Rule 3.19 (Exchange of antecedent rule)

$$\frac{U \vdash A \to (B \to C)}{U \vdash B \to (A \to C)}.$$

Exchanging the antecedent simply means that it doesn't matter in which order we use the lemmas necessary in a proof.

Theorem 3.20 $\vdash \neg A \to (A \to B)$.

Proof

1.	$\{\neg A\} \vdash \neg A \to (\neg B \to \neg A)$	Axiom 1
2.	$\{\neg A\} \vdash \neg A$	Assumption
3.	$\{\neg A\} \vdash \neg B \to \neg A$	MP 1, 2
4.	$\{\neg A\} \vdash (\neg B \to \neg A) \to (A \to B)$	Axiom 3
5.	$\{\neg A\} \vdash A \to B$	MP 3, 4
6.	$\vdash \neg A \to (A \to B)$	Deduction 5

■

Theorem 3.21 $\vdash A \to (\neg A \to B)$.

Proof

1.	$\vdash \neg A \to (A \to B)$	Theorem 3.20
2.	$\vdash A \to (\neg A \to B)$	Exchange 1

■

These two theorems are of major theoretical importance. They say that if you can prove some formula A and its negation $\neg A$, then you can prove *any* formula B! If you can prove any formula then there are no unprovable formulas so the concept of proof becomes meaningless.

Theorem 3.22 $\vdash \neg\neg A \to A$.

Proof

1.	$\{\neg\neg A\} \vdash \neg\neg A \to (\neg\neg\neg\neg A \to \neg\neg A)$	Axiom 1
2.	$\{\neg\neg A\} \vdash \neg\neg A$	Assumption
3.	$\{\neg\neg A\} \vdash \neg\neg\neg\neg A \to \neg\neg A$	MP 1, 2
4.	$\{\neg\neg A\} \vdash \neg A \to \neg\neg\neg A$	Contrapositive 3
5.	$\{\neg\neg A\} \vdash \neg\neg A \to A$	Contrapositive 4
6.	$\{\neg\neg A\} \vdash A$	MP 2, 5
7.	$\vdash \neg\neg A \to A$	Deduction 6

■

Theorem 3.23 $\vdash A \to \neg\neg A$.

Proof

1.	$\vdash \neg\neg\neg A \to \neg A$	Theorem 3.22
2.	$\vdash A \to \neg\neg A$	Contrapositive 1

■

Rule 3.24 (Double negation rule)

$$\frac{U \vdash \neg\neg A}{U \vdash A}, \qquad \frac{U \vdash A}{U \vdash \neg\neg A}.$$

Double negation is a very intuitive rule. We expect that 'it is raining' and 'it is not true that it is not raining' will have the same truth value, and that the second formula can be simplified to the first. Nevertheless, some logicians reject the rule because it is not constructive. Suppose that we can prove for some number n, 'it is not true that n is prime' which is the same as 'it is not true that n is not composite'. This double negation could be reduced by the rule to 'n is composite', but we have not actually demonstrated any factors of n.

Theorem 3.25 $\vdash (A \to B) \to (\neg B \to \neg A)$.

Proof

1.	$\{A \to B\} \vdash A \to B$	Assumption
2.	$\{A \to B\} \vdash \neg\neg A \to A$	Theorem 3.22
3.	$\{A \to B\} \vdash \neg\neg A \to B$	Transitivity 2, 1
4.	$\{A \to B\} \vdash B \to \neg\neg B$	Theorem 3.23
5.	$\{A \to B\} \vdash \neg\neg A \to \neg\neg B$	Transitivity 3, 4
6.	$\{A \to B\} \vdash \neg B \to \neg A$	Contrapositive 5
7.	$\vdash (A \to B) \to (\neg B \to \neg A)$	Deduction 6

■

Rule 3.26 (Contrapositive rule)

$$\frac{U \vdash A \to B}{U \vdash \neg B \to \neg A}.$$

This is the other direction of the contrapositive rule shown earlier.

Recall from Sect. 2.3.3 the definition of the logical constants *true* as an abbreviation for $p \vee \neg p$ and *false* as an abbreviation for $p \wedge \neg p$. These can be expressed using implication and negation alone as $p \to p$ and $\neg(p \to p)$.

Theorem 3.27

$$\vdash \mathit{true},$$
$$\vdash \neg \mathit{false}.$$

Proof $\vdash \mathit{true}$ is an instance of Theorem 3.10. $\vdash \neg \mathit{false}$, which is $\vdash \neg\neg(p \to p)$, follows by double negation. ■

Theorem 3.28 $\vdash (\neg A \to \mathit{false}) \to A$.

Proof

1.	$\{\neg A \to \mathit{false}\} \vdash \neg A \to \mathit{false}$	Assumption
2.	$\{\neg A \to \mathit{false}\} \vdash \neg \mathit{false} \to \neg\neg A$	Contrapositive
3.	$\{\neg A \to \mathit{false}\} \vdash \neg \mathit{false}$	Theorem 3.27
4.	$\{\neg A \to \mathit{false}\} \vdash \neg\neg A$	MP 2, 3
5.	$\{\neg A \to \mathit{false}\} \vdash A$	Double negation 4
6.	$\vdash (\neg A \to \mathit{false}) \to A$	Deduction 5

■

Rule 3.29 (Reductio ad absurdum)

$$\frac{U \vdash \neg A \to \mathit{false}}{U \vdash A}.$$

Reductio ad absurdum is a very useful rule in mathematics: Assume the negation of what you wish to prove and show that it leads to a contradiction. This rule is also controversial because proving that $\neg A$ leads to a contradiction provides no reason that directly justifies A.

Here is an example of the use of this rule:

Theorem 3.30 $\vdash (A \to \neg A) \to \neg A$.

Proof

1.	$\{A \to \neg A, \neg\neg A\} \vdash \neg\neg A$	Assumption
2.	$\{A \to \neg A, \neg\neg A\} \vdash A$	Double negation 1
3.	$\{A \to \neg A, \neg\neg A\} \vdash A \to \neg A$	Assumption
4.	$\{A \to \neg A, \neg\neg A\} \vdash \neg A$	MP 2, 3
5.	$\{A \to \neg A, \neg\neg A\} \vdash A \to (\neg A \to \mathit{false})$	Theorem 3.21
6.	$\{A \to \neg A, \neg\neg A\} \vdash \neg A \to \mathit{false}$	MP 2, 5
7.	$\{A \to \neg A, \neg\neg A\} \vdash \mathit{false}$	MP 4, 6
8.	$\{A \to \neg A\} \vdash \neg\neg A \to \mathit{false}$	Deduction 7
9.	$\{A \to \neg A\} \vdash \neg A$	Reductio ad absurdum 8
10.	$\vdash (A \to \neg A) \to \neg A$	Deduction 9

■

We leave the proof of the following theorem as an exercise.

Theorem 3.31 $\vdash (\neg A \to A) \to A$.

These two theorems may seem strange, but they can be understood on the semantic level. For the implication of Theorem 3.31 to be false, the antecedent $\neg A \to A$ must be true and the consequent A false. But if A is false, then so is $\neg A \to A \equiv A \vee A$, so the formula is true.

3.5 Theorems for Other Operators

So far we have worked with only negation and implication as operators. These two operators are adequate for defining all others (Sect. 2.4), so we can use these definitions to prove theorems using other operators. Recall that $A \wedge B$ is defined as $\neg(A \to \neg B)$, and $A \vee B$ is defined as $\neg A \to B$.

Theorem 3.32 $\vdash A \to (B \to (A \wedge B))$.

Proof

1.	$\{A, B\} \vdash (A \to \neg B) \to (A \to \neg B)$	Theorem 3.10
2.	$\{A, B\} \vdash A \to ((A \to \neg B) \to \neg B)$	Exchange 1
3.	$\{A, B\} \vdash A$	Assumption
4.	$\{A, B\} \vdash (A \to \neg B) \to \neg B$	MP 2, 3
5.	$\{A, B\} \vdash \neg\neg B \to \neg(A \to \neg B)$	Contrapositive 4
6.	$\{A, B\} \vdash B$	Assumption
7.	$\{A, B\} \vdash \neg\neg B$	Double negation 6
8.	$\{A, B\} \vdash \neg(A \to \neg B)$	MP 5, 7
9.	$\{A\} \vdash B \to \neg(A \to \neg B)$	Deduction 8
10.	$\vdash A \to (B \to \neg(A \to \neg B))$	Deduction 9
11.	$\vdash A \to (B \to (A \wedge B))$	Definition of $\wedge$

■

Theorem 3.33 (Commutativity) $\vdash A \vee B \leftrightarrow B \vee A$.

Proof

1.	$\{\neg A \to B, \neg B\} \vdash \neg A \to B$	Assumption
2.	$\{\neg A \to B, \neg B\} \vdash \neg B \to \neg\neg A$	Contrapositive 1
3.	$\{\neg A \to B, \neg B\} \vdash \neg B$	Assumption
4.	$\{\neg A \to B, \neg B\} \vdash \neg\neg A$	MP 2, 3
5.	$\{\neg A \to B, \neg B\} \vdash A$	Double negation 4
6.	$\{\neg A \to B\} \vdash \neg B \to A$	Deduction 5
7.	$\vdash (\neg A \to B) \to (\neg B \to A)$	Deduction 6
8.	$\vdash A \vee B \to B \vee A$	Def. of $\vee$

The other direction is similar. ■

The proofs of the following theorems are left as exercises.

Theorem 3.34 (Weakening)

$\vdash A \to A \vee B$,
$\vdash B \to A \vee B$,
$\vdash (A \to B) \to ((C \vee A) \to (C \vee B))$.

Theorem 3.35 (Associativity)

$\vdash A \vee (B \vee C) \leftrightarrow (A \vee B) \vee C$.

Theorem 3.36 (Distributivity)

$\vdash A \vee (B \wedge C) \leftrightarrow (A \vee B) \wedge (A \vee C)$,
$\vdash A \wedge (B \vee C) \leftrightarrow (A \wedge B) \vee (A \wedge C)$.

3.6 Soundness and Completeness of $\mathscr{H}$

We now prove the soundness and completeness of the Hilbert system $\mathscr{H}$. As usual, soundness is easy to prove. Proving completeness will not be too difficult because we already know that the Gentzen system $\mathscr{G}$ is complete so it is sufficient to show how to transform any proof in $\mathscr{G}$ into a proof in $\mathscr{H}$.

Theorem 3.37 *The Hilbert system $\mathscr{H}$ is sound*: *If* $\vdash A$ *then* $\models A$.

Proof The proof is by structural induction. First we show that the axioms are valid, and then we show that *MP* preserves validity. Here are closed semantic tableaux for the *negations* of Axioms 1 and 3:

$$\begin{array}{c} \neg[A \to (B \to A)] \\ \downarrow \\ A, \neg(B \to A) \\ \downarrow \\ A, B, \neg A \\ \times \end{array} \qquad \begin{array}{c} \neg[(\neg B \to \neg A) \to (A \to B)] \\ \downarrow \\ \neg B \to \neg A, \neg(A \to B) \\ \downarrow \\ \neg B \to \neg A, A, \neg B \\ \swarrow \quad \searrow \\ \neg\neg B, A, \neg B \qquad \neg A, A, \neg B \\ \downarrow \qquad\qquad \times \\ B, A, \neg B \\ \times \end{array}$$

The construction of a tableau for the negation of Axiom 2 is left as an exercise.

Suppose that *MP* were not sound. There would be a set of formulas $\{A, A \to B, B\}$ such that A and $A \to B$ are valid, but B is not valid. Since B is not valid, there is an interpretation $\mathscr{I}$ such that $v_{\mathscr{I}}(B) = F$. Since A and $A \to B$ are valid, for *any* interpretation, in particular for $\mathscr{I}$, $v_{\mathscr{I}}(A) = v_{\mathscr{I}}(A \to B) = T$. By definition of $v_{\mathscr{I}}$ for implication, $v_{\mathscr{I}}(B) = T$, contradicting $v_{\mathscr{I}}(B) = F$. ■

There is no circularity in the final sentence of the proof: We are not using the syntactical proof rule *MP*, but, rather, the semantic definition of truth value in the presence of the implication operator.

Theorem 3.38 *The Hilbert system $\mathscr{H}$ is complete*: *If* $\models A$ *then* $\vdash A$.

By the completeness of the Gentzen system $\mathscr{G}$ (Theorem 3.8), if $\models A$, then $\vdash A$ in $\mathscr{G}$. The proof of the theorem showed how to construct the proof of A by first constructing a semantic tableau for $\neg A$; the tableau is guaranteed to close since A is valid. The completeness of $\mathscr{H}$ is proved by showing how to transform a proof in $\mathscr{G}$ into a proof in $\mathscr{H}$. Note that all three steps can be carried out algorithmically: Given an arbitrary valid formula in propositional logic, a computer can generate its proof.

We need a more general result because a proof in $\mathscr{G}$ is a sequence of *sets* of formulas, while a proof in $\mathscr{H}$ is a sequence of formulas.

Theorem 3.39 *If* $\vdash U$ *in* $\mathscr{G}$, *then* $\vdash \bigvee U$ *in* $\mathscr{H}$.

The difficulty arises from the clash of the data structures used: U is a set while $\bigvee U$ is a single formula. To see why this is a problem, consider the base case of the induction. The set $\{\neg p, p\}$ is an axiom in $\mathscr{G}$ and we immediately have $\vdash \neg p \vee p$ in $\mathscr{H}$ since this is simply $\vdash p \to p$. But if the axiom in $\mathscr{G}$ is $\{q, \neg p, r, p, s\}$, we can't immediately conclude that $\vdash q \vee \neg p \vee r \vee p \vee s$ in $\mathscr{H}$.

Lemma 3.40 *If* $U' \subseteq U$ *and* $\vdash \bigvee U'$ *in* $\mathscr{H}$ *then* $\vdash \bigvee U$ *in* $\mathscr{H}$.

Proof The proof is by induction using weakening, commutativity and associativity of disjunction (Theorems 3.34–3.35). We give the outline here and leave it as an exercise to fill in the details.

Suppose we have a proof of $\bigvee U'$. By repeated application of Theorem 3.34, we can transform this into a proof of $\bigvee U''$, where U'' is a permutation of the elements of U. By repeated applications of commutativity and associativity, we can move the elements of U'' to their proper places. ∎

Example 3.41 Let $U' = \{A, C\} \subset \{A, B, C\} = U$ and suppose we have a proof of $\vdash \bigvee U' = A \vee C$. This can be transformed into a proof of $\vdash \bigvee U = A \vee (B \vee C)$ as follows, where Theorems 3.34–3.35 are used as derived rules:

1.	$\vdash A \vee C$	Assumption
2.	$\vdash (A \vee C) \vee B$	Weakening, 1
3.	$\vdash A \vee (C \vee B)$	Associativity, 2
4.	$\vdash (C \vee B) \to (B \vee C)$	Commutativity
5.	$\vdash A \vee (C \vee B) \to A \vee (B \vee C)$	Weakening, 4
6.	$\vdash A \vee (B \vee C)$	MP 3, 5

∎

Proof of Theorem 3.39 The proof is by induction on the structure of the proof in $\mathscr{G}$. If U is an axiom, it contains a pair of complementary literals and $\vdash \neg p \vee p$ can be proved in $\mathscr{H}$. By Lemma 3.40, this can be transformed into a proof of $\bigvee U$.

Otherwise, the last step in the proof of U in $\mathscr{G}$ is the application of a rule to an α- or β-formula. As usual, we will use disjunction and conjunction as representatives of α- and β-formulas.

Case 1: A rule in $\mathscr{G}$ was applied to obtain an α-formula $\vdash U_1 \cup \{A_1 \vee A_2\}$ from $\vdash U_1 \cup \{A_1, A_2\}$. By the inductive hypothesis, $\vdash ((\bigvee U_1) \vee A_1) \vee A_2$ in $\mathscr{H}$ from which we infer $\vdash \bigvee U_1 \vee (A_1 \vee A_2)$ by associativity.

Case 2: A rule in $\mathscr{G}$ was applied to obtain a β-formula $\vdash U_1 \cup U_2 \cup \{A_1 \wedge A_2\}$ from $\vdash U_1 \cup \{A_1\}$ and $\vdash U_2 \cup \{A_2\}$. By the inductive hypothesis, $\vdash (\bigvee U_1) \vee A_1$ and $\vdash (\bigvee U_2) \vee A_2$ in $\mathscr{H}$. We leave it to the reader to justify each step of the following deduction of $\vdash \bigvee U_1 \vee \bigvee U_2 \vee (A_1 \wedge A_2)$:

1. $\vdash \bigvee U_1 \vee A_1$
2. $\vdash \neg \bigvee U_1 \rightarrow A_1$
3. $\vdash A_1 \rightarrow (A_2 \rightarrow (A_1 \wedge A_2))$
4. $\vdash \neg \bigvee U_1 \rightarrow (A_2 \rightarrow (A_1 \wedge A_2))$
5. $\vdash A_2 \rightarrow (\neg \bigvee U_1 \rightarrow (A_1 \wedge A_2))$
6. $\vdash \bigvee U_2 \vee A_2$
7. $\vdash \neg \bigvee U_2 \rightarrow A_2$
8. $\vdash \neg \bigvee U_2 \rightarrow (\neg \bigvee U_1 \rightarrow (A_1 \wedge A_2))$
9. $\vdash \bigvee U_1 \vee \bigvee U_2 \vee (A_1 \wedge A_2)$

■

Proof of Theorem 3.38 If $\models A$ then $\vdash A$ in $\mathscr{G}$ by Theorem 3.8. By the remark at the end of Definition 3.2, $\vdash A$ is an abbreviation for $\vdash \{A\}$. By Theorem 3.39, $\vdash \bigvee\{A\}$ in $\mathscr{H}$. Since A is a single formula, $\vdash A$ in $\mathscr{H}$. ■

3.7 Consistency

What would mathematics be like if both $1 + 1 = 2$ and $\neg(1 + 1 = 2) \equiv 1 + 1 \neq 2$ could be proven? An inconsistent deductive system is useless, because *all* formulas are provable and the concept of proof becomes meaningless.

Definition 3.42 A set of formulas U is *inconsistent* iff for some formula A, both $U \vdash A$ and $U \vdash \neg A$. U is *consistent* iff it is not inconsistent. A deductive system is *inconsistent* iff it contains an inconsistent set of formulas. ■

Theorem 3.43 *U is inconsistent iff for all A, $U \vdash A$.*

Proof Let A be an arbitrary formula. If U is inconsistent, for some formula B, $U \vdash B$ and $U \vdash \neg B$. By Theorem 3.21, $\vdash B \rightarrow (\neg B \rightarrow A)$. Using *MP* twice, $U \vdash A$. The converse is trivial. ■

Corollary 3.44 *U is consistent if and only if for some A, $U \nvdash A$.*

If a deductive system is sound, then $\vdash A$ implies $\models A$, and, conversely, $\not\models A$ implies $\nvdash A$. Therefore, if there is even a single falsifiable formula A in a sound system, the system must be consistent! Since $\not\models$ *false* (where *false* is an abbreviation for $\neg(p \rightarrow p)$), by the soundness of $\mathscr{H}$, $\nvdash$ *false*. By Corollary 3.44, $\mathscr{H}$ is consistent.

The following theorem is another way of characterizing inconsistency.

Theorem 3.45 *$U \vdash A$ if and only if $U \cup \{\neg A\}$ is inconsistent.*

Proof If $U \vdash A$, obviously $U \cup \{\neg A\} \vdash A$, since the extra assumption will not be used in the proof. $U \cup \{\neg A\} \vdash \neg A$ because $\neg A$ is an assumption. By Definition 3.42, $U \cup \{\neg A\}$ is inconsistent.

Conversely, if $U \cup \{\neg A\}$ is inconsistent, then $U \cup \{\neg A\} \vdash A$ by Theorem 3.43. By the deduction theorem, $U \vdash \neg A \rightarrow A$, and $U \vdash A$ follows by *MP* from $\vdash (\neg A \rightarrow A) \rightarrow A$ (Theorem 3.31). ■

3.8 Strong Completeness and Compactness *

The construction of a semantic tableau can be generalized to an infinite set of formulas $S = \{A_1, A_2, \ldots\}$. The label of the root is $\{A_1\}$. Whenever a rule is applied to a leaf of depth n, A_{n+1} will be added to the label(s) of its child(ren) in addition to the α_i or β_i.

Theorem 3.46 *A set of formulas $S = \{A_1, A_2, \ldots\}$ is unsatisfiable if and only if a semantic tableau for S closes.*

Proof Here is an outline of the proof that is given in detail in Smullyan (1968, Chap. III).

If the tableau closes, there is only a finite subset $S_0 \subset S$ of formulas on each closed branch, and S_0 is unsatisfiable. By a generalization of Theorem 2.46 to an infinite set of formulas, it follows that $S = S_0 \cup (S - S_0)$ is unsatisfiable.

Conversely, if the tableau is open, it can be shown that there must be an infinite branch containing all formulas in S, and the union of formulas in the labels of nodes on the branch forms a Hintikka set, from which a satisfying interpretation can be found. ■

The completeness of propositional logic now generalizes to:

Theorem 3.47 (Strong completeness) *Let U be a finite or countably infinite set of formulas and let A be a formula. If $U \models A$ then $U \vdash A$.*

The same construction proves the following important theorem.

Theorem 3.48 (Compactness) *Let S be a countably infinite set of formulas, and suppose that every finite subset of S is satisfiable. Then S is satisfiable.*

Proof Suppose that S were unsatisfiable. Then a semantic tableau for S must close. There are only a finite number of formulas labeling nodes on each closed branch. Each such set of formulas is a finite unsatisfiable subset of S, contracting the assumption that all finite subsets are satisfiable. ■

3.9 Variant Forms of the Deductive Systems *

$\mathscr{G}$ and $\mathscr{H}$, the deductive systems that we presented in detail, are two of many possible deductive systems for propositional logic. Different systems are obtained by changing the operators, the axioms or the representations of proofs. In propositional logic, all these systems are equivalent in the sense that they are sound and complete. In this section, we survey some of these variants.

3.9.1 Hilbert Systems

Hilbert systems almost invariably have *MP* as the only rule. They differ in the choice of primitive operators and axioms. For example, $\mathscr{H}'$ is an Hilbert system where Axiom 3 is replaced by:

$$\text{Axiom } 3' \quad \vdash (\neg B \to \neg A) \to ((\neg B \to A) \to B).$$

Theorem 3.49 *$\mathscr{H}$ and $\mathscr{H}'$ are equivalent in the sense that a proof in one system can be transformed into a proof in the other.*

Proof We prove Axiom $3'$ in $\mathscr{H}$. It follows that any proof in $\mathscr{H}'$ can be transformed into a proof in $\mathscr{H}$, by starting with this proof of the new axiom and using it as a previously proved theorem.

1.	$\{\neg B \to \neg A, \neg B \to A, \neg B\} \vdash \neg B$	Assumption
2.	$\{\neg B \to \neg A, \neg B \to A, \neg B\} \vdash \neg B \to A$	Assumption
3.	$\{\neg B \to \neg A, \neg B \to A, \neg B\} \vdash A$	MP 1, 2
4.	$\{\neg B \to \neg A, \neg B \to A, \neg B\} \vdash \neg B \to \neg A$	Assumption
5.	$\{\neg B \to \neg A, \neg B \to A, \neg B\} \vdash A \to B$	Contrapositive 4
6.	$\{\neg B \to \neg A, \neg B \to A, \neg B\} \vdash B$	MP 3, 5
7.	$\{\neg B \to \neg A, \neg B \to A\} \vdash \neg B \to B$	Deduction 7
8.	$\{\neg B \to \neg A, \neg B \to A\} \vdash (\neg B \to B) \to B$	Theorem 3.31
9.	$\{\neg B \to \neg A, \neg B \to A\} \vdash B$	MP 8, 9
10.	$\{\neg B \to \neg A\} \vdash (\neg B \to A) \to B$	Deduction 9
11.	$\vdash (\neg B \to \neg A) \to ((\neg B \to A) \to B)$	Deduction 10

The use of the deduction theorem is legal because its proof in $\mathscr{H}$ does not use Axiom 3, so the identical proof can be done in $\mathscr{H}'$.

We leave it as an exercise to prove Axiom 3 in $\mathscr{H}'$. ■

Either conjunction or disjunction may replace implication as the binary operator in the formulation of a Hilbert system. Implication can then be defined by $\neg(A \wedge \neg B)$ or $\neg A \vee B$, respectively, and *MP* is still the only inference rule. For disjunction, a set of axioms is:

Axiom 1 $\vdash A \vee A \rightarrow A$,
Axiom 2 $\vdash A \rightarrow A \vee B$,
Axiom 3 $\vdash A \vee B \rightarrow B \vee A$,
Axiom 4 $\vdash (B \rightarrow C) \rightarrow (A \vee B \rightarrow A \vee C)$.

The steps needed to show the equivalence of this system with $\mathscr{H}$ are given in Mendelson (2009, Exercise 1.54).

Finally, Meredith's axiom:

$$\vdash (\{[(A \rightarrow B) \rightarrow (\neg C \rightarrow \neg D)] \rightarrow C\} \rightarrow E) \rightarrow [(E \rightarrow A) \rightarrow (D \rightarrow A)],$$

together with *MP* as the rule of inference is a complete deductive system for propositional logic. Adventurous readers are invited to prove the axioms of $\mathscr{H}$ from Meredith's axiom following the 37-step plan given in Monk (1976, Exercise 8.50).

3.9.2 Gentzen Systems

$\mathscr{G}$ was constructed in order to simplify the theoretical treatment by using a notation that is identical to that of semantic tableaux. We now present a deductive system similar to the one that Gentzen originally proposed; this system is taken from Smullyan (1968, Chap. XI).

Definition 3.50 If U and V are (possibly empty) sets of formulas, then $U \Rightarrow V$ is a *sequent*. ■

Intuitively, a sequent represents 'provable from' in the sense that the formulas in U are assumptions for the set of formulas V that are to be proved. The symbol $\Rightarrow$ is similar to the symbol $\vdash$ in Hilbert systems, except that $\Rightarrow$ is part of the object language of the deductive system being formalized, while $\vdash$ is a metalanguage notation used to reason about deductive systems.

Definition 3.51 Axioms in the Gentzen sequent system $\mathscr{S}$ are sequents of the form:

$$U \cup \{A\} \Rightarrow V \cup \{A\}.$$

The rules of inference are shown in Fig. 3.2. ■

The semantics of the sequent system $\mathscr{S}$ are defined as follows:

Definition 3.52 Let $S = U \Rightarrow V$ be a sequent where $U = \{U_1, \ldots, U_n\}$ and $V = \{V_1, \ldots, V_m\}$, and let $\mathscr{I}$ be an interpretation for $U \cup V$. Then $v_{\mathscr{I}}(S) = T$ if and only if $v_{\mathscr{I}}(U_1) = \cdots = v_{\mathscr{I}}(U_n) = T$ implies that for some i, $v_{\mathscr{I}}(V_i) = T$. ■

This definition relates sequents to formulas: Given an interpretation $\mathscr{I}$ for $U \cup V$, $v_{\mathscr{I}}(U \Rightarrow V) = T$ if and only if $v_{\mathscr{I}}(\bigwedge U \rightarrow \bigvee V) = T$.

op	Introduction into consequent	Introduction into antecedent
$\wedge$	$\dfrac{U \Rightarrow V \cup \{A\} \qquad U \Rightarrow V \cup \{B\}}{U \Rightarrow V \cup \{A \wedge B\}}$	$\dfrac{U \cup \{A, B\} \Rightarrow V}{U \cup \{A \wedge B\} \Rightarrow V}$
$\vee$	$\dfrac{U \Rightarrow V \cup \{A, B\}}{U \Rightarrow V \cup \{A \vee B\}}$	$\dfrac{U \cup \{A\} \Rightarrow V \qquad U \cup \{B\} \Rightarrow V}{U \cup \{A \vee B\} \Rightarrow V}$
$\rightarrow$	$\dfrac{U \cup \{A\} \Rightarrow V \cup \{B\}}{U \Rightarrow V \cup \{A \rightarrow B\}}$	$\dfrac{U \Rightarrow V \cup \{A\} \qquad U \cup \{B\} \Rightarrow V}{U \cup \{A \rightarrow B\} \Rightarrow V}$
$\neg$	$\dfrac{U \cup \{A\} \Rightarrow V}{U \Rightarrow V \cup \{\neg A\}}$	$\dfrac{U \Rightarrow V \cup \{A\}}{U \cup \{\neg A\} \Rightarrow V}$

Fig. 3.2 Rules of inference for sequents

3.9.3 Natural Deduction

The advantage of working with sequents is that the deduction theorem is a rule of inference: introduction into the consequent of $\rightarrow$. The convenience of Gentzen systems is apparent when proofs are presented in a format called *natural deduction* that emphasizes the role of assumptions.

Look at the proof of Theorem 3.30, for example. The assumptions are dragged along throughout the entire deduction, even though each is used only twice, once as an assumption and once in the deduction rule. The way we reason in mathematics is to set out the assumptions once when they are first needed and then to *discharge* them by using the deduction rule. A natural deduction proof of Theorem 3.30 is shown in Fig. 3.3.

The boxes indicate the scope of assumptions. Just as in programming where local variables in procedures can only be used within the procedure and disappear when the procedure is left, an assumption can only be used within the scope of its box, and once it is discharged by using it in a deduction, it is no longer available.

3.9.4 Subformula Property

Definition 3.53 A deductive system has the *subformula property* iff any formula appearing in a proof of A is either a subformula of A or the negation of a subformula of A. ■

The systems $\mathscr{G}$ and $\mathscr{S}$ have the subformula property while $\mathscr{H}$ does not. For example, in the proof of the theorem of double negation $\vdash \neg\neg A \rightarrow A$, the formula $\vdash \neg\neg\neg\neg A \rightarrow \neg\neg A$ appeared even though it is obviously not a subformula of the theorem.

Gentzen proposed his deductive system in order to obtain a system with the subformula property. Then he defined the system $\mathscr{S}'$ by adding an additional rule of inference, the *cut rule*:

$$\frac{U, A \Rightarrow V \qquad U \Rightarrow V, A}{U \Rightarrow V}$$

1.	$A \to \neg A$	Assumption
2.	$\neg \neg A$	Assumption
3.	A	Double negation 2
4.	$\neg A$	MP 1, 3
5.	$A \to (\neg A \to \mathit{false})$	Theorem 3.21
6.	$\neg A \to \mathit{false}$	MP 3, 5
7.	false	MP 4, 6
8.	$\neg \neg A \to \mathit{false}$	Deduction 2, 7
9.	$\neg A$	Reductio ad absurdum 8
10.	$(A \to \neg A) \to \neg A$	Deduction 1, 9

Fig. 3.3 A natural deduction proof

to the system $\mathscr{S}$ and showed that proofs in $\mathscr{S}'$ can be mechanically transformed into proofs in $\mathscr{S}$. See Smullyan (1968, Chap. XII) for a proof of the following theorem.

Theorem 3.54 (Gentzen's Hauptsatz) *Any proof in $\mathscr{S}'$ can be transformed into a proof in $\mathscr{S}$ not using the cut rule.*

3.10 Summary

Deductive systems were developed to formalize mathematical reasoning. The structure of Hilbert systems such as $\mathscr{H}$ imitates the style of mathematical theories: a small number of axioms, *modus ponens* as the sole rule of inference and proofs as linear sequences of formulas. The problem with Hilbert systems is that they offer no guidance on how to find a proof of a formula. Gentzen systems such as $\mathscr{G}$ (and variants that use sequents or natural deduction) facilitate finding proofs because all formulas that appear are subformulas of the formula to be proved or their negations.

Both the deductive systems $\mathscr{G}$ and $\mathscr{H}$ are sound and complete. Completeness of $\mathscr{G}$ follows directly from the completeness of the method of semantic tableaux as a decision procedure for satisfiability and validity in propositional logic. However, the method of semantic tableaux is not very efficient. Our task in the next chapters is to study more efficient algorithms for satisfiability and validity.

3.11 Further Reading

Our presentation is based upon Smullyan (1968) who showed how Gentzen systems are closely related to tableaux. The deductive system $\mathscr{H}$ is from Mendelson (2009); he develops the theory of $\mathscr{H}$ (and later its generalization to first-order logic) without recourse to tableaux. Huth and Ryan (2004) base their presentation of logic on natural deduction. Velleman (2006) will help you learn how to prove theorems in mathematics.

3.12 Exercises

3.1 Prove in $\mathscr{G}$:

$$\vdash (A \to B) \to (\neg B \to \neg A),$$
$$\vdash (A \to B) \to ((\neg A \to B) \to B),$$
$$\vdash ((A \to B) \to A) \to A.$$

3.2 Prove that if $\vdash U$ in $\mathscr{G}$ then there is a closed semantic tableau for $\bar{U}$ (the forward direction of Theorem 3.7).

3.3 Prove the derived rule *modus tollens*:

$$\frac{\vdash \neg B \qquad \vdash A \to B}{\vdash \neg A}.$$

3.4 Give proofs in $\mathscr{G}$ for each of the three axioms of $\mathscr{H}$.

3.5 Prove $\vdash (\neg A \to A) \to A$ (Theorem 3.31) in $\mathscr{H}$.

3.6 Prove $\vdash (A \to B) \vee (B \to C)$ in $\mathscr{H}$.

3.7 Prove $\vdash ((A \to B) \to A) \to A$ in $\mathscr{H}$.

3.8 Prove $\{\neg A\} \vdash (\neg B \to A) \to B$ in $\mathscr{H}$.

3.9 Prove Theorem 3.34 in $\mathscr{H}$:

$$\vdash A \to A \vee B,$$
$$\vdash B \to A \vee B,$$
$$\vdash (A \to B) \to ((C \vee A) \to (C \vee B)).$$

3.10 Prove Theorem 3.35 in $\mathscr{H}$:

$$\vdash A \vee (B \vee C) \leftrightarrow (A \vee B) \vee C.$$

3.11 Prove Theorem 3.36 in $\mathscr{H}$:

$$\vdash A \vee (B \wedge C) \leftrightarrow (A \vee B) \wedge (A \vee C),$$
$$\vdash A \wedge (B \vee C) \leftrightarrow (A \wedge B) \vee (A \wedge C).$$

3.12 Prove that Axiom 2 of $\mathscr{H}$ is valid by constructing a semantic tableau for its negation.

3.13 Complete the proof that if $U' \subseteq U$ and $\vdash \bigvee U'$ then $\vdash \bigvee U$ (Lemma 3.40).

3.14 Prove the last two formulas of Exercise 3.1 in $\mathscr{H}$.

3.15 * Prove Axiom 3 of $\mathscr{H}$ in $\mathscr{H}'$.

3.16 * Prove that the Gentzen sequent system $\mathscr{S}$ is sound and complete.

3.17 * Prove that a set of formulas U is inconsistent if and only if there is a finite set of formulas $\{A_1, \ldots, A_n\} \subseteq U$ such that $\vdash \neg A_1 \vee \cdots \vee \neg A_n$.

3.18 A set of formulas U is *maximally consistent* iff every proper superset of U is not consistent. Let S be a countable, consistent set of formulas. Prove:

1. Every finite subset of S is satisfiable.
2. For every formula A, at least one of $S \cup \{A\}$, $S \cup \{\neg A\}$ is consistent.
3. S can be extended to a maximally consistent set.

References

M. Huth and M.D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems (Second Edition)*. Cambridge University Press, 2004.
E. Mendelson. *Introduction to Mathematical Logic (Fifth Edition)*. Chapman & Hall/CRC, 2009.
J.D. Monk. *Mathematical Logic*. Springer, 1976.
R.M. Smullyan. *First-Order Logic*. Springer-Verlag, 1968. Reprinted by Dover, 1995.
D.J. Velleman. *How to Prove It: A Structured Approach (Second Edition)*. Cambridge University Press, 2006.

Chapter 4
Propositional Logic: Resolution

The method of resolution, invented by J.A. Robinson in 1965, is an efficient method for searching for a proof. In this section, we introduce resolution for the propositional logic, though its advantages will not become apparent until it is extended to first-order logic. It is important to become familiar with resolution, because it is widely used in automatic theorem provers and it is also the basis of logic programming (Chap. 11).

4.1 Conjunctive Normal Form

Definition 4.1 A formula is in *conjunctive normal form (CNF)* iff it is a conjunction of disjunctions of literals. ■

Example 4.2 The formula:

$$(\neg p \vee q \vee r) \wedge (\neg q \vee r) \wedge (\neg r)$$

is in CNF while the formula:

$$(\neg p \vee q \vee r) \wedge ((p \wedge \neg q) \vee r) \wedge (\neg r)$$

is not in CNF, because $(p \wedge \neg q) \vee r$ is not a disjunction.

The formula:

$$(\neg p \vee q \vee r) \wedge \neg (\neg q \vee r) \wedge (\neg r)$$

is not in CNF because the second disjunction is negated. ■

Theorem 4.3 *Every formula in propositional logic can be transformed into an equivalent formula in CNF.*

Proof To convert an arbitrary formula to a formula in CNF perform the following steps, each of which preserves logical equivalence:

M. Ben-Ari, *Mathematical Logic for Computer Science*,
DOI 10.1007/978-1-4471-4129-7_4, © Springer-Verlag London 2012

1. Eliminate all operators except for negation, conjunction and disjunction by substituting logically equivalent formulas:

$$\begin{aligned} A \leftrightarrow B &\equiv (A \rightarrow B) \wedge (B \rightarrow A), \\ A \oplus B &\equiv \neg (A \rightarrow B) \vee \neg (B \rightarrow A), \\ A \rightarrow B &\equiv \neg A \vee B, \\ A \uparrow B &\equiv \neg (A \wedge B), \\ A \downarrow B &\equiv \neg (A \vee B). \end{aligned}$$

2. Push negations inward using De Morgan's laws:

$$\begin{aligned} \neg (A \wedge B) &\equiv (\neg A \vee \neg B), \\ \neg (A \vee B) &\equiv (\neg A \wedge \neg B), \end{aligned}$$

   until they appear only before atomic propositions or atomic propositions preceded by negations.
3. Eliminate sequences of negations by deleting double negation operators:

$$\neg \neg A \equiv A.$$

4. The formula now consists of disjunctions and conjunctions of literals. Use the distributive laws:

$$\begin{aligned} A \vee (B \wedge C) &\equiv (A \vee B) \wedge (A \vee C), \\ (A \wedge B) \vee C &\equiv (A \vee C) \wedge (B \vee C) \end{aligned}$$

   to eliminate conjunctions within disjunctions. ■

Example 4.4 The following sequence of formulas shows the four steps applied to the formula $(\neg p \rightarrow \neg q) \rightarrow (p \rightarrow q)$:

$$\begin{aligned} (\neg p \rightarrow \neg q) \rightarrow (p \rightarrow q) &\equiv \neg (\neg \neg p \vee \neg q) \vee (\neg p \vee q) \\ &\equiv (\neg \neg \neg p \wedge \neg \neg q) \vee (\neg p \vee q) \\ &\equiv (\neg p \wedge q) \vee (\neg p \vee q) \\ &\equiv (\neg p \vee \neg p \vee q) \wedge (q \vee \neg p \vee q). \end{aligned}$$

■

4.2 Clausal Form

The clausal form of formula is a notational variant of CNF. Recall (Definition 2.57) that a *literal* is an atom or the negation of an atom.

Definition 4.5

- A *clause* is a set of literals.
- A clause is considered to be an implicit disjunction of its literals.
- A *unit clause* is a clause consisting of exactly one literal.
- The empty set of literals is the *empty clause*, denoted by $\square$.
- A formula in *clausal form* is a set of clauses.
- A formula is considered to be an implicit conjunction of its clauses.
- The formula that is the *empty set of clauses* is denoted by $\emptyset$. ■

The only significant difference between clausal form and the standard syntax is that clausal form is defined in terms of sets, while our standard syntax was defined in terms of trees. A node in a tree may have multiple children that are identical subtrees, but a set has only one occurrence of each of its elements. However, this difference is of no logical significance.

Corollary 4.6 *Every formula ϕ in propositional logic can be transformed into an logically equivalent formula in clausal form.*

Proof By Theorem 4.3, ϕ can be transformed into a logically equivalent formula ϕ' in CNF. Transform each disjunction in ϕ' into a clause (a set of literals) and ϕ' itself into the set of these clauses. Clearly, the transformation into sets will cause multiple occurrences of literals and clauses to collapse into single occurrences. Logical equivalence is preserved by idempotence: $A \wedge A \equiv A$ and $A \vee A \equiv A$. ■

Example 4.7 The CNF formula:

$$(p \vee r) \wedge (\neg q \vee \neg p \vee q) \wedge (p \vee \neg p \vee q \vee p \vee \neg p) \wedge (r \vee p)$$

is logically equivalent to its clausal form:

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

The clauses corresponding to the first and last disjunctions collapse into a single set, while in the third disjunction multiple occurrences of p and $\neg p$ have been collapsed to obtain the third clause. ■

Trivial Clauses

A formula in clausal form can be simplified by removing trivial clauses.

Definition 4.8 A clause if *trivial* if it contains a pair of clashing literals. ■

Since a trivial clause is valid ($p \vee \neg p \equiv \mathit{true}$), it can be removed from a set of clauses without changing the truth value of the formula.

Lemma 4.9 *Let S be a set of clauses and let $C \in S$ be a trivial clause. Then $S - \{C\}$ is logically equivalent to S.*

Proof Since a clause is an implicit disjunction, C is logically equivalent to a formula obtained by weakening, commutativity and associativity of a valid disjunction $p \vee \neg p$ (Theorems 3.34–3.35). Let $\mathscr{I}$ be any interpretation for $S - \{C\}$. Since $S - \{C\}$ is an implicit conjunction, the value $v_{\mathscr{I}}(S - \{C\})$ is not changed by adding the clause C, since $v_{\mathscr{I}}(C) = T$ and $A \wedge T \equiv A$. Therefore, $v_{\mathscr{I}}(S - \{C\}) = v_{\mathscr{I}}(S)$. Since $\mathscr{I}$ was arbitrary, it follows that $S - \{C\} \equiv S$. ■

Henceforth, we will assume that all trivial clauses have been deleted from formulas in clausal form.

The Empty Clause and the Empty Set of Clauses

The following results may be a bit hard to understand at first, but they are very important. The proof uses reasoning about vacuous sets.

Lemma 4.10

$\square$, the empty clause, is unsatisfiable. *$\emptyset$, the empty set of clauses, is valid.*

Proof A clause is satisfiable iff there is *some* interpretation under which *at least one literal* in the clause is true. Let $\mathscr{I}$ be an arbitrary interpretation. Since there are no literals in $\square$, there are *no* literals whose value is true under $\mathscr{I}$. But $\mathscr{I}$ was an arbitrary interpretation, so $\square$ is unsatisfiable.

A set of clauses is valid iff *every* clause in the set is true in *every* interpretation. But there are no clauses in $\emptyset$ that need be true, so $\emptyset$ is valid. ■

Notation

When working with clausal form, the following additional notational conventions will be used:

- An abbreviated notation will be used for a formula in clausal form. The set delimiters { and } are removed from each clause and a negated literal is denoted by a bar over the atomic proposition. In this notation, the formula in Example 4.7 becomes:

$$\{pr,\ \bar{q}\,\bar{p}q,\ p\bar{p}q\}.$$

- S is a formula in clausal form, C is a clause and l is a literal. The symbols will be subscripted and primed as necessary.
- If l is a literal l^c is its complement: if $l = p$ then $l^c = \bar{p}$ and if $l = \bar{p}$ then $l^c = p$.

- The concept of an interpretation is generalized to literals. Let l be a literal defined on the atomic proposition p, that is, l is p or l is $\bar{p}$. Then an interpretation $\mathscr{I}$ for a set of atomic propositions including p is extended to l as follows:
  - $\mathscr{I}(l) = T$, if $l = p$ and $\mathscr{I}(p) = T$,
  - $\mathscr{I}(l) = F$, if $l = p$ and $\mathscr{I}(p) = F$,
  - $\mathscr{I}(l) = T$, if $l = \bar{p}$ and $\mathscr{I}(p) = F$,
  - $\mathscr{I}(l) = F$, if $l = \bar{p}$ and $\mathscr{I}(p) = T$.

The Restriction of CNF to 3CNF *

Definition 4.11 A formula is in *3CNF* iff it is in CNF and each disjunction has exactly three literals. ■

The problem of finding a model for a formula in CNF belongs to an important class of problems called $\mathscr{NP}$-complete problems (Sect. 6.7). This important theoretical result holds even if the formulas are restricted to 3CNF. To prove this, an efficient algorithm is needed to transform a CNF formula into one in 3CNF.

Algorithm 4.12 (CNF to 3CNF)
Input: A formula in CNF.
Output: A formula in 3CNF.

For each disjunction $C_i = l_i^1 \vee l_i^2 \vee \cdots \vee l_i^{n_i}$, perform the appropriate transformation depending of the value of n_i:

- If $n_i = 1$, create two new atoms p_i^1, p_i^2 and replace C_i by:

$$(l_i^1 \vee p_i^1 \vee p_i^2) \wedge (l_i^1 \vee \neg p_i^1 \vee p_i^2) \wedge (l_i^1 \vee p_i^1 \vee \neg p_i^2) \wedge (l_i^1 \vee \neg p_i^1 \vee \neg p_i^2).$$

- If $n_i = 2$, create one new atom p_i^1 and replace C_i by:

$$(l_i^1 \vee l_i^2 \vee p_i^1) \wedge (l_i^1 \vee l_i^2 \vee \neg p_i^1).$$

- If $n_i = 3$, do nothing.
- If $n_i > 3$, create $n - 3$ new atoms $p_i^1, p_i^2, \ldots, p_i^{n-3}$ and replace C_i by:

$$(l_i^1 \vee l_i^2 \vee p_i^1) \wedge (\neg p_i^1 \vee l_i^3 \vee p_i^2) \wedge \cdots \wedge (\neg p_i^{n-3} \vee l_i^{n-1} \vee l_i^n).$$

■

We leave the proof of the following theorem as an exercise.

Theorem 4.13 *Let A be a formula in CNF and let A' be the formula in 3CNF constructed from A by Algorithm 4.12. Then A is satisfiable if and only if A' is satisfiable. The length of A' (the number of symbols in A') is a polynomial in the length of A.*

4.3 Resolution Rule

Resolution is a refutation procedure used to check if a formula in clausal form is unsatisfiable. The resolution procedure consists of a sequence of applications of the resolution rule to a set of clauses. The rule maintains satisfiability: if a set of clauses is satisfiable, so is the set of clauses produced by an application of the rule. Therefore, if the (unsatisfiable) empty clause is ever obtained, the original set of clauses must have been unsatisfiable.

Rule 4.14 (Resolution rule) *Let* C_1, C_2 *be clauses such that* $l \in C_1$, $l^c \in C_2$. *The clauses* C_1, C_2 *are said to be* clashing clauses *and to* clash *on the complementary pair of literals* l, l^c. C, *the* resolvent *of* C_1 *and* C_2, *is the clause*:

$$Res(C_1, C_2) = (C_1 - \{l\}) \cup (C_2 - \{l^c\}).$$

C_1 *and* C_2 *are the* parent clauses *of* C. ■

Example 4.15 The pair of clauses $C_1 = ab\bar{c}$ and $C_2 = bc\bar{e}$ clash on the pair of complementary literals $c, \bar{c}$. The resolvent is:

$$C = (ab\bar{c} - \{\bar{c}\}) \cup (bc\bar{e} - \{c\}) = ab \cup b\bar{e} = ab\bar{e}.$$

Recall that a clause is a set so duplicate literals are removed when taking the union: $\{a, b\} \cup \{b, \bar{e}\} = \{a, b, \bar{e}\}$. ■

Resolution is only performed if the pair of clauses clash on *exactly* one pair of complementary literals.

Lemma 4.16 *If two clauses clash on more than one literal, their resolvent is a trivial clause* (*Definition* 4.8).

Proof Consider a pair of clauses:

$$\{l_1, l_2\} \cup C_1, \qquad \{l_1^c, l_2^c\} \cup C_2,$$

and suppose that we perform the resolution rule because the clauses clash on the pair of literals $\{l_1, l_1^c\}$. The resolvent is the trivial clause:

$$\{l_2, l_2^c\} \cup C_1 \cup C_2.$$

■

It is not strictly incorrect to perform resolution on such clauses, but since trivial clauses contribute nothing to the satisfiability or unsatisfiability of a set of clauses (Theorem 4.9), we agree to delete them from any set of clauses and not to perform resolution on clauses with two clashing pairs of literals.

Theorem 4.17 *The resolvent C is satisfiable if and only if the parent clauses C_1 and C_2 are both satisfiable.*

Proof Let C_1 and C_2 be satisfiable under an interpretation $\mathscr{I}$. Since l, l^c are complementary, either $\mathscr{I}(l) = T$ or $\mathscr{I}(l^c) = T$. Suppose that $\mathscr{I}(l) = T$; then $\mathscr{I}(l^c) = F$ and C_2, the clause containing l^c, can be satisfied only if $\mathscr{I}(l') = T$ for some *other* literal $l' \in C_2, l' \neq l^c$. By construction in the resolution rule, $l' \in C$, so $\mathscr{I}$ is also a model for C. A symmetric argument holds if $\mathscr{I}(l^c) = T$.

Conversely, let $\mathscr{I}$ be an interpretation which satisfies C; then $\mathscr{I}(l') = T$ for at least one literal $l' \in C$. By the resolution rule, $l' \in C_1$ or $l' \in C_2$ (or both). If $l' \in C_1$, then $v_{\mathscr{I}}(C_1) = T$. Since neither $l \in C$ nor $l^c \in C$, $\mathscr{I}$ is not defined on either l or l^c, and we can extend $\mathscr{I}$ to an interpretation $\mathscr{I}'$ by defining $\mathscr{I}(l^c) = T$. Since $l^c \in C_2$, $v_{\mathscr{I}'}(C_2) = T$ and $v_{\mathscr{I}'}(C_1) = v_{\mathscr{I}}(C_1) = T$ (because $\mathscr{I}$ is an extension of v) so $\mathscr{I}'$ is a model for both C_1 and C_2. A symmetric argument holds if $l' \in C_2$. ■

Algorithm 4.18 (Resolution procedure)
Input: A set of clauses S.
Output: S is satisfiable or unsatisfiable.

Let S be a set of clauses and define $S_0 = S$.

Repeat the following steps to obtain S_{i+1} from S_i until the procedure terminates as defined below:

- *Choose* a pair of clashing clauses $\{C_1, C_2\} \subseteq S_i$ that has not been chosen before.
- Compute $C = Res(C_1, C_2)$ according to the resolution rule.
- If C is not a trivial clause, let $S_{i+1} = S_i \cup \{C\}$; otherwise, $S_{i+1} = S_i$.

Terminate the procedure if:

- $C = \square$.
- All pairs of clashing clauses have be resolved. ■

Example 4.19 Consider the set of clauses:

$$S = \{(1)\, p,\ (2)\, \bar{p}q,\ (3)\, \bar{r},\ (4)\, \bar{p}\bar{q}r\},$$

where the clauses have been numbered. Here is a resolution derivation of $\square$ from S, where the justification for each line is the pair of the numbers of the parent clauses that have been resolved to give the resolvent clause:

5.	$\bar{p}\bar{q}$	3, 4
6.	$\bar{p}$	5, 2
7.	$\square$	6, 1

■

It is easier to read a resolution derivation if it is presented as a tree. Figure 4.1 shows the tree that represents the derivation of Example 4.19. The clauses of S label leaves, and the resolvents label interior nodes whose children are the parent clauses used in the resolution.

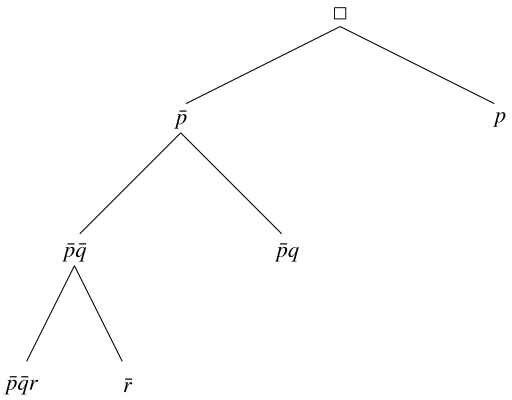


Fig. 4.1 A resolution refutation represented as a tree

Definition 4.20 A derivation of □ from a set of clauses S is a *refutation by resolution* of S or a *resolution refutation* of S. ■

Since □ is unsatisfiable, by Theorem 4.17 if there exists a refutation of S by resolution then S is unsatisfiable.

In Example 4.19, we derived the unsatisfiable clause □, so we conclude that the set of clauses S is unsatisfiable. We leave it to the reader to check that S is the clausal form of $\neg A$ where A is an instance of Axiom 2 of $\mathscr{H}$ $(p \to (q \to r)) \to ((p \to q) \to (p \to r))$. Since $\neg A$ is unsatisfiable, A is valid.

4.4 Soundness and Completeness of Resolution *

The soundness of resolution follows easily from Theorem 4.17, but completeness is rather difficult to prove, so you may want to skip the this section on your first reading.

Theorem 4.21 *If the set of clauses labeling the leaves of a resolution tree is satisfiable then the clause at the root is satisfiable.*

The proof is by induction using Theorem 4.17 and is left as an exercise.

The converse to Theorem 4.21 is not true because we have no way of ensuring that the extensions made to $\mathscr{I}$ on all branches are consistent. In the tree in Fig. 4.2, the set of clauses on the leaves $S = \{r, pq\bar{r}, \bar{r}, p\bar{q}r\}$ is not satisfiable even though the clause p at the root is satisfiable. Since S is unsatisfiable, it has a refutation: whenever the pair of clashing clauses r and $\bar{r}$ is chosen, the resolvent will be □.

Resolution is a refutation procedure, so soundness and completeness are better expressed in terms of unsatisfiability, rather than validity.

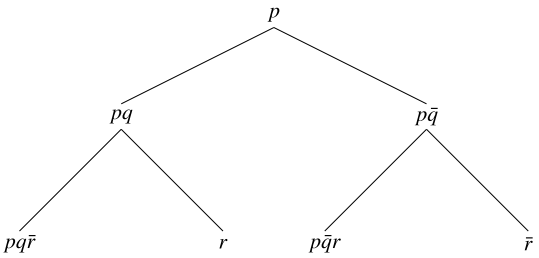


Fig. 4.2 Incomplete resolution tree

Corollary 4.22 (Soundness) *Let S be a set of clauses. If there is a refutation by resolution for S then S is unsatisfiable.*

Proof Immediate from Theorem 4.21 and Lemma 4.10. ■

Theorem 4.23 (Completeness) *If a set of clauses is unsatisfiable then the empty clause $\square$ will be derived by the resolution procedure.*

We have to prove that given an unsatisfiable set of clauses, the resolution procedure will eventually terminate producing $\square$, rather than continuing indefinitely or terminating but failing to produce $\square$. The resolution procedure was defined so that the same pair of clauses is never chosen more than once. Since there are only a finite number of distinct clauses on the finite set of atomic propositions appearing in a set of clauses, the procedure terminates. We need only prove that when the procedure terminates, the empty clause is produced.

Semantic Trees

The proof will use *semantic trees* (which must not be confused with semantic tableaux). A semantic tree is a data structure for recording assignments of T and F to the atomic propositions of a formula in the process of searching for a model (satisfying interpretation). If the formula is unsatisfiable, the search for a model must end in failure. Clauses that are created during a resolution refutation will be associated with nodes of the tree called failure nodes; these nodes represent assignments that falsify the associated clauses. Eventually, the root node (associated with the empty clause $\square$) will be shown to be a failure node.

Definition 4.24 (Semantic tree) Let S be a set of clauses and let $P_S = \{p_1, \ldots, p_n\}$ be the set of atomic propositions appearing in S. $\mathscr{T}$, the *semantic tree* for S, is a complete binary tree of depth n such that for $1 \leq i \leq n$, every left-branching edge from a node at depth $i - 1$ is labeled p_i and every right-branching edge is labeled by $\bar{p}_i$.

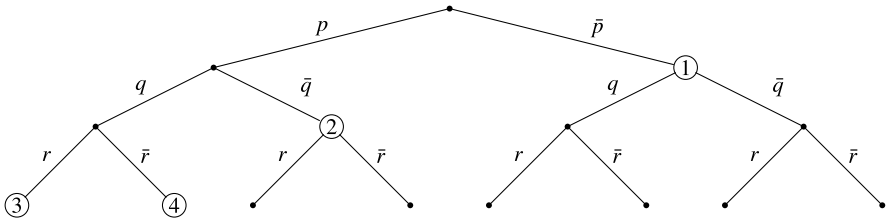


Fig. 4.3 Semantic tree

Every branch b from the root to a leaf in $\mathcal{T}$ is labeled by a sequence of literals $\{l_1, \ldots, l_n\}$, where $l_i = p_i$ or $l_i = \bar{p}_i$. b defines an interpretation by:

$$\begin{aligned} \mathcal{I}_b(p_i) &= T \quad \text{if } l_i = p_i, \\ \mathcal{I}_b(p_i) &= F \quad \text{if } l_i = \bar{p}_i. \end{aligned}$$

A branch b is *closed* if $v_b(S) = F$, otherwise b is *open*. $\mathcal{T}$ is *closed* if all branches are closed, otherwise $\mathcal{T}$ is *open*. ■

Example 4.25 The semantic tree for $S = \{p,\ \bar{p}q,\ \bar{r},\ \bar{p}\bar{q}r\}$ is shown in Fig. 4.3 where the numbers on the nodes will be explained later. The branch b ending in the leaf labeled 4 defines the interpretation:

$$\mathcal{I}_b(p) = T, \qquad \mathcal{I}_b(q) = T, \qquad \mathcal{I}_b(r) = F.$$

Since $v_{\mathcal{I}_b}(\bar{p}\bar{q}r) = F$, $v_{\mathcal{I}_b}(S) = F$ (a set of clauses is the conjunction of its members) and the branch b is closed. We leave it to the reader to check that every branch in this tree is closed. ■

Lemma 4.26 *Let S be a set of clauses and let $\mathcal{T}$ a semantic tree for S. Every interpretation $\mathcal{I}$ for S corresponds to $\mathcal{I}_b$ for some branch b in $\mathcal{T}$, and conversely, every $\mathcal{I}_b$ is an interpretation for S.*

Proof By construction. ■

Theorem 4.27 *The semantic tree $\mathcal{T}$ for a set of clauses S is closed if and only if the set S is unsatisfiable.*

Proof Suppose that $\mathcal{T}$ is closed and let $\mathcal{I}$ be an arbitrary interpretation for S. By Lemma 4.26, $\mathcal{I}$ is $\mathcal{I}_b$ for some branch in $\mathcal{T}$. Since $\mathcal{T}$ is closed, $v_b(S) = F$. But $\mathcal{I} = \mathcal{I}_b$ was arbitrary so S is unsatisfiable.

Conversely, let S be an unsatisfiable set of clauses, $\mathcal{T}$ the semantic tree for S and b an arbitrary branch in $\mathcal{T}$. Then v_b is an interpretation for S by Lemma 4.26, and $v_b(S) = F$ since S is unsatisfiable. Since b was arbitrary, $\mathcal{T}$ is closed. ■

Failure Nodes

When traversing a branch of the semantic tree top-down, a (partial) branch from the root to a node represents a partial interpretation (Definition 2.18) defined by the labels of the edges that were traversed. It is possible that this partial interpretation is sufficiently defined to evaluate the truth value of some clauses; in particular, some clause might evaluate to F. Since a set of clauses is an implicit conjunction, if even one clause evaluates to F, the partial interpretation is sufficient to conclude that the entire set of clauses is false. In a *closed* semantic tree, there must be such a node on every branch. However, if a clause contains the literal labeling the edge to a leaf, a (full) interpretation may be necessary to falsify the clause.

Example 4.28 In the semantic tree for $S = \{p,\ \bar{p}q,\ \bar{r},\ \bar{p}\bar{q}r\}$ (Fig. 4.3), the partial branch $b_{p\bar{q}}$ from the root to the node numbered 2 defines a partial interpretation $\mathscr{I}_{b_{p\bar{q}}}(p) = T$, $\mathscr{I}_{b_{p\bar{q}}}(q) = F$, which falsifies the clause $\bar{p}q$ and thus the entire set of clauses S.

Consider now the partial branches b_p and b_{pq} and the full branch b_{pqr} that are obtained by always taking the child labeled by a positive literal. The partial interpretation $\mathscr{I}_{b_p}(p) = T$ does not falsify any of the clauses, nor does the partial interpretation $\mathscr{I}_{b_{pq}}(p) = T$, $\mathscr{I}b_{pq}(q) = T$. Only the full interpretation $\mathscr{I}_{b_{pqr}}$ that assigns T to r falsifies one of the clauses ($\bar{r}$). ■

Definition 4.29 Let $\mathscr{T}$ be a closed semantic tree for a set of clauses S and let b be a branch in $\mathscr{T}$. The node in b closest to the root which falsifies S is a *failure node*.

Example 4.30 Referring again to Fig. 4.3, the node numbered 2 is a failure node since neither its parent node (which defines the partial interpretation $\mathscr{I}_{b_p}$) nor the root itself falsifies any of the clauses in the set. We leave it to the read to check that all the numbered nodes are failure nodes. ■

Since a failure node falsifies S (an implicit conjunction of clauses), it must falsify at least once clause in S.

Definition 4.31 A clause falsified by a failure node is a *clause associated with the node*. ■

Example 4.32 The failure nodes in Fig. 4.3 are labeled with the number of a clause associated with it; the numbers were given in Examples 4.19. ■

It is possible that more than one clause is associated with a failure node; for example, if q is added to the set of clauses, then q is another clause associated with failure node numbered 2.

We can characterize the clauses associated with failure nodes. For C to be falsified at a failure node n, *all* the literals in C must be assigned F in the partial interpretation.

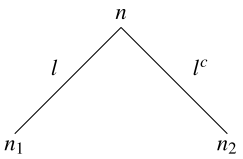


Fig. 4.4 Inference and failure nodes

Example 4.33 In Fig. 4.3, $\bar{r}$ is a clause associated with the failure node numbered 3. $\{\bar{r}\}$ is a proper subset of $\{\bar{p}, \bar{q}, \bar{r}\}$, the set of *complements* of the literals assigned to on the branch. ■

Lemma 4.34 *A clause C associated with a failure node n is a subset of the complements of the literals appearing on the partial branch b from the root to n.*

Proof Let $C = l_1 \cdots l_k$ and let $E = \{e_1, \ldots, e_m\}$ be the set of literals labeling edges in the branch. Since C is the clause associated with the failure node n, $v_b(C) = F$ for the interpretation $\mathscr{I}_b$ defined by $\mathscr{I}_b(e_j) = T$ for all $e_j \in E$. C is a disjunction so for each $l_i \in C$, $\mathscr{I}_b(l_i)$ must be assigned F. Since $\mathscr{I}_b$ only assigns to the literals in E, it follows that $l_i = e_j^c$ for some $e_j \in E$. Therefore, $C = l_1 \cdots l_k \subseteq \{e_1^c, \ldots, e_m^c\}$. ■

Inference Nodes

Definition 4.35 n is an *inference node* iff its children are failure nodes. ■

Example 4.36 In Fig. 4.3, the parent of nodes 3 and 4 is an inference node. ■

Lemma 4.37 *Let $\mathscr{T}$ be a closed semantic tree for a set of clauses S. If there are at least two failure nodes in $\mathscr{T}$, then there is at least one inference node.*

Proof Suppose that n_1 is a failure node and that its sibling n_2 is not (Fig. 4.4). Then no ancestor of n_2 can be a failure node, because its ancestors are also ancestors of n_1, which is, by assumption, a failure node and thus the node *closest* to the root on its branch which falsifies S.

$\mathscr{T}$ is closed so every branch in $\mathscr{T}$ is closed, in particular, any branch b that includes n_2 is closed. By definition of a closed branch, $\mathscr{I}_b$, the full interpretation associated with the leaf of b, must falsify S. Since neither n_2 nor any ancestor of n_2 is a failure node, some node below n_2 on b (perhaps the leaf itself) must be the highest node which falsifies a clause in S.

We have shown that given an arbitrary failure node n_1, either its sibling n_2 is a failure node (and hence their parent is an inference node), or there is a failure node at a *greater* depth than n_1 and n_2. Therefore, if there is no inference node, there must be an infinite sequence of failure nodes. But this is impossible, since a semantic tree is finite (its depth is the number of different atomic propositions in S). ■

Lemma 4.38 *Let $\mathcal{T}$ be closed semantic tree and let n be an inference node whose children n_1 and n_2 of n are (by definition) failure nodes with clauses C_1 and C_2 associated with them, respectively. Then C_1, C_2 clash and the partial interpretation defined by the branch from the root to n falsifies their resolvent.*

Proof of the Notation follows Fig. 4.4. Let b_1 and b_2 be the partial branches from the root to the nodes n_1 and n_2, respectively. Since n_1 and n_2 are failure nodes and since C_1 and C_2 are clauses associated with the nodes, they are *not* falsified by any node higher up in the tree. By Lemma 4.34, the clauses C_1 and C_2 are subsets of the complements of the literals labeling the nodes of b_1 and b_2, respectively. Since b_1 and b_2 are identical except for the edges from n to n_1 and n_2, we must have $\bar{l} \in C_1$ and $\bar{l^c} \in C_2$ so that the clauses are falsified by the assignments to the literals.

Since the nodes n_1 and n_2 are failure nodes, $v_{\mathscr{I}_{b_1}}(C_1) = v_{\mathscr{I}_{b_2}}(C_2) = F$. But clauses are disjunctions so $v_{\mathscr{I}_{b_1}}(C_1 - \{\bar{l}\}) = v_{\mathscr{I}_{b_2}}(C_2 - \{\bar{l^c}\}) = F$ and this also holds for the interpretation $\mathscr{I}_b$. Therefore, their resolvent is also falsified:

$$v_{\mathscr{I}_b}(\,(C_1 - \{\bar{l}\}) \cup (C_2 - \{\bar{l^c}\})\,) = F.$$

■

Example 4.39 In Fig. 4.3, $\bar{r}$ and $\bar{p}\bar{q}r$ are clauses associated with failure nodes 3 and 4, respectively. The resolvent $\bar{p}\bar{q}$ is falsified by $\mathscr{I}_{pq}(p) = T$, $\mathscr{I}_{pq}(q) = T$, the partial interpretation associated with the parent node of 3 and 4. The parent node is now a failure node for the set of clauses $S \cup \{\bar{p}\bar{q}\}$. ■

There is a technicality that must be dealt with before we can prove completeness. A semantic tree is defined by choosing an ordering for the set of atoms that appear in *all* the clauses in a set; therefore, an inference node may not be a failure node.

Example 4.40 The semantic tree in Fig. 4.3 is also a semantic tree for the set of clauses $\{p,\ \bar{p}q,\ \bar{r},\ \bar{p}r\}$. Node 3 is a failure node associated with $\bar{r}$ and 4 is a failure node associated with $\bar{p}r$, but their parent is *not* a failure node for their resolvent $\bar{p}$, since it is already falsified by a node higher up in the tree. (Recall that a failure node was defined to be the node *closest* to the root which falsifies the set of clauses.) ■

Lemma 4.41 *Let n be an inference node, $C_1, C_2 \in S$ clauses associated with the failure nodes that are the children of n, and C their resolvent. Then $S \cup \{C\}$ has a failure node that is either n or an ancestor of n and C is a clause associated with the failure node.*

Proof By Lemma 4.38, $v_{\mathscr{I}_b}(C) = F$, where $\mathscr{I}_b$ is the partial interpretation associated with the partial branch b from the root to the inference node. By Lemma 4.34, $C \subseteq \{l_1^c, \ldots, l_n^c\}$, the set of complements of the literals labeling b. Let j be the smallest index such $C \cap \{l_{j+1}^c, \ldots, l_n^c\} = \emptyset$. Then $C \subseteq \{l_1^c, \ldots, l_j^c\} \subseteq \{l_1^c, \ldots, l_n^c\}$ so $v_{\mathscr{I}_b^j}(C) = v_{\mathscr{I}_b}(C) = F$ where $\mathscr{I}_b^j$ is the partial interpretation defined by the partial branch from the root to node j. It follows that j is a failure node and C is a clause associated with it. ■

Example 4.42 Returning to the set of clauses $\{p,\ \bar{p}q,\ \bar{r},\ \bar{p}r\}$ in Example 4.40, the resolvent at the inference node is $C = \{\bar{p}\}$. Now $C = \{\bar{p}\} \subseteq \{\bar{p}, \bar{q}\}$, the complements of the literals on the partial branch from the root to the inference node. Let $j = 1$. Then $\{\bar{p}\} \cap \{\bar{q}\} = \emptyset$, $\{\bar{p}\} \subseteq \{\bar{p}\}$ and $C = \{\bar{p}\}$ is falsified by the partial interpretation $\mathscr{I}_{b_p}(p) = T$. ■

We now have all the machinery needed to proof completeness.

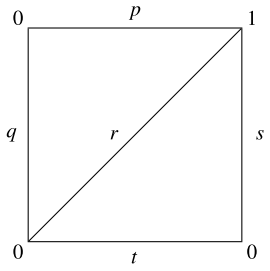
Proof of Completeness of resolution If S is an unsatisfiable set of clauses, there is a closed semantic tree $\mathscr{T}$ for S. If S is unsatisfiable and does not already contain $\square$, there must be at least two failure nodes in $\mathscr{T}$ (exercise), so by Lemma 4.37, there is at least one inference node in $\mathscr{T}$.

An application of the resolution rule at the inference node adds the resolvent to the set, creating a failure node by Lemma 4.41 and deleting two failure nodes, thus decreasing the number of failure nodes. When the number of failure nodes has decreased to one, it must be the root which is associated with the derivation of the empty clause by the resolution rule. ■

4.5 Hard Examples for Resolution *

If you try the resolution procedure on formulas in clausal form, you will find that is usually quite efficient. However, there are families of formulas on which *any* resolution refutation is necessarily inefficient. We show how an unsatisfiable set of clauses can be associated with an arbitrarily large graph such that a resolution refutation of a set of clauses from this family produces an exponential number of new clauses.

Let G be an undirected graph. Label the nodes with 0 or 1 and the edges with distinct atoms. The following graph will be used as an example throughout this section.



Definition 4.43

- The *parity* of a natural number i is 0 if i is even and 1 if i is odd.
- Let C be a clause. $\Pi(C)$, the *parity of* C, is the parity of the number of complemented literals in C.
- Let $\mathscr{I}$ be an interpretation for a set of atomic propositions $\mathscr{P}$. $\Pi(\mathscr{I})$, the *parity of* $\mathscr{I}$, is the parity of the number of atoms in $\mathscr{P}$ assigned T in $\mathscr{I}$. ■

Example 4.44 $\Pi(p\bar{r}\bar{s}) = 2$ and $\Pi(\bar{p}\bar{r}\bar{s}) = 3$. For the interpretation $\mathscr{I}$ defined by $\mathscr{I}(p) = T$, $\mathscr{I}(q) = T$, $\mathscr{I}(r) = F$, the parity $\Pi(\mathscr{I})$ is 2. ■

With each graph we associate a set of clauses.

Definition 4.45 Let G be an undirected, connected graph, whose nodes are labeled with 0 or 1 and whose edges are labeled with distinct atomic propositions. Let n be a node of G labeled a_n (0 or 1) and let $\mathscr{P}_n = \{p_1, \ldots, p_k\}$ be the set of atoms labeling edges incident with n.

$C(n)$, the *set of clauses associated with* n, is the set of all clauses C that can be formed as follows: the literals of C are *all* the atoms in $\mathscr{P}_n$, some of which are negated so that $\Pi(C) \neq a_n$.

$C(G)$, the *set of clauses associated with* G, is $\bigcup_{n \in G} C(n)$.

Let $\mathscr{I}$ be an interpretation on all the atomic propositions $\bigcup_n \mathscr{P}_n$ in G. $\mathscr{I}_n$ is the *restriction* of $\mathscr{I}$ to node n which assigns truth values only to the literals in $C(n)$. ■

Example 4.46 The sets of clauses associated with the four nodes of the graph are (clockwise from the upper-left corner):

$$\{\bar{p}q,\ p\bar{q}\}, \qquad \{prs,\ \bar{p}\bar{r}s,\ \bar{p}r\bar{s},\ p\bar{r}\bar{s}\}, \qquad \{\bar{s}t,\ s\bar{t}\}, \qquad \{\bar{q}rt,\ q\bar{r}t,\ qr\bar{t},\ \bar{q}\bar{r}\bar{t}\}.$$

By definition, the parity of each clause associated with a node n must be opposite the parity of n. For example:

$$\begin{aligned} \Pi(\bar{p}\bar{r}s) = 0 &\neq 1, \\ \Pi(\bar{q}rt) = 1 &\neq 0. \end{aligned}$$

■

Lemma 4.47 *$\mathscr{I}_n$ is a model for $C(n)$ if and only if $\Pi(\mathscr{I}_n) = a_n$.*

Proof Suppose that $\Pi(\mathscr{I}_n) \neq a_n$ and consider the clause $C \in C(n)$ defined by:

$$\begin{aligned} l_i &= p_i \qquad \text{if } \mathscr{I}_n(p_i) = F, \\ l_i &= \bar{p}_i \qquad \text{if } \mathscr{I}_n(p_i) = T. \end{aligned}$$

Then:

$$\begin{aligned} \Pi(C) &= \text{parity of negated atoms of } C && \text{(by definition)} \\ &= \text{parity of literals assigned } T && \text{(by construction)} \\ &= \Pi(\mathscr{I}_n) && \text{(by definition)} \\ &\neq a_n && \text{(by assumption).} \end{aligned}$$

But $\mathscr{I}_n(C) = F$ since $\mathscr{I}_n$ assigns F to each literal $l_i \in C$ (T to negated literals and F to atoms). Therefore, $\mathscr{I}_n$ does not satisfy all clauses in $C(n)$.

We leave the proof of the converse as an exercise. ■

Example 4.48 Consider an interpretation $\mathscr{I}$ such that $\mathscr{I}_n$ is:

$$\mathscr{I}_n(p) = \mathscr{I}_n(r) = \mathscr{I}_n(s) = T$$

for n the upper right node in the graph. For such interpretations, $\Pi(\mathscr{I}_n) = 1 = a_n$, and it is easy to see that $v_n(prs) = v_n(\bar{p}\bar{r}s) = v_n(\bar{p}q\bar{s}) = v_n(p\bar{r}\bar{s}) = T$ so $\mathscr{I}$ is a model for $C(n)$.

Consider an interpretation $\mathscr{I}$ such that $\mathscr{I}_n$ is:

$$\mathscr{I}_n(p) = \mathscr{I}_n(r) = \mathscr{I}_n(s) = F.$$

$\Pi(\mathscr{I}_n) = 0 \neq a_n$ and $v_n(prs) = F$ so $\mathscr{I}$ is not a model for $C(n)$. ■

$C(G)$ is the set of clauses obtained by taking the union of the clauses associated with all the nodes in the graph. Compute the sum modulo 2 (denoted $\sum$ in the following lemma) of the labels of the nodes and the sum of the parities of the restrictions of an interpretation to each node. Since each atom appears twice, the sum of the parities of the restricted interpretations must be 0. By Lemma 4.47, for the clauses to be satisfiable, the sum of the node labels must be the same as the sum of the parities of the interpretations, namely zero.

Lemma 4.49 *If* $\sum_{n\in G} a_n = 1$ *then* $C(G)$ *is unsatisfiable.*

Proof Suppose that there exists a model $\mathscr{I}$ for $C(G) = \bigcup_{n\in G} C(n)$. By Lemma 4.47, for all n, $\Pi(\mathscr{I}_n) = a_n$, so:

$$\sum_{n\in G} \Pi(\mathscr{I}_n) = \sum_{n\in G} a_n = 1.$$

Let p_e be the atom labeling an arbitrary edge e in G; it is incident with (exactly) two nodes, n_1 and n_2. The sum of the parities of the restricted interpretations can be written:

$$\sum_{n\in G} \Pi(\mathscr{I}_n) = \Pi(\mathscr{I}_{n_1}) + \Pi(\mathscr{I}_{n_2}) + \sum_{n\in(G-\{n_1,n_2\})} \Pi(\mathscr{I}_n).$$

Whatever the value of the assignment of $\mathscr{I}$ to p_e, it appears once in the first term, once in the second term and not at all in the third term above. By modulo 2 arithmetic, the total contribution of the assignment to p_e to $\sum_{n\in G} \Pi(\mathscr{I}_n)$ is 0. Since e was arbitrary, this is true for all atoms, so:

$$\sum_{n\in G} \Pi(\mathscr{I}_n) = 0,$$

contradicting $\sum_{n\in G} \Pi(\mathscr{I}_n) = 1$ obtained above. Therefore, $\mathscr{I}$ cannot be a model for $C(G)$, so $C(G)$ must be unsatisfiable. ■

Tseitin (1968) defined a family G_n of graphs of arbitrary size n and showed that for a restricted form of resolution the number of distinct clauses that appear a resolution refutation of $C(G_n)$ is *exponential* in n. About twenty years later, the restriction was removed by Urquhart (1987).

4.5.1 Tseitin Encoding

The standard procedure for transforming a formula into CNF (Sect. 4.1) can lead to formulas that are significantly larger than the original formula. In practice, an alternate transformation by Tseitin (1968) yields a more compact set of clauses at the expense of adding new atoms.

Algorithm 4.50 (Tseitin encoding) Let A be a formula in propositional logic. Define a sequence of formulas $A = A_0, A_1, A_2, \ldots$ by repeatedly performing the transformation:

- Let $B_i' \circ B_i''$ be a subformula of A_i, where B_i', B_i'' are literals.
- Let p_i be a new atom that does not appear in A_i. Construct A_{i+1} by replacing the subformula $B_i' \circ B_i''$ by p_i and adding the CNF of:

$$p_i \leftrightarrow B_i' \circ B_i''.$$

- Terminate the transformation when A_n is in CNF. ■

Theorem 4.51 *Let A be a formula in propositional logic and apply Algorithm 4.50 to obtain the CNF formula A_n. Then A is satisfiable if and only if A_n is satisfiable.*

Example 4.52 Let n be a node labeled 1 with five incident edges labeled by the atoms p, q, r, s, t. $C(n)$ consists of all clauses of even parity defined on these atoms:

$$\begin{array}{l} pqrst, \\ \bar{p}\bar{q}rst,\ \bar{p}q\bar{r}st,\ \ldots,\ pq\bar{r}s\bar{t},\ pqr\bar{s}\bar{t} \\ p\bar{q}\bar{r}\bar{s}\bar{t},\ \bar{p}q\bar{r}\bar{s}\bar{t},\ \bar{p}\bar{q}r\bar{s}\bar{t},\ \bar{p}\bar{q}\bar{r}s\bar{t},\ \bar{p}\bar{q}\bar{r}\bar{s}t. \end{array}$$

There are 16 clauses in $C(n)$ since there $2^5 = 32$ clauses on five atoms and half of them have even parity: one clause with parity 0, $\frac{5!}{2!\cdot(5-2)!} = 10$ clauses with parity 2 and five clauses with parity 4. We leave it to the reader to show that this set of clauses is logically equivalent to the formula:

$$(p \leftrightarrow (q \leftrightarrow (r \leftrightarrow (s \leftrightarrow t)))),$$

where we have used parentheses to bring out the structure of subformulas. Applying the Tseitin encoding, we choose four new atoms a, b, c, d and obtain the set of formulas:

$$\{a \leftrightarrow (s \leftrightarrow t),\ b \leftrightarrow (r \leftrightarrow a),\ c \leftrightarrow (q \leftrightarrow b),\ d \leftrightarrow (s \leftrightarrow c)\}.$$

Each of the new formulas is logically equivalent to one in CNF that contains four disjunctions of three literals each; for example:

$$a \leftrightarrow (s \leftrightarrow t) \equiv \{a \vee s \vee t,\ \bar{a} \vee \bar{s} \vee t,\ \bar{a} \vee s \vee \bar{t},\ a \vee \bar{s} \vee \bar{t}\}.$$

Sixteen clauses of five literals have been replaced by the same number of clauses but each clause has only three literals. ■

4.6 Summary

Resolution is a highly efficient refutation procedure that is a decision procedure for unsatisfiability in propositional logic. It works on formulas in clausal form, which is a set representation of conjunctive normal form (a conjunction of disjunctions of literals). Each resolution step takes two clauses that clash on a pair of complementary literals and produces a new clause called the resolvent. If the formula is unsatisfiable, the empty clause will eventually be produced.

4.7 Further Reading

Resolution for propositional logic is presented in the advanced textbooks by Nerode and Shore (1997) and Fitting (1996).

4.8 Exercises

4.1 A formula is in *disjunctive normal form (DNF)* iff it is a disjunction of conjunctions of literals. Show that every formula is equivalent to one in DNF.

4.2 A formula A is in *complete DNF* iff it is in DNF and each propositional letter in A appears in a literal in each conjunction. For example, $(p \wedge q) \vee (\bar{p} \wedge q)$ is in complete DNF. Show that every formula is equivalent to one in complete DNF.

4.3 Simplify the following sets of literals, that is, for each set S find a simpler set S', such that S' is satisfiable if and only if S is satisfiable.

$$\{p\bar{q},\ q\bar{r},\ rs,\ p\bar{s}\},$$
$$\{pqr,\ \bar{q},\ p\bar{r}s,\ qs,\ p\bar{s}\},$$
$$\{pqrs,\ \bar{q}rs,\ \bar{p}rs,\ qs,\ \bar{p}s\},$$
$$\{\bar{p}q,\ qrs,\ \bar{p}\bar{q}rs,\ \bar{r},\ q\}.$$

4.4 Given the set of clauses $\{\bar{p}\bar{q}r,\ pr,\ qr,\ \bar{r}\}$ construct two refutations: one by resolving the literals in the order $\{p, q, r\}$ and the other in the order $\{r, q, p\}$.

4.5 Transform the set of formulas

$$\{p,\ \ p \to ((q \vee r) \wedge \neg\,(q \wedge r)),\ \ p \to ((s \vee t) \wedge \neg\,(s \wedge t)),\ \ s \to q,\ \neg r \to t,\ \ t \to s\,\}$$

into clausal form and refute using resolution.

4.6 * The half-adder of Example 1.2 implements the pair of formulas:

$$s \leftrightarrow \neg\,(b1 \wedge b2) \wedge (b1 \vee b2), \qquad c \leftrightarrow b1 \wedge b2.$$

Transform the formulas to a set of clauses. Show that the addition of the unit clauses $\{b1,\ b2,\ \bar{s},\ \bar{c}\}$ gives an unsatisfiable set while the addition of $\{b1,\ b2,\ \bar{s},\ c\}$ gives a satisfiable set. Explain what this means in terms of the behavior of the circuit.

4.7 Prove that if the set of clauses labeling the leaves of a resolution tree is satisfiable then the clause at the root is satisfiable (Theorem 4.21).

4.8 Construct a resolution refutation for the set of Tseitin clauses given in Example 4.46.

4.9 * Construct the set of Tseitin clauses corresponding to a labeled complete graph on five vertices and give a resolution refutation of the set.

4.10 * Construct the set of Tseitin clauses corresponding to a labeled complete bipartite graph on three vertices on each side and give a resolution refutation of the set.

4.11 * Show that if $\Pi(v_n) = b_n$, then v_n satisfies all clauses in $C(n)$ (the converse direction of Lemma 4.47).

4.12 * Let $\{q_1, \ldots, q_n\}$ be literals on *distinct* atoms. Show that $q_1 \leftrightarrow \cdots \leftrightarrow q_n$ is satisfiable iff $\{p \leftrightarrow q_1, \ldots, p \leftrightarrow q_n\}$ is satisfiable, where p is a new atom. Construct an efficient decision procedure for formulas whose only operators are $\neg$, $\leftrightarrow$ and $\oplus$.

4.13 Prove Theorem 4.13 on the correctness of the CNF-to-3CNF algorithm.

4.14 Carry out the Tseitin encoding on the formula $(a \to (c \wedge d)) \vee (b \to (c \wedge e))$.

References

M. Fitting. *First-Order Logic and Automated Theorem Proving* (*Second Edition*). Springer, 1996.
A. Nerode and R.A. Shore. *Logic for Applications* (*Second Edition*). Springer, 1997.
G.S. Tseitin. On the complexity of derivation in propositional calculus. In A.O. Slisenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125. Steklov Mathematical Institute, 1968.
A. Urquhart. Hard examples for resolution. *Journal of the ACM*, 34:209–219, 1987.