Mathematical Logic for Computer Science is a mathematics textbook with theorems and proofs, but the choice of topics has been guided by the needs of computer science students. The method of semantic tableaux provides a elegant way to teach logic that is both theoretically sound and yet sufficiently elementary for undergraduates. To provide a balanced treatment of logic, tableaux are related to deductive proof systems.

The logical systems presented are:

- Propositional calculus (including binary decision diagrams)
- Predicate calculus
- Resolution
- Hoare logic
- Z
- Temporal logic

Answers to exercises as well as Prolog source code for algorithms are available via the Springer London website:
http://www.springer.co.uk/com_pub/ct_mics.html

Mordechai Ben-Ari is Associate Professor at the Department of Science Teaching of the Weizmann Institute of Science. He has published textbooks on concurrent programming and programming languages.

ISBN 1-85233-319-7

Mordechai Ben-Ari

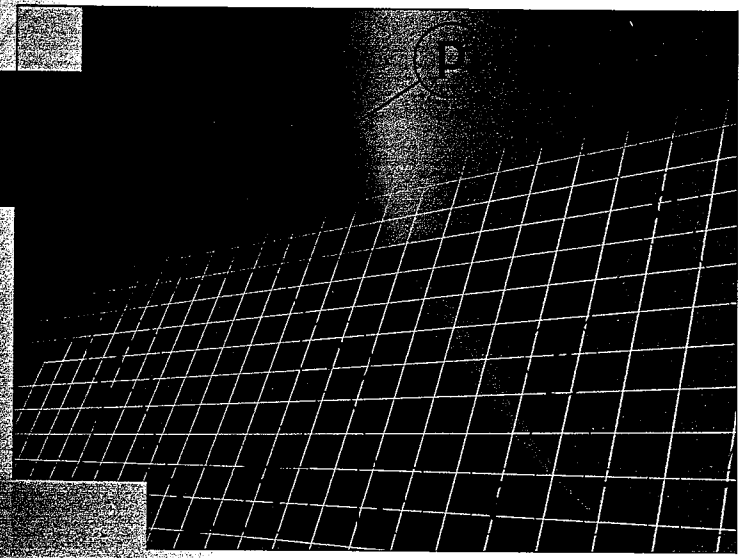
Mathematical Logic *for* Computer Science

http://www.springer.co.uk

Mordechai Ben-Ari

Mathematical Logic *for* Computer Science

[Second edition]



Springer

Mordechai Ben-Ari

Mathematical Logic for Computer Science

(Second edition)

Springer
London
Berlin
Heidelberg
New York
Barcelona
Hong Kong
Milan
Paris
Singapore

Mordechai Ben-Ari, PhD
Department of Science Teaching, Weizmann Institute of Science,
Rehovot 76100, Israel

ISBN 1-85233-319-7 2nd edition Springer-Verlag London Berlin Heidelberg
ISBN 0-13-564139-X 1st edition

British Library Cataloguing in Publication Data
Ben-Ari, M., 1948-
Mathematical logic for computer science. - 2nd rev. ed.
1.Logic, Symbolic and mathematical
I.Title
511.3
ISBN 1852333197

Library of Congress Cataloging-in-Publication Data
Ben-Ari., 1948-
Mathematical logic for computer science / Mordechai Ben-Ari.--2nd rev. ed.
p. cm.
Includes bibliographical references and indexes.
ISBN 1-85233-319-7 (acid-free paper)
1. Logic, Symbolic and mathematical. I. Title
QA9 .B3955 2001
511.3--dc21 00-066113

First published 1993

Typesetting: camera-ready by author
Printed and bound at the Athenæum Press Ltd, Gateshead, Tyne and Wear

For Anita

Minä rakastan sinua

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) are used. Prolog implementations of many of the algorithms are given; for a computer science student, the study of a concrete program can reinforce the study of an abstract algorithm. An ideal course in logic would supplement the theory in this book with practical study of logic programming.

My approach can be characterized as *broad*, *elementary* and *rigorous*: I want to cover many topics rigorously, but I have chosen elementary material, generally the minimal subset of a logic that can be profitably studied. Examples are: the predicate calculus without equality, a future fragment of temporal logic and partial rather than total correctness. The student who has learned the results and techniques in these elementary cases will be well-prepared to study the more general systems in advanced courses.

Organization

Chapter 1 is introductory and surveys the topics in the book. Appendix A summarizes the elementary set theory that the reader should know, and Appendix B contains a guide to the literature. The rest of the book covers five main topics:

- Propositional calculus (Chapters 2, 3, 4).
- Predicate calculus (Chapters 5, 6).
- Resolution and logic programming (Chapters 7, 8).
- Program specification and verification (Chapters 9, 10).
- Temporal logic (Chapters 11, 12).

The first two topics form the core of classical mathematical logic, though I have augmented them with algorithms and programs, and material of interest to computer scientists. The other three topics are chosen for their specific relevance in modern computer science.

The general progression in each main topic is: syntax and semantics of formulas, semantic tableaux, deductive systems and algorithms. While many modern textbooks are heavily weighted in favor of algorithmic semantic methods, I have tried to give equal time to syntactic deduction. Deduction is still the language of mathematical reasoning and is the only fall-back when semantic methods fail.

The propositional and predicate calculi are the central topics in logic and should be taught in any course. The other three topics are independent of each other and the instructor can select or rearrange the topics as desired. Sections marked * contain advanced material that may be skipped in lower-level undergraduate classes, as well as interesting results presented without proof that are beyond the scope of this book. Sections marked ^{P} contain the Prolog programs. The printed programs are only fragments; the full source code of the programs (including routines for input-output and testing) is available online (http://www.springer.co.uk/com_pubs/ct_mlcs.htm). The web site will also include answers to exercises. The programs have been run on the free implementation SWI Prolog (http://www.swi.psy.uva.nl/projects/SWI-Prolog/), but should run on any implementation of the Prolog standard.

Second edition

The second edition has been totally rewritten. Major additions are sections on binary decision diagrams, constraint logic programming and the completeness of Hoare logic. One curiosity: I had used Fermat's Last Theorem as an example of a formula whose truth is not known. Since then, Andrew Wiles has proved the theorem (Singh 1997)! I have chosen to replace it by Goldbach's conjecture.

Notation

'If and only if' is abbreviated iff. Definitions by convention use iff to emphasize that the definition is restrictive. Consider the definition: a natural number is even iff it can be expressed as $2k$ for some natural number k. In this context, iff means that numbers expressible 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 proof is denoted by ▮ and the end of an example or definition is denoted by □.

Acknowledgements

I am very grateful to Mark Ryan and Anna LaBella for their merciless comments on the manuscript. Amir Pnueli helped bring me up to date on temporal logic, as did Ehud Shapiro on logic programming. I appreciate receiving corrections to the first edition from Arnon Avron, Anna LaBella and Rüdiger Reischuk.

I would like to thank Beverley Ford and her team at Springer-Verlag London for their enthusiasm and support during the past two years. Rebecca Mowat deserves a special citation for expertly fielding dozens of email messages.

Finally, the students in my logic class at the Weizmann Institute of Science (in particular Dana Fisman) were extremely helpful in debugging a preliminary draft.

M. Ben-Ari
Rehovot, 2000

Contents

1 Introduction

1.1 The origins of mathematical logic

The study of logic was begun by the ancient Greeks whose educational system stressed competence in philosophy and rhetoric. Logic was used to formalize *deduction*: the derivation of true statements, called *conclusions*, from statements that are assumed to be true, called *premises*. Rhetoric, the art of public speaking, included the study of logic so that all sides in a debate would use the same rules of deduction.

Rules of logic were classified and named. A famous rule is the *syllogism*:

> **Premise** All men are mortal.
> **Premise** X is a man.
> **Conclusion** Therefore, X is mortal.

If we assume the truth of the premises, the syllogism rule claims that the conclusion is true, whatever the identity of X. In particular, if X is a specific man such as Socrates, we can *deduce* that Socrates is mortal.

Natural language is not precise, so the careless use of logic can lead to claims that false statements are true, or to claims that a statement is true, even though its truth does not necessarily follow from the premises. A clever example is the following 'syllogism' given by Smullyan:

> **Premise** Some cars rattle.
> **Premise** My car is some car.
> **Conclusion** Therefore, my car rattles.

We still use many Greek words in logic such as *axiom* and *theorem*, but until the nineteenth century, logic remained a philosophical, rather than a mathematical and scientific, tool, perhaps because it lacked a sufficiently developed symbolic notation. Familiarity with logic is unfortunately no longer required in our educational system.

Mathematicians revived the study of logic in order to study the foundations of math-

the legitimacy of the entire deductive process used to prove theorems in mathematics. Mathematical deduction can be justified by formalizing a system of logic in which the set of provable statements is the same as the set of true statements. In other words, (i) every statement that can be proved is true, and (ii) if a statement is in fact true, there is a proof somewhere out there just waiting for a bright mathematician to discover it.

The research spurred by this plan, called *Hilbert's program*, resulted in the development, not just of systems of logic, but also of theories of the nature of logic itself. Ironically, Hilbert's hopes were dashed when Gödel showed that there are true statements of arithmetic that are not provable.

While mathematical logic remains an important branch of pure mathematics, it is being extensively applied in computer science. In turn, the application of logic to computer science has spurred the development of new systems of logic. The situation is similar to the cross-fertilization between continuous mathematics (calculus and differential equations) and applications in the physical sciences. The remainder of this chapter gives an overview of the theoretical topics and the applications that will be presented in this book. Examples of logical formulas will be given, though you are not expected to understand them at this stage.

1.2 Propositional calculus

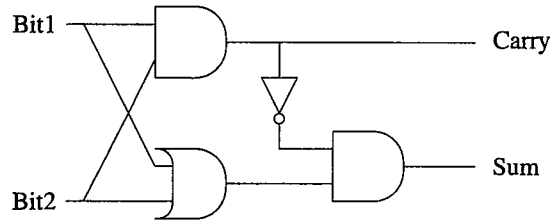
In general, mathematical logic studies two-valued expressions; conventionally, the two values are called *true* and *false* from their origin in the study of philosophy, but this is arbitrary and we could call the values 0 and 1 or even ♣ and ♠. Given any sentence, we assign it a value *true* or *false*. The study of logic commences with the study of the *propositional calculus* whose sentences are built from *atomic propositions*, which are sentences that have no internal structure.

Propositions can be combined using *Boolean operators*. Again, these operators have conventional names derived from natural language such as *and*, *or* and *implies*, but they are given a formal meaning. The Boolean operator *and* is defined as the operator that gives the value *true* if and only if applied to two expressions whose values are *true*. This mimics usage in natural language: since 'One plus one equals two' and 'The earth revolves around the sun' are true statements, 'One plus one equals two *and* the earth revolves around the sun' is also a true statement. Since 'The sun revolves around the earth' is a false statement, so is 'One plus one equals two *and* the sun revolves around the earth'.

Formulas of the propositional calculus are defined by *syntactical rules*, and meaning (*semantics*) is associated with each formula by defining *interpretations* which assign a value *true* or *false* to every formula. Syntax is also used to define the concept of *proof*, the symbolic manipulation of formulas in order to deduce a theorem. The central theoretical result that we prove is that the set of provable formulas is the same as the set of formulas which are always true.

The propositional calculus can be applied to computers because digital computers work with two voltage levels that are arbitrarily assigned the symbols 0 and 1. Circuits are described by idealized elements called *logic gates*. An and-gate produces a certain voltage level called 1 at its output terminal if and only if all its input terminals are held at the same voltage level 1. This is an idealized description because various continuous phenomena (such as rise times and stray capacitance) cannot be ignored, but as a first approximation, logic gates are an extremely useful abstraction. Since gates correspond to Boolean operators, *logic design*—building circuits from gates—can be studied using the propositional calculus.

Example 1.1 Here is a half-adder constructed from and, or- and not-gates.



The following expressions give the relationship between the input and output values:

$$Sum \leftrightarrow \neg\,(Bit1 \wedge Bit2) \wedge (Bit1 \vee Bit2) \qquad Carry \leftrightarrow Bit1 \wedge Bit2.$$

□

1.3 Predicate calculus

The propositional calculus is not sufficiently expressive for mathematical theories such as arithmetic. An arithmetic expression such as $x > y$ is neither true nor false. Its truth depends on the values of x and y; more formally, the operator $>$ is a function from pairs of integers (or real numbers) to the set of Boolean values $\{true, false\}$. The system of logic that includes functions from domains such as numbers to Boolean values is called the *predicate calculus* or *first-order logic*. The predicate calculus is sufficient for most applications of logic to mathematics, such as formalizing arithmetic and algebra. Similarly, most applications of logic to computer science use either the predicate calculus or a system of logic that can be formulated within the predicate calculus.

An extremely important use of the predicate calculus is to formalize the semantics of programming languages and to specify and verify programs. First let us note that the syntax of a programming language is specified by a *grammar*, a set of rules for constructing syntactically legal programs. The properties of grammars are studied in the subject called *formal languages*.

Example 1.2 An `if`-statement in Pascal is described by the grammar rule:

if-statement ::= `if` *expression* `then` *statement* [`else` *statement*]

which says that an `if`-statement consists of the keyword `if`, followed by an expression (such as `x>=0`), followed by the keyword `then` and another statement and finally an optional `else`-clause. □

Since programs perform computation on domains such as numbers or strings, the predicate calculus is used to formalize the semantics of a program.

Example 1.3 Given the statement

```
if x >= 0 then y := sqrt(x) else y := abs(x),
```

we can give a formula of the predicate calculus that relates x' and y' (the values of `x` and `y` after the execution of the statement) to x and y (the values of `x` and `y` before the execution of the statement):

$$\forall x \forall y (x' = x \wedge (x >= 0 \rightarrow y' = \sqrt{x}) \wedge (\neg (x >= 0) \rightarrow y' = |x|)).$$

The formula $x' = x$ specifies that the value of `x` does not change during the execution of the statement. □

Mathematical logic is also used to write a formal *specification* of the execution of a program and then to *verify* programs, that is, to prove the correctness of a program relative to the specification.

Example 1.4 Here is a Pascal program `P` which computes the greatest common denominator of two non-negative integers.

```
while a <> b do
  if a > b then a := a - b else a := b - a;
```

The specification of the program is given by the formula

$$\{a \geq 0 \wedge b \geq 0\}\ \mathtt{P}\ \{a = \gcd(a, b)\},$$

read

> If the initial values of `a` and `b` are such that $a \geq 0$ and $b \geq 0$, and if the program terminates, then the final value of `a` is $\gcd(a, b)$.

We will show how to verify the correctness of the program by proving this formula using the formal semantics of Pascal, a deduction system for proving programs and the theory of arithmetic. □

1.4 Theorem proving and logic programming

We are fascinated by the possibility of a computer imitating human thought processes, or at least imitating intelligent human behavior. This field of study is called *artificial intelligence*. Can a computer 'do' mathematics, that is, can a computer discover proofs of mathematical theorems? Given a set of axioms for, say, number theory, can we turn the computer loose and return a week later to find proofs of all the known theorems, as well as statements and proofs of new interesting 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:

$$4 = 2 + 2,\ 6 = 3 + 3,\ \ldots,\ 100 = 3 + 97,\ 102 = 51 + 51,\ 104 = 3 + 101,\ \ldots.$$

It is not know if Goldbach's Conjecture is true or not, though no even number has been found which is not the sum of two prime numbers.

Automated theorem provers have been developed; they have even discovered new theorems, though usually with the interactive assistance of a mathematician. Research into automated theorem proving led to a new and efficient method of proving formulas in the predicate calculus called *resolution*, which is relatively easy to implement on a computer. More importantly, certain variants 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, that is, $A'(I) \leq A'(J)$ for $I < J$.

Suppose further that given any specific array A, the theorem prover happens to actually construct the array A'. Then the formula is, in effect, a *program* for sorting.

The use of theorem provers for computation is called *logic programming*. Rather than program the computational steps needed to solve a problem, you 'simply' write a logical formula that describes the relation between the input and the output, and then let the theorem prover search for the answer. Logic programming is *descriptive* or *non-procedural*, as opposed to programming with languages like Pascal which are *operational* or *procedural*.

The most widespread logic programming language is called Prolog. It is expressive enough to execute non-procedural programs such as the sort program given above, and yet also contains enough compromises with the real world to allow it to execute many programs efficiently. Non-procedural programming improves the reliability of software by narrowing the gap between the specification of the program and its implementation.

1.5 Systems of logic

First-order predicate logic is the language of most of mathematics. Nevertheless, other systems of logic have been studied, some for philosophical reasons, and others because of their importance in applications, including computer science. This section surveys some of these systems.

As computer scientists, we know that everything can be encoded in bits and this justifies the restriction to Boolean (two-valued) logic. Occasionally it is convenient to be able to directly refer to three or more discrete values. For example, a logic gate may be in an undetermined state before it settles into a stable voltage level. This can be formalized in a *three-valued logic* with a value X in addition to *true* and *false*. The definition of the operators has to be extended for the new values, for example, $X \text{ and } true = X$.

The philosophy behind *intuitionistic logic* is appealing to a computer scientist. For an intuitionist, a mathematical object (such as the solution of an equation) does not exist unless a finite construction (algorithm) can be given for that object. In terms of propositional logic, this means rejecting commonly used methods of reasoning such as the law of the excluded middle: Any proposition is either true or false.

Example 1.5 Let G be the statement of Goldbach's Conjecture. An intuitionist would not accept the truth of the following statement: The proposition G is either true or false. We can construct neither a proof of G nor a counterexample of an even number which cannot be expressed as the sum of two primes. □

Much of standard mathematics can be done within the framework of intuitionistic logic, but the task is very difficult, so almost all mathematicians use methods of the ordinary predicate calculus.

Sometimes, the predicate calculus is adequate but clumsy to use.

Example 1.6 Consider the two statements, '$1 < 2$' and 'It is raining'. The first statement is *always* true, whereas the second one is *sometimes* true. These can be expressed in the predicate calculus as: 'For all times t, the value of "$1 < 2$" at time t is *true*', and 'For some times t, the value of "It is raining" at time t is *true*'. □

Rather than endlessly repeat the dependence of a statement on the time variable, *temporal logic* implicitly introduces time by defining concepts such as *always* (denoted $\Box$) and *eventually* (denoted $\Diamond$) as primitive concepts in the logic. Temporal logic and its close cousin *modal logic* are used in computer science to describe the dynamic behavior of a circuit element or program. In particular, it is extensively used to formulate properties of *reactive* programs like operating systems and real-time systems, which do not compute an 'answer', but instead are intended to run indefinitely while exhibiting correct dynamic behavior in response to external stimuli.

Example 1.7 An operating system is a reactive system. Some temporal properties that we would like to prove are: $\Box \neg$ *deadlock*, the system will *never* (= *always not*) deadlock, and *request* $\rightarrow \Diamond$*print*, if you request the printing of a file, *eventually* it will be printed. □

1.6 Exercise

1. What is wrong with Smullyan's 'syllogism'?

2 Propositional Calculus: Formulas, Models, Tableaux

2.1 Boolean operators

A data type consists of a set of values and a set of predefined operators on those values. For example, in integer arithmetic the values are $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ and the operators are $\{+, -, *, /\}$. The selection of these operators is arbitrary in the sense that other operators such as mod and abs could be added to the set. The definition of the operators is not arbitrary because these four are interesting, and suffice for defining and proving theorems in arithmetic and for manipulating arithmetic expressions in practice. It would be possible to reduce the number of operators by defining multiplication and division as repeated addition and subtraction, respectively, but convention and convenience dictate this choice of operators.

The propositional calculus is concerned with expressions over the *Boolean* type which has two values denoted T and F. Since the set of Boolean values is finite, the number of possible n-place operators is finite for each n. There are 2^{2^n} n-place Boolean operators $op(x_1, \ldots, x_n)$, 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 $\circ_n(x)$.

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

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 called *negation* and is denoted by $\neg$, read *not*. Negation appears in all applications of logic: in electronics, not-gates are used to transform a signal between two discrete voltage levels, and in programming, the

`not` operator is used explicitly in the operator not-equal (denoted `!=`, `<>`, or `/=`), and implicitly in `if`-statements where the `else` part is executed if the condition is *not* true.

There are $2^{2^2} = 16$ two-place operators:

x_1	x_2	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_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	o_9	o_{10}	o_{11}	o_{12}	o_{13}	o_{14}	o_{15}	o_{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

Several of the operators are trivial: o_1 and o_{16} are constant; o_4 and o_6 are projection operators, that is, their value is determined only by the value of one operand; o_{11} and o_{13} are the negations of the projection operators.

The interesting operators and their negations are shown in the following table:

op	name	symbol	op	name	symbol
o_2	disjunction	$\vee$	o_{15}	nor	$\downarrow$
o_8	conjunction	$\wedge$	o_9	nand	$\uparrow$
o_5	implication	$\rightarrow$	o_{12}		
o_3	reverse implication	$\leftarrow$	o_{14}		
o_7	equivalence	$\leftrightarrow$	o_{10}	exclusive or	$\oplus$

o_{10}, the negation of equivalence, is called *non-equivalence* in logic, but in computer science it is called *exclusive or*. For reference, we extract the definitions of the most common Boolean operators:

x	y	$\wedge$	$\vee$	$\rightarrow$	$\leftrightarrow$	$\oplus$	$\uparrow$	$\downarrow$
T	T	T	T	T	T	F	F	F
T	F	F	T	F	F	T	T	F
F	T	F	T	T	F	T	T	F
F	F	F	F	T	T	F	T	T

The familiar readings of the names of the operators can cause confusion. Disjunction is 'inclusive or' as distinct from 'exclusive or'. We say sentences like

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

where 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 sentences are true. In both versions, it is sufficient for one sentence to be true for the compound sentence to be true. Thus, the following strange sentence is true because the truth of the first clause by itself is sufficient to ensure the truth of the sentence.

> The earth revolves around the sun *or* $1 + 1 = 3$.

The operator of $p \rightarrow q$ is called *material implication*, p is called the *antecedent* and q is called the *consequent*. Material implication does not contain an element of causation; it merely states that if the antecedent is true, so is the consequent. Thus it can be falsified only if the antecedent is true and the consequent false.

> The earth revolves around the sun *implies* that $1 + 1 = 3$.

is false, as expected, but

> The sun revolves around the earth *implies* that $1 + 1 = 3$.

is true, because the falsity of the antecedent by itself is sufficient to ensure the truth of the sentence. Confusion can be avoided by referring back to the definition of these operators rather than translating to natural language.

We will see later that this set of operators is highly redundant and that the first five binary operators can all be defined in terms of any one of them plus negation, and either nand or nor by itself is sufficient to define all other operators. The choice of an interesting set of operators depends on the application. Mathematics is generally interested in one-way logical deductions (given a set of axioms, what do they imply?), so implication together with negation are chosen as the basic operators and the others are defined from them.

In digital hardware design, the logic 'gates' commonly implemented are negation and the symmetric operators conjunction and disjunction. Their negations, nand and nor, are also important because they allow not gates to be absorbed into adjacent gates, that is, the two gates needed to implement $\neg (p \wedge q)$ can be replaced by the single gate $p \uparrow q$. Also, since nand and nor are sufficient to define all other gates, a single type of component can be used to construct all possible logic circuits.

The binary operators $\leftrightarrow$ and $\oplus$ are their own inverses, that is, $((p \leftrightarrow q) \leftrightarrow q)$ and $((p \oplus q) \oplus q)$ are the same as p. (They are equivalent—a term to be defined later.) This property is used in circuits and programs that implement encoding for error-correcting and cryptography. A coded message is created by performing an exclusive-or operation between the data and the code. If *CodedMessage* is *Data* $\oplus$ *Code*, then *Data* can be retrieved by performing an exclusive-or operation on *CodedMessage* and *Code*.

$$CodedMessage \oplus Code = (Data \oplus Code) \oplus Code = Data.$$

Notation

Unfortunately, it is rare to find two books on mathematical logic that use the same notation. To increase confusion, programming languages use a different notation from mathematics textbooks. Here are some alternate notations that you are likely to find:

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

In Section 2.4, we introduce $\equiv$ to denote the metalogical concept of logical equivalence which must not be confused with the Boolean operator $\leftrightarrow$. Be careful, as some books make the opposite choice of symbols for these two concepts.

2.2 Propositional formulas

In computer science, the term expression is used to denote the construction of a complex value from elementary values. In the propositional calculus, the term *propositional formula* (shortened to *formula* if the context is clear) is used instead. The syntactically correct formulas are described by giving a context-free grammar similar to the BNF grammars used to describe programming languages.

Grammar rules of the form

$$symbol \quad ::= \quad symbol_1 \cdots symbol_N$$

mean that *symbol* may be replaced by that sequence of N symbols. Rules of the form

$$symbol \quad ::= \quad symbol_1 \mid \cdots \mid symbol_N$$

mean that *symbol* may be replaced by one of the symbols on the right-hand side of the rule. Symbols that occur on the left-hand side of a rule are called *non-terminals* and represent grammatical classes; symbols that never occur on the left-hand side are called *terminals* and represent the symbols of the language.

A word in a language is obtained from a *derivation* that starts with an initial non-terminal. Repeatedly, choose a non-terminal and a rule with that non-terminal on its left-hand side, and replace it with the right-hand side of the rule as described above. The construction terminates when the sequence of symbols consists only of terminals.

1.	*fml*	Initial non-terminal
2.	$fml \leftrightarrow fml$	Rule 6
3.	$fml \rightarrow fml \leftrightarrow fml$	Rule 5
4.	$p \rightarrow fml \leftrightarrow fml$	Rule 1
5.	$p \rightarrow q \leftrightarrow fml$	Rule 1
6.	$p \rightarrow q \leftrightarrow fml \rightarrow fml$	Rule 5
7.	$p \rightarrow q \leftrightarrow \neg fml \rightarrow fml$	Rule 2
8.	$p \rightarrow q \leftrightarrow \neg p \rightarrow fml$	Rule 1
9.	$p \rightarrow q \leftrightarrow \neg p \rightarrow \neg fml$	Rule 2
10.	$p \rightarrow q \leftrightarrow \neg p \rightarrow \neg q$	Rule 1

Figure 2.1 Derivation of $p \rightarrow q \leftrightarrow \neg p \rightarrow \neg q$

The sequence of symbols is called a *syntactically correct* or *well-formed word* in the language defined by the grammar.

In the propositional calculus the terminals are the Boolean operators and an unbounded set of arbitrary symbols $\mathcal{P}$ called *propositional letters* or *atomic propositions* (shortened to *atoms*). Atoms will be denoted by lower case letters in the set $\{p, q, r, \ldots\}$, possibly with subscripts.

Definition 2.1 A *formula* in the propositional calculus is a word that can be derived from the following grammar, starting from the initial non-terminal *fml*.

1.	*fml*	::=	p	for any $p \in \mathcal{P}$
2.	*fml*	::=	$\neg$ *fml*	
3.	*fml*	::=	*fml* $\vee$ *fml*	
4.	*fml*	::=	*fml* $\wedge$ *fml*	
5.	*fml*	::=	*fml* $\rightarrow$ *fml*	
6.	*fml*	::=	*fml* $\leftrightarrow$ *fml*	
7.	*fml*	::=	*fml* $\oplus$ *fml*	
8.	*fml*	::=	*fml* $\uparrow$ *fml*	
9.	*fml*	::=	*fml* $\downarrow$ *fml*.	

The set of formulas that can be derived from this grammar is denoted $\mathcal{F}$. □

Each derivation of a formula from a grammar can be represented by a *derivation tree* (Hopcroft & Ullman 1979, Section 4.3) that displays the application of the grammar rules to the non-terminals.

Example 2.2 The derivation of the formula $p \rightarrow q \leftrightarrow \neg p \rightarrow \neg q$ is given in Figure 2.1; its derivation tree is displayed in Figure 2.2. □

From the derivation tree we obtain a *formation tree* (Figure 2.3) for the derived formula by replacing an *fml* non-terminal by the child that is an operator or an atom. We

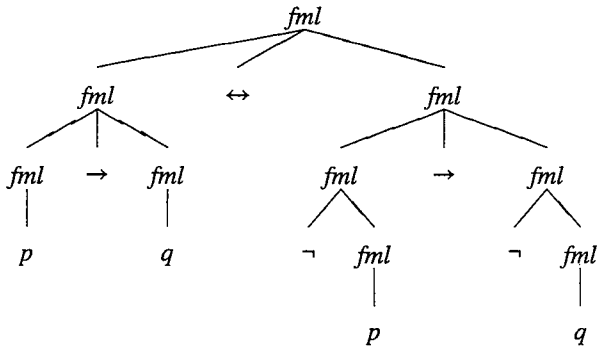


Figure 2.2 Derivation tree for $p \to q \leftrightarrow \neg p \to \neg q$

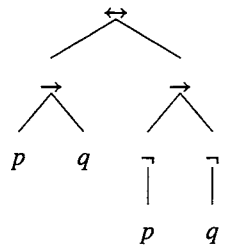


Figure 2.3 Formation tree for $p \to q \leftrightarrow \neg p \to \neg q$

leave it as an exercise to show that a unique formation tree is associated with each derivation tree.

The formula—the word in the language generated by the grammar—can be read left to right off the leaves of the derivation tree, or by an inorder traversal of the formation tree: visit the left subtree, visit the root, visit the right subtree. (For a node labeled by negation, the left subtree is considered to be empty.) The formula represented by this tree is:

$$p \to q \leftrightarrow \neg p \to \neg q.$$

Unfortunately, this linear sequence of symbols is also obtained from the formation tree shown in Figure 2.4 which represents an entirely different formula.

In other words, the linear representation of formulas is ambiguous, even though the formation trees are not. Since we prefer to deal with linear sequences of symbols, we need some way to resolve ambiguities if we are to use a linear representation of a formula independent of its derivation. There are three ways of doing this.

There will be no ambiguity if the linear sequence of symbols is created by a preorder traversal of the formation tree: visit the root, visit the left subtree, visit the right sub-

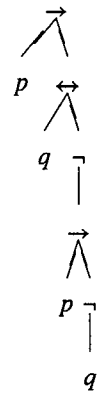


Figure 2.4 Another formation tree

tree. The linear representations of the two formulas are

$$\leftrightarrow \to pq \to \neg p \neg q$$

for the first tree and

$$\to p \leftrightarrow q \neg \to p \neg q$$

for the second tree, and there is no longer any ambiguity. The formulas are said to be in *Polish notation*, named after the group of Polish logicians led by J. Łukasiewicz. The notation is difficult for most of us to read because the operators can be very far from the operands; furthermore, infix notation is easier for us to mentally parse. Polish notation is used in the internal representation of an expression in a computer and in the operation of some calculators. The advantage of Polish notation is that the expression can be executed or calculated in the linear order the symbols appear. If we rewrite the first formula from backwards:

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

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

```
Load q
Negate
Load p
Negate
Imply
Load q
Load p
Imply
Equiv
```

The second way of resolving ambiguity is to use parentheses. The grammar would be changed to:

$$\begin{array}{llll} 1. & \mathit{fml} & ::= & p \qquad \text{for any } p \in \mathcal{P} \\ 2. & \mathit{fml} & ::= & (\neg \mathit{fml}) \\ 3. & \mathit{fml} & ::= & (\mathit{fml} \vee \mathit{fml}) \\ \ldots & & & \end{array}$$

The two formulas are represented by different strings and there is no ambiguity:

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

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

The third 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 $(((a*b)*c)+(d*e))$. In propositional formulas the order of precedence from high to low is as follows: negation, conjunction, nand, disjunction, nor, implication, equivalence. 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 precedence, as in arithmetic where $a*(b+c)$ needs parentheses to denote that the addition is done before the multiplication, while $a*b+c$ does not need them to denote the multiplication before addition. With minimal use of parentheses, the propositional formulas above can be written:

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

Whatever syntax is used for the linear representation of a formula, it should be understood as a shorthand for the unambiguous formation tree.

Theorem 2.3 *Let $A \in \mathcal{F}$ be a formula. Then A is either an atom or it is of the form $\neg A_1$ or A_1 op A_2 for some formulas A_1, A_2 and operator op.*

Proof: Immediate from Definition 2.1. ∎

Definition 2.4 If A is not an atom, the operator at the root of the formation tree for A is called the *principal operator* of the formula. □

Structural induction is used to prove that a property holds for all formulas: first show that it holds for atoms and then show that the property is preserved when formulas are constructed from simpler formulas using the operators. These two steps are called the *base case* and the *induction step* in analogy with mathematical induction used to prove that a property holds for all natural numbers (Appendix A.6).

Theorem 2.5 (Structural induction) *To show property(A) for all formulas $A \in \mathcal{F}$, it suffices to show:*

1. *property(p) for all atoms p.*
2. *Assuming property(A), then property($\neg A$) holds.*
3. *Assuming property(A_1) and property(A_2), then property(A_1 op A_2) hold, for each of the operators op.*

Proof: Suppose that the base case and two inductive steps have been proved and let A be an arbitrary formula. We show *property*(A) by (arithmetical) induction on n, the height of the formation tree for A. If A is a leaf ($n = 0$), then A is an atom p and *property*(p) holds by the base case. Otherwise, the principal operator of A is either $\neg$ or one of the binary operators *op*. The subtrees of the tree for A are of height $n - 1$, so by (arithmetical) induction *property* holds for the formulas labeling them. By the inductive steps, *property*(A) holds. ∎

We will later show that all the binary operators can be defined in terms of any one of them plus negation, so that a structural inductive proof of a property of a formula reduces to the base case and two inductive steps.

2.3 Interpretations

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 the propositional calculus, truth values are assigned to the atoms of a formula in order to evaluate the truth value of the formula.

Definition 2.6 An *assignment* is a function $v : \mathcal{P} \mapsto \{T, F\}$, that is, v assigns one of the *truth values* T or F to every atom.

An assignment v can be extended to a function $v : \mathcal{F} \mapsto \{T, F\}$, mapping formulas to truth values by the inductive definitions in Figure 2.5. v is called an *interpretation*. □

Example 2.7 Let $A = (p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$, and let v the assignment such that $v(p) = F$ and $v(q) = T$, and $v(p_i) = T$ for all other $p_i \in \mathcal{P}$. Extend v to an interpretation. The truth value of A can be calculated inductively using Figure 2.5:

$$v(p \rightarrow q) = T$$
$$v(\neg q) = F$$
$$v(\neg p) = T$$
$$v(\neg q \rightarrow \neg p) = T$$
$$v((p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)) = T.$$

□

A	$v(A_1)$	$v(A_2)$	$v(A)$
$\neg A_1$	T		F
$\neg A_1$	F		T
$A_1 \vee A_2$	F	F	F
$A_1 \vee A_2$	otherwise		T
$A_1 \wedge A_2$	T	T	T
$A_1 \wedge A_2$	otherwise		F
$A_1 \rightarrow A_2$	T	F	F
$A_1 \rightarrow A_2$	otherwise		T

A	$v(A_1)$	$v(A_2)$	$v(A)$
$A_1 \uparrow A_2$	T	T	F
$A_1 \uparrow A_2$	otherwise		T
$A_1 \downarrow A_2$	F	F	T
$A_1 \downarrow A_2$	otherwise		F
$A_1 \leftrightarrow A_2$	$v(A_1) = v(A_2)$		T
$A_1 \leftrightarrow A_2$	$v(A_1) \neq v(A_2)$		F
$A_1 \oplus A_2$	$v(A_1) \neq v(A_2)$		T
$A_1 \oplus A_2$	$v(A_1) = v(A_2)$		F

Figure 2.5 Evaluation of truth values of formulas

Since each formula A is represented by a unique formation tree, v is well-defined, that is, given A and v, $v(A)$ has exactly one value according to the inductive definition.

Example 2.8 $v(p \rightarrow (q \rightarrow p)) = T$ but $v((p \rightarrow q) \rightarrow p) = F$ under the above interpretation, emphasizing that the linear string $p \rightarrow q \rightarrow p$ is ambiguous. □

The use of v to denote both an assignment and its extension to an interpretation is justified by the following theorem whose proof we leave as an exercise.

Theorem 2.9 *An assignment can be extended to exactly one interpretation.*

Furthermore, an assignment need not assign to all possible atoms in $\mathcal{P}$.

Theorem 2.10 *Let $\mathcal{P}' = \{p_{i_1}, \ldots, p_{i_n}\} \subseteq \mathcal{P}$ be the atoms appearing in $A \in \mathcal{F}$. Let v_1 and v_2 be assignments that agree on $\mathcal{P}'$, that is, $v_1(p_{i_k}) = v_2(p_{i_k})$ for all $p_{i_k} \in \mathcal{P}'$. Then the interpretations agree on A, that is, $v_1(A) = v_2(A)$.*

Proof: Exercise. ∎

Definition 2.11 Let $S = \{A_1, \ldots, A_n\}$ be a set of formulas and let v be an assignment that assigns truth values to all atoms that appear in any A_i. Any interpretation obtained by extending v to all atoms in $\mathcal{P}$ is called an *interpretation for* S. □

Example 2.12 Let $S = \{p \rightarrow q,\ p,\ p \vee s \leftrightarrow s \wedge q\}$, and let v be the assignment given by $v(p) = T$, $v(q) = F$, $v(r) = T$, $v(s) = T$. v is an interpretation for S and assigns the truth values

$$v(p \rightarrow q) = F,$$
$$v(p) = T,$$
$$v(p \vee s \leftrightarrow s \wedge q) = F.$$

□

2.4 Logical equivalence and substitution

Definition 2.13 Let $A_1, A_2 \in \mathcal{F}$. If $v(A_1) = v(A_2)$ for all interpretations v, then A_1 is *logically equivalent* to A_2, denoted $A_1 \equiv A_2$. □

Of course, it is sufficient to check the (finite number of) interpretations that assign truth values just to atoms that appear in either formula.

Example 2.14 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:

p	q	$v(p \vee q)$	$v(q \vee p)$
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 theorem can be generalized to show that the disjunction of any two formulas, not just the disjunction of the atoms p and q, is commutative.

Theorem 2.15 *Let A_1 and A_2 be any formulas. Then $A_1 \vee A_2 \equiv A_2 \vee A_1$.*

Proof: Let v be an arbitrary interpretation for $A_1 \vee A_2$, that is, v assigns truth values to the union of the sets of atoms in A_1 and A_2. Obviously, v is also an interpretation for $A_2 \vee A_1$. Furthermore, since the set of atoms in A_1 is a subset of those in $A_1 \vee A_2$, v is an interpretation for A_1 and similarly for A_2. $v(A_1 \vee A_2) = T$ iff either $v(A_1) = T$ or $v(A_2) = T$ iff $v(A_2 \vee A_1) = T$ by definition (Figure 2.5). Since v 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 v be an *arbitrary* interpretation and then write a proof without using any property that distinguishes one interpretation from another.

Is $\equiv$ a Boolean operator? The answer is no. It is simply a notation for the phrase 'is logically equivalent to', whereas $\leftrightarrow$ is a Boolean operator in the propositional calculus. There is potential for confusion when studying logic because we are using a similar vocabulary both for the *object language* under discussion, in this case propositional calculus, and for the semi-formal *metamathematical language* (or *metalanguage*) used to reason about the object language. Similarly, we must distinguish between a propositional formula like $p \vee q$ in the object language and a formula like $A_1 \vee A_2$ in the metalanguage.

Despite this disclaimer that $\equiv$ and $\leftrightarrow$ are not the same, the two concepts are closely related as shown by the following theorem:

Theorem 2.16 *$A_1 \equiv A_2$ if and only if $A_1 \leftrightarrow A_2$ is true in every interpretation.*

Proof: Suppose that $A_1 \equiv A_2$ and let ν be an arbitrary interpretation. Then $\nu(A_1) = \nu(A_2)$ by definition of logical equivalence, and $\nu(A_1 \leftrightarrow A_2) = T$ by the inductive definition in Figure 2.5. Since ν was arbitrary, $\nu(A_1 \leftrightarrow A_2) = T$ in all interpretations. The proof of the converse is similar. ∎

Logical equivalence justifies substitution of one formula for another.

Definition 2.17 A is a *subformula* of B if the formation tree for A occurs as a subtree of the formation tree for B. A is a *proper subformula* of B if A is a subformula of B, but A is not identical to B. □

Example 2.18 The formula $(p \rightarrow q) \leftrightarrow (\neg p \rightarrow \neg q)$ contains the following proper subformulas: $p \rightarrow q$, $\neg p \rightarrow \neg q$, $\neg p$, $\neg q$, p and q. □

Definition 2.19 If A is a subformula of B and A' is any formula, then B', the *substitution of A' for A in B*, denoted $B\{A \leftarrow A'\}$, is the formula obtained by replacing all occurrences of the subtree for A in B by the tree for A'. □

Example 2.20 Let $B = (p \rightarrow q) \leftrightarrow (\neg p \rightarrow \neg q)$, $A = p \rightarrow q$ and $A' = \neg p \vee q$.

$$\begin{aligned} B' &= B\{A \leftarrow A'\} \\ &= B\{A \leftarrow \neg p \vee q\} \\ &= (\neg p \vee q) \leftrightarrow (\neg q \rightarrow \neg p). \end{aligned}$$

□

Theorem 2.21 *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 ν be an arbitrary interpretation. We know that $\nu(A) = \nu(A')$ and we must show that $\nu(B) = \nu(B')$. The proof will be by induction on the depth d of the root of the subtree of the highest occurrence of A in the formation tree of B. If $d = 0$, there is only one occurrence of A, namely B itself, and $A' = B'$. Obviously, $\nu(B) = \nu(A) = \nu(A') = \nu(B')$. If $d \neq 0$, then B must be $\neg B_1$, 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, $\nu(B_1) = \nu(B_1') = B_1\{A \leftarrow A'\})$, and similarly for B_2. By the inductive definition of ν on the Boolean operators, $\nu(B) = \nu(B')$. ∎

Logically equivalent formulas

Figure 2.6 contains a list of logical equivalences. Their proofs are elementary from the definitions and are left as exercises.

$A \vee \mathit{true} \equiv \mathit{true}$	$A \wedge \mathit{true} \equiv A$
$A \vee \mathit{false} \equiv A$	$A \wedge \mathit{false} \equiv \mathit{false}$
$A \rightarrow \mathit{true} \equiv \mathit{true}$	$\mathit{true} \rightarrow A \equiv A$
$A \rightarrow \mathit{false} \equiv \neg A$	$\mathit{false} \rightarrow A \equiv \mathit{true}$
$A \leftrightarrow \mathit{true} \equiv A$	$A \oplus \mathit{true} \equiv \neg A$
$A \leftrightarrow \mathit{false} \equiv \neg A$	$A \oplus \mathit{false} \equiv A$
$A \equiv \neg\neg A$	
$A \equiv A \wedge A$	$A \equiv A \vee A$
$A \vee \neg A \equiv \mathit{true}$	$A \wedge \neg A \equiv \mathit{false}$
$A \rightarrow A \equiv \mathit{true}$	
$A \leftrightarrow A \equiv \mathit{true}$	$A \oplus A \equiv \mathit{false}$
$\neg A \equiv A \uparrow A$	$\neg A \equiv A \downarrow A$
$A \vee B \equiv B \vee A$	$A \wedge B \equiv B \wedge A$
$A \leftrightarrow B \equiv B \leftrightarrow A$	$A \oplus B \equiv B \oplus A$
$A \uparrow B \equiv B \uparrow A$	$A \downarrow B \equiv B \downarrow A$
$A \rightarrow B \equiv \neg B \rightarrow \neg A$	
$A \vee (B \vee C) \equiv (A \vee B) \vee C$	$A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$
$A \leftrightarrow (B \leftrightarrow C) \equiv (A \leftrightarrow B) \leftrightarrow C$	$A \oplus (B \oplus C) \equiv (A \oplus B) \oplus C$
$A \uparrow (B \uparrow C) \equiv (A \uparrow B) \uparrow C$	$A \downarrow (B \downarrow C) \equiv (A \downarrow B) \downarrow C$
$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$	$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
$A \wedge (A \vee B) \equiv A$	$A \vee (A \wedge B) \equiv A$
$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$	$A \oplus B \equiv \neg(A \rightarrow B) \vee \neg(B \rightarrow A)$
$A \rightarrow B \equiv \neg A \vee B$	$A \rightarrow B \equiv \neg(A \wedge \neg B)$
$A \vee B \equiv \neg(\neg A \wedge \neg B)$	$A \wedge B \equiv \neg(\neg A \vee \neg B)$
$A \vee B \equiv \neg A \rightarrow B$	$A \wedge B \equiv \neg(A \rightarrow \neg B)$
$A \rightarrow B \equiv A \leftrightarrow (A \wedge B)$	$A \rightarrow B \equiv B \leftrightarrow (A \vee B)$
$A \wedge B \equiv (A \leftrightarrow B) \leftrightarrow (A \vee B)$	$A \leftrightarrow B \equiv (A \vee B) \rightarrow (A \wedge B)$

Figure 2.6 Logical equivalences

We have extended the syntax of Boolean formulas to include the two constant atomic propositions *true* and *false*.

$$formula \quad ::= \quad true \mid false,$$

interpreted as $v(true) = T$ and $v(false) = F$ for any v. Alternatively, it is possible to regard *true* and *false* as abbreviations for the formulas $p \vee \neg p$ and $p \wedge \neg p$, respectively. Do not confuse these symbols in the object language of the propositional calculus with the truth values T and F used to define interpretations.

Simplification is one of the most important applications of substitution of logical equivalences. Given a formula A, substitute for subformulas until a simpler formula is obtained:

$$\begin{array}{ll} p \wedge (\neg p \vee q) & \equiv \\ (p \wedge \neg p) \vee (p \wedge q) & \equiv \\ false \vee (p \wedge q) & \equiv \\ p \wedge q. & \end{array}$$

Many of the equivalences describe familiar mathematical properties of Boolean operators. Except for $\rightarrow$, they are all associative and commutative, so we can freely omit parentheses and rearrange sequences of these operators. $\vee$ and $\wedge$ are *idempotent*, that is, they collapse identical operands:

$$A \vee A \equiv A \qquad A \wedge A \equiv A,$$

so we can freely insert or remove additional copies of a subformula. $\uparrow$ and $\downarrow$ also collapse copies of identical operands but introduce a negation:

$$A \uparrow A \equiv \neg A \qquad A \downarrow A \equiv \neg A.$$

Finally, equivalence operators erase identical operands:

$$A \leftrightarrow A \equiv true \qquad A \oplus A \equiv false.$$

Definition 2.22 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 $\{\circ_1, \ldots, \circ_n\}$. Similarly, the (only nontrivial) 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. □

Equivalence can be defined from implication and conjunction:

$$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A),$$

and implication can be defined from negation and either conjunction or disjunction:

$$A \rightarrow B \equiv \neg A \vee B \qquad A \rightarrow B \equiv \neg (A \wedge \neg B).$$

Similarly, there are formulas for defining disjunction and conjunction from implication and negation, and formulas for defining disjunction from conjunction and negation, and conversely. These formulas are called *De Morgan's laws*:

$$A \wedge B \equiv \neg (\neg A \vee \neg B) \qquad A \vee B \equiv \neg (\neg A \wedge \neg B).$$

From the above discussion, it is clear that all unary and binary Boolean operators can be defined from negation and one of disjunction, conjunction or implication. It may come as a surprise that it is possible to define all Boolean operators from either nand or nor alone. The formula $\neg A \equiv A \uparrow A$ is used to define negation from nand and the following sequence of equivalences shows how conjunction can be defined:

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

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

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

Suppose that $\circ$ can define all other operators. In particular, negation must be defined by a formula $\neg A \equiv A \circ \cdots \circ A$, for some number of applications of $\circ$, and for any binary operator *op*,

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

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 v be any interpretation that assigns T to A. Then

$$F = v(\neg A) = v(A \circ \cdots \circ A).$$

Prove by induction on the number of occurrences of $\circ$ that in the definition of $\circ$, $v(A_1 \circ A_2) = F$ when $v(A_1) = T$ and $v(A_2) = T$. Similarly, let v assign F to A, so that $v(\neg A) = T$, and prove that $v(A_1 \circ A_2) = T$ when $v(A_1) = F$ and $v(A_2) = F$.

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?
		F?
F	T	T?
		F?
F	F	T

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

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

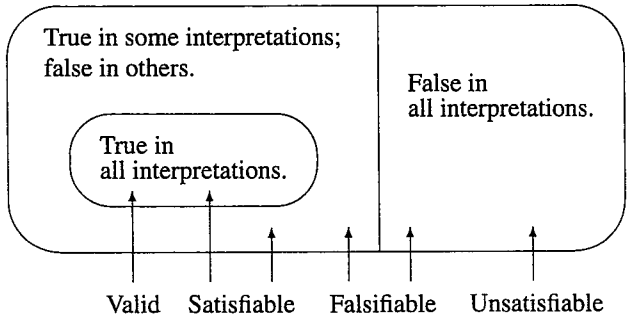
for some i and zero or more negations. ▮

2.5 Satisfiability, validity and consequence

Definition 2.24 A propositional formula A is *satisfiable* iff $v(A) = T$ for *some* interpretation v. A satisfying interpretation is called a *model* for A. A is *valid*, denoted $\models A$, iff $v(A) = T$ for *all* interpretations v. A valid propositional formula is also called a *tautology*.

A propositional formula A is *unsatisfiable* or *contradictory*, iff it is not satisfiable, that is, if $v(A) = F$ for *all* interpretations v. A is *not-valid* or *falsifiable*, denoted $\not\models A$, iff it is not valid, that is, if $v(A) = F$ for *some* interpretation v. □

The relationship among these concepts is shown in the following diagram.



Theorem 2.25 *A is valid if and only if $\neg A$ is unsatisfiable. A is satisfiable if and only if $\neg A$ is falsifiable.*

Proof: Let v be an *arbitrary* interpretation. $v(A) = T$ if and only if $v(\neg A) = F$ by the definition of interpretation for negation. Since v is 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. Similarly for satisfiability, if v is *some* interpretation such that $v(A) = T$, then $v(\neg A) = F$ in this interpretation; conversely, if $v(\neg A) = F$ then $v(A) = T$. ▮

Validity and unsatisfiability are duals: to prove a theorem 'A is valid', it is sufficient to prove the theorem '$\neg A$ is unsatisfiable'.

Definition 2.26 Let $\mathcal{U}$ be a set of formulas. An algorithm is a *decision procedure* for $\mathcal{U}$ if given an arbitrary formula $A \in \mathcal{F}$, it terminates and returns the answer 'yes' if $A \in \mathcal{U}$ and the answer 'no' if $A \notin \mathcal{U}$. □

By Theorem 2.25, 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 are usually more efficient, 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 the propositional calculus is trivial. Since any formula contains a finite number of atoms, there are a finite number of different interpretations (Theorem 2.10) and we can check them all. This algorithm is called the *method of truth tables* because the computation can be arranged in tabular form, one row for each assignment to the atoms of the formula. There is a column for each atom and a column with the truth value of the formula. It is convenient to have additional columns for subformulas to assist in the computation.

Example 2.27 The formula $(p \to q) \to (\neg q \to \neg p)$ is valid because each line of its truth table evaluates to T:

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

□

Example 2.28 The formula $p \wedge q$ is satisfiable but not valid because its truth table contains a line that evaluates to T as well as lines that evaluate to F.

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

□

Example 2.29 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 algorithm because we evaluate every formula for each of the 2^n possible interpretations, where n is the number of distinct atoms in the formula. In the following chapters we will discuss more efficient algorithms for satisfiability, though it is extremely unlikely that there is an algorithm that is always efficient (see Section 4.4).

Definition 2.30 A set of formulas $U = \{A_1, \ldots, A_n\}$ is *(simultaneously) satisfiable* iff there exists an interpretation v such that $v(A_1) = \cdots = v(A_n) = T$. The satisfying interpretation is called a *model* of U. U is *unsatisfiable* iff for every interpretation v, there exists an i such that $v(A_i) = F$. □

Example 2.31 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. Note that 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. In all the theorems, let $U = \{A_1, \ldots, A_n\}$.

Theorem 2.32 *If U is satisfiable, then so is $U - \{A_i\}$ for any $1 \leq i \leq n$.*

Theorem 2.33 *If U is satisfiable and B is valid, then $U \cup \{B\}$ is satisfiable.*

Theorem 2.34 *If U is unsatisfiable, then for any formula B, $U \cup \{B\}$ is unsatisfiable.*

Theorem 2.35 *If U is unsatisfiable and for some $1 \leq i \leq n$, A_i is valid, then $U - \{A_i\}$ is unsatisfiable.*

Definition 2.36 Let U be a set of formulas and A a formula. If A is true in every model of U, then A is a *logical consequence* of U. Notation: $U \models A$. □

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 2.37 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 such that $v(p) = T$ and $v(q) = F$. However, A is not valid, since it is not true in the interpretation $v(p) = F$, $v(q) = T$, $v(r) = T$. □

The caveat concerning $\leftrightarrow$ and $\equiv$ also applies to $\rightarrow$ and $\models$. $\rightarrow$ is an operator in the object language and $\models$ is a symbol for a concept in the metalanguage. As before, the two concepts are connected:

Theorem 2.38 *$U \models A$ if and only if $\models A_1 \wedge \cdots \wedge A_n \rightarrow A$.*

The proofs of this theorem as well as the following two are left as exercises.

Theorem 2.39 *If $U \models A$ then $U \cup \{B\} \models A$ for any formula B.*

Theorem 2.40 *If $U \models A$ and B is valid then $U - \{B\} \models A$.*

Theories

Logical consequence is the central concept in the foundations of mathematics. Valid formulas such as $p \vee q \leftrightarrow q \vee p$ are of minor mathematical interest since they are self-evident. 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 2.41 A set of formulas $\mathcal{T}$ is a *theory* iff it is *closed* under logical consequence. $\mathcal{T}$ is closed under logical consequence iff for all formulas A, if $\mathcal{T} \models A$ then $A \in \mathcal{T}$. The elements of $\mathcal{T}$ are called *theorems*.

Let U be a set of formulas. $\mathcal{T}(U) = \{A \mid U \models A\}$ is the called the *theory* of U. The formulas of U are called *axioms* and the theory $\mathcal{T}(U)$ is *axiomatizable*. □

We leave it as an exercise to show that $\mathcal{T}(U)$ is in fact a theory, that is, that it is closed under logical consequence.

Implementation[P]

Though the method of truth tables is easy to implement by hand, we give a Prolog program for the method as a first example of the implementation of an algorithm.

The Boolean operators $\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \oplus$ will be represented in the Prolog programs by new operators `neg`, `or`, `and`, `imp`, `eqv` and `xor`. For reference, we give the declarations of the operators in Prolog, though they will be of interest only to Prolog experts:

```
:- op(650, xfy, xor).
:- op(650, xfy, eqv).
:- op(640, xfy, imp).
:- op(630, xfy, or).
:- op(620, xfy, and).
:- op(610, fy,  neg).
```

For brevity, we will not use the operators `nand` and `nor` in the programs in the text; you can easily add them if you wish.

Formulas written with this notation are not easy to read or write. The source archive contains programs to translate this notation to and from a notation that corresponds more closely to mathematical notation. We have not used it directly in the program because there are clashes with important predefined operators in Prolog.

Example 2.42 The internal and external representations of the formula

$$(p \oplus q) \leftrightarrow (\neg(p \rightarrow q) \vee \neg(q \rightarrow p))$$

are:

```
(p xor q) eqv (neg (p imp q) or neg (q imp p) )
```

and

```
(p + q) <-> ( ~ (p -> q) v ~ (q -> p) ),
```

respectively. □

The predicate `tt(Fml, V, TV)` returns the truth value `TV` of formula `Fml` under the assignment `V`. The assignment is a list of pairs `(A,TV)`, where `A` is an atom and `TV` is `t` or `f`, for example, `[(p,f),(q,t)]`. `tt` recurses on the structure of the formula. For atoms, it returns the truth value by lookup in the list; for negations, `neg` is called to negate the value; for formulas with a binary operator, `opr` is called to compute the truth value from the truth values of the subformulas.


```
tt(A eqv B, V, TV) :-
    tt(A, V, TVA), tt(B, V, TVB), opr(eqv, TVA, TVB, TV).
tt(A xor B, V, TV) :-
    tt(A, V, TVA), tt(B, V, TVB), opr(xor, TVA, TVB, TV).
tt(A imp B, V, TV) :-
    tt(A, V, TVA), tt(B, V, TVB), opr(imp, TVA, TVB, TV).
tt(A or  B, V, TV) :-
    tt(A, V, TVA), tt(B, V, TVB), opr(or,  TVA, TVB, TV).
tt(A and B, V, TV) :-
    tt(A, V, TVA), tt(B, V, TVB), opr(and, TVA, TVB, TV).

tt(neg A, V, TV) :- tt(A, V, TVA), negate(TVA, TV).
tt(A, V, TV)     :- member((A,TV), V).
```

`opr` and `negate` implement the computation of truth values from Figure 2.5.

```
opr(or, t,t,t). opr(or, t,f,t). opr(or, f,t,t). opr(or, f,f,f).
opr(and,t,t,t). opr(and,t,f,f). opr(and,f,t,f). opr(and,f,f,f).
opr(xor,t,t,f). opr(xor,t,f,t). opr(xor,f,t,t). opr(xor,f,f,f).
opr(eqv,t,t,t). opr(eqv,t,f,f). opr(eqv,f,t,f). opr(eqv,f,f,t).
opr(imp,t,t,t). opr(imp,t,f,f). opr(imp,f,t,t). opr(imp,f,f,t).

negate(t,f).
negate(f,t).
```

`eval(Fml)` prints the truth table for `Fml`.

```
eval(Fml) :-
    get_atoms(Fml, Atoms),
    generate(Atoms, V),
    tt(Fml, V, TV),
    write_tt_line(Fml, V, TV),
    fail.
```

`get_atoms(Fml,Atoms)` returns a sorted list of the atoms occurring in `Fml`, and `generate(Atoms,V)` generates assignments for this set of atoms. (The programming of these two procedures requires advanced Prolog techniques; see the source archive.) As each assignment is generated, `tt(Fml, V, TV)` is called, the value of `TV` is printed and then the predicate `fail` causes backtracking into `generate`. Eventually, the entire truth table will be printed.

2.6 Semantic tableaux

The method of *semantic tableaux* is a relatively efficient algorithm for deciding satisfiability (and by duality validity) in the propositional calculus. Furthermore, the method

is important because it will be the main tool for proving general theorems about the calculus. The principle is very simple: search *systematically* for a model. If one is found, the formula is satisfiable; otherwise, it is unsatisfiable. We begin with the definition of some terms, and then analyze the satisfiability of two formulas to motivate the construction of semantic tableaux.

Definition 2.43 A *literal* is an atom or a 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 2.44 Let us analyze the satisfiability of the formula $A = p \wedge (\neg q \vee \neg p)$ in an arbitrary interpretation v, using the inductive rules for the evaluation of the truth value of a formula.

- $v(A) = T$ if and only if both $v(p) = T$ and $v(\neg q \vee \neg p) = T$.
- Hence, $v(A) = T$ if and only if either:
  1. $v(p) = T$ and $v(\neg q) = T$, or
  2. $v(p) = T$ and $v(\neg p) = T$.

Thus, 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 questions about the satisfiability of sets of literals. It is easy to see that a set of literals is satisfiable if and only if it does *not* contain a complementary pair of literals. Since any formula contains a finite number of atoms, there are at most a finite number of sets of literals built from these atoms. It is trivial to decide if the condition holds for any one set. In the example, the second set of literals $\{p, \neg p\}$ is complementary and hence unsatisfiable, but the first set $\{p, \neg q\}$ contains no complementary pair of literals, hence the set is satisfiable and we can conclude that A is also satisfiable. Furthermore, we can trivially construct a model of A by assigning T to positive literals and F to negative literals:

$$v(p) = T,\ v(q) = F.$$

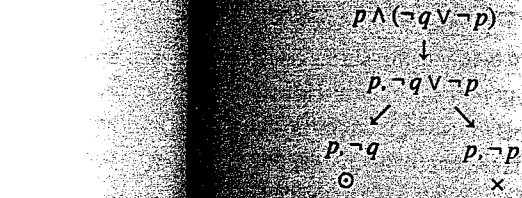
We leave it to the reader to check that for this interpretation, $v(A) = T$. □

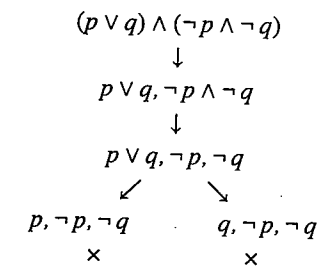
Example 2.45 Now consider the formula $B = (p \vee q) \wedge (\neg p \wedge \neg q)$.

- $v(B) = T$ if and only if $v(p \vee q) = T$ and $v(\neg p \wedge \neg q) = T$.
- Hence, $v(B)$=T if and only if $v(p \vee q) = v(\neg p) = v(\neg q) = T$.
- Hence, $v(B) = T$ if and only if either
  1. $v(p) = v(\neg p) = v(\neg q) = T$, or
  2. $v(q) = v(\neg p) = v(\neg q) = T$.

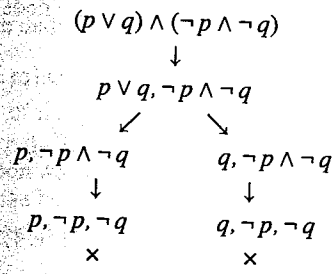
Since both sets of literals $\{p, \neg p, \neg q\}$ and $\{q, \neg p, \neg q\}$ contain complementary pairs, neither set of literals is satisfiable, and we conclude that it is impossible to find a model for B, that is, B is unsatisfiable. □

The systematic search is easy to conduct if a data structure is used to keep track of the assignments that must be made to subformulas. In semantic tableaux, trees are used: the original formula labels the root of the tree, and the sets of formulas created by the search label interior nodes of the tree. The leaves will be labeled by sets of literals that must be satisfied. A leaf containing a complementary set of literals will be marked ×, while a satisfiable leaf will be marked ⊙. Here are semantic tableaux for the formulas A and B from the examples:





The tableau construction is not unique; here is another tableau for $(p \vee q) \wedge (\neg p \wedge \neg q)$. It is constructed by branching to search for a satisfying assignment for $p \vee q$ before searching for one for $\neg p \wedge \neg q$. Clearly, the first tableau is more efficient because it contains fewer nodes.



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

α	α_1	α_2
$\neg\neg A_1$	A_1	
$A_1 \wedge A_2$	A_1	A_2
$\neg(A_1 \vee A_2)$	$\neg A_1$	$\neg A_2$
$\neg(A_1 \to A_2)$	A_1	$\neg A_2$
$\neg(A_1 \uparrow A_2)$	A_1	A_2
$A_1 \downarrow A_2$	$\neg A_1$	$\neg A_2$
$A_1 \leftrightarrow A_2$	$A_1 \to A_2$	$A_2 \to A_1$
$\neg(A_1 \oplus A_2)$	$A_1 \to A_2$	$A_2 \to A_1$

β	β_1	β_2
$\neg(B_1 \wedge B_2)$	$\neg B_1$	$\neg B_2$
$B_1 \vee B_2$	B_1	B_2
$B_1 \to B_2$	$\neg B_1$	B_2
$B_1 \uparrow B_2$	$\neg B_1$	$\neg B_2$
$\neg(B_1 \downarrow B_2)$	B_1	B_2
$\neg(B_1 \leftrightarrow B_2)$	$\neg(B_1 \to B_2)$	$\neg(B_2 \to B_1)$
$B_1 \oplus B_2$	$\neg(B_1 \to B_2)$	$\neg(B_2 \to B_1)$

We now give the construction of a semantic tableau.

Algorithm 2.46 (Construction of a semantic tableau)
Input: A formula A of the propositional calculus.
Output: A semantic tableau $\mathcal{T}$ for A all of whose leaves are marked.

A semantic tableau $\mathcal{T}$ for A is a tree each node of which will be labeled with a set of formulas. Initially, $\mathcal{T}$ consists of a single node, the root, labeled with the singleton set $\{A\}$. The tableau is built inductively by *choosing* an unmarked leaf l labeled with a set of formulas $U(l)$, and applying one of the following rules. The construction terminates when all leaves are marked $\times$ or $\odot$.

- If $U(l)$ is a set of literals, check if there is a complementary pair of literals in $U(l)$. If so, mark the leaf *closed* $\times$; if not, mark the leaf as *open* $\odot$.
- If $U(l)$ is not a set of literals, *choose* a formula in $U(l)$ which is not a literal.
  - If the formula is an α-formula, create a new node l' as a child of l and label l' with
    $$U(l') = (U(l) - \{\alpha\}) \cup \{\alpha_1, \alpha_2\}.$$
    (In the case that α is $\neg\neg A_1$, there is no α_2.)
  - If the formula is a β-formula, create two new nodes l' and l'' as children of l. Label l' with
    $$U(l') = (U(l) - \{\beta\}) \cup \{\beta_1\},$$
    and label l'' with
    $$U(l'') = (U(l) - \{\beta\}) \cup \{\beta_2\}.$$

□

The algorithm is not deterministic since there is a choice of leaf at each step and a choice of which formula to expand within the label of the chosen leaf.

Definition 2.47 A tableau whose construction has terminated is called a *completed tableau*. A completed tableau is *closed* if all leaves are marked closed. Otherwise (that is, if some leaf is marked open), it is *open*. □

Theorem 2.48 *The construction of a semantic tableau terminates.*

Proof: Let $\mathcal{T}$ be the tableau for formula A at any stage of its construction and let us assume for now that $\leftrightarrow$ and $\oplus$ do not occur in the formula A. For any leaf $l \in T$, let $b(l)$ be the number of binary operators in formulas in $U(l)$ and let $n(l)$ be the number of negations in $U(l)$. Define

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

We claim that any step of the construction creates a new node l' or nodes l', l'' such that $W(l) > W(l')$ and $W(l) > W(l'')$. For example, if we apply the α rule to $\neg(A_1 \vee A_2)$ to obtain $\neg A_1$ and $\neg A_2$, then

$$W(l) = k + 3 \cdot 1 + 1 \;>\; k + 3 \cdot 0 + 2 = W(l'),$$

where k is the sum of the number of operators in A_1 and A_2. Obviously, $W(l) \geq 0$, so no branch of $\mathcal{T}$ can be extended indefinitely. We leave it to the reader to check the correctness of the claim for the other rules and to modify the definition of $W(l)$ in the case that A contains $\leftrightarrow$ or $\oplus$. ∎

In practice, the construction of semantic tableaux can be made more efficient:

1. Define a leaf to be *non-atomically closed* if it contains a complementary pair of *formulas* and define it to be *atomically closed* if it contains a complementary pair of *literals*. Change the algorithm to mark a leaf closed if it is non-atomically closed. It can be shown (exercise) that the method of semantic tableaux remains sound and complete under this (more efficient) definition.
2. It is not necessary to copy unmodified formulas from one node to the next. Significant savings of time and memory can be achieved if the label of a node is a set of pointers to the formulas themselves.
3. Various heuristics can be used to shorten the tableau. For example, always use α-rules before β-rules to avoid duplication of formulas.

2.7 Soundness and completeness

Theorem 2.49 (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.*

We delay the proof in order to present some consequences of this theorem.

Corollary 2.50 *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 2.49) $\mathcal{T}$ is not closed iff (by definition) $\mathcal{T}$ is open. ∎

Corollary 2.51 *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 2.52 *The method of semantic tableaux is a decision procedure for validity in the propositional calculus.*

Proof: Let A be a formula of the propositional calculus. By Theorem 2.48, 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 2.51 is called *completeness*, which means that if A is valid, we can discover this fact by constructing a closed tableau for $\neg A$. The converse direction is called *soundness*, which means that 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. For example, the following algorithm is vacuously sound, but far from complete!

Algorithm 2.53 (Incomplete algorithm for validity)
Input: A formula A of the propositional calculus.
Output: A is not valid. □

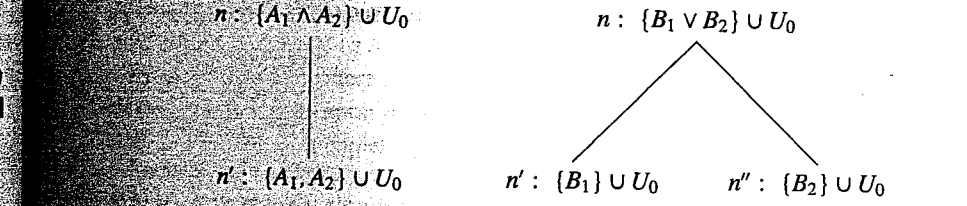
Example 2.54 If the rule for $\neg(A_1 \vee A_2)$ is omitted, the construction is still sound, but it is not complete, because it is impossible to construct a closed tableau for the obviously valid formula $\neg p \vee p$. □

Proof of soundness: To make the proof easier to follow, we will use $\wedge$ and $\vee$ as representatives of the classes of α- and β-formulas, respectively.

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

If $h = 0$, n is a leaf. Since $\mathcal{T}$ closes, $U(n)$ contains a complementary set of literals. Hence $U(n)$ is unsatisfiable.

If $h > 0$, then some α- or β-rule was used in creating the child(ren) of n:



Case 1: An α-rule was used. $U(n) = \{A_1 \wedge A_2\} \cup U_0$ and $U(n') = \{A_1, A_2\} \cup U_0$ for some (possibly empty) set of formulas U_0. But the height of n' is $h - 1$, so by the inductive hypothesis, $U(n')$ is unsatisfiable since the subtree rooted at n' closes. Let v be an arbitrary interpretation. Since $U(n')$ is unsatisfiable, $v(A') = F$ for some $A' \in U(n')$, by Definition 2.30. There are three possibilities:

- For some formula $A_0 \in U_0$, $v(A_0) = F$. But $A_0 \in U_0 \subseteq U(n)$.
- $v(A_1) = F$. By definition of v on $\wedge$, $v(A_1 \wedge A_2) = F$, and $A_1 \wedge A_2 \in U(n)$.
- $v(A_2) = F$. Similarly, $v(A_1 \wedge A_2) = F$, and $A_1 \wedge A_2 \in U(n)$.

Thus $v(A) = F$ for some $A \in U(n)$; since v was arbitrary, $U(n)$ is unsatisfiable.

Case 2: A β-rule was used. $U(n) = \{B_1 \vee B_2\} \cup U_0$, $U(n') = \{B_1\} \cup U_0$, and $U(n'') = \{B_2\} \cup U_0$. By the inductive hypothesis, both $U(n')$ and $U(n'')$ are unsatisfiable. Let v be an arbitrary interpretation. There are two possibilities:

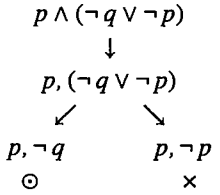
- $U(n')$ and $U(n'')$ are unsatisfiable because $v(B_0) = F$ for some $B_0 \in U_0$. But $B_0 \in U_0 \subseteq U(n)$.
- Otherwise, $v(B_0) = T$ for all $B_0 \in U_0$. Since *both* $U(n')$ and $U(n'')$ are unsatisfiable, $v(B_1) = v(B_2) = F$. By definition of v on $\vee$, $v(B_1 \vee B_2) = F$, and $B_1 \vee B_2 \in U(n)$.

Thus $v(B) = F$ for some $B \in U(n)$; since v was arbitrary, $U(n)$ is unsatisfiable. ∎

Proof of completeness

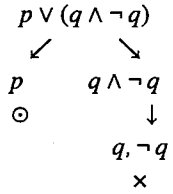
The theorem to be proved is: if A is unsatisfiable then *every* tableau for A closes. Completeness is much more difficult to prove than soundness. There we had a single (though arbitrary) closed tableau and we proved unsatisfiability by a simple induction on the structure of a tableau. Here we have to 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 2.50) and show that if some tableau is open, that is if some tableau has an open branch, then the formula is satisfiable. We have a single (though arbitrary) open branch in a tableau and we can use induction on the length of the branch to prove that A is satisfiable.

Example 2.55 Let $A = p \wedge (\neg q \vee \neg p)$. We have already constructed the tableau for A which is reproduced here.



The interpretation $v(p) = T$, $v(q) = F$ defined by assigning T to the literals labeling the leaf of the open branch is clearly a model for A. □

Example 2.56 Now let $A = p \vee (q \wedge \neg q)$. Here is a tableau for A.



From the open branch of the tableau, we can conclude that any model for A must define $v(p) = T$. However, an interpretation for A must also define an assignment to q. It is obvious in this case that *any* interpretation which assigns T to p is a model for A, regardless of what is assigned to q. □

Completeness will be proved if we can 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 2.57 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 $\alpha \in U$ is an α-formula, then $\alpha_1 \in U$ and $\alpha_2 \in U$.
3. If $\beta \in U$ is a β-formula, then $\beta_1 \in U$ or $\beta_2 \in U$.

□

Example 2.58 U, the union of the set of formulas labeling the open branch of Example 2.56, 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 in U. Condition (2) is vacuous and condition (3) holds as follows: $p \vee (q \wedge \neg q) \in U$, so either $p \in U$ or $q \wedge \neg q \in U$ must be true. In fact, $p \in U$, completing the demonstration that U is a Hintikka set. □

If U is a Hintikka set, requirements (2) and (3) ensure that U is *downward saturated*, meaning that U contains 'sufficient' subformulas so that the decomposition of a formula to be satisfied will not take us out of U. When the decomposition terminates, the set will not contain a complementary pair of literals (by (1)), so the formula must be satisfiable.

Theorem 2.59 *Let l be an open leaf in a completed tableau $\mathcal{T}$. 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 m. Thus if m appears for the first time in $U(n)$ for some n, then $m \in U(k)$ for all nodes k on the branch from n to l, in particular, $m \in U(l)$. This means that all literals in U appear in $U(l)$. Since the branch is open, no complementary pair of literals appear in $U(l)$, so (1) holds for U.

Let $\alpha \in U$. Since the tableau is completed, at some node n an α-rule was used on α. By construction, $\{\alpha_1, \alpha_2\} \in U(n') \subseteq U$ so (2) holds.

Let $\beta \in U$. Since the tableau is completed, at some node n a β-rule was used on β. By construction, $\beta_1 \in U(n')$ and $\beta_2 \in U(n'')$. But the branch from the root to l is constructed by choosing either n' or n'' as part of the branch. Thus either $U(n') \subseteq U$ or $U(n'') \subseteq U$, that is $\beta_1 \in U$ or $\beta_2 \in U$, proving (3). ∎

It now remains to take the Hintikka set defined by an open branch in the tableau and show how the (partial) assignment it defines can be extended to a model.

Theorem 2.60 (Hintikka's Lemma) *Let U be a Hintikka set. Then U is satisfiable.*

Proof: Let $\mathcal{P} = \{p_1, \ldots, p_m\}$ be the set of atoms appearing in all formulas in U. Define an interpretation for U as follows:

$$\begin{array}{ll} v(p) = T & \text{if } p \in U \\ v(p) = F & \text{if } \neg p \in U \\ v(p) = T & \text{if } p \notin U \text{ and } \neg p \notin U. \end{array}$$

Since U is a Hintikka set, by (1) v is well-defined, that is, every atom in $\mathcal{P}$ is given exactly one value. Example 2.56 demonstrates the third case: the variable q appears in a *formula* of U, but neither the literal q *nor* its complement $\neg q$ appear in U.

We now show by structural induction that for any $A \in U$, $v(A) = T$.

- If A is an atom p, then $v(A) = v(p) = T$ since $p \in U$.
- If A is a negated atom $\neg p$, $v(p) = F$ since $\neg p \in U$, so $v(A) = v(\neg p) = T$.
- If A is α, by (2) $\alpha_1 \in U$ and $\alpha_2 \in U$. By the inductive hypothesis, $v(\alpha_1) = v(\alpha_2) = T$, so $v(\alpha) = T$ by definition of the conjunctive operators.
- If A is β, by (3) $\beta_1 \in U$ or $\beta_2 \in U$. By the inductive hypothesis, either $v(\beta_1) = T$ or $v(\beta_2) = T$, so $v(\beta) = T$ by definition of the disjunctive operators.

Since A was an arbitrary formula in U, we have shown that all formulas in U are true in this interpretation. ∎

Proof of completeness: Let $\mathcal{T}$ 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 2.59 and a model can be found for U by Theorem 2.60. Since A is the formula labeling the root, $A \in U$, so the interpretation is a model of A. ∎

2.8 Implementation^P

A tableau will be represented by a predicate `t(Fmls, Left, Right)`, where `Fmls` is a list of the formulas labeling the root of the tableau, and `Left` and `Right` are the subtrees of the root which recursively contain terms on the same predicate. `Right` is ignored for an α-rule. Here is the Prolog term for the tableau for $p \wedge (\neg q \vee \neg p)$ given on page 31.

```
t([p and (neg q or neg p)],
  t([p, neg q or neg p],
    t([p,neg q],open,empty),
    t([p,neg p],closed,empty)
  ),
  empty
).
```

We never explicitly construct the term for a tableau; instead, we write a Prolog program to construct the tableau and another program to print it out in a readable form.

The tableau for a formula `Fml` is created by starting with `t([Fml],_,_)` and then extending the tableau by instantiating the logical variables for the subtrees.

```
create_tableau(Fml, Tab) :-
  Tab = t([Fml], _, _),
  extend_tableau(Tab).
```


The predicate `extend_tableau` performs one step of the tableau construction. First, it checks for a pair of contradictory formulas in `Fmls`, and then it checks if `Fmls` contains only literals. Only then does it perform an alpha or a beta rule, with alpha rules given precedence.

```
extend_tableau(t(Fmls, closed, empty)) :-
  check_closed(Fmls), !.
extend_tableau(t(Fmls, open, empty)) :-
  contains_only_literals(Fmls), !.

extend_tableau(t(Fmls, Left, empty)) :-
  alpha_rule(Fmls, Fmls1), !,
  Left = t(Fmls1, _, _),
  extend_tableau(Left).

extend_tableau(t(Fmls, Left, Right)) :-
  beta_rule(Fmls, Fmls1, Fmls2),
  Left = t(Fmls1, _, _),
  Right = t(Fmls2, _, _),
  extend_tableau(Left),
  extend_tableau(Right).
```

To check if a branch is closed, a formula is chosen from the list of formulas labeling the node, and then a search is made for its complement. Backtracking will ensure that all possibilities are tried.

```
check_closed(Fmls) :-
  member(F, Fmls), member(neg F, Fmls).
```

To check if a branch is open, check if all elements of the label are literals.

```
contains_only_literals([]).
contains_only_literals([Fml | Tail]) :-
  literal(Fml),
  contains_only_literals(Tail).

literal(Fml) :- atom(Fml).
literal(neg Fml) :- atom(Fml).
```

To perform an α- or β-rule, we nondeterministically select a formula, pattern-match it against the database of rules, delete the formula from the node and add the subformulas. The rule for double negation is implemented separately.

```
alpha_rule(Fmls, [A1, A2 | Fmls1]) :-
  member(A, Fmls),
  alpha(A, A1, A2), !,
  delete(Fmls, A, Fmls1).

alpha_rule(Fmls, [A1 | Fmls1]) :-
  member(A, Fmls),
  A = neg neg A1,
  delete(Fmls, A, Fmls1).

beta_rule(Fmls, [B1 | Fmls1], [B2 | Fmls1]) :-
  member(B, Fmls),
  beta(B, B1, B2),
  delete(Fmls, B, Fmls1).
```

The database of rules is copied directly from the tables on page 32.

```
alpha(A1 and A2, A1, A2).
alpha(neg (A1 imp A2), A1, neg A2).
alpha(neg (A1 or A2), neg A1, neg A2).
alpha(A1 eqv A2, A1 imp A2, A2 imp A1).

beta(B1 or B2, B1, B2).
beta(B1 imp B2, neg B1, B2).
beta(neg (B1 and B2), neg B1, neg B2).
beta(neg (B1 eqv B2), neg (B1 imp B2), neg (B2 imp B1)).
```

2.9 Exercises

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

3. Prove that an assignment can be extended to exactly one interpretation (Theorem 2.9) and that assignments that agree on the atoms in a formula extend to the same interpretation (Theorem 2.10).

4. Prove (some of) the logical equivalences in Figure 2.6, especially:

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

5. Prove $((p \oplus q) \oplus q) \equiv p$ and $((p \leftrightarrow q) \leftrightarrow q) \equiv p$.

6. Complete the proof that nand and nor can each define all unary and binary Boolean operators (Theorem 2.23).

7. Prove that $\wedge$ and $\vee$ cannot define negation.

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

9. Prove Theorems 2.32–2.35 on the satisfiability of sets of formulas.

10. Prove Theorems 2.38–2.40 on logical consequence.

11. Prove that for a set of axioms U, $\mathcal{T}(U)$ is closed under logical consequence (see Definition 2.41).

12. For the four equivalences on $\leftrightarrow$ at the bottom of Figure 2.6: (a) prove the equivalences by the method of truth tables; (b) prove them by building semantic tableaux for their negations; (c) prove graphically using Venn diagrams.

13. Complete the proof that the construction of a semantic tableau terminates (Theorem 2.48).

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

15. P Extend the implementation of semantic tableaux to include $\oplus$, $\uparrow$ and $\downarrow$.

16. P Find a more efficient implementation of the check that a node is closed.

3 Propositional Calculus: Deductive Systems

3.1 Deductive proofs

The theorems of a theory $\mathcal{T}(U)$ are the logical consequences of the set of axioms U. Suppose we have a formula A and we want to know if it belongs to the theory $\mathcal{T}(U)$. By Theorem 2.38, $U \models A$ if and only if $\models A_1 \wedge \cdots \wedge A_n \rightarrow A$, where $U = \{A_1, \ldots, A_n\}$ is the set of axioms. Thus $A \in \mathcal{T}(U)$ iff a decision procedure for validity answers 'yes' on the formula. However, there are several problems with this approach:

- The set of axioms may be infinite, for example, in an axiomatization of arithmetic, we may specify that *all* formulas of the form $(x = y) \rightarrow (x + 1 = y + 1)$ are axioms.
- Very few logics have decision procedures like the propositional calculus.
- 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 1994, Section 3.7). This understanding can also help us propose other formulas that might be theorems.
- A decision procedure just produces a 'yes/no' answer, so it is difficult to recognize intermediate results, lemmas. Obviously, the millions of mathematical theorems in existence could not have been inferred directly from axioms.

There is another approach to logic called deductive proofs. Instead of working with *semantic* concepts like interpretation and consequence, we choose a set of axioms and a set of *syntactical* rules for deducing new formulas from the axioms.

Definition 3.1 A *deductive system* is a set of axioms and a set of rules of inference. A *proof* in a deductive system is a sequence of sets of formulas such that each element is either an axiom or it can be inferred from previous elements of the sequence using a rule of inference. If $\{A\}$ is the last element of the sequence, A is a *theorem*, the sequence is a *proof* of A, and A is *provable*, denoted $\vdash A$. □

The concept of deducing theorems from a set of axioms and rules is very old and is familiar to every high-school student who has studied Euclidean geometry. Modern mathematics, with its millions of theorems, 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 are formalizations of the reasoning used in mathematics, and were developed in an attempt to justify mathematical reasoning.

Deduction is purely syntactical. This approach solves the problems described above:

- There may be an infinite number of axioms, but only a finite number will appear in any proof.
- Any particular proof consists of a finite sequence of sets of formulas, and the legality of each individual deduction can be easily and efficiently determined from the syntax of the formulas.
- The proof of a formula clearly shows which axioms, theorems and rules are used and for what purposes. Such a pattern can then be transferred to other similar proofs, or modified to prove different results.
- Once a theorem has been proved, it can be used in proofs just like an axiom.

Deduction introduces new problems. Though deduction is defined purely in terms of syntactical formula manipulation, it is not amenable to systematic search procedures. The semantic tableau rules only create *subformulas* of the formula to be proved (or their negations). In most deductive systems, any axiom can be used, regardless of whether it is a subformula of the formula to be proved. This makes deduction more difficult because it requires ingenuity rather than brute-force search, though programs called *automatic theorem provers* use heuristics to guide the search for a proof.

In the next sections we define the notion of proof in the propositional calculus and then prove soundness and completeness: a formula is valid if and only if it can be proved (deduced) in the axiom system. We will do this twice, first for a Gentzen deductive system which has only one form of axiom but many rules. The completeness will turn out to be trivial because Gentzen systems are just semantic tableaux turned upside-down. Then we will present a Hilbert deductive system which has several forms of axioms but only one rule. The completeness of the Hilbert system will be shown by giving an algorithm to translate any Gentzen proof into a Hilbert proof.

3.2 The Gentzen system $\mathcal{G}$

Definition 3.2 The *Gentzen system* $\mathcal{G}$ is a deductive system. The axioms are the sets of formulas containing a complementary pair of literals. The rules of inference are:

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

where the classification into α- and β-formulas is the dual of the classification for semantic tableaux.

α	α_1	α_2
A	$\neg\neg A$	
$\neg(A_1 \wedge A_2)$	$\neg A_1$	$\neg A_2$
$A_1 \vee A_2$	A_1	A_2
$A_1 \rightarrow A_2$	$\neg A_1$	A_2
$A_1 \uparrow A_2$	$\neg A_1$	$\neg A_2$
$\neg(A_1 \downarrow A_2)$	A_1	A_2
$\neg(A_1 \leftrightarrow A_2)$	$\neg(A_1 \rightarrow A_2)$	$\neg(A_2 \rightarrow A_1)$
$A_1 \oplus A_2$	$\neg(A_1 \rightarrow A_2)$	$\neg(A_2 \rightarrow A_1)$

β	β_1	β_2
$B_1 \wedge B_2$	B_1	B_2
$\neg(B_1 \vee B_2)$	$\neg B_1$	$\neg B_2$
$\neg(B_1 \rightarrow B_2)$	B_1	$\neg B_2$
$\neg(B_1 \uparrow B_2)$	B_1	B_2
$B_1 \downarrow B_2$	$\neg B_1$	$\neg B_2$
$B_1 \leftrightarrow B_2$	$B_1 \rightarrow B_2$	$B_2 \rightarrow B_1$
$\neg(B_1 \oplus B_2)$	$B_1 \rightarrow B_2$	$B_2 \rightarrow B_1$

The set or sets of formulas above the line are called *premises* and the set of formulas below the line is called the *conclusion.* □

A set of formulas in $\mathcal{G}$ is an implicit disjunction, so an axiom containing a complementary pair of literals is obviously valid. For an α-rule, the inference from $U_1 \cup \{A_1, A_2\}$ to $U_1 \cup \{A_1 \vee A_2\}$ (or any other of the disjunctive operators) is simply a formalization of the intuitive meaning of a set as a disjunction. For a β-rule, if we have proved both $\bigvee U_1 \vee B_1$ and $\bigvee U_2 \vee B_2$, then $\bigvee U_1 \vee \bigvee U_2 \vee (B_1 \wedge B_2)$ is inferred using the distribution of disjunction over conjunction. (The notation $\bigvee U$ means the disjunction over all the formulas in U.)

A proof is 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 as an α- or β-rule on the principal operator of the conclusion and the number or numbers of the lines containing the premises. In the system $\mathcal{G}$ we will write $\vdash \{A_1, \ldots, A_n\}$ without the braces as $\vdash A_1, \ldots, A_n$.

Example 3.3 $\vdash (p \vee q) \rightarrow (q \vee p)$. The proof is:

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

□

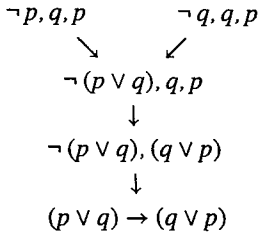
Example 3.4 $\vdash (p \vee (q \wedge r)) \rightarrow (p \vee q) \wedge (p \vee r)$. The proof is:

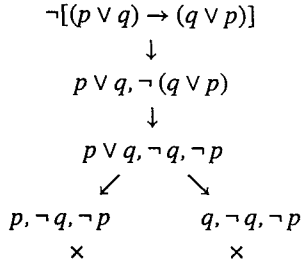
1.	$\vdash \neg p, p, q$	Axiom
2.	$\vdash \neg p, (p \vee q)$	$\alpha\vee$, 1
3.	$\vdash \neg p, p, r$	Axiom
4.	$\vdash \neg p, (p \vee r)$	$\alpha\vee$, 3
5.	$\vdash \neg p, (p \vee q) \wedge (p \vee r)$	$\beta\wedge$, 2, 4
6.	$\vdash \neg q, \neg r, p, q$	Axiom
7.	$\vdash \neg q, \neg r, (p \vee q)$	$\alpha\vee$, 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

□

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.

Example 3.5 The proof of the theorem in Example 3.3 is displayed in tree form on the left below.





If this looks familiar, it should. What we have done is to turn the semantic tableau

upside down and reversed the signs of the formulas in the labels on the nodes, as shown to the right of the Gentzen derivation in the figure. □

If the label of a leaf in a semantic tableau containing an α-formula is extended with a node whose label contains $\{\alpha_1, \alpha_2\}$, then from $\vdash \{\alpha_1, \alpha_2\}$ we can deduce $\vdash \alpha$ in $\mathcal{G}$ (remember that the formulas in $\mathcal{G}$ are duals of those in the semantic tableau). Similarly, if the label of a leaf containing a β-formula is extended with nodes whose labels contain β_1 and β_2, then from $\vdash \beta_1$ and $\vdash \beta_2$, we can deduce $\vdash \beta$ in $\mathcal{G}$. The reader should check these claims by comparing the proof tree with the semantic tableau in Example 3.5.

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

Theorem 3.6 *Let U be a set of formulas and $\bar{U}$ be the set of complements of formulas in U. Then $\vdash U$ in $\mathcal{G}$ if and only if there is a closed semantic tableau for $\bar{U}$.*

Proof: Let $\mathcal{T}$ be a closed semantic tableau for $\bar{U}$. We prove $\vdash U$ by induction on h, the height of T. If $h = 0$, then T consists of a single node labeled by $\bar{U}$, a set of literals containing a complementary pair $\{p, \neg p\}$, that is, $\bar{U} = \bar{U}_0 \cup \{p, \neg p\}$. Obviously, $U = U_0 \cup \{\neg p, p\}$ is an axiom in $\mathcal{G}$, hence $\vdash U$.

If $h > 0$, then some tableau α- or β-rule was used at the root n of T on a formula $\bar{A} \in \bar{U}$, that is, $\bar{U} = \bar{U}_0 \cup \{\bar{A}\}$. (In the following, be sure to distinguish between applications of the tableau rules and applications of the Gentzen rules of the same name.)

Case 1: A tableau α-rule was used on (a formula such as) $\bar{A} = \neg (A_1 \vee A_2)$ to produce the node n' labeled $\bar{U}' = \bar{U}_0 \cup \{\neg A_1, \neg A_2\}$. The subtree rooted at n' is a closed tableau for $\bar{U}'$, so by the inductive hypothesis, $\vdash U_0 \cup \{A_1, A_2\}$. Using the α-rule in $\mathcal{G}$, $\vdash U_0 \cup \{A_1 \vee A_2\}$, that is, $\vdash U$.

Case 2: A tableau β-rule was used on (a formula such as) $\bar{A} = \neg (A_1 \wedge A_2)$ to produce the nodes n' and n'' labeled $\bar{U}' = \bar{U}_0 \cup \{\neg A_1\}$ and $\bar{U}'' = \bar{U}_0 \cup \{\neg A_2\}$, respectively. By the inductive hypothesis, $\vdash U_0 \cup \{A_1\}$, and $\vdash U_0 \cup \{A_2\}$. Using β-rule in $\mathcal{G}$, $\vdash U_0 \cup \{A_1 \wedge A_2\}$, that is, $\vdash U$.

The other direction is left as an exercise. ∎

Corollary 3.7 *$\vdash A$ in $\mathcal{G}$ if and only if there is a closed semantic tableau for $\neg A$.*

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

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 $\mathcal{G}$. ∎

The proof is very simple because we did all the hard work in the proof of the completeness of the tableau method. The Gentzen system $\mathcal{G}$ described in this section is not

very useful; other versions (surveyed in Section 3.6) are more convenient for proving theorems and are closer to Gentzen's original formulation. We introduced $\mathcal{G}$ as a theoretical stepping stone to Hilbert systems which we now describe.

3.3 The Hilbert system $\mathcal{H}$

Hilbert systems are deductive systems for single formulas, unlike Gentzen systems which are deductive systems for sets of formulas. In Gentzen systems there is one axiom and many rules, while in a Hilbert system there are several axioms but only one rule. This textbook (like most others) contains only one theorem (Theorem 3.10) that is proved directly; practical use of the system depends on the use of derived rules, especially the deduction rule.

Definition 3.9 $\mathcal{H}$ is a deductive system with three axiom schemes and one rule of inference. For any formulas A, B, C, the following formulas are axioms:

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 called *modus ponens* (*MP* for short). For any formulas A, B:

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

□

Here is a proof in $\mathcal{H}$ that for any formula A, $\vdash A \to A$. When an axiom is given as the justification, make sure that you can identify which formulas are substituted for the formula letters in the axiom.

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

■

The proof 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*. For each derived rule we prove that the rule is *sound*: the use of the derived rule does not augment the set of provable theorems in $\mathcal{H}$. We show how to mechanically transform a proof using the derived rule into another (usually longer) proof using just the original axioms and *MP*. Of course, once a derived rule is proved to be sound, it can be used in the justification of other derived rules.

The most important derived rule is the *deduction rule*: assume the premise of the implication you want to prove and then prove the consequence.

Example 3.11 Suppose that you want to prove that *the sum of two odd numbers is even*, expressed formally as: $odd(x) \wedge odd(y) \to even(x + y)$, for every x and y. Let us *assume* the formula $odd(x) \wedge odd(y)$ as an additional 'axiom'. Now 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 $2k + 1$. Then

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

an even number. The implication $odd(x) \wedge odd(y) \to even(x + y)$ follows from the deduction rule which 'discharges' the assumption. □

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. If $A_i \in U$, a proof of $U \vdash A$ may include an element of the form $U \vdash A_i$. □

Rule 3.13 (Deduction rule)

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

□

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 $U \cup \{A\} \vdash B$, 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 $\mathcal{H}$. If B is A, then $\vdash A \to B$ by Theorem 3.10, so certainly $U \vdash A \to B$. Otherwise, the following is a proof of $U \vdash A \to B$ which does not use the deduction rule:

1.	$U \vdash B$	Assumption or Axiom
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 $n = 1$. If *MP* was used, then there is a formula C such that formula i in the proof is $U \cup \{A\} \vdash C$

and formula j is $U \cup \{A\} \vdash C \rightarrow B$, for $i, j < n$. By the inductive hypothesis, $U \vdash A \rightarrow C$ and $U \vdash A \rightarrow (C \rightarrow B)$. A proof of $U \vdash A \rightarrow B$ is given by:

i'.	$U \vdash A \rightarrow C$	Inductive hypothesis
j'.	$U \vdash A \rightarrow (C \rightarrow B)$	Inductive hypothesis
$j'+1$.	$U \vdash (A \rightarrow (C \rightarrow B)) \rightarrow ((A \rightarrow C) \rightarrow (A \rightarrow B))$	Axiom 2
$j'+2$.	$U \vdash (A \rightarrow C) \rightarrow (A \rightarrow B)$	MP $j', j'+1$
$j'+3$.	$U \vdash A \rightarrow B$	MP $i', j'+2$

∎

Theorems and derived rules in $\mathcal{H}$

We will now prove a series of important theorems that are also used as justifications for derived rules. Any theorem of the form $U \vdash A \rightarrow B$ justifies a derived rule of the form $\dfrac{U \vdash A}{U \vdash B}$ simply by using *MP* on A and $A \rightarrow B$.
The contrapositive rule is justified by Axiom 3.

Rule 3.15 (Contrapositive rule)

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

□

The contrapositive is used extensively in mathematics. For example, 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:

If a formula is unsatisfiable (not satisfiable), the tableau is closed (not open)

that we had intended to prove.

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

Proof:

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

∎

Rule 3.17 (Transitivity rule)

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

□

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

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

Proof:

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

∎

Rule 3.19 (Exchange of antecedent rule)

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

□

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

Proof:

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

∎

If you can prove some formula and its negation, then you can prove anything! We will study the implications of this result in Section 3.6.

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

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

Rule 3.23 (Double negation rule)

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

Theorem 3.24 $\vdash (A \to B) \to (\neg B \to \neg A)$.

Proof:

1.	$\{A \to B, \neg B, \neg\neg A\} \vdash \neg\neg A$	Assumption
2.	$\{A \to B, \neg B, \neg\neg A\} \vdash A$	Double neg. 1
3.	$\{A \to B, \neg B, \neg\neg A\} \vdash A \to B$	Assumption
4.	$\{A \to B, \neg B, \neg\neg A\} \vdash B$	MP 2, 3
5.	$\{A \to B, \neg B, \neg\neg A\} \vdash \neg B$	Assumption
6.	$\{A \to B, \neg B, \neg\neg A\} \vdash \neg B \to (B \to \neg\neg B)$	Theorem 3.20
7.	$\{A \to B, \neg B, \neg\neg A\} \vdash B \to \neg\neg B$	MP 5, 6
8.	$\{A \to B, \neg B, \neg\neg A\} \vdash \neg\neg B$	MP 4, 7
9.	$\{A \to B, \neg B\} \vdash \neg\neg A \to \neg\neg B$	Deduction 8
10.	$\{A \to B, \neg B\} \vdash \neg B \to \neg A$	Contrapositive 9
11.	$\{A \to B, \neg B\} \vdash \neg B$	Assumption
12.	$\{A \to B, \neg B\} \vdash \neg A$	MP 10, 11
13.	$\{A \to B\} \vdash \neg B \to \neg A$	Deduction 12
14.	$\vdash (A \to B) \to (\neg B \to \neg A)$	Deduction 13

The contrapositive rule can be formulated in the opposite direction. Similarly, the other direction of the double negation rule is justified by the following theorem.

Theorem 3.25 $\vdash A \to \neg\neg A$.

Proof:

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

Let *true* be an abbreviation for $p \to p$ and *false* be an abbreviation for $\neg(p \to p)$. We have $\vdash$ *true* by Theorem 3.10 and $\vdash \neg$*false* by double negation.

Theorem 3.26 $\vdash (A \to \mathit{false}) \to \neg A$.

Proof:

1.	$\{A \to \mathit{false}\} \vdash A \to \mathit{false}$	Assumption
2.	$\{A \to \mathit{false}\} \vdash \neg\mathit{false} \to \neg A$	Contrapositive
3.	$\{A \to \mathit{false}\} \vdash \neg\mathit{false}$	Theorem 3.10, Double neg.
4.	$\{A \to \mathit{false}\} \vdash \neg A$	MP 2, 3
5.	$\vdash (A \to \mathit{false}) \to \neg A$	Deduction

Rule 3.27 (Reductio ad absurdum)

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

This is a very useful (but controversial) rule of mathematical inference: assume the negation of what you wish to prove and show that it leads to a contradiction.

Theorem 3.28 $\vdash (A \to \neg A) \to \neg A$.

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 neg. 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

Theorem 3.29 $\vdash (\neg A \to A) \to A$.

Proof: Exercise. ∎

These two theorems may seem strange, but they can be understood on the semantic level. For the implication of Theorem 3.29 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 $\neg A \to A$, so the formula is true.

Theorems for other operators

So far we have worked with implication as the only binary operator. In Section 3.6 we discuss alternate axiomatizations of Hilbert systems using other operators, but here we just *define* $A \wedge B$, $A \vee B$ and $A \leftrightarrow B$ as abbreviations for $\neg(A \to \neg B)$, $\neg A \to B$ and $(A \to B) \wedge (B \to A)$, respectively. The theorems can also be used implicitly as justifications for appropriate derived rules.

Theorem 3.30 $\vdash A \to (B \to (A \wedge B))$.

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 neg. 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.	$\vdash A \to (B \to (A \wedge B))$	Def. of $\wedge$

∎

Theorem 3.31 (Weakening)

$\vdash A \to A \vee B$.

$\vdash B \to A \vee B$.

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

Proof: Exercise. ∎

Theorem 3.32 (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 neg. 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. ∎

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

Proof:

1.	$\{\neg A \to (\neg B \to C), \neg(\neg A \to B)\} \vdash \neg(\neg A \to B)$	Assumption
2.	$\{\neg A \to (\neg B \to C), \neg(\neg A \to B)\} \vdash B \to (\neg A \to B)$	Axiom 1
3.	$\{\neg A \to (\neg B \to C), \neg(\neg A \to B)\} \vdash \neg(\neg A \to B) \to \neg B$	Contrapositive 2
4.	$\{\neg A \to (\neg B \to C), \neg(\neg A \to B)\} \vdash \neg B$	MP 1, 3
5.	$\{\neg A \to (\neg B \to C), \neg(\neg A \to B)\} \vdash A \to (\neg A \to B)$	Theorem 3.21
6.	$\{\neg A \to (\neg B \to C), \neg(\neg A \to B)\} \vdash \neg(\neg A \to B) \to \neg A$	Contrapositive 5
7.	$\{\neg A \to (\neg B \to C), \neg(\neg A \to B)\} \vdash \neg A$	MP 1, 7
8.	$\{\neg A \to (\neg B \to C), \neg(\neg A \to B)\} \vdash \neg A \to (\neg B \to C)$	Assumption
9.	$\{\neg A \to (\neg B \to C), \neg(\neg A \to B)\} \vdash \neg B \to C$	MP 7, 8
10.	$\{\neg A \to (\neg B \to C), \neg(\neg A \to B)\} \vdash C$	MP 4, 9
11.	$\{\neg A \to (\neg B \to C)\} \vdash \neg(\neg A \to B) \to C$	Deduction 10
12.	$\vdash (\neg A \to (\neg B \to C)) \to (\neg(\neg A \to B) \to C)$	Deduction 11
13.	$\vdash A \vee (B \vee C) \to (A \vee B) \vee C$	Def. of $\vee$

The other direction is similar. ∎

Commutativity and associativity theorems can also be proved for $\wedge$ and $\leftrightarrow$.

3.4 Soundness and completeness of $\mathcal{H}$

Theorem 3.34 *The Hilbert system $\mathcal{H}$ is sound, that is, if $\vdash A$ then $\models A$.*

Proof: The proof is by structural induction. We show that the axioms are valid and that if the premises of *MP* are valid, so is the conclusion. Here are closed semantic tableaux for the negations of Axioms 1 and 3; a tableau for Axiom 2 is left as an exercise.

$$\neg[A \to (B \to A)] \quad \neg[(\neg B \to \neg A) \to (A \to B)]$$

$$\downarrow \qquad\qquad \downarrow$$

$$A, \neg(B \to A) \qquad \neg B \to \neg A, \neg(A \to B)$$

$$\downarrow \qquad\qquad \downarrow$$

$$A, B, \neg A \qquad\qquad \neg B \to \neg A, A, \neg B$$

$$\times \qquad\qquad \swarrow \quad \searrow$$

$$\neg\neg B, A, \neg B \qquad \neg A, A, \neg B$$

$$\downarrow \qquad\qquad \times$$

$$B, A, \neg B$$

$$\times$$

Suppose that *MP* were not sound. Then 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. If B is not valid, there is an interpretation v such that $v(B) = F$. Since A and $A \to B$ are valid, for *any* interpretation, in particular for v, $v(A) = v(A \to B) = T$. From this we deduce that $v(B) = T$ contradicting the choice of v. ∎

Theorem 3.35 *The Hilbert system $\mathcal{H}$ is complete, that is, if $\models A$ then $\vdash A$.*

Any valid formula can be proved in $\mathcal{G}$ (Theorem 3.8). We will show how a proof in $\mathcal{G}$ can be mechanically transformed into a proof in $\mathcal{H}$.

The exact correspondence is that if the *set* of formulas U is provable in $\mathcal{G}$ then the *single* formula $\bigvee U$ is provable in $\mathcal{H}$. The only real difficulty arises from the clash of the data structures used: U is a set while $\bigvee U$ is a formation tree. To see why this is a problem, consider the base case of the induction. The set $\{\neg p, p\}$ is an axiom in $\mathcal{G}$ and we immediately have that $\vdash \neg p \vee p$ in $\mathcal{H}$ since this is simply Theorem 3.10. But if the axiom in $\mathcal{G}$ is $\{q, \neg p, r, p, s\}$, we can't immediately conclude that $\vdash q \vee \neg p \vee r \vee p \vee s$.

Lemma 3.36 *If $U' \subseteq U$ and $\vdash \bigvee U'$ (in $\mathcal{H}$) then $\vdash \bigvee U$ (in $\mathcal{H}$).*

Proof: The proof is by induction using Theorems 3.31 through 3.33. 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 3.31, we can transform this into a proof of $\bigvee U''$, where U'' is a permutation of the elements of U. Now by repeated applications of the commutativity and the associativity of disjunction, we can move the elements of U'' to their proper places. ∎

Example 3.37 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:

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 completeness: The proof is by induction on the structure of the proof in $\mathcal{G}$. If U is an axiom, it contains a pair of complementary literals and $\vdash \neg p \vee p$ can be proved in $\mathcal{H}$. By Lemma 3.36, this may be transformed into a proof of $\bigvee U$.

Otherwise, the last step in the proof of U in $\mathcal{G}$ is the application of an α- or β-rule.

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

Case 2: A β-rule was used in $\mathcal{G}$ to infer $U_1 \cup U_2 \cup \{A_1 \wedge A_2\}$ from $U_1 \cup \{A_1\}$ and $U_2 \cup \{A_2\}$. By the inductive hypothesis, $\vdash \bigvee U_1 \vee A_1$ and $\vdash \bigvee U_2 \vee A_2$ in $\mathcal{H}$. 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. $\vdash \neg \bigvee U_1 \to A_1$
3. $\vdash A_1 \to (A_2 \to (A_1 \wedge A_2))$
4. $\vdash \neg \bigvee U_1 \to (A_2 \to (A_1 \wedge A_2))$
5. $\vdash A_2 \to (\neg \bigvee U_1 \to (A_1 \wedge A_2))$
6. $\vdash \bigvee U_2 \vee A_2$
7. $\vdash \neg \bigvee U_2 \to A_2$
8. $\vdash \neg \bigvee U_2 \to (\neg \bigvee U_1 \to (A_1 \wedge A_2))$
9. $\vdash \bigvee U_1 \vee \bigvee U_2 \vee (A_1 \wedge A_2)$

∎

Consistency

A deductive system that could prove both a formula and its negation is of little use.

Definition 3.38 A set of formulas U is *inconsistent* iff for some formula A, $U \vdash A$ and $U \vdash \neg A$. U is *consistent* iff it is not inconsistent. □

Theorem 3.39 *U is inconsistent iff for all A, $U \vdash A$.*

Proof: Let A be an arbitrary formula. Since 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 3.40 *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 $\nvdash A$. So if there is a falsifiable formula in a sound system, it must be consistent! Since $\not\models$ *false* (where *false* is an abbreviation for $\neg (p \rightarrow p)$), by the soundness of $\mathcal{H}$, $\nvdash$ *false*. By the corollary, the axioms of $\mathcal{H}$ are consistent.

Theorem 3.41 *$U \vdash A$ 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, $U \cup \{\neg A\}$ is inconsistent.

Conversely, if $U \cup \{\neg A\}$ is inconsistent, then $U \cup \{\neg A\} \vdash A$ by Theorem 3.39. 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 3.29). ∎

*Strong completeness and compactness**

The construction of a semantic tableau can be generalized for 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. If the tableau closes, then there is only a finite subset $S_0 \subset S$ of formulas on each closed branch, and S_0 is unsatisfiable, as is $S = S_0 \cup (S - S_0)$ by Theorem 2.34. 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. For details, see Smullyan (1995, Chapter III).

Theorem 3.42 (Strong completeness) *Let U be a finite or countably infinite set of formulas and A a formula. If $U \models A$ then $U \vdash A$.*

The same construction proves the following important theorem.

Theorem 3.43 (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. ∎

3.5 A proof checker^P

Just as we wrote a program to *generate* a semantic tableau from a formula, it would be nice if we could write a program to generate a proof of a formula in $\mathcal{H}$. However, this is far from straightforward as quite a lot of ingenuity goes into producing a concise proof. About the best we could do is to generate a proof using the construction of the completeness proof, but such proofs would be long and unintuitive. In this section we present a *proof checker* for $\mathcal{H}$: a program which receives a list of formulas and their assumptions as its input, and checks if the list is a correct proof. It checks that each element of the list is either an axiom or assumption, or follows from previous elements by *MP* or deduction. The program writes out the justification of each element in the list.

The axioms are facts with the axiom number as an additional argument.

```
axiom(A imp (_ imp A), 1).
axiom((A imp (B imp C)) imp ( (A imp B) imp (A imp C)), 2).
axiom(((neg B) imp (neg A)) imp (A imp B), 3).
```

The data structure used is a list whose elements are of the form `deduce(A,F)`, where `A` is a list of formulas that is the current set of assumptions, and `F` is the formula that has been proved. The predicate `proof` has two additional arguments, a line number used on output and a list of the formulas proved so far.

```
proof(List) :- proof(List, 0, []).
```

Checking an axiom or an assumption is trivial and involves just checking the database of axioms or the list of assumptions.

```
proof([], _, _).
proof([Fml | Tail], Line, SoFar) :-
  Line1 is Line + 1,
  Fml = deduce(_, A),
  axiom(A, N),
  write_proof_line(Line1, Fml, ['Axiom ', N]),
  proof(Tail, Line1, [Fml | SoFar]).
```

```
proof([Fml | Tail], Line, SoFar) :-
  Line1 is Line + 1,
  Fml = deduce(Assump, A),
  member(A, Assump),
  write_proof_line(Line1, Fml, ['Assumption']),
  proof(Tail, Line1, [Fml | SoFar]).
```

To check if A can be justified by *MP*, the predicate nth1 nondeterministically searches SoFar for a formula of the form B imp A and then for the formula B.

```
proof([Fml | Tail], Line, SoFar) :-
  Line1 is Line + 1,
  Fml = deduce(_, A),
  nth1(L1, SoFar, deduce(_, B imp A)),
  nth1(L2, SoFar, deduce(_, B)),
  MP1 is Line1 - L1,
  MP2 is Line1 - L2,
  write_proof_line(Line1, Fml, ['MP ', MP1, ',', MP2]),
  proof(Tail, Line1, [Fml | SoFar]).
```

A formula can be justified by the deduction rule if it is an implication A imp B. Nondeterministically choose a formula from SoFar that has B as its formula, and check that A is in its list of assumptions. The formula A is deleted from Assump, the list of assumptions of A imp B.

```
proof([Fml | Tail], Line, SoFar) :-
  Line1 is Line + 1,
  Fml = deduce(Assump, A imp B),
  nth1(L, SoFar, deduce(Previous, B)),
  member(A, Previous),
  delete(Previous, A, Assump),
  D is Line1 - L,
  write_proof_line(Line1, Fml, ['Deduction ', D]),
  proof(Tail, Line1, [Fml | SoFar]).
```

3.6 Variant forms of the deductive systems*

In this section, we survey some variants of $\mathcal{G}$ and $\mathcal{H}$.

Hilbert systems

Hilbert systems almost invariably have *MP* as the only rule. They differ in the choice of primitive operators and axioms. For example, Axiom 3 can be replaced by:

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

to define a new Hilbert system $\mathcal{H}'$ for the propositional calculus.

Theorem 3.44 *$\mathcal{H}$ and $\mathcal{H}'$ are equivalent.*

Proof: Here is a proof of Axiom 3' in $\mathcal{H}$:

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 3.29
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

We leave it as an exercise to prove Axiom 3 in $\mathcal{H}'$. Note that you may use the deduction theorem without further proof, since its proof in $\mathcal{H}$ does not use Axiom 3, so the identical proof holds in $\mathcal{H}'$. ∎

Either conjunction or disjunction may replace implication as the primitive binary operator in the formulation of a Hilbert system. Implication is 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 $\vdash A \vee A \to A$.

Axiom 2 $\vdash A \to A \vee B$.

Axiom 3 $\vdash A \vee B \to B \vee A$.

Axiom 4 $\vdash (B \to C) \to (A \vee B \to A \vee C)$.

The steps needed to show the equivalence of this system with $\mathcal{H}$ are given in Exercise 1.54 of Mendelson (1997).

Finally, the following axiom together with *MP* as the rule of inference is a complete axiom system for the propositional calculus.

Meredith's axiom $(\{[(A \to B) \to (\neg C \to \neg D)] \to C\} \to E) \to [(E \to A) \to (D \to A)]$.

Adventurous readers are invited to prove the axioms of $\mathcal{H}$ from Meredith's axiom alone following the 37-step plan given in Exercise 8.50 of Monk (1976).

Gentzen systems

$\mathcal{G}$ was constructed in order to simplify the theoretical treatment by using a notation that is identical to that of semantic tableaux. Gentzen's original system is based on sequents; we present a similar system described in Smullyan (1995, Chapter XI).

Definition 3.45 If U and V are (possibly empty) sets of formulas, then $U \Rightarrow V$ is called a *sequent*. □

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

Definition 3.46 Axioms in the Gentzen sequent system $\mathcal{S}$ are sequents of the form: $U \cup \{A\} \Rightarrow V \cup \{A\}$. The rules of inference are:

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

□

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

Definition 3.47 Let $S = U \Rightarrow V$ be a sequent where $U = \{U_1, \ldots, U_n\}$ and $V = \{V_1, \ldots, V_m\}$, and let v be an interpretation for the atomic formulas in S. $v(S) = T$ if and only if $v(U_1) = \cdots = v(U_n) = T$ implies that for some i, $v(V_i) = T$. □

There is a simple relationship between $\mathcal{S}$ and $\mathcal{H}$: a sequent S is true if and only if $(U_1 \wedge \cdots \wedge U_n) \rightarrow (V_1 \vee \cdots \vee V_m)$ is true.

Natural deduction

The advantage of working with sequents is that the deduction theorem is built into the rules of inference (introduction into the consequence of $\rightarrow$). Sequent-based Gentzen systems are often called systems of *natural deduction*.

The convenience of Gentzen systems is apparent when proofs are presented in a format that emphasizes the role of assumptions. Look at the proof of Theorem 3.28, 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 needed and then to *discharge* them by using the deduction rule. Here is a natural deduction proof of Theorem 3.28:

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

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, here an assumption can be used only within the scope of the box, and once it is discharged by using it in a deduction, it is no longer available.

For a presentation of logic based on natural deduction, see Huth & Ryan (2000).

Subformula property

Definition 3.48 A deductive system has the *subformula property* if any formula appearing in a proof of A is either a subformula of A or the negation of a subformula of A. □

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

Gentzen invented his system in order to obtain a formulation of predicate calculus with the subformula property. In addition to the system $\mathcal{S}$, he defined the system $\mathcal{S}'$ with the *cut rule*:

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

and then showed that proofs in $\mathcal{S}'$ can be mechanically transformed into proofs in $\mathcal{S}$.

Theorem 3.49 (Gentzen's Hauptsatz) *Any proof in $\mathcal{S}'$ can be transformed into a proof in $\mathcal{S}$ not using the cut rule.*

Proof: See Smullyan (1995, Chapter XII). ∎

This can be immediately proved indirectly by showing that the cut rule is sound and then invoking the completeness theorem for the cut-free system $\mathcal{G}$. A direct proof by induction on the number of cuts and structural induction on the formula is more complex.

3.7 Exercises

1. Prove in $\mathcal{G}$:
$$\vdash (A \to B) \to (\neg B \to \neg A),$$
$$\vdash (A \to B) \to ((\neg A \to B) \to B),$$
$$\vdash ((A \to B) \to A) \to A.$$

2. Prove that if $\vdash U$ in $\mathcal{G}$ then there is a closed semantic tableau for $\bar{U}$ (the forward direction of Theorem 3.6).

3. Prove the derived rule called *modus tollens*:
$$\frac{\vdash \neg B \qquad \vdash A \to B}{\vdash \neg A}.$$

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

5. Prove $\vdash (\neg A \to A) \to A$ (Theorem 3.29).

6. Prove $\vdash A \to A \vee B$ and the other parts of Theorem 3.31.

7. Prove $\vdash (A \vee B) \vee C \to A \vee (B \vee C)$ (the converse direction of Theorem 3.33).

8. Formulate and prove derived rules based on Theorems 3.30–3.33.

9. Construct a semantic tableau that shows that Axiom 2 of $\mathcal{H}$ is valid.

10. Complete the proof that if $U' \subseteq U$ and $\vdash \bigvee U'$ then $\vdash \bigvee U$ (Lemma 3.36).

11. Prove the formulas of Exercise 1 in $\mathcal{H}$.

12. * Prove Axiom 3 of $\mathcal{H}$ in $\mathcal{H}'$.

13. * Show that the Gentzen sequent system $\mathcal{S}$ is sound and complete.

14. * 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 $\vdash \neg A_1 \vee \cdots \vee \neg A_n$.

15. 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:
    (a) Every finite subset of S is satisfiable.
    (b) For every formula A, at least one of $S \cup \{A\}$, $S \cup \{\neg A\}$ is consistent.
    (c) S can be extended to a maximally consistent set.

16. ^{P} Implement a proof checker for $\mathcal{G}$.

4 Propositional Calculus: Resolution and BDDs

4.1 Resolution

One desirable property of a deductive system is that it should be easy to mechanize an efficient proof search. It is very difficult to search for a proof in a Hilbert system because there is no obvious connection between the formula and its proof. Proof search in the propositional calculus is easy and efficient with semantic tableaux and the equivalent (cut-free) Gentzen systems. However, as we shall see in the next chapter, the method of semantic tableaux in the predicate calculus becomes arbitrary and inefficient. The method of resolution, invented by J. A. Robinson in 1965, is frequently an efficient method for searching for a proof. In this section, we introduce resolution for the propositional calculus, though its advantages will not be apparent until it is extended to the predicate calculus.

CNF and clausal 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 $(\neg p \vee q \vee r) \wedge ((p \wedge \neg q) \vee r) \wedge (\neg r)$ is not in CNF, because of the conjunction within the second 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 the propositional calculus can be transformed into an equivalent formula in CNF.*

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

1. Eliminate all operators except for negation, conjunction and disjunction, using the equivalences in Figure 2.6.

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

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

3. Eliminate double negations using the equivalence $\neg\neg A \equiv A$.

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 $(\neg p \to \neg q) \to (p \to q)$:

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

□

Definition 4.5 A *clause* is a set of literals which is considered to be an implicit disjunction. A *unit clause* is a clause consisting of exactly one literal. A formula in *clausal form* is a set of clauses which is considered to be an implicit conjunction. □

Corollary 4.6 *Every formula in the propositional calculus can be transformed into an equivalent formula in clausal form.*

Proof: In the transformation from a formula to a set of sets of literals, identical literals in a clause and identical clauses in the set will be removed. By idempotence, $A \equiv A \wedge A$ and $A \equiv A \vee A$, logical equivalence is preserved. ∎

Example 4.7 The CNF formula

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

is equivalent to the clausal form $\{\{\neg q, \neg p, q\}, \{p, \neg p, q\}\}$. □

Notation

We sometimes use an abbreviated notation, removing the set delimiters { and } from a clause and denoting negation by a bar over the propositional letter $\bar{p}$. The formula above is written $\{\bar{q}\bar{p}q, p\bar{p}q\}$ in the abbreviated notation.

The following symbols will be used: S for a set of clauses (that is, a formula in clausal form), C for a clause and l for a literal. The symbols will be subscripted and primed as necessary. If l is a literal, l^c is its complement. This means that if $l = p$ then $l^c = \bar{p}$ and if $l = \bar{p}$ then $l^c = p$. The concept of an assignment is generalizaed so that it can be defined on literals in the obvious way: $\nu(l) = T$ means $\nu(p) = T$ if $l = p$ and $\nu(p) = F$ if $l = \bar{p}$.

Properties of clausal form

Definition 4.8 Let S, S' be sets of clauses. $S \approx S'$ denotes that S is satisfiable if and only if S' is satisfiable. □

In the following sequence of lemmas, we show various ways a formula can be transformed without changing its satisfiability.

Lemma 4.9 *Suppose that a literal l appears in (some clause of) S, but l^c does not appear in (any clause of) S. Let S' be obtained from S by deleting every clause containing l. Then $S \approx S'$.*

Proof: If S' is satisfiable, there is a model ν for S' such that $\nu(C') = T$ for every $C' \in S'$. (Recall that a set of clauses is an implicit conjunction so that all clauses must be true for ν to be a model.) Extend ν by defining $\nu(l) = T$. Then $\nu(C) = T$ for every $C \in S - S'$. (Recall that a clause is an implicit disjunction so that it is sufficient if one literal is true for a clause to evaluate to true.) Conversely, if S is satisfiable, S' is obviously satisfiable since $S' \subset S$. ∎

Example 4.10 Let $S = \{pq\bar{r}, p\bar{q}, \bar{p}q\}$ and $S' = \{p\bar{q}, \bar{p}q\}$, where S' is obtained from S by deleting the clause $pq\bar{r}$ containing $\bar{r}$ since $\bar{r}^c = r$ does not appear in S. S' is satisfied by the interpretation $\nu(p) = F$, $\nu(q) = F$, which can be extended to an interpretation of S by defining $\nu(\bar{r}) = T$ so that $\nu(pq\bar{r}) = T$. Note that S is not *logically equivalent* to S', since under the interpretation $\nu(p) = F$, $\nu(q) = F$, $\nu(r) = T$, S evaluates to T and S' evaluates to F. □

Lemma 4.11 *Let $C = \{l\} \in S$ be a unit clause and let S' be obtained from S by deleting every clause containing l and by deleting l^c from every (remaining) clause. Then $S \approx S'$.*

Proof: Let v be a model for S. We will prove that v is also a model for S' (ignoring the assignment to l), by showing that for every C_i' such that $C_i' = C_i - \{l^c\}$, $v(C_i') = T$. $v(C) = v(l) = T$ since v is a model for S, so $v(l^c) = F$. Since v is a model for S, $v(C_i) = T$ and there must be some *other* literal $l_i \in C_i$ such that $v(l_i) = T$. Therefore, $v(C_i') = T$. The proof of the converse is similar to the proof of the previous lemma and is left as an exercise. ∎

Example 4.12 Let $S = \{r, pq\bar{r}, p\bar{q}, \bar{q}p\}$ and $S' = \{pq, p\bar{q}, \bar{q}p\}$, where $\{r\}$ is the unit clause deleted. $v(r) = T$ in any model for S, so if $v(pq\bar{r}) = T$ then either $v(p) = T$ or $v(q) = T$. Therefore, $v(pq) = T$ in S'. □

Lemma 4.13 *Suppose that both $l \in C$ and $l^c \in C$ for some $C \in S$, and let $S' = S - \{C\}$. Then $S \approx S'$.*

Proof: *Any* interpretation satisfies C. ∎

We will assume that all such valid clauses are deleted from a formula in clausal form.

Definition 4.14 If $C_1 \subseteq C_2$, C_1 *subsumes* C_2 and C_2 *is subsumed by* C_1. □

Lemma 4.15 *Let $C_1, C_2 \in S$ be clauses such that C_1* subsumes *C_2, and let $S' = S - \{C_2\}$. Then $S \approx S'$, that is, the* larger *clause can be deleted.*

Proof: Since a clause is an implicit disjunction, any interpretation that satisfies C_1 must also satisfy C_2. ∎

The concept of subsumption may initially seem non-intuitive. The set of interpretations that satisfy the small clause is contained in the set that satisfies the larger clause, so that once we have found an interpretation satisfying the smaller clause it automatically satisfies the larger one.

Example 4.16 Let $S = \{pq, pq\bar{r}, p\bar{q}, \bar{p}q\}$ and $S' = \{pq, p\bar{q}, \bar{p}q\}$, where $pq\bar{r}$ is subsumed by pq. If S' is satisfiable then $v(pq) = T$, so obviously $v(pq\bar{r}) = T$. □

Definition 4.17 The *empty clause* is denoted by $\Box$. The *empty set of clauses* is denoted by $\emptyset$. □

Lemma 4.18 *$\emptyset$ is valid. $\Box$ is unsatisfiable.*

Proof: The proof uses reasoning about vacuous sets and may be a bit hard to follow. A set of clauses is valid iff *every* clause in the set is true under every interpretation. But there are no clauses in $\emptyset$ that need be true, so $\emptyset$ is valid. A clause is satisfiable iff there is some interpretation under which at least one literal in the clause is true. Since there are no literals in $\Box$, under any interpretation there are no literals which will be true, so $\Box$ is unsatisfiable.

Alternatively, we can use the previous lemmas to give a proof. Consider the set of clauses $\{\{p\}\}$. By Lemma 4.9, $\{\{p\}\} \approx \emptyset$. Since $\{\{p\}\}$ is satisfiable, so is $\emptyset$, say by some interpretation v. But by Theorem 2.10, $v'(\emptyset) = v(\emptyset) = T$ for *every* interpretation v', so $\emptyset$ is valid.

$\{\{p\}, \{\neg p\}\}$ is the clausal form of the unsatisfiable formula $p \wedge \neg p$. If we apply Lemma 4.11, the first clause $\{p\}$ is deleted from the formula and the literal $p^c = \neg p$ is deleted from the second clause. Then $\{\{\}\} \approx \{\{p\}, \{\neg p\}\}$, so the set $\{\{\}\} = \{\Box\}$ is unsatisfiable; since $\Box$ is the only clause in the set, it must be unsatisfiable. ∎

Definition 4.19 Let S be a set of clauses and U a set of propositional letters. $R_U(S)$, the *renaming* of S by U, is obtained from S by replacing each literal l whose propositional letter is in U by l^c. □

Lemma 4.20 $S \approx R_U(S)$.

Proof: Let v be a model for S. Define an interpretation v' for $R_U(S)$ by:

$$v'(p) = v(\bar{p}), \text{ if } p \in U$$
$$v'(p) = v(p), \text{ if } p \notin U.$$

Let $C \in S$ and $C' = R_U(\{C\})$. Since v is a model for S, $v(C) = T$ and $v(l) = T$ for some $l \in C$. If the letter p of l is not in U then $l \in C'$ so $v'(l) = v(l) = T$ and $v'(C') = T$. If $p \in U$ then $l^c \in C'$ so $v'(l^c) = v(l) = T$ and $v'(C') = T$. The converse is similar. ∎

Example 4.21 The set of clauses $S = \{pqr, \bar{p}q, \bar{q}\bar{r}, r\}$ is satisfied by the interpretation $v(p) = F$, $v(q) = F$, $v(r) = T$. The renaming $R_{\{p,q\}}(S) = \{\bar{p}\bar{q}r, p\bar{q}, q\bar{r}, r\}$ is satisfied by $v(p) = T$, $v(q) = T$, $v(r) = T$. □

The resolution rule

Resolution is a refutation procedure which is used to show that 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 4.22 (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

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 4.23 The pair of clauses $C_1 = ab\bar{c}$ and $C_2 = bc\bar{e}$ clash on the pair $c, \bar{c}$ and 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 and duplicate literals are removed in the union.

Theorem 4.24 *The resolvent C is satisfiable if and only if the parent clauses C_1 and C_2 are (mutually) satisfiable.*

Proof: Let C_1 and C_2 both be satisfiable under interpretation v. Since l, l^c are complementary, either $v(l) = T$ or $v(l^c) = T$. If $v(l) = T$, then $v(l^c) = F$ so C_2 is satisfied only if $v(l') = T$ for some literal $l' \in C_2, l' \neq l^c$. By construction in the resolution rule, $l' \in C$, so v is also a model for C. A symmetric argument holds if $v(l^c) = T$.

Conversely, if v is an interpretation which satisfies C, $v(l') = T$ for at least one literal $l' \in C$. By the resolution rule, $l' \in C_1$ or $l' \in C_2$ (or both). Suppose $l' \in C_1$, then $v(C_1) = T$. Since neither $l \in C$ nor $l^c \in C$, v is not defined on l and we can extend v to an interpretation v' by defining $v'(l^c) = T$. Then $v'(C_2) = T$ and $v'(C_1) = v(C_1) = T$ because v' is an extension of v. A symmetric argument holds if $l' \in C_2$. ▮

Algorithm 4.25 (Resolution procedure)
Input: A set of clauses S.
Output: S is satisfiable or unsatisfiable.

Let S be a set of clauses and define $S_0 = S$. Assume that S_i has been constructed. *Choose* a pair of clashing clauses $C_1, C_2 \in S_i$ that has not been chosen before. Let C be the clause $Res(C_1, C_2)$ defined by the resolution rule and let $S_{i+1} = S_i \cup \{C\}$. If $C = \square$, terminate the procedure—S is unsatisfiable. If $S_{i+1} = S_i$ for all possible choices of clashing clauses, terminate the procedure—S is satisfiable. □

Example 4.26 Let S be the set of clauses $\{(1)\, p,\ (2)\, \bar{p}q,\ (3)\, \bar{r},\ (4)\, \bar{p}\bar{q}r\}$ where the clauses have been numbered for reference. 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 in the line.

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

□

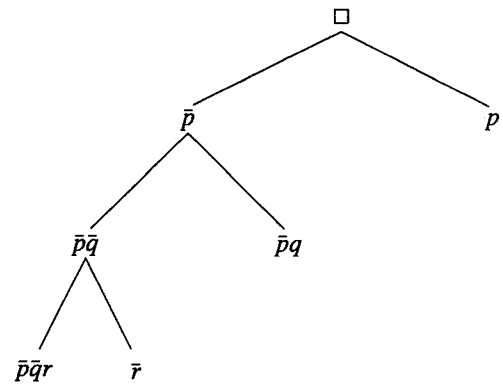


Figure 4.1 Resolution tree

It is easier to read a resolution derivation if it is presented as a tree (Figure 4.1), where the original clauses label leaves, and resolvents label interior nodes whose children are the clauses used in the resolution.

In the example, we have derived the unsatisfiable clause □ so we can 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 $\mathcal{H}$: $(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$. Hence, $\neg A$ is unsatisfiable, that is A is valid and we have used resolution to prove that this instance of Axiom 2 is valid.

Definition 4.27 A derivation of □ from S is called a *refutation* of S. □

Soundness and completeness of resolution

Theorem 4.28 *If the set of clauses labeling the leaves of a resolution tree is satisfiable then the clause at the root is satisfiable.*

Proof: The proof is by induction using Theorem 4.24 and is left as an exercise. ▮

Note that the converse to Theorem 4.28 is not necessarily true because we have no way of ensuring that the extensions made to v on all branches are consistent. In the tree in Figure 4.2, the set of clauses on the leaves $\{r, pq\bar{r}, \bar{r}, p\bar{q}r\}$ is not satisfiable even though the clause on the root p is satisfiable. In this case, we have made a poor choice of clauses to resolve since we could obtain □ immediately by resolving r and $\bar{r}$.

Since resolution is a refutation procedure, the soundness and completeness are expressed in terms of unsatisfiability, rather than validity.

Corollary 4.29 (Soundness) *If the empty clause □ is derived from a set of clauses by the resolution procedure, then the set of clauses is unsatisfiable.*

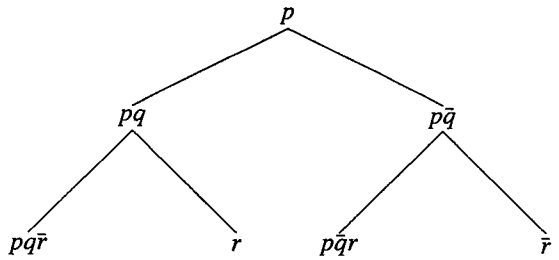


Figure 4.2 Incomplete resolution tree

Proof: Immediate from Theorem 4.28 and Lemma 4.18. ∎

Theorem 4.30 (Completeness) *If a set of clauses is unsatisfiable then the empty clause □ 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 □, rather than continuing indefinitely or terminating but failing to produce □. We defined the resolution procedure to be systematic in that a pair of clauses is never chosen more than once. Since there are only a finite number of distinct clauses on the finite set of propositional letters appearing in the set of clauses, the procedure terminates. Thus, we need only prove that when the procedure terminates, the empty clause is produced.

To prove this, we define a construct called a *semantic tree* (which must not be confused with a semantic tableau). A semantic tree is a data structure for assignments to the atomic propositions of a formula. If the formula is unsatisfiable, all assignments must falsify the formula. We will associate clauses that are created during a resolution refutation with nodes of the tree called failure nodes, which represent assignments that falsify clauses. Eventually the empty clause □ must be created and is associated with the root node that is the failure note representing all assignments.

Algorithm 4.31 (Construction of a semantic tree)
Input: A set of clauses S.
Output: A semantic tree $\mathcal{T}$ for S which is either open or closed.

Let $\{p_1, \ldots, p_n\}$ be the propositional letters appearing in S. Form the complete binary tree $\mathcal{T}$ of depth n and label the left-branching edges from a node of depth $i-1$ by p_i and the right-branching edges by $\bar{p}_i$. Each branch b in $\mathcal{T}$ defines an interpretation v_b by $v_b(p_i) = T$ if p_i labels the i'th edge in b, otherwise, $\bar{p}_i$ labels the i'th edge in b, and $v_b(p_i) = F$. 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*. □

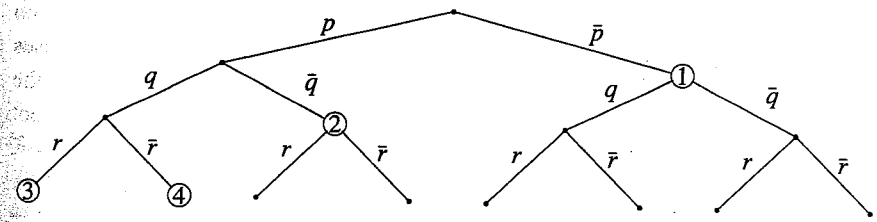


Figure 4.3 Semantic tree

Example 4.32 The semantic tree for $S = \{p, \bar{p}q, \bar{r}, \bar{p}\bar{q}r\}$ is shown in Figure 4.3 where the numbers on the nodes are explained later. The second branch from the left defines the interpretation $v(p) = v(q) = T$, $v(r) = F$, and $v(S) = F$ since the fourth clause $\bar{p}\bar{q}r$ is false in this interpretation. We leave it to the reader to check that every branch in this tree is closed. □

Lemma 4.33 *Let S be a set of clauses and let $\mathcal{T}$ a semantic tree for S. Every interpretation v for S corresponds to v_b for some $b \in \mathcal{T}$, and conversely, every v_b is an interpretation for S.*

Proof: By construction. ∎

Theorem 4.34 *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 v be an arbitrary interpretation for S. v is v_b for some branch by Lemma 4.33. Since $\mathcal{T}$ is closed, $v_b(S) = F$. But $v = v_b$ is arbitrary so S is unsatisfiable. Conversely, let S be an unsatisfiable set of clauses and let $\mathcal{T}$ be a semantic tree for S. Let b be an arbitrary branch. Then v_b is an interpretation for S by Lemma 4.33, and $v_b(S) = F$ since S is unsatisfiable. Since b was arbitrary, $\mathcal{T}$ is closed. ∎

Traversing a branch of the semantic tree top-down, at each node there is a (partial) interpretation defined by the edges already traversed. It is possible that this interpretation is sufficiently defined to evaluate some of the clauses. In particular, some clause might evaluate to F. Since a set of clauses is an implicit conjunction, if 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: the node may be a leaf or it may be an interior node.

Definition 4.35 Let $\mathcal{T}$ be a closed semantic tree for a set of clauses S and let b be a branch in $\mathcal{T}$. The node in b closest to the root which falsifies S is a *failure node*. □

Example 4.36 In Figure 4.3, the node numbered 2 defines a partial interpretation $v(p) = T$, $v(q) = F$, which falsifies the clause $\bar{p}q$ and thus the entire set of clauses S. Neither the parent node (which defines the partial interpretation $v(p) = T$) nor the root itself falsify any of the clauses in the set, so node 2 is the node closest to the root on its branches which falsifies S. Hence, node 2 is a failure node. □

Lemma 4.37 *Let $\mathcal{T}$ be a closed tree for S. Then each failure node in $\mathcal{T}$ falsifies at least one clause in S.*

Proof: Immediate. ∎

Definition 4.38 A clause falsified by a failure node is called a *clause associated with the node*. □

Example 4.39 In the semantic tree in Figure 4.3, each failure node is circled and labeled with the number of the clause associated with it. 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. □

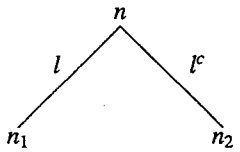
We can be explicit on the relationship between failure nodes and clauses in S.

Lemma 4.40 *A clause C associated with a failure node n is a subset of the complements of the literals appearing on the branch b to n.*

Proof: Intuitively, for C to be falsified at a failure node n, *all* the literals in C must be assigned to in the partial interpretation and furthermore they must all be assigned F because C is a disjunction. Since the partial interpretation is defined by assigning T to the edge labels, the lemma follows.

Formally, let $C = l_1 \cdots l_k$ and let $\{e_1, \ldots, e_m\}$ be the set of literals labeling edges in the branch. Since C is the clause associated with the failure node n, for each i, $v(l_i) = F$ where v is defined by $v(e_i) = T$ on the corresponding edges. Therefore, $l_i = e_i^c \in \{e_1^c, \ldots, e_m^c\}$ and $C = \bigcup_{i=1}^{k} \{l_i\} \subseteq \{e_1^c, \ldots, e_m^c\}$. ∎

Definition 4.41 Let n_1, n_2 be failure nodes which are children of node n. n is called an *inference node*:



□

Example 4.42 In Figure 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 4.43 *In a closed semantic tree $\mathcal{T}$ for a set of clauses S, there is at least one inference node.*

Proof: Suppose that n_1 is a failure node and that its sibling n_2 is not. Then no ancestor of n_2 is a failure node, because its ancestors are also ancestors of n_1 which is, by assumption, the node *closest* to the root on its branch which falsifies the set of clauses. But every branch through n_2 defines an interpretation and they must all falsify S by the assumption that $\mathcal{T}$ is closed. So the assignments defined at the leaves of the subtree rooted at n_2 falsify S; at least one node n' which is a descendant of n_2 (a leaf or an ancestor of a leaf) must be a failure node.

By induction, either there is an inference node (a pair of sibling failure nodes), or we obtain an infinite sequence of failure nodes $n_1, n', n'', \ldots$ of increasing depth which is impossible in a finite tree. ∎

Example 4.44 In Figure 4.3, the parent of nodes 3 and 4 is an inference node. □

Lemma 4.45 *In a closed semantic tree, let b be the branch from the root terminating at inference node n. The children n_1 and n_2 of n are failure nodes, so let C_1 and C_2 be any clauses associated with them, respectively. Then C_1, C_2 clash, and v_b, the partial interpretation associated with n, falsifies C their resolvent clause.*

Proof: Let b_1 and b_2 be the branches that terminate at failure nodes n_1 and n_2, respectively. Since b_1 and b_2 are identical except for the edges from n to n_1 and n_2, by Lemma 4.40 the nodes are associated with clauses C_1, C_2 which clash on the literals l, l^c of the atom p. The partial interpretation v_b is the same as v_{b_1} and v_{b_2}, except that it does not assign to p. But $v_{b_1}(C_1) = v_{b_2}(C_2) = F$ since n_1 and n_2 are failure nodes, so certainly, $v_{b_1}(C_1 - \{l\}) = F$ and $v_{b_2}(C_2 - \{l^c\})$. Clearly, $v_b(C) = v_b((C_1 - \{l\}) \cup (C_2 - \{l^c\})) = F$. ∎

Example 4.46 In Figure 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 $v(p) = T$, $v(q) = T$, the partial interpretation associated with the parent of 3 and 4. Note that if we add the resolvent $\bar{p}\bar{q}$ to S to obtain S_1, the parent node is a failure node for S_1. □

There is one more technicality that must be overcome in the general case. The same semantic tree (Figure 4.3) is also a semantic tree for the set of clauses $\{p, \bar{p}q, \bar{r}, \bar{p}r\}$, where 3 is a failure node associated with $\bar{r}$ and 4 is a failure node associated with $\bar{p}r$. However, their parent is *not* a failure node, since the resolvent $\bar{p}$ is already falsified by

a node higher up in the tree, while a failure node was defined to be the node closest to the root which falsifies the set of clauses.

Lemma 4.47 *Let n be an inference node, let $C_1, C_2 \in S$ be clauses associated with the failure nodes that are the children of n, and let C be 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 4.45, $v(C) = F$, where v is the partial interpretation associated with the inference node. By Lemma 4.40, $C \subseteq \{l_1^c, \ldots, l_n^c\}$, the complements of the literals labeling b, the path to the inference node. Let j be the smallest index such $C \cap \{l_{j+1}^c, \ldots, l_n^c\}$ is empty. Then $C \subseteq \{l_1^c, \ldots, l_j^c\}$ and $v_j(C) = F$ where v_j is the partial interpretation defined at node j. j is a failure node and C is a clause associated with it. ∎

Proof of the completeness of resolution: If S is an unsatisfiable set of clauses, there is a closed semantic tree $\mathcal{T}$ for S. Clauses of S can be associated with failure nodes in $\mathcal{T}$. By Lemma 4.43, there is at least one inference node in $\mathcal{T}$. An application of the resolution rule at that node adds the resolvent to the set, creating a failure node by Lemma 4.47 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. ∎

Implementation[P]

We start with a program to convert a formula into CNF and then give a program to perform resolution. The program for conversion to CNF performs three steps one after another: elimination of operators other than negation, disjunction and conjunction, reducing the scope of negations using De Morgan's laws and double negation, and finally distribution of disjunction over conjunction.

```
cnf(A, A3) :-
  eliminate(A, A1),
  demorgan(A1, A2),
  distribute(A2, A3).
```

Elimination of operators is done by a recursive traversal of the formation tree. The first three clauses eliminate `imp`, `eqv` and `neqv`.

```
eliminate(A eqv B, (A1 and B1) or ((neg A1) and (neg B1))) :-
    eliminate(A, A1), eliminate(B, B1).
eliminate(A xor B, (A1 and (neg B1)) or ((neg A1) and B1)) :-
    eliminate(A, A1), eliminate(B, B1).
eliminate(A imp B, (neg A1) or B1) :-
    eliminate(A, A1), eliminate(B, B1).
```


The next four clauses simply traverse the tree for the other operators.

```
eliminate(A or B, A1 or B1) :-
    eliminate(A, A1), eliminate(B, B1).
eliminate(A and B, A1 and B1) :-
    eliminate(A, A1), eliminate(B, B1).
eliminate(neg A, neg A1) :-
    eliminate(A, A1).
eliminate(A, A).
```

The application of De Morgan's laws is similar. Two clauses apply the laws for negations of conjunction and disjunction.

```
demorgan(neg (A and B), A1 or B1) :-
    demorgan(neg A, A1), demorgan(neg B, B1).
demorgan(neg (A or B), A1 and B1) :-
    demorgan(neg A, A1), demorgan(neg B, B1).
```

The next two clauses traverse the formula for non-negated formulas.

```
demorgan(A and B, A1 and B1) :-
    demorgan(A, A1), demorgan(B, B1).
demorgan(A or B, A1 or B1) :-
    demorgan(A, A1), demorgan(B, B1).
```

The final two clauses eliminate double negation and terminate the recursion at a literal.

```
demorgan((neg (neg A)), A1) :-
    demorgan(A, A1).
demorgan(A, A).
```

Distribution of disjunction over conjunction is more tricky, because one step of the distribution may produce additional structures that must be handled. For example, one step of distribution applied to $p \vee ((q \wedge r) \wedge r)$ gives $(p \vee (q \wedge r)) \wedge (p \vee r)$ and the distribution rule must be applied again.

```
distribute(A or (B and C), ABC) :- !,
  distribute(A or B, AB),
  distribute(A or C, AC),
  distribute(AB and AC, ABC).
distribute((A and B) or C, ABC) :- !,
  distribute(A or C, AC),
  distribute(B or C, BC),
  distribute(AC and BC, ABC).
```

```
distribute(A or B, A1 or B1) :-
  distribute(A, A1),
  distribute(B, B1).
distribute(A and B, A1 and B1) :-
  distribute(A, A1),
  distribute(B, B1).
distribute(A, A).
```

To perform resolution we first convert a CNF formula to clausal form—a set of sets of literals—using a simple program not given here. The program also discards valid clauses like $\{p, r, \bar{p}\}$. The goal clause `resolution(S)` succeeds if the set of clauses S can be refuted, that is, if the empty clause can be produced by resolution. We start with two rules, one that fails if the set of clauses is empty which means that the formula is valid, and one that succeeds if the empty clause is contained in the set.

```
resolution([]) :- !, fail.
resolution(S) :- member([], S), !.
```

The other rule nondeterministically selects two clauses in the set, creates their resolvent, and recursively calls the predicate with the resolvent added to the set. The predicate will fail and backtrack in three cases:

- The two clauses cannot be resolved because they do not have clashing literals.
- The resolvent is a useless valid clause (a clause that clashes with itself).
- The resolvent already exists in the set.

```
resolution(S) :-
  member(C1, S),
  member(C2, S),
  clashing(C1, L1, C2, L2),
  delete(C1, L1, C1P),
  delete(C2, L2, C2P),
  union(C1P, C2P, C),
  \+ clashing(C, _, C, _),
  \+ member(C, S),
  resolution([C | S]).
```

Using nondeterminism, it is trivial to check for clashing literals.

```
clashing(C1, L, C2, neg L) :-
    member(L, C1), member(neg L, C2), !.
clashing(C1, neg L, C2, L) :-
    member(neg L, C1), member(L, C2), !.
```


4.2 Binary decision diagrams (BDDs)

Suppose that you want to decide if two formulas A_1 and A_2 are logically equivalent. You could construct truth tables for both formulas and check if they are identical. Alternatively, you can check if $A_1 \leftrightarrow A_2$ is valid by constructing a semantic tableau for its negation, or by trying to refute the clausal form of its negation using resolution. All these methods can be inefficient if there are many atoms or if the formula is large.

In this section we describe the *binary decision diagram (BDD)*, which is a data structure for propositional formulas. These data structures have the property that (under a certain condition) there is a unique structure for equivalent formulas. Algorithms for BDDs have been found to be surprisingly efficient in many cases. BDDs are extensively used in applications such as circuit design and program verification, where you want to prove that one formula—which describes the operation of a circuit or program—is equivalent to another formula—which specifies the behavior of the circuit or program.

In this section we present a few rather long-winded examples intended to help you understand the intuition behind the algorithms. The formal description of the algorithms and Prolog implementations are given in the next section.

Efficient truth tables

Consider the truth table for $p \vee (q \wedge r)$:

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

From the first two rows, we can see that when p and q are assigned T, the formula evaluates to T regardless of the value of r. Similarly, for the second two rows. Thus the first four rows can be condensed into two rows:

p	q	r	$p \vee (q \wedge r)$
T	T	$*$	T
T	F	$*$	T

where $*$ indicates that the value assigned to r is immaterial. In fact, we now see that the value assigned to q is immaterial, so these two rows collapse into one:

p	q	r	$p \vee (q \wedge r)$
T	$*$	$*$	T

After collapsing the last two rows, the entire truth table can be expressed in four rows:

p	q	r	$p \vee (q \wedge r)$
T	$*$	$*$	T
F	T	T	T
F	T	F	F
F	F	$*$	F

Let us try another example, this time for the formula $p \oplus q \oplus r$. It is easy to compute the truth table for a formula whose only operator is $\oplus$, since a row evaluates to true if and only if an odd number of atoms are assigned true.

p	q	r	$p \oplus q \oplus r$
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	T
F	T	T	F
F	T	F	T
F	F	T	T
F	F	F	F

Here, adjacent rows cannot be collapsed, but careful examination reveals that rows 5 and 6 show the same dependence on r as do rows 3 and 4. Rows 7 and 8 are similarly related to rows 1 and 2. Instead of explicitly writing the truth table entries for these rows, we can simply refer to the previous entries:

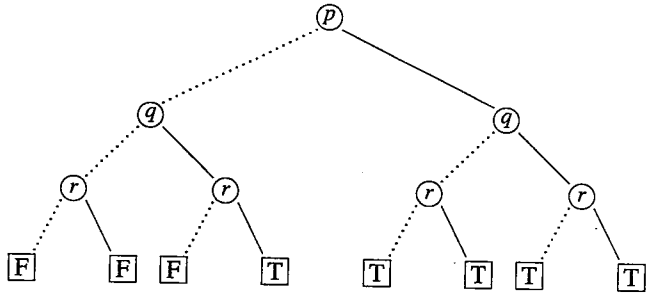
p	q	r	$p \oplus q \oplus r$
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	T
F	T	$*$	(See rows 3 and 4.)
F	F	$*$	(See rows 1 and 2.)

We turn now to an alternate representation of the semantics of a propositional formula that is much more efficient than truth tables.

Reduction of BDDs

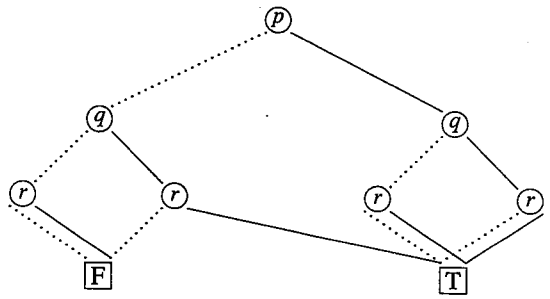
A binary decision diagram, like a truth table, is a representation of the value of a formula under all possible assignments to atomic propositions in the formula. We start with a graph that is simply a notational variant of a semantic tree: each node of the tree is labeled with an atom, and the solid and dotted edges leaving the node represent the assignment of true or false, respectively, to this atom. Along each branch, there is an edge for every atom in the formula, so there is a one-to-one correspondence between branches and assignments. The leaf of a branch is labeled with the value of the formula under its assignment.

Here is a BDD for $p \vee (q \wedge r)$; check that the value of the formula for each assignment is the same as that given in the truth table above.



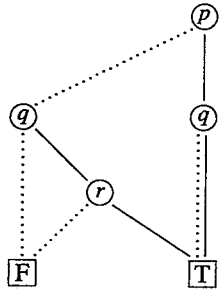
We can modify the structure of this tree to obtain a more concise representation without losing the ability to evaluate the formula under any assignment. The modifications are called *reductions*; when no more reductions can be made, the diagram is called a *reduced BDD*.

The first reduction is to coalesce all the leaves into just two, one for T and one for F. The tree is now a dag (directed acyclic graph), where the direction of an edge is implicitly from a node to its child.

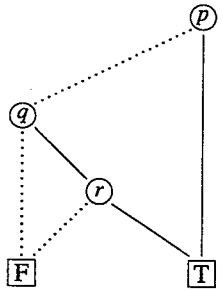


The second reduction is to remove nodes that are not needed to evaluate the formula. Once on the left-hand side of the diagram and twice on the right-hand side, the node

for r has both outgoing edges leading to the same node. This means that the partial assignment to p and q is sufficient to determine the value of the formula. In each of the three cases, the r node and its outgoing edges can be deleted and the incoming edge to the r node connected directly to the joint target node.

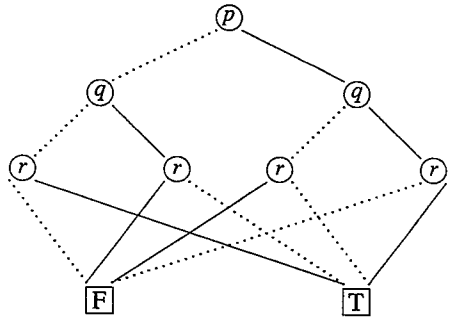


Now, the right-hand node for q becomes redundant and can be eliminated using the same type of reduction.

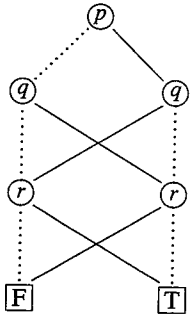


No more reductions can be made on this diagram which is now reduced.Evaluation of the formula for a given truth assignment can be efficiently computed because many nodes have been deleted. For example, when evaluating $p \vee (q \wedge r)$, if the assignment begins by assigning T to p, we immediately reach the T leaf.

Let us now consider the formula $p \oplus q \oplus r$. Here is the dag after the leaves of the tree have been coalesced by the first type of reduction.



The second type of reduction (removing nodes with both outgoing edges pointing to a single node) is not applicable, but examination of the BDD reveals that the two outermost nodes for r have the same structure as do the two innermost nodes, because their true and false edges point to the same subgraphs, in this case the leaves. So the true (solid) edge from the right-hand node for q can be made to point to the leftmost node for r, and the false (dotted) edge from that node can be made to point to the second r node from the left. Deleting the two r nodes on the right and their outgoing edges gives the following reduced BDD.



We invite you to check that these BDDs represent the same evaluations of the formulas as do the truth tables.

Definition of BDDs

Definition 4.48 A *Binary Decision Diagram (BDD)* for a formula A is a rooted directed binary acyclic graph. Each nonterminal is labeled with an atom and each leaf is labeled with a truth value. No atom appears more than once in a path from the root to an edge. One outgoing edge is the *false edge* and is denoted by a dotted line; the other outgoing edge is the *true edge* and is denoted by a solid line.

With each path from the root to a leaf is associated an assignment to the atoms of A. Assign F to the atom if the false edge is taken from the node labeled by the atom, and assign T if the true edge is taken. The leaf gives the value of A under this assignment. The path need not include all atoms in A, but it must include assignments to enough atoms to enable the value of A to be computed. □

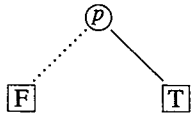
Example 4.49 There are four paths in the reduced BDD for $p \vee (q \wedge r)$ on page 84. Written as literals to denote the assignments, they are (from left to right):

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

The first two paths represent all the assignments that give the value F to the formula, and the second two represent assignments that give T. □

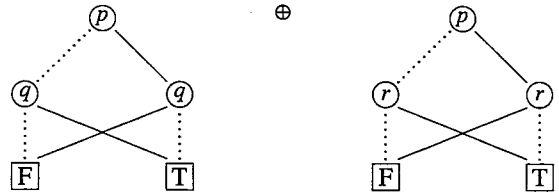
The apply operation

It hardly seems worthwhile to create a BDD if we start from the full binary tree whose size is about the same as the size of the truth table. The power of BDDs comes from the ability to perform propositional operations directly on the two reduced BDDs. That is, there is an algorithm **Apply** that constructs the BDD for $A_1\ op\ A_2$ *directly* and efficiently from the reduced BDDs for A_1 and A_2. The algorithm is also used to construct an initial BDD for an arbitrary formula by building it up from the BDDs for atoms. The BDD for the atom p is simply:



We now construct the BDD for $(p \oplus q) \oplus (p \oplus r)$ from the BDDs for $p \oplus q$ and $p \oplus r$, by applying the operator $\oplus$ to the pair of BDDs. In the following diagram, we have drawn the two BDDs with the operator $\oplus$ between them.

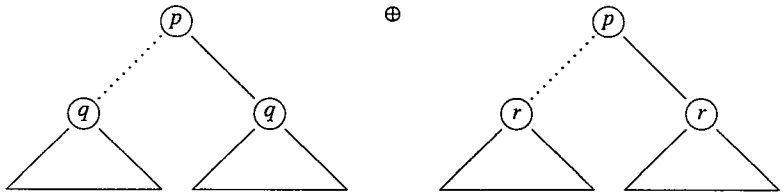
BDD for $p \oplus q$ $\quad\oplus\quad$ BDD for $p \oplus r$



Since $(p \oplus q) \oplus (p \oplus r) \equiv q \oplus r$, the result of the construction should be the BDD for $q \oplus r$.

The algorithm uses structural induction on the pair of BDDs. Given a BDD, the nodes joined to the root are themselves BDDs of formulas obtained by substituting T and F into the formula. The two BDDs above can be considered to have the following structure:

BDD for $p \oplus q$ $\quad\oplus\quad$ BDD for $p \oplus r$



BDD for $F \oplus q$ BDD for $T \oplus q$ BDD for $F \oplus r$ BDD for $T \oplus r$

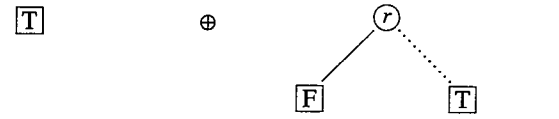
If p is assigned F in both $p \oplus q$ and $p \oplus r$, it will be assigned F in $(p \oplus q) \oplus (p \oplus r)$. So to compute $(p \oplus q) \oplus (p \oplus r)$ for this assignment, we need to compute $(F \oplus q) \oplus (F \oplus r)$, which simplifies to $q \oplus r$. Similarly, if p is assigned T, we need to compute $(T \oplus q) \oplus (T \oplus r)$, which simplifies to $\neg q \oplus \neg r$. Graphically, this can be represented by recursing on the BDDs. The following diagram shows the pair of BDDs obtained by taking the right-hand edges that assign T to p.

BDD for $\neg q$ $\quad\oplus\quad$ BDD for $\neg r$

BDD for $\neg T$ BDD for $\neg F$ BDD for $\neg T$ BDD for $\neg F$

The recursion can be continued by assigning T and then F to q. Since the right-hand formula $\neg r$ does not depend on the assignment to q, it does not split into two recursive subcases as does the left-hand formula $\neg q$. Instead, the algorithm must be applied recursively for each sub-BDD of $\neg q$ together with the *entire* BDD for $\neg r$. The following diagram shows the computation that must be done if the right-hand (false) branch of the BDD for $\neg q$ is taken.

BDD for $\neg F$ $\quad$ BDD for $\neg r$



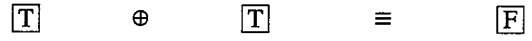
BDD for $\neg T$ BDD for $\neg F$

The left-hand BDD has reached the base case of the recursion. Recursing now on the BDD for $\neg r$ also gives base cases, one for the left-hand (true) branch:

BDD for $\neg F$ $\quad$ BDD for $\neg T$ $\quad$ BDD for $\neg F \oplus \neg T$

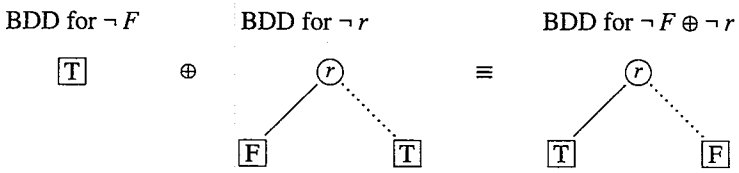
and one for the right-hand (false) branch:

BDD for $\neg F$ $\quad$ BDD for $\neg F$ $\quad$ BDD for $\neg F \oplus \neg F$

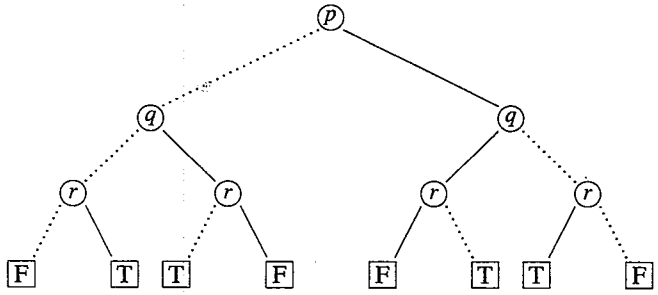


On these base cases, the computation of the resultant BDD is immediate, as shown.

Returning up the recursive evaluation sequence, these two results can be combined to give the result for the non-leaf BDD.

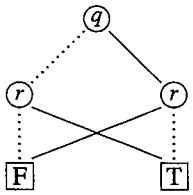


The BDD produced by continuing the algorithm to completion is:



and when it is reduced the expected answer is obtained:

BDD for $q \oplus r$



Many optimizations can be made to improve the efficiency of the algorithm as described in the section on implementation.

4.3 Algorithms on BDDs

The algorithm **Reduce** constructs an canonical BDD if the atoms on each path have compatible orderings.

Definition 4.50 The ordered sequence of atoms labeling the nodes on a path is called an *ordering* of the atoms. A set of orderings $\{O_1, \ldots, O_n\}$ is *compatible* iff there are no atoms p, p' such that p comes before p' in O_i and p' comes before p in O_j, $i \neq j$. □

Definition 4.51 An *Ordered Binary Decision Diagram (OBDD)* is a BDD such that the set of orderings of the atoms of all paths is compatible. □

Example 4.52 The orderings of atoms along the four paths in the reduced BDD for $p \vee (q \wedge r)$ on page 84 are (p, q), (p, q, r), (p, q, r), (p), so the BDD is an OBDD. □

The algorithm Reduce

Algorithm 4.53 (Reduce)
Input: An ordered binary decision diagram *bdd*.
Output: An ordered binary decision diagram in reduced form.

If *bdd* has more than two distinct leaves (one labeled T and one labeled F), remove duplicate leaves and direct all edges that point to leaves to the single remaining leaf for each truth value. Then perform the following steps as long as possible:

1. If both the false and the true edges of a node labeled v_i point to the same node labeled v_j, delete this node for v_i and direct v_i's incoming edges to v_j.
2. If two nodes labeled v_i are the roots of identical sub-BDDs, delete one sub-BDD and direct its incoming edges to the other node.

□

When the algorithm terminates (as it must, why?), the BDD is said to be *reduced.*

Theorem 4.54 (Bryant) *The algorithm* ***Reduce*** *is correct. It returns a reduced OBDD that is equivalent to the OBDD in the input, in the sense that they give the same value to each assignment.*

For a given ordering of atoms, the reduced OBDDs for logically equivalent formulas are structurally identical.

The proofs of the theorems in this section can be found in Bryant (1986).

The algorithm **Reduce** and Theorem 4.54 immediately provide a set of trivial algorithms for properties of propositional formulas:

- A formula is satisfiable iff $\boxed{T}$ appears in its reduced OBDD.
- A formula is valid iff its OBDD is simply $\boxed{T}$.
- A formula is unsatisfiable iff its OBDD is simply $\boxed{F}$.
- $A \equiv B$ if their OBDD's are identical.

The usefulness of OBDDs depends of course on the efficiency of the algorithm **Reduce** (and others that we will describe), which in turn depends on the size of reduced OBDDs. It turns out that in many useful cases the size is quite small. However, the size of the reduced OBDD for a formula depends on the ordering.

Theorem 4.55 (Bryant) *The OBDD for the formula* $(p_1 \wedge p_2) \vee \ldots \vee (p_{2n-1} \wedge p_{2n})$. *has* $2n + 2$ *nodes under the ordering* $p_1, \ldots, p_{2n}$, *and* 2^{n+1} *nodes under the ordering* $p_1, p_{n+1}, p_2, p_{n+2}, \ldots, p_n, p_{2n}$.

Fortunately, you can generally use heuristics to choose an efficient ordering. Unfortunately, not all formulas have orderings that lead to small OBDDs.

Theorem 4.56 (Bryant) *There is a formula A with n atoms such that the reduced OBDD for any ordering of the atoms has at least* 2^{cn} *nodes for some* $c > 0$.

The theorem is constructive in that Bryant gave a specific formula and bound c. Theorem 4.56 is not surprising for reasons to be discussed in Section 4.4.

The algorithm Apply

The following algorithm returns an OBDD for $A_1 \; op \; A_2$ constructed from the OBDDs for the formulas A_1 and A_2.

Algorithm 4.57 (Apply)
Input: A Boolean operator *op*, an OBDD bdd_1 for formula A_1, and an OBDD bdd_2 for formula A_2, such that the union of the set of orderings of atoms in the two OBDDs is compatible.
Output: An OBDD for the formula $A_1 \; op \; A_2$.

- If bdd_1 and bdd_2 are both leaves labeled v_1 and v_2, respectively, return the leaf labeled by $v_1 \; op \; v_2$.
- If the roots of bdd_1 and bdd_2 are both labeled by the same atom p, then return the BDD whose root is labeled by p and whose left (right) sub-BDD is obtained by recursively performing this algorithm on the left (right) sub-BDDs of bdd_1 and bdd_2.
- If the root of bdd_1 is labeled by an atom p_1, and the root of bdd_2 is labeled by some *other* atom p_2 such that $p_1 < p_2$ in the ordering, then return the BDD whose root is labeled by p_1 and whose left (right) sub-BDD is obtained by recursively performing this algorithm with bdd_1 and the left (right) sub-BDD of bdd_2. This construction is also performed if bdd_2 is a leaf, but bdd_1 is not.

- Otherwise, we have a symmetrical case to the previous one. The BDD returned has its root labeled by p_2 and its left (right) sub-BDD obtained by recursively performing this algorithm on the left (right) sub-BDD of bdd_1 and bdd_2.

□

*Restriction and quantification**

Definition 4.58 The *restriction* operation takes a formula A, an atom p and a value $v = T$ or $v = F$, and returns the formula obtained by substituting v for p and partially evaluating A. Notation: $A|_{p=v}$. □

Example 4.59 Let $A = p \vee (q \wedge r)$.

$$A|_{r=T} = p \vee (q \wedge T) = p \vee q$$
$$A|_{r=F} = p \vee (q \wedge F) = p \vee F = p.$$

□

The algorithm **Reduce** is justified by appealing to the following theorem which expresses the application of an operator in terms of its application to restrictions.

Theorem 4.60 (Shannon expansion)

$$A_1 \; op \; A_2 \equiv (p \wedge (A_1|_{p=T} \; op \; A_2|_{p=T})) \vee (\neg p \wedge (A_1|_{p=F} \; op \; A_2|_{p=F})).$$

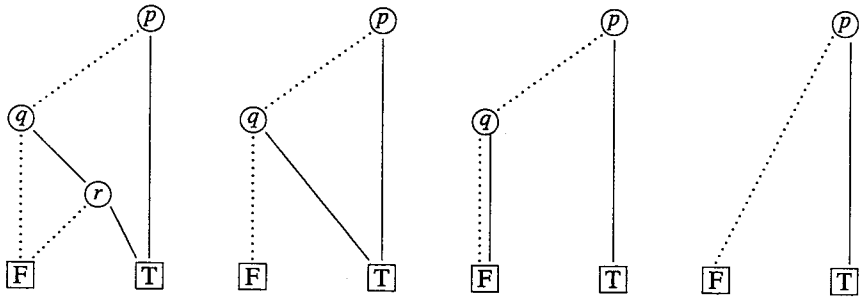
Proof: Exercise. ∎

Restriction is very easy to implement on OBDDs.

Algorithm 4.61 (Restrict)
Input: An ordered binary decision diagram *bdd* for a formula A, an atom p occurring in A and a value v.
Output: An ordered binary decision diagram for $A|_{p=v}$.
The restriction is obtained by a recursive traverse of the OBDD.

1. If the root of *bdd* is a leaf, return the leaf.
2. If the root of *bdd* is labeled with p, then return the false or true sub-BDD, according to the value of v.
3. Otherwise (the root is labeled with $q \neq p$), apply the algorithm to the left and right sub-BDDs, and return the tree whose root is q and whose left and right sub-BDDs are those returned by the recursive calls.

Example 4.62 The OBDD of $A = p \vee (q \wedge r)$ is shown in (a) below. (b) is $A|_{r=T}$ and (c) is $A|_{r=F}$. Note that the restriction may not be reduced; the reduction of the OBDD in (c) is shown in (d).



(a) (b) (c) (b)

Compare the OBDDs in (b) and (d) with the formulas in Example 4.59. □

Is $A = p \vee (q \wedge r)$ true for some value of r or for all values of r? Of course we could construct a truth table and check, but it is much more efficient to use BDDs.

Definition 4.63 Let A be a formula and p an atom. The *existential quantification of* A is the formula denoted $\exists pA$ that is true iff for *some* assignment to p, A is true. The *universal quantification of* A is the formula denoted $\forall pA$ that is true iff for *all* assignments to p, A is true. □

Theorem 4.64 $\exists pA = A|_{p=F} \vee A|_{p=T}$ *and* $\forall pA = A|_{p=F} \wedge A|_{p=T}$.

Proof: Exercise. ∎

Quantification is easily computed using OBDDs:

$$\exists pA = apply(restrict(A, p, F), or, restrict(A, p, T)),$$
$$\forall pA = apply(restrict(A, p, F), and, restrict(A, p, T)).$$

Example 4.65

$$\exists r\,(p \vee (q \wedge r)) = p \vee (p \vee q)) = p \vee q$$
$$\forall r\,(p \vee (q \wedge r)) = p \wedge (p \vee q)) = p.$$

We leave it as an exercise to perform these computations using OBDDs. □

Implementation[P]

Atoms are represented by integers: think of `N` as standing for the atom p_N. BDDs are represented by the predicate `bdd(N,False,True)`, where `N` is the atom labeling the root, `False` is the sub-BDD when `N` is assigned F and `True` is the sub-BDD when `N` is assigned T.

The algorithm **Reduce** does a recursive traversal of the BDD: calling `Reduce(B, BR)` with a BDD `B` returns the reduced BDD in `BR`. A cache of reduced BDDs is maintained as a dynamic database, so that it is easy to check if a BDD already exists as required by the second type of reduction in the algorithm. `retraceall` should be called before executing `reduce` in order to initialize the database.

The first clause checks if the current BDD is in the cache; if so, unification is used to return the existing BDD.

```
reduce(B, B) :- B, !.
```

Next we check if the BDD is a leaf; if so, it is placed into the cache and returned.

```
reduce(B, B) :- B = bdd(leaf, _, _), !, assert(B).
```

The third clause recurses on the sub-BDDs, but before returning it calls `remove` to perform the first type of reduction.

```
reduce(bdd(N, False, True), NewNode) :-
  reduce(False, NewFalse),
  reduce(True, NewTrue),
  remove(bdd(N, NewFalse, NewTrue), NewNode).
```

If the two edges from this node `N` point to the same subBDD, `N` must be removed and one copy of the subBDD returned instead.

```
remove(bdd(_, SubBDD, SubBDD), SubBDD) :- !.
```

Otherwise, the new BDD formed by `N` and the subBDDs returned by the recursion is returned unchanged, but stored in the cache for future use.

```
remove(B, B) :- assert(B).
```

The algorithm **Apply** requires a simultaneous recursive traversal of two BDDs. The base case is if both BDDs are leaves. In this case, simply apply the operator to the values in the leaves.

```
apply(bdd(leaf, Val1, _), Opr, bdd(leaf, Val2, _),
                bdd(leaf, ValResult, x)) :- !,
  opr(Opr, Val1, Val2, ValResult).
```

If the same atom is at the root of both BDDs, a simultaneous recursion is done and the resultant BDD constructed from the BDDs that are returned.

```
apply(bdd(N, False1, True1), Opr, bdd(N, False2, True2),
          bdd(N, FalseResult, TrueResult)) :- !,
  apply(False1, Opr, False2, FalseResult),
  apply(True1, Opr, True2, TrueResult).
```

The difficult part of the implementation is when one of the BDDs has a atom at the root and the other is a leaf, or when the roots are labeled with different atoms. The first clause is taken if the right-hand sub-BDD is a leaf or has a higher-numbered atom; in this case, the sub-BDDs of the left-hand node are applied to the *entire* right-hand node `Node2`. The two cases can be treated together, using the `;` operator in Prolog which succeeds if either of its operand does.

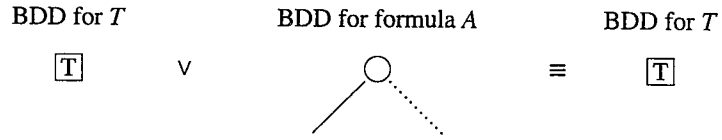
```
apply(bdd(N1, False1, True1), Opr, Node2,
          bdd(N1, FalseResult, TrueResult)) :-
  Node2 = bdd(N2, _, _),
  (N2 = leaf ; (N1 \= leaf, N1 < N2)), !,
  apply(False1, Opr, Node2, FalseResult),
  apply(True1, Opr, Node2, TrueResult).
```

(The check that `N1` is not a leaf is not needed by the algorithm; it just ensures that we don't try to evaluate `leaf<N2` which is illegal in Prolog.) The second clause is symmetrical if the left-hand node is a leaf or has a higher-numbered atom.

```
apply(Node1, Opr, bdd(N2, False2, True2),
          bdd(N2, FalseResult, TrueResult)) :-
  apply(Node1, Opr, False2, FalseResult),
  apply(Node1, Opr, True2, TrueResult).
```

The source archive extends the implementation to include optimizations that are essential for practical use of the algorithm:

- Create a cache `bdd_pair(B1, B2, B)` of *pairs* of BDDs and the result of applying the operator to the pair. As in the algorithm for `reduce`, first check if the result is in the database before performing the traversal.
- If one of the BDDs is a leaf, check if its value is a *controlling operand* for the operator. A value is controlling if the result of the operation does not depend on the other operand. T is controlling for $\vee$ (as shown in the diagram), F is controlling for $\wedge$, and F is controlling for the left operand of $\rightarrow$.



In the presence of a controlling value, the other sub-BDD can be ignored, and the BDD for the controlling value returned immediately.

- Integrate `reduce` with `apply` instead of first creating an unreduced BDD.

4.4 Complexity*

Let us review some of the definitions in algorithmic complexity.

An algorithm is *deterministic* if its computation (and hence its result) is fully determined by its input. A deterministic algorithm is correct iff the (single) result is correct.

Example 4.66 The method of truth tables is a deterministic algorithm for deciding both satisfiability and validity in the propositional calculus: the algorithm constructs the truth table for a formula and checks if T appears in some or all of the rows of the table. □

An algorithm is *nondeterministic* if it is not deterministic, that is, if there may be more than one computation for a given input. A nondeterministic algorithm is correct iff the result of *some* computation is correct.

Example 4.67 The construction of a semantic tableau is a nondeterministic algorithm because at any stage of the construction, we can *choose* a leaf to expand, and further, *choose* a formula in the label of the leaf to which a rule will be applied. If fact, *every* computation (construction) produces a correct answer, so the algorithm is not characteristic of a nondeterministic algorithm. □

Example 4.68 Here is a nondeterministic algorithm for deciding the satisfiability of a formula A.

Choose an interpretation v for A. If $v(A) = T$, then A is satisfiable.

If A is satisfiable, for *some* computation (choice of v), the result is correct. Of course, other choices may not give the correct answer, but that does not affect the correctness of the nondeterministic algorithm. □

An algorithm is said to run in *polynomial time* if its running time can be bounded from above by a polynomial in n, where n is the size of the input. An algorithm is said to run in *exponential time* if its running time can be bounded from below by 2^{cn} for some positive c.

The truth-table method is not an efficient algorithm for satisfiability or validity in the propositional calculus because we construct 2^n rows, where n is the number of variables. For a formula whose size is polynomial in the number of variables, the

complexity of the method will be exponential. The method of semantic tableaux and the resolution procedure are usually more efficient than truth tables; nevertheless, there are families of formulas for which these methods have exponential complexity.

The nondeterministic algorithm in Example 4.68 is a polynomial algorithm for satisfiability: *evaluation* of the truth value of a propositional formula under any particular interpretation is very efficient. Searching for a satisfying interpretation has been replaced by nondeterministic choice of the correct answer.

This shows that the problem of satisfiability in the propositional calculus is in the class *NP* of problems solvable by a *N*ondeterministic algorithm in *P*olynomial time. It is not known if satisfiability is in the class *P* of problems solvable in *P*olynomial time by a deterministic algorithm. Though the clairvoyance of the nondeterministic algorithm seems to be a powerful tool for defining efficient algorithms, no one has been able to prove that there exists a problem in *NP* which is not in *P*. This is called the *P=NP?*-problem.

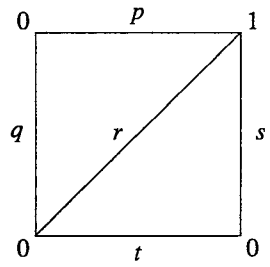
The evidence is overwhelming that *P≠NP*. In 1971, S. A. Cook proved that satisfiability is an *NP*-complete problem, that is, if satisfiability is in *P*, then *every* problem in *NP* is in *P*. Since then, hundreds of problems have been shown to be *NP*-complete. The discovery of a deterministic polynomial-time algorithm for *any* of these problems, including satisfiability, implies the existence of such an algorithm for *all* of them! Many of these problems are famous and have been studied for years by researchers seeking efficient algorithms; thus it is highly unlikely that *P=NP* and hence it is highly unlikely that there is a polynomial algorithm for satisfiability.

Consider now the complementary problem, unsatisfiability or validity. Unsatisfiability is ostensibly a much more difficult problem than satisfiability, because to prove unsatisfiability, we have to show that there is *no* satisfying interpretation. Unsatisfiability is in the class *co-NP* of problems whose complement (here satisfiability) is in *NP*. It can be shown that *co-NP=NP* if and only if unsatisfiability is in *NP*. It is not known if there is a nondeterministic polynomial decision procedure for unsatisfiability, and hence it is also not known if *co-NP=NP* or not.

Hard examples for resolution

The complexity of specific algorithms has been extensively studied. It can be shown that methods of truth tables, semantic tableaux and resolution are all of exponential complexity by exhibiting families of formulas on which the algorithms are inefficient. Here we give an example of this area of research by defining a family of sets of clauses for which resolution is exponential.

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 4.69 The *parity* of a number is 0 if it is even and 1 if it is odd. $\Pi(C)$, the *parity* of a clause C, is the parity of the number of complemented literals in C. $\Pi(v)$, the *parity* of an interpretation v, is the parity of the number of atoms assigned T in v. □

With each graph we associate a set of clauses.

Definition 4.70 Let n be a node of G, let b_n be the label (0 or 1) of n and let $P(n) = \{p_1, \ldots, p_k\}$ be the set of atoms labeling edges incident with n. $C(n)$, the *set of clauses associated with n*, is the set of clauses C whose literals are all the atoms in $P(n)$, some of which are negated so that $\Pi(C) \neq b_n$. v_n is an *interpretation associated with n* if v_n assigns truth values to exactly the literals in $C(n)$. □

Example 4.71 The sets of clauses associated with the four nodes of the graph are (clockwise from the upper-left corner):

$$\{\bar{p}q,\ p\bar{q}\},\qquad \{prs,\ \bar{p}\bar{r}s,\ \bar{p}r\bar{s},\ p\bar{r}\bar{s}\},\qquad \{\bar{s}t,\ s\bar{t}\},\qquad \{\bar{q}rt,\ q\bar{r}t,\ qr\bar{t},\ \bar{q}\bar{r}\bar{t}\}.$$

Checking some of the clauses against the definition:

$$\Pi(\bar{p}\bar{r}s) = 0 \neq 1 = b_n,$$
$$\Pi(\bar{q}rt) = 1 \neq 0 = b_n.$$

□

Lemma 4.72 *v_n satisfies all clauses in $C(n)$ if and only if $\Pi(v_n) = b_n$.*

Proof: Suppose $\Pi(v_n) \neq b_n$. Consider the clause C defined by:

$$l_i = p_i \text{ if } v(p_i) = F \qquad \text{and} \qquad l_i = \bar{p}_i \text{ if } v(p_i) = T.$$

Then

$\Pi(C)$	=	(by definition)
parity of negated atoms of C	=	(by construction)
parity of literals assigned T	=	(by definition)
$\Pi(v_n)$	$\neq$	(by assumption)
b_n,		

so $C \in C(n)$. But $v_n(C) = F$ since v_n assigns F to each literal $l_i \in C$, therefore v_n does not satisfy all clauses in $C(n)$.

We leave the proof of the converse as an exercise. ∎

Example 4.73 Consider an assignment to the atoms adjacent to the upper right node n defined by $v(p) = v(r) = v(s) = T$. For this interpretation, $\Pi(v) = 1 = b_n$ and it is easy to see that $v(C) = T$ for all clauses in $C(n)$. Conversely, $v(p) = v(r) = v(s) = F$ is an interpretation such that $\Pi(v) = 0 \neq b_n$ and $v(prs) = F$ so v does not satisfy all clauses in $C(n)$. □

Lemma 4.74 *If $\sum_{n\in G} b_n = 1$ where $\sum$ is modulo two sum, then $C(G) = \cup_{n\in G} C(n)$ is unsatisfiable.*

Proof: By the previous lemma, if v is an arbitrary model for $C(G)$, then $\Pi(v_n) = b_n$ where v_n is v restricted to the literals incident with n, so $\sum_{n\in G} \Pi(v_n) = \sum_{n\in G} b_n$. Suppose that an atom labels an edge whose endpoints are n' and n''. Then each assignment to a atom appears twice in the sum, once for $v(n')$ and once for $v(n'')$. Thus the number of literals assigned T in $\sum_{n\in G} \Pi(v_n)$ is even and $0 = \sum_{n\in G} \Pi(v_n) = \sum_{n\in G} b_n$. Therefore, if $\sum_{n\in G} b_n = 1$ there cannot be a model for $C(G)$ so the set of clauses is unsatisfiable. ∎

Example 4.75 Let n' be the upper left corner and n'' the upper right corner and v be such that $v(p) = T$. Then $v(p) = T$ is counted twice and its contribution to the total parity is 0. □

There are unsatisfiable sets of clauses associated with arbitrarily large graphs. If the graphs have just a few edges incident with each node—such as a grid with at most four edges per node—the size of the set of clauses will be small. With N nodes and four edges per node, there will be at most $8N$ clauses of four literals each. An appropriate set of clauses was defined in 1968 by G. S. Tseitin, but not until 1987 was the following theorem finally proved:

Theorem 4.76 (Haken and Urquhart) *For arbitrarily large N, there is a graph and a set of associated clauses of size about N, such that the number of distinct clauses created in* any *resolution refutation is greater than 2^{cN} for some fixed $c > 0$.*

The proof is extremely difficult and beyond the scope of this textbook. The difficulty stems from the choice in the resolution procedure—at every step, any two clashing clauses can be chosen. A combinatorical argument is used to show that no sequence of choices gives a non-exponential refutation.

Ironically, the formulas obtained by Tseitin's clauses are problematic only when represented in clausal form. They can be expressed as sets of equivalences, and there is a simple and efficient algorithm for checking the validity of such formulas. Therefore, the theorem shows only that resolution is an exponential algorithm. It does not preclude the (unlikely) existence of a polynomial algorithm for satisfiability.

4.5 Exercises

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.

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.

3. ^{P} Write a program to transform a formula into an equivalent formula in complete DNF.

4. Simplify the following sets of literals, that is, for each set S find a simpler set S', such that $S \approx S'$.

$$\{p\bar{q},\ q\bar{r},\ rs,\ p\bar{s}\},$$
$$\{pqr,\ \bar{q},\ p\bar{r}s,\ qs,\ p\bar{s}\},$$
$$\{pqrs,\ \bar{q}rs,\ \bar{p}rs,\ qs,\ \bar{p}s\},$$
$$\{\bar{p}q,\ qrs,\ \bar{p}\bar{q}rs,\ \bar{r},\ q\}.$$

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

6. 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.

7. * The half-adder of Example 1.1 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.

8. Prove that adding a unit clause on a new atom to a set of clauses and adding its complement to clauses in the set preserves satisfiability (the converse direction of Lemma 4.11).

9. 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 4.28).

10. Prove the Shannon Expansion (Theorem 4.60) and the formula for propositional quantification (Theorem 4.64).

11. Show that $\exists r(p \vee (q \wedge r)) = p \vee q$ and $\forall r(p \vee (q \wedge r)) = p$ using BDDs (Example 4.65).

12. ^{P} Implement the optimizations of the BDD algorithms discussed as the end of Section 4.3.

13. * Construct a resolution refutation for the set of clauses associated with the graph on page 97.

14. * Construct the set of Tseitin clauses corresponding to a labeled complete graph on five vertices and give a resolution refutation of the set.

15. * Show that if $\Pi(v_n) = b_n$, then v_n satisfies all clauses in $C(n)$ (the converse direction of Lemma 4.72).

16. * 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 sets of formulas whose only operators are $\neg$, $\leftrightarrow$ and $\oplus$.

5 Predicate Calculus: Formulas, Models, Tableaux

5.1 Relations and predicates

The axioms and theorems of mathematics are defined on arbitrary sets such as the set of integers $\mathcal{Z}$. We need to be able to write and manipulate logical formulas that contain relations on values from arbitrary sets. The predicate calculus extends the propositional calculus with predicate letters that are interpreted as relations on a domain. (You may wish to review Appendix A on set theory at this time.)

Let $\mathcal{R}$ be an n-ary relation on a domain D, that is, $\mathcal{R}$ is a subset of D^n.

Example 5.1

$\mathcal{P}r(x) \subset \mathcal{N}$ is the set of prime numbers: $\{2, 3, 5, 7, 11, \ldots\}$.

$\mathcal{S}q(x, y) \subset \mathcal{N}^2$ is the set of pairs (x, y) such that $y = x^2$:

$\{(0, 0), (1, 1), (2, 4), (3, 9), \ldots\}$. □

Definition 5.2 A relation can be *represented* by a Boolean-valued function $R : D^n \mapsto \{T, F\}$, by mapping an n-tuple to T if and only if it is included in the relation:

$$R(d_1, \ldots, d_n) = T \text{ iff } (d_1, \ldots, d_n) \in \mathcal{R}.$$

□

Example 5.3 $\mathcal{S}q$ and $\mathcal{P}r$ are represented by the functions Pr and Sq, respectively:

$Pr(0) = F$	$Pr(1) = F$	$Pr(2) = F$	$\cdots$

$Sq(0,0) = T$	$Sq(0,1) = F$	$Sq(0,2) = F$	$\cdots$
$Sq(1,0) = F$	$Sq(1,1) = T$	$Sq(1,2) = F$	$\cdots$
$Sq(2,0) = F$	$Sq(2,1) = F$	$Sq(2,2) = F$	$\cdots$
$\cdots$	$\cdots$	$\cdots$	$\cdots$

□

This correspondence is trivial but it provides the link necessary for a logical formalization of mathematics. All the logical machinery—formulas, interpretations, proofs, etc.—that we developed for the propositional calculus can be applied to predicates. The presence of a domain upon which predicates are interpreted considerably complicates the technical details but not the basic concepts.

An overview of the development is as follows:

- Syntax: Predicate letters are used to represent relations (functions from a domain to truth values). Quantifiers allow a purely syntactical expression of the statement that the relation represented by a predicate is true for *some* or *all* elements of the domain.
- Semantics: An interpretation consists of a domain and an assignment of relations to the predicate letters. The semantics of the Boolean operators remains unchanged, but the evaluation of the truth value of the formula must take the quantifiers into account.
- Semantic tableaux: The systematic search for a model is potentially infinite because domains like the integers are infinite. The construction of a tableau may not terminate, so there is no decision procedure for satisfiability in the predicate calculus. However, if a tableau happens to close, the formula is unsatisfiable; conversely, a systematic tableau for an unsatisfiable formula will close.
- There are Gentzen and Hilbert deductive systems which are sound and complete. A valid formula is provable and we can construct a proof of the formula using tableaux, but given an *arbitrary* formula we cannot decide if it is valid and hence provable.
- The syntax of the predicate calculus is extended with function letters that are interpreted as functions on the domain. With functions we can reason about mathematical operations, for example, $(x > 0 \wedge y > 0) \rightarrow (x \cdot y > 0)$. The predicate calculus with function letters is used in the resolution procedure discussed in Chapter 8.
- There are canonical interpretations called Herbrand interpretations. If a formula has a model, it has a model which is an Herbrand interpretation, so to check satisfiability, it is sufficient to check if there is a Herbrand model for a formula.

5.2 Predicate formulas

Let $\mathcal{P}$, $\mathcal{A}$ and $\mathcal{V}$ be countable sets of symbols called *predicate letters*, *constants* and *variables*, respectively. By convention, the following lower-case letters, possibly with subscripts, will denote these sets: $\mathcal{P} = \{p, q, r\}, \mathcal{A} = \{a, b, c\}, \mathcal{V} = \{x, y, z\}$.

Definition 5.4 The following grammar defines *atomic formulas* and *formulas* in the predicate calculus:

argument	::=	x	for any $x \in \mathcal{V}$
argument	::=	a	for any $a \in \mathcal{A}$
argument_list	::=	*argument*	
argument_list	::=	*argument*, *argument_list*	
atomic_formula	::=	$p \mid p$ (*argument_list*)	for any $p \in \mathcal{P}$
formula	::=	*atomic_formula*	
formula	::=	$\neg$ *formula*	
formula	::=	*formula* $\vee$ *formula*	similarly for $\wedge, \cdots$
formula	::=	$\forall x$ *formula*	for any $x \in \mathcal{V}$
formula	::=	$\exists x$ *formula*	for any $x \in \mathcal{V}$

A predicate letter $p \in \mathcal{P}$ can have no arguments or any finite number of arguments. We regard two predicate letters with different arities as distinct. □

The definition of derivation and formation trees, and the concept of induction on the structure of a formula are taken over unchanged from the propositional calculus. The quantifiers are considered to have higher precedence than the Boolean operators.

Example 5.5 Here are some examples of formulas in the predicate calculus. We will discuss their meaning in the next section after we have defined interpretations.

1. $\forall x \forall y (p(x, y) \rightarrow p(y, x))$.
2. $\forall x \exists y p(x, y)$.
3. $\exists x \exists y (p(x) \wedge \neg p(y))$.
4. $\forall x p(a, x)$.
5. $\forall x (p(x) \wedge q(x)) \leftrightarrow (\forall x p(x) \wedge \forall x q(x))$.
6. $\exists x (p(x) \vee q(x)) \leftrightarrow (\exists x p(x) \vee \exists x q(x))$.
7. $\forall x (p(x) \rightarrow q(x)) \rightarrow (\forall x p(x) \rightarrow \forall x q(x))$.
8. $(\forall x p(x) \rightarrow \forall x q(x)) \rightarrow \forall x (p(x) \rightarrow q(x))$.

□

Definition 5.6 $\forall$ is the *universal quantifier* and is read 'for all'. $\exists$ is the *existential quantifier* and is read 'there exists'. In a quantified formula $\forall xA$, x is the *quantified variable* and A is the *scope* of the quantified variable. It is not required that x actually appear in the scope of its quantification. □

```
program Main;
var X: Integer;
    procedure p;
    var X: Integer;
    begin
        X := 1;   writeln(X)
    end;
    procedure q;
    begin
        writeln(X)
    end;
begin
  X := 5;   p;   q
end.
```

Figure 5.1 Global and local variables

The concept of scope in formulas of the predicate calculus is similar to the concept of scope of variables in a block-structured programming language like Pascal. Consider the program in Figure 5.1. The variable X is declared twice, once globally and once locally in procedure p. The scope of the global declaration includes p, but the local declaration *hides* the global one. The `writeln(X)` statement in p will print 1. In procedure q, the global variable X is in scope and is not hidden; the procedure will print 5. As in programming, hiding a quantified variable within its scope is confusing and should be avoided by giving different names to each quantified variable.

Definition 5.7 Let A be a formula. An occurrence of a variable x in A is a *free variable* of A iff x is not within the scope of a quantified variable x. Notation: $A(x_1, \ldots, x_n)$ indicates that the set of free variables of the formula A is a subset of $\{x_1, \ldots, x_n\}$. A variable which is not free is *bound*.

If a formula has no free variables, it is *closed*. If $\{x_1, \ldots, x_n\}$ are all the free variables of A, the *universal closure* of A is $\forall x_1 \cdots \forall x_n A$ and the *existential closure* is $\exists x_1 \cdots \exists x_n A$. □

Example 5.8 $p(x, y)$ has two free variables x and y, $\exists y p(x, y)$ has one free variable x and $\forall x \exists y p(x, y)$ is closed. The universal closure of $p(x, y)$ is $\forall x \forall y p(x, y)$ and the existential closure is $\exists x \exists y p(x, y)$. □

Example 5.9 In $\forall x p(x) \wedge q(x)$, the occurrence of x in $p(x)$ is bound and the occurrence in $q(x)$ is free. The universal closure is $\forall x(\forall x p(x) \wedge q(x))$. Obviously, it would have been better to write the formula as $\forall x p(x) \wedge q(y)$ where y is the free variable; its closure is $\forall y(\forall x p(x) \wedge q(y))$. □

5.3 Interpretations

Definition 5.10 Let U be a set of formulas such that $\{p_1, \ldots, p_m\}$ are all the predicate letters and $\{a_1, \ldots, a_k\}$ are all the constant symbols appearing in U. An *interpretation* $\mathcal{I}$ is a triple

$$(D, \{R_1, \ldots, R_m\}, \{d_1, \ldots, d_k\}),$$

where D is a *non-empty* domain, R_i is an assignment of an n_i-ary relation on D to the n_i-ary predicate letter p_i and $d_i \in D$ is an assignment of an element of D to the constant a_i. □

Example 5.11 Here are three numerical interpretations for the formula $\forall x p(a, x)$:

$$\mathcal{I}_1 = (\mathcal{N}, \{\leq\}, \{0\}), \qquad \mathcal{I}_2 = (\mathcal{N}, \{\leq\}, \{1\}), \qquad \mathcal{I}_3 = (\mathcal{Z}, \{\leq\}, \{0\}).$$

The formula can also be interpreted over strings $\mathcal{I}_4 = (S, \{substr\}, \{''\})$, where S is the set of strings on some alphabet, *substr* is the binary relation substring and $''$ is the null string. □

Definition 5.12 Let $\mathcal{I}$ be an interpretation. An *assignment* $\sigma_{\mathcal{I}} : \mathcal{V} \mapsto D$ is a function which maps every variable to an element of the domain of $\mathcal{I}$. $\sigma_{\mathcal{I}}[x_i \leftarrow d_i]$ is an assignment that is the same as $\sigma_{\mathcal{I}}$ except that x_i is mapped to d_i. □

Definition 5.13 Let A be a formula, $\mathcal{I}$ an interpretation and $\sigma_{\mathcal{I}}$ an assignment. $v_{\sigma_{\mathcal{I}}}(A)$, the *value* of A *under* $\sigma_{\mathcal{I}}$, is defined by induction on the structure of A:

- Let $A = p_k(c_1, \ldots, c_n)$ be an atomic formula where each c_i is either a variable x_i or a constant a_i. $v_{\sigma_{\mathcal{I}}}(A) = T$ iff $\{d_1, \ldots, d_n\} \in R_k$ where R_k is the relation assigned by $\mathcal{I}$ to p_k, and d_i is the domain element assigned to c_i, either by $\mathcal{I}$ if c_i is a constant or by $\sigma_{\mathcal{I}}$ if c_i is a variable.
- $v_{\sigma_{\mathcal{I}}}(\neg A_1) = T$ iff $v_{\sigma_{\mathcal{I}}}(A_1) = F$.
- $v_{\sigma_{\mathcal{I}}}(A_1 \vee A_2) = T$ iff $v_{\sigma_{\mathcal{I}}}(A_1) = T$ or $v_{\sigma_{\mathcal{I}}}(A_2) = T$, and similarly for the other Boolean operators.
- $v_{\sigma_{\mathcal{I}}}(\forall x A_1) = T$ iff $v_{\sigma_{\mathcal{I}}[x \leftarrow d]}(A_1) = T$ for *all* $d \in D$.
- $v_{\sigma_{\mathcal{I}}}(\exists x A_1) = T$ iff $v_{\sigma_{\mathcal{I}}[x \leftarrow d]}(A_1) = T$ for *some* $d \in D$.

□

Theorem 5.14 *Let A be a* closed *formula. Then $v_{\sigma_\mathcal{I}}(A)$ does not depend on $\sigma_\mathcal{I}$.*

Proof: Call a formula independent of $\sigma_\mathcal{I}$ if its value does not depend on $\sigma_\mathcal{I}$. Let $A' = \forall x A_1(x)$ (respectively $\exists x A_1(x)$) be a (not necessarily proper) subformula of A, where A' is not contained in the scope of any other quantifier. Then $v_{\sigma_\mathcal{I}}(A') = T$ iff $v_{\sigma_\mathcal{I}[x \leftarrow d]}(A_1)$ for all (respectively some) $d \in D$. But x is the only free variable in A_1, so A_1 is independent of $\sigma_\mathcal{I}[x \leftarrow d]$. The theorem can now be proved by induction on the depth of the quantifiers and by structural induction, using the fact that a formula constructed using Boolean operators on independent formulas is also independent. ∎

By the theorem, we can talk about the value of a closed formula, denoted $v_\mathcal{I}(A)$, without mentioning a specific assignment. In fact, there is rarely any need to consider non-closed formulas by the following theorem whose proof is left as an exercise.

Theorem 5.15 *Let $A' = A(x_1, \ldots, x_n)$ be a non-closed formula and let $\mathcal{I}$ be an interpretation. Then:*

- $v_{\sigma_\mathcal{I}}(A') = T$ *for* some *assignment* $\sigma_\mathcal{I}$ *iff* $v_\mathcal{I}(\exists x_1 \cdots \exists x_n A') = T$.
- $v_{\sigma_\mathcal{I}}(A') = T$ *for* all *assignments* $\sigma_\mathcal{I}$ *iff* $v_\mathcal{I}(\forall x_1 \cdots \forall x_n A') = T$.

Now we can give the central semantic definitions for the predicate calculus.

Definition 5.16 A closed formula A is *true* in $\mathcal{I}$ or $\mathcal{I}$ is a *model* for A, if $v_\mathcal{I}(A) = T$. Notation: $\mathcal{I} \models A$. □

Example 5.17 For the formula $A = \forall x p(a, x)$ and the interpretations defined above:

- $\mathcal{I}_1 \models A$ since for *all* $n \in \mathcal{N}, 0 \leq n$.
- $\mathcal{I}_2 \not\models A$ since it is not true that for *all* $n \in \mathcal{N}$, $1 \leq n$. If $n = 0$ then $1 \not\leq 0$.
- $\mathcal{I}_3 \not\models A$, because there is no smallest integer.
- $\mathcal{I}_4 \models A$. By the usual definition, the null string '' is a substring of every string.

□

Definition 5.18 A closed formula A is *satisfiable* if for some interpretation $\mathcal{I}$, $\mathcal{I} \models A$. A is *valid* if for *all* interpretations $\mathcal{I}$, $\mathcal{I} \models A$. Notation: $\models A$. A is *unsatisfiable* if it is not satisfiable, and *falsifiable* if it is not valid. □

Example 5.19 $\forall x p(x) \rightarrow p(a)$ is valid. If it is not, there must be an interpretation $\mathcal{I} = (D, \{R\}, \{d\})$ such that $v_\mathcal{I}(\forall x p(x)) = T$ and $v_\mathcal{I}(p(a)) = F$. By Theorem 5.15, $v_{\sigma_\mathcal{I}}(p(x)) = T$ for all assignments $\sigma_\mathcal{I}$, in particular for the assignment $\sigma'_\mathcal{I}$ that assigns d to x. But $p(a)$ is closed, so $v_{\sigma'_\mathcal{I}}(p(a)) = v_\mathcal{I}(p(a)) = F$, a contradiction. □

Example 5.20 Here is a semantic analysis of the formulas from Example 5.5:

- $\forall x \forall y(p(x, y) \rightarrow p(y, x))$
  The formula is satisfiable in an interpretation where p is assigned a symmetric relation like $=$.
- $\forall x \exists y p(x, y)$
  The formula is satisfiable in an interpretation where p is assigned a relation that is a total function, such as $(x, y) \in R$ iff $y = x + 1$ for $x, y \in \mathcal{Z}$.
- $\exists x \exists y(p(x) \wedge \neg p(y))$
  This formula is satisfiable only in a domain with at least two elements.
- $\forall x p(a, x)$
  This expresses the existence of a special element. For example, if p is interpreted by the relation $\leq$ on the domain $\mathcal{N}$, then the formula is true for $a = 0$. If we change the domain to $\mathcal{Z}$ the formula is false for the same assignment of $\leq$ to p. Thus a change of domain alone can falsify a formula.
- $\forall x(p(x) \wedge q(x)) \leftrightarrow (\forall x p(x) \wedge \forall x q(x))$
  The formula is valid. We prove the forward direction and leave the converse as an exercise. Let $\mathcal{I} = (D, \{R_1, R_2\}, \{\})$ be an arbitrary interpretation. By Theorem 5.15, $v_{\sigma_\mathcal{I}}(p(x) \wedge q(x)) = T$ for all all assignments $\sigma_\mathcal{I}$, and by the inductive definition of an interpretation, $v_{\sigma_\mathcal{I}}(p(x)) = T$ and $v_{\sigma_\mathcal{I}}(q(x)) = T$ for all assignments $\sigma_\mathcal{I}$. Again by Theorem 5.15, $v_\mathcal{I}(\forall x p(x)) = T$ and $v_\mathcal{I}(\forall x q(x)) = T$, and by the definition of interpretation $v_\mathcal{I}(\forall x p(x) \wedge \forall x q(x)) = T$.

  Show that $\forall$ does not distribute over disjunction by constructing a falsifying interpretation for $\forall x(p(x) \vee q(x)) \leftrightarrow (\forall x p(x) \vee \forall x q(x))$.
- $\forall x(p(x) \rightarrow q(x)) \rightarrow (\forall x p(x) \rightarrow \forall x q(x))$
  This is a valid formula, but its converse is not.

□

5.4 Logical equivalence and substitution

Definition 5.21 Given two closed formulas A_1, A_2, if $v_\mathcal{I}(A_1) = v_\mathcal{I}(A_2)$ for all interpretations $\mathcal{I}$, then A_1 is *logically equivalent* to A_2. Notation: $A_1 \equiv A_2$.

Let A be a closed formula and U a set of closed formulas. If for all interpretations $\mathcal{I}$, $v_\mathcal{I}(A) = T$ whenever $v_\mathcal{I}(A_i) = T$ for all $A_i \in U$, then A is a *logical consequence* of U. Notation: $U \models A$. □

$\forall x A(x) \leftrightarrow \neg \exists x \neg A(x)$	$\exists x A(x) \leftrightarrow \neg \forall x \neg A(x)$
$\forall x \forall y A(x,y) \leftrightarrow \forall y \forall x A(x,y)$ $\exists x \forall y A(x,y) \rightarrow \forall y \exists x A(x,y)$	$\exists x \exists y A(x,y) \leftrightarrow \exists y \exists x A(x,y)$
$(\exists x A(x) \vee B) \leftrightarrow \exists x (A(x) \vee B)$	$(\forall x A(x) \vee B) \leftrightarrow \forall x (A(x) \vee B)$
$(B \vee \exists x A(x)) \leftrightarrow \exists x (B \vee A(x))$	$(B \vee \forall x A(x)) \leftrightarrow \forall x (B \vee A(x))$
$(\exists x A(x) \wedge B) \leftrightarrow \exists x (A(x) \wedge B)$	$(\forall x A(x) \wedge B) \leftrightarrow \forall x (A(x) \wedge B)$
$(B \wedge \exists x A(x)) \leftrightarrow \exists x (B \wedge A(x))$	$(B \wedge \forall x A(x)) \leftrightarrow \forall x (B \wedge A(x))$
$\forall x (A \rightarrow B(x)) \leftrightarrow (A \rightarrow \forall x B(x))$	$\forall x (A(x) \rightarrow B) \leftrightarrow (\exists x A(x) \rightarrow B)$
$(\exists x (A(x) \vee B(x)) \leftrightarrow (\exists x A(x) \vee \exists x B(x))$	$\forall x (A(x) \wedge B(x)) \leftrightarrow (\forall x A(x) \wedge \forall x B(x))$
$\forall x A(x) \vee \forall x B(x)) \rightarrow \forall x (A(x) \vee B(x))$	$\exists x (A(x) \wedge B(x)) \rightarrow (\exists x A(x) \wedge \exists x B(x))$
$\forall x (A(x) \leftrightarrow B(x)) \rightarrow (\forall x A(x) \leftrightarrow \forall x B(x))$	$\forall x (A(x) \leftrightarrow B(x)) \rightarrow (\exists x A(x) \leftrightarrow \exists x B(x))$
$\exists x (A(x) \rightarrow B(x)) \leftrightarrow (\forall x A(x) \rightarrow \exists x B(x))$	$(\exists x A(x) \rightarrow \forall x B(x)) \rightarrow \forall x (A(x) \rightarrow B(x))$
$\forall x (A(x) \vee B(x)) \rightarrow (\forall x A(x) \vee \exists x B(x))$	$\forall x (A(x) \rightarrow B(x)) \rightarrow (\forall x A(x) \rightarrow \forall x B(x))$
$\forall x (A(x) \rightarrow B(x)) \rightarrow (\exists x A(x) \rightarrow \exists x B(x))$	$\forall x (A(x) \rightarrow B(x)) \rightarrow (\forall x A(x) \rightarrow \exists x B(x))$

Figure 5.2 Valid formulas in the predicate calculus

Theorem 5.22

$$A \equiv B \quad \text{iff} \quad \models A \leftrightarrow B.$$
$$U \models A \quad \text{iff} \quad \models (A_1 \wedge \cdots \wedge A_n) \rightarrow A.$$

Proof: Exercise. ∎

Figure 5.2 contains a list of valid formulas in the predicate calculus. The rest of this section motivates and explains some important formulas from this list.

The two quantifiers are duals and one can be defined in terms of the other:

$$\forall x A(x) \leftrightarrow \neg \exists x \neg A(x) \qquad \exists x A(x) \leftrightarrow \neg \forall x \neg A(x).$$

Universal quantifiers distribute over conjunction, and existential quantifiers distribute over disjunction, but only one direction holds for universal quantifiers over disjunction and existential quantifiers over conjunction:

$$(\forall x A(x) \vee \forall x B(x)) \rightarrow \forall x (A(x) \vee B(x))$$

$$\exists x (A(x) \wedge B(x)) \rightarrow (\exists x A(x) \wedge \exists x B(x)).$$

If a subformula does not contain a free variable then quantifiers on that variable may be freely passed through the subformula, for example, $\exists x A(x) \vee B \leftrightarrow \exists x (A(x) \vee B)$.

Passing quantifiers through implications is not simple. The formulas in the table can be derived from formulas for disjunction and conjunction by replacing the implication by the equivalent disjunction and observing the alternation of quantifiers as negation is passed through them. Here is an example:

$$\begin{aligned} \exists x (A(x) \rightarrow B(x)) &\equiv \exists x (\neg A(x) \vee B(x)) \\ &\equiv \exists x \neg A(x) \vee \exists x B(x) \\ &\equiv \neg \exists x \neg A(x) \rightarrow \exists x B(x) \\ &\equiv \forall x A(x) \rightarrow \exists x B(x). \end{aligned}$$

Example 5.23 Let us prove the validity of

$$\forall x (A(x) \vee B(x)) \rightarrow (\forall x A(x) \vee \exists x B(x)).$$

Replace the second disjunction by implication and then $\neg \forall x A(x)$ by $\exists x \neg A(x)$ using duality. Exchange of antecedents gives:

$$\exists x \neg A(x) \rightarrow (\forall x (A(x) \vee B(x)) \rightarrow \exists x B(x))).$$

For the formula to be valid, it must be true under all interpretations. Clearly, if $v_{\mathcal{I}}(\exists x \neg A(x)) = F$ or $v_{\mathcal{I}}(\forall x (A(x) \vee B(x))) = F$, the formula is true, so we need only show $v_{\mathcal{I}}(\exists x B(x)) = T$ for interpretations $v_{\mathcal{I}}$ under which these subformulas are true. By Theorem 5.15, for some assignment $\sigma'_{\mathcal{I}}$, $v_{\sigma'_{\mathcal{I}}}(\neg A(x)) = T$ and thus $v_{\sigma'_{\mathcal{I}}}(A(x)) = F$. Using Theorem 5.15 again, $v_{\sigma_{\mathcal{I}}}(A(x) \vee B(x)) = F$ under all assignments, in particular under $\sigma'_{\mathcal{I}}$. By definition of an interpretation for disjunction, $v_{\sigma'_{\mathcal{I}}}(B(x)) = T$, and using Theorem 5.15 yet again, $v_{\mathcal{I}}(\exists x B(x)) = T$. □

5.5 Semantic tableaux

Before presenting the formal construction of semantic tableaux for the predicate calculus, we will informally construct several tableaux to study the difficulties that must be dealt with. Remember that a tableau is a systematic search for a counterexample.

Informal construction

The first example is the negation of the valid formula:

$$\forall x(p(x) \to q(x)) \to (\forall x p(x) \to \forall x q(x)).$$

Applying two α-rules to formulas of the form $\neg(A \to B)$, we obtain the following set of formulas to which no further propositional tableau rules can be applied:

$$\forall x(p(x) \to q(x)),\ \forall x p(x),\ \neg \forall x q(x).$$

The formula $\neg \forall x q(x)$ is equivalent to $\exists x \neg q(x)$, so if we are to create a counterexample, we must instantiate the formula with some domain element substituted for x. Rather than choose some specific domain such as integers or strings, we will use the set of constant symbols of the predicate calculus itself as a domain. Let $a \in \mathcal{A}$ and extend the tableau with a new node whose label replaces the quantified formula with $\neg q(a)$:

$$\forall x(p(x) \to q(x)),\ \forall x p(x),\ \neg q(a).$$

The first two formulas are universally quantified so they impose requirements on *every* element of the proposed domain. Therefore, we instantiate each of them for the domain element a we have introduced. In two steps this gives:

$$\begin{array}{c} \forall x(p(x) \to q(x)),\ \forall x p(x),\ \neg q(a) \\ \downarrow \\ \forall x(p(x) \to q(x)),\ p(a),\ \neg q(a) \\ \downarrow \\ p(a) \to q(a),\ p(a),\ \neg q(a). \end{array}$$

Applying the β-rule to the implication immediately gives a closed tableau.

Next let us try constructing a tableau (Figure 5.3) for the negation of the formula:

$$\forall x(p(x) \vee q(x)) \to (\forall x p(x) \vee \forall x q(x)),$$

which is satisfiable but not valid, so its negation should also be satisfiable. We obtain a closed tableau for a satisfiable formula. What went wrong? The answer is that a should not have been chosen as the domain element for the instantiation of $\neg \forall x p(x)$ (which is an existential formula since it is equivalent to $\exists x \neg p(x)$), once it had already been chosen for $\neg \forall x q(x)$. In fact the formula is true in all interpretations over domains of a single element. When creating domain elements for *existential* formulas, we must choose a new element each time. If we do so, the fifth line of the tableau is:

$$\forall x(p(x) \vee q(x)),\ \neg p(b),\ \neg q(a).$$

Instantiating the universal quantifier with a gives:

$$p(a) \vee q(a),\ \neg p(b),\ \neg q(a).$$

$$\begin{array}{c} \neg(\forall x(p(x) \vee q(x)) \to (\forall x p(x) \vee \forall x q(x))) \\ \downarrow \\ \forall x(p(x) \vee q(x)),\ \neg(\forall x p(x) \vee \forall x q(x)) \\ \downarrow \\ \forall x(p(x) \vee q(x)),\ \neg \forall x p(x),\ \neg \forall x q(x) \\ \downarrow \\ \forall x(p(x) \vee q(x)),\ \neg \forall x p(x),\ \neg q(a) \\ \downarrow \\ \forall x(p(x) \vee q(x)),\ \neg p(a),\ \neg q(a) \\ \downarrow \\ p(a) \vee q(a),\ \neg p(a),\ \neg q(a) \\ \swarrow \qquad\qquad \searrow \\ p(a),\ \neg p(a),\ \neg q(a) \qquad\qquad q(a),\ \neg p(a),\ \neg q(a) \\ \times \qquad\qquad\qquad\qquad \times \end{array}$$

Figure 5.3 Semantic tableau for a satisfiable formula

We should now instantiate the universal formula $\forall x(p(x) \vee q(x))$ again with b since it must be true for *all* domain elements, but, unfortunately, the formula has been 'used up' by the tableau construction. To prevent this, universal formulas will never be deleted from the label of a node. They remain in the label of all descendant nodes so as to constrain the possible interpretations of every new constant that is introduced. The label of the sixth node should be:

$$\forall x(p(x) \vee q(x)),\ p(a) \vee q(a),\ \neg p(b),\ \neg q(a),$$

and the universal formula can be instantiated again with b giving:

$$\forall x(p(x) \vee q(x)),\ p(b) \vee q(b),\ p(a) \vee q(a),\ \neg p(b),\ \neg q(a).$$

We leave it to the reader to extend the tableau using β-rules and to show that exactly one branch of the tableaux is open, defining the model $(\{a, b\}, \{P, Q\}, \{\})$, where $a \in P$, $a \notin Q$, $b \notin P$, $b \in Q$. Thus there is a counterexample to the negation of the formula, so the formula is falsifiable, that is, not valid.

Next, let us search for a model for the formula $A = A_1 \wedge A_2 \wedge A_3$ where:

$$\begin{aligned} A_1 &= \forall x \exists y p(x, y), \\ A_2 &= \forall x \neg p(x, x), \\ A_3 &= \forall x \forall y \forall z (p(x, y) \wedge p(y, z) \to p(x, z)). \end{aligned}$$

Obviously, the first two steps can be α-rules to decompose the conjunction to obtain the set of formulas A_1, A_2, A_3. Now there is a minor problem since we must instantiate the universally quantified formulas but no constants have been introduced by

existential formulas. Recall that the definition of an interpretation required that the domain be non-empty, so we can arbitrarily choose an element a_1 to be in the domain. The tableau construction continues by instantiating A_1 to obtain $\exists y p(a_1, y)$ and then instantiating the existential formula with a new constant:

$$\begin{array}{c} \forall x \exists y p(x,y),\ A_2,\ A_3 \\ \downarrow \\ \forall x \exists y p(x,y),\ \exists y p(a_1,y),\ A_2,\ A_3 \\ \downarrow \\ \forall x \exists y p(x,y),\ p(a_1,a_2),\ A_2,\ A_3 \end{array}$$

Note that since A_1 is universal, it is retained in the label of the descendant nodes. The new constant a_2 must eventually be used to instantiate the universal formula A_1; this results in an existential formula which will be instantiated with a new constant a_3.

$$\begin{array}{c} \forall x \exists y p(x,y),\ p(a_1,a_2),\ A_2,\ A_3 \\ \downarrow \\ \forall x \exists y p(x,y),\ \exists y p(a_2,y),\ p(a_1,a_2),\ A_2,\ A_3 \\ \downarrow \\ \forall x \exists y p(x,y),\ p(a_2,a_3),\ p(a_1,a_2),\ A_2,\ A_3 \end{array}$$

The construction will not terminate; if you continue the tableau construction, applying rules also to A_2 and A_3, an infinite branch is obtained. The set of atomic formulas on the branch will be $p(a_i, a_j)$, where $a_1, a_2, \ldots$ is an infinite sequence and $i < j$. The tableau neither closes nor terminates; instead it defines an *countably infinite* model, that is, a model with a countable domain. In fact, $(\mathcal{N}, \{<\})$ is a model for A.

Theorem 5.24 *The formula $A = A_1 \wedge A_2 \wedge A_3$ has no finite model.*

Proof: Suppose that A had a finite model. The domain of an interpretation is non-empty so it has at least one element. By A_1, there is a sequence of elements $a_1, \ldots$ such that $v_{\sigma_I[x \leftarrow a_i][y \leftarrow a_j]}(p(x,y)) = T$ for all i and $j = i+1$. By A_3, this holds for all $j \neq i$. Since the model is finite, for some k, $a_k = a_i$, contradicting A_2 which requires that $v_{\sigma_I[x \leftarrow a_i]}(p(x,x)) = F$. ∎

The construction of semantic tableaux is *not* a decision procedure for validity in the predicate calculus. We can never know if a branch that does not close defines an infinite model or if it will eventually close, say, after one million further applications of the tableau rules.

A final complication in the construction of semantic tableau is given by:

$$A_1 \wedge A_2 \wedge A_3 \wedge \forall x (q(x) \wedge \neg q(x)).$$

After applying the α-rules, the node label is:

$$A_1, A_2, A_3, \forall x (q(x) \wedge \neg q(x))$$

In the propositional calculus, formulas always simplify so every formula in the label of a node eventually has a rule applied to it. However, once we require that universally quantified formulas not be deleted, we could endlessly apply rules to A_1 and never get around to using the fourth subformula which immediately closes the tableau. Thus a *systematic* construction is needed to make sure that rules are eventually applied to all formulas labeling a node of the tableau.

Formal construction

The formal presentation of the construction of semantic tableaux for the predicate calculus is given in two stages. First rules are presented for the universal and existential quantifiers and soundness is proved. Then a systematic construction of tableaux is described and used to prove completeness of the procedure. As in the propositional case, we generalize the rules into two cases, γ-rules for universal formulas and δ-rules for existential formulas:

γ	$\gamma(a)$
$\forall x A(x)$	$A(a)$
$\neg \exists x A(x)$	$\neg A(a)$

δ	$\delta(a)$
$\exists x A(x)$	$A(a)$
$\neg \forall x A(x)$	$\neg A(a)$

Definition 5.25 A *literal* is a closed atomic formula $p(a_1, \ldots, a_k)$, that is, an atomic formula all of whose arguments are constants, or the negation of such a formula. □

Algorithm 5.26 (Construction of a semantic tableau)
Input: A formula A of the predicate calculus.
Output: A semantic tableau $\mathcal{T}$ for A: all branches are either infinite, or finite with leaves marked closed or open.

A semantic tableau for A is a tree $\mathcal{T}$ each node of which will be labeled with a set of formulas. Initially, $\mathcal{T}$ consists of a single node, the root, labeled with the singleton set $\{A\}$. The tableau is built inductively by *choosing* an unmarked leaf l labeled with a set of formulas $U(l)$, and applying one of the following rules.

- If $U(l)$ is a set of literals, check if there is a complementary pair of literals $\{p(a_1, \ldots, a_k), \neg p(a_1, \ldots, a_k)\}$ in $U(l)$. If so, mark the leaf *closed* $\times$; if not, mark the leaf as *open* $\odot$.
- If $U(l)$ is not a set of literals, *choose* a formula A in $U(l)$ which is not a literal.
  - If A is an α-formula, create a new node l' as a child of l and label l' with

$$U(l') = (U(l) - \{A\}) \cup \{\alpha_1, \alpha_2\}$$

(In the case that A is $\neg\neg A_1$, there is no α_2.)

- If A is a β-formula, create two new nodes l' and l'' as children of l. Label l' and l'' with

$$U(l') = (U(l) - \{A\}) \cup \{\beta_1\}$$
$$U(l'') = (U(l) - \{A\}) \cup \{\beta_2\}$$

- If A is a γ-formula (such as $\forall x A_1(x)$), create a new node l' as a child of l and label l' with

$$U(l') = U(l) \cup \{\gamma(a)\}$$

where a is some constant that *appears* in $U(l)$. If $U(l)$ consists only of literals and γ-formulas and $U(l') = U(l)$ for all choices of a, then mark the leaf as *open* $\odot$.

- If A is a δ-formula (such as $\exists x A_1(x)$), create a new node l' as a child of l and label l' with

$$U(l') = (U(l) - \{A\}) \cup \{\delta(a)\}$$

where a is some constant that *does not appear* in $U(l)$.

□

The γ-rule for universal quantifiers is the only rule that does not replace a formula by one or more simpler formulas. Instead it adds a simpler formula while leaving the quantified formula as part of the label of the node. This is used to ensure that the universal quantifier will be applied to new constants that have not yet been introduced. If the only rule that applies is a γ-rule and the rule produces no new subformulas, then the branch is open. For example, $\{\forall x p(a,x)\}$ gives $\{p(a,a), \forall x p(a,x)\}$ and there is no point in continuing, as $(\{a\}, \{R\}, \{a\})$ with $(a,a) \in R$ is clearly a model.

Definition 5.27 A branch in a tableau is *closed* iff it terminates in a leaf marked closed. Otherwise (it is infinite or it terminates in a leaf marked open), the branch is *open*. □

Soundness

Theorem 5.28 (Soundness) *Let A be a formula in the predicate calculus and let $\mathcal{T}$ be a tableau for A. If $\mathcal{T}$ closes, then A is unsatisfiable.*

Proof: The theorem is a special case of the following more general statement that will be proved: if a subtree rooted at a node n of $\mathcal{T}$ closes, the set of formulas $U(n)$ is unsatisfiable. The proof is by induction on the height h of n. The proof of the base case for $h = 0$ and the inductive cases 1 and 2 for α- and β-rules is the same as the proof in the propositional calculus (see Section 2.6).

Case 3: The γ-rule was used. Then

$$U(n) = U_0 \cup \{\forall x A(x)\} \quad \text{and} \quad U(n') = U_0 \cup \{\forall x A(x), A(a)\}.$$

Assume that $U(n)$ is satisfiable and let $\mathcal{I}$ be a model for $U(n)$, so that $v_{\mathcal{I}}(A_i) = T$ for all $A_i \in U_0(n)$ and also $v_{\mathcal{I}}(\forall x A(x)) = T$. By Theorem 5.15, $v_{\mathcal{I}}(\forall x A(x)) = T$ iff $v_{\sigma_{\mathcal{I}}}(A(x)) = T$ for all assignments $\sigma_{\mathcal{I}}$, in particular for any assignment that assigns the same domain element to x that $\mathcal{I}$ does to a. (By the tableau construction, a appears in some formula of $U(n)$, so $\mathcal{I}$ in fact assigns it a domain element.) But $v_{\mathcal{I}}(A(a)) = T$ contradicts the inductive hypothesis that $U(n')$ is unsatisfiable.

Case 4: The δ-rule was used. Then

$$U(n) = U_0 \cup \{\exists x A(x)\} \quad \text{and} \quad U(n') = U_0 \cup \{A(a)\},$$

for some constant a which does not occur in a formula of $U(n)$. Assume that $U(n)$ is satisfiable and let

$$\mathcal{I} = (D, \{R_1, \ldots, R_n\}, \{d_1, \ldots, d_k\})$$

be a satisfying interpretation. Then $v_{\mathcal{I}}(\exists x A(x)) = T$, so for the relation R_i assigned to A and for some $d \in D$, $(d) \in R_i$. (This assumes that A is unary; the modification for n-ary predicates is immediate.) Extend $\mathcal{I}$ to the interpretation

$$\mathcal{I}' = (D, \{R_1, \ldots, R_n\}, \{d_1, \ldots, d_k, d\})$$

by assigning d to the constant a. $\mathcal{I}'$ is well-defined: since a does not occur in $U(n)$, it is not among the constants $\{a_1, \ldots, a_k\}$ already assigned to by $\mathcal{I}$. Then $v_{\mathcal{I}'}(A(a)) = T$, and since $v_{\mathcal{I}'}(U_0) = v_{\mathcal{I}}(U_0) = T$, we can conclude that $v_{\mathcal{I}'}(U(n')) = T$, contradicting the inductive hypothesis that $U(n')$ is unsatisfiable. ∎

Systematic construction and completeness

Soundness shows that if $\mathcal{T}$ closes, then A is unsatisfiable regardless of the order in which the rules were applied. However, as the examples above show, the construction of the tableau is not complete unless it is built systematically. The aim of the systematic construction is to ensure that rules are eventually applied to all formulas in the label of a node and, in the case of universally quantified formulas, that an instance is created for all constants that appear.

Algorithm 5.29 (Systematic construction of a semantic tableau)
Input: A formula A of the predicate calculus.
Output: A semantic tableau $\mathcal{T}$ for A: all branches are either infinite, or finite with leaves marked closed or open.

A semantic tableau for A is a tree $\mathcal{T}$ each node of which is labeled by a pair $W(n) = (U(n), C(n))$, where $U(n) = \{A_1, \ldots, A_k\}$ is a set of formulas and $C(n) = \{a_1, \ldots, a_m\}$

is a set of constants. $C(n)$ contains the list of constants that appear in the formulas in $U(n)$. Initially, $\mathcal{T}$ consists of a singe node, the root, labeled with $(\{A\}, \{a_1, \ldots, a_k\})$, where $\{a_1, \ldots, a_k\}$ is the set of constants that appear in A. If A has no constants, choose an arbitrary constant a and label the node with $(\{A\}, \{a\})$.

The tableau is built inductively by repeatedly *choosing* an unmarked leaf l labeled with $W(l) = (U(l), C(l))$, and applying one of the following rules *in the order given.*

- If $U(l)$ is a set of literals, check it contains a complementary pair of literals $\{p(a_1, \ldots, a_k), \neg p(a_1, \ldots, a_k)\}$. If so, mark the leaf *closed* $\times$; if not, mark the leaf as *open* $\odot$.
- If $U(l)$ is not a set of literals, *choose* a formula A in $U(l)$ which is an α-, β- or δ-formula.
  - If A is an α-formula, create a new node l' as a child of l and label l' with
    $$W(l') = (U(l'), C(l')) = (U(l) - \{A\}) \cup \{\alpha_1, \alpha_2\}, C(l)).$$
    (In the case that A is $\neg\neg A_1$, there is no α_2.)
  - If A is a β-formula, create two new nodes l' and l'' as children of l. Label l' and l'' with
    $$\begin{aligned} W(l') &= (U(l'), C(l')) = ((U(l) - \{A\}) \cup \{\beta_1\}, C(l)), \\ W(l'') &= (U(l''), C(l'')) = ((U(l) - \{A\}) \cup \{\beta_2\}, C(l)). \end{aligned}$$
  - If A is a δ-formula, create a new node l' as a child of l and label l' with
    $$W(l') = (U(l'), C(l')) = ((U(l) - \{A\}) \cup \{\delta(a)\}, C(l) \cup \{a\}),$$
    where a is some constant that does not appear in $U(l)$.
- Let $\{\gamma_1, \ldots, \gamma_m\} \subseteq U(l)$ be all the γ-formulas in $U(l)$ and let $C(l) = \{a_1, \ldots, a_k\}$. Create a new node l' as a child of l and label l' with
  $$W(l') = (U(l'), C(l')) = (U(l) \cup \cup_{i=1,j=1}^{i=m,j=k} \{\gamma_i(a_j)\}, C(l)).$$
  If $U(l)$ consists only of literals and γ-formulas and $U(l') = U(l)$, then mark the leaf as *open* $\odot$.

□

In the systematic construction, α-, β- and δ-formulas are 'used up' when a rule is applied to them. Eventually a leaf must be labeled with formulas that are either literals or γ-formulas. To ensure that all universal formulas are eventually instantiated with every new constant, the γ-rule makes all these instantiations at once.

Lemma 5.30 *Let b be an open branch of a systematic tableau, n a node on b, and A a formula in $U(n)$. Then for some node m which is a descendant of n on b, a rule is applied to A. Furthermore, if A is a γ-formula and $a \in C(n)$, then $\gamma(a) \in U(m')$.*

Proof: Exercise. ∎

It is important to note that the algorithm is *not* a search procedure for a satisfying interpretation, because it may choose to infinitely expand one branch. Semantic tableaux in the predicate calculus can only be used to prove the validity of a formula by showing that a tableau for its negation closes. Since all branches close in a closed tableau, the order that the rules are applied doesn't matter. The systematic construction is needed for the proof of completeness.

Definition 5.31 Let U be a set of formulas in the predicate calculus. U is a *Hintikka set* iff the following conditions hold for all formulas $A \in U$:

1. If A is a closed atomic formula, either $A \notin U$ or $\neg A \notin U$.
2. If A is an α-formula, $\alpha_1 \in U$ and $\alpha_2 \in U$.
3. If A is a β-formula, $\beta_1 \in U$ or $\beta_2 \in U$.
4. If A is a γ-formula, $\gamma(a) \in U$ for *all* constants a appearing in formulas in U.
5. If A is a δ-formula, $\delta(a) \in U$ for *some* constant a.

□

Theorem 5.32 (Hintikka's Lemma) *Let b be an open branch of a systematic tableau and $U = \cup_{n \in b} U(n)$. The U is a Hintikka set.*

Proof: Let $A \in U$. Since b is open, if A is a closed atomic formula, $\neg A \notin U$, so Condition 1 holds. If A is not atomic and not a γ-formula, by Lemma 5.30 eventually a rule is applied to A, and Conditions 2, 3 and 5 hold. Let $A \in U(n)$ be a γ-formula and let a be an arbitrary constant appearing in a formula of U. Then $a \in C(m)$ for some node m in b. By construction, the set of γ-formulas and the set of constants are non-decreasing along the path, so $A \in U(k)$ and $a \in C(k)$ for $k = \max(n, m)$. By Lemma 5.30, $\gamma(a) \in U(k') \subseteq U$, for some $k' > k$. ∎

Lemma 5.33 *Let U be a Hintikka set. Then there is a model for U.*

Proof: Let $\mathcal{A} = \{a_1, \ldots\}$ be the set of constants appearing in formulas of U. Define an interpretation $\mathcal{I}$ as follows. The domain is the same as the set of symbols $\{a_1, \ldots\}$ used as constants in U; the domain element a_i is assigned to the constant a_i. For each n-ary predicate letter p_i in U, define an n-ary relation R_i by:

$$
\begin{array}{ll}
(a_1,\ldots,a_n) \in R_i & \text{if} \quad p(a_1,\ldots,a_n) \in U, \\
(a_1,\ldots,a_n) \notin R_i & \text{if} \quad \neg p(a_1,\ldots,a_n) \in U, \\
(a_1,\ldots,a_n) \in R_i & \text{otherwise.}
\end{array}
$$

The relations are well-defined by Condition 1 of the definition of Hintikka sets. We leave as an exercise to show that $\mathcal{I} \models A$ for all $A \in U$ by induction on the structure of A using the conditions defining a Hintikka set. ∎

Theorem 5.34 (Completeness) *Let A be a valid formula. Then the systematic semantic tableau for $\neg A$ closes.*

Proof: Let A be a valid formula and suppose that the systematic tableau for $\neg A$ does not close. By Lemma 5.32, there is an open branch b such that $U = \cup_{n \in b} U(n)$ is a Hintikka set. By Lemma 5.33, there is a model $\mathcal{I}$ for U. But $\neg A \in U$ so $\mathcal{I} \models \neg A$ contradicting the assumption that A is valid. ∎

5.6 Implementation^P

In this section, we extend the program to construct a tableau for the propositional calculus to the predicate calculus.

`check_closed` has to be modified to prevent unification of atomic formulas; instead, syntactical identity has to be checked using the Prolog operator ==.

```
check_closed(Fmls) :-
  member(F1, Fmls), member(neg F2, Fmls), F1 == F2.
```

To implement the systematic search, the rules are ordered so that α-, β- and δ-rules are performed before attempting a γ-rule. The tableau predicate has an extra argument to hold the list of constants `C`. This is updated whenever a δ-rule is used.

```
extend_tableau(t(Fmls, Left, empty, C)) :-
  delta_rule(Fmls, Fmls1, Const), !,
  Left = t(Fmls1, _, _, [Const|C]),
  extend_tableau(Left).

extend_tableau(t(Fmls, Left, empty, C)) :-
  gamma_rule(Fmls, Fmls1, C), !,
  Left = t(Fmls1, _, _, C),
  extend_tableau(Left).
```

In the procedure for `delta_rule`, the Prolog procedure `gensym` is used to generate a new constant symbol. `instance(A1,A2,X,C)` (see source code below) returns in `A2` the instance of `A1` that can be obtained by replacing the variable `X` by the constant `C`.


```
delta_rule(Fmls, [A2 | Fmls1], C) :-
  member(A, Fmls),
  delta(A, X, A1), !,
  gensym(a, C),
  instance(A1, A2, X, C),
  delete(Fmls, A, Fmls1).
```

`delta` recognizes the two forms of a δ-formula and returns the subformula.

```
delta(ex(X, A1), X, A1).
delta(neg all(X, A1), X, neg A1).
```

The application of a γ-rule is rather complicated. First we recognize that `A` is a γ-formula, but throw away the dummy instance created. Next, `gamma_all` is called to create in `AList` a list of all the instances formed with `A` and the current constant list `C`. Then `A` is moved to the end of the list of formulas so other γ-formulas will be systematically processed, and the new instances are placed at the *front* of the list so that non-γ-rules will be used if possible. `list_to_set` is called to remove duplicates.

```
gamma_rule(Fmls, Fmls4, C) :-
  member(A, Fmls),
  gamma(A, _, dummy), !,
  gamma_all(C, A, AList),
  delete(Fmls, A, Fmls1),
  append(Fmls1, [A], Fmls2),
  append(AList, Fmls2, Fmls3),
  list_to_set(Fmls3, Fmls4).
```

`gamma` recognizes the γ-formulas and returns instances.

```
gamma(all(X, A1),     A2,      C) :- instance(A1, A2, X, C).
gamma(neg ex(X, A1), neg A2, C) :- instance(A1, A2, X, C).
```

`gamma_all(C, A, AList)` applies the γ-rule to `A` with all of the constants in the constant list `C` and returns the list of formulas in `AList`.

```
gamma_all([C | Rest], A, [A1 | AList]) :-
  gamma(A, A1, C),
  gamma_all(Rest, A, AList).
gamma_all([], _, []).
```

To create an instance, the formula is recursively traversed until an atomic formula is reached; then the substitution is performed.

```
instance(all(X, A), all(X, A1), Y, C) :-
  instance(A, A1, Y, C).
instance(ex(X, A), ex(X, A1), Y, C) :-
  instance(A, A1, Y, C).

instance(A v B, A1 v B1, X, C) :-
  instance(A, A1, X, C), instance(B, B1, X, C).
% Similarly for the other Boolean operators.

instance(A, A1, X, C) :-
  A =..  [F | Vars],
  subst_constant(X, C, Vars, Vars1),
  A1 =..  [F | Vars1].
```

To implement substitution, the operator == must be used to prevent substitution to a variable that might just be unifiable with the variable we want to substitute for.

```
subst_constant(X1, C, [X2 | Tail], [C | Tail1]) :-
  X1 == X2, !,
  subst_constant(X1, C, Tail, Tail1).
subst_constant(X, C, [Y | Tail], [Y | Tail1]) :- !,
  subst_constant(X, C, Tail, Tail1).
subst_constant(_, _, [], []).
```

5.7 Finite and infinite models*

(Read this section and the next one only after studying Section 7.1.)

Definition 5.35 A formula of the predicate calculus is *pure* if it contains no function symbols (including constants which are 0-ary function symbols). □

From the proof of the completeness theorem, we can extract more information on the structure of models. Here we survey some of these results that form the basis of an advanced topic in mathematical logic called *model theory*.

Definition 5.36 A set of formulas U has the *finite model property* iff: U is satisfiable iff it is satisfiable in an interpretation whose domain is a finite set. □

Theorem 5.37 *Let U be a set of pure formulas of the form*

$$\exists x_1 \cdots \exists x_k \forall y_1 \cdots \forall y_l A(x_1, \ldots, x_k, y_1, \ldots, y_l),$$

where A does not contain any quantifiers. Then U has the finite model property.

Proof: During the construction of a tableau for U, once the δ-rules have been applied to the existential quantifiers, no more existential quantifiers remain. Thus the set of constants will be finite and the tableau will terminate once all substitutions using these constants have been made for universal quantifiers. ∎

Theorem 5.38 (Löwenheim) *If a formula is satisfiable then it is satisfiable in a countable domain.*

Proof: The domain D constructed in the proof of completeness is a countable set. ∎

This can be generalized by noting that a tableau can be constructed for a countably infinite set of formulas $U = \{A_0, A_1, A_2, \ldots\}$. Commence the construction of a semantic tableau from the root labeled $(\{A_0\}, C_0)$, where C_0 is the set of constants in A_0 or $\{a\}$ if there are none. Let n be a node to which a rule is applied during the construction of the semantic tableau to create a node n' (and similarly for n'' in a β-rule). Label n' by $(U(n') \cup \{A_n\}, C(n'))$, that is, when creating the n'th node on a branch, throw in the n'th formula from the sequence. If the tableau does not close, eventually, every A_i will appear on the branch, and the labels will form a Hintikka set. Hintikka's Lemma and completeness can be proved as before giving:

Theorem 5.39 (Löwenheim–Skolem) *If a countable set of formulas is satisfiable then it is satisfiable in a countable domain.*

Uncountable sets such as the real numbers can be described by countably many axioms (formulas). Thus formulas that describe real numbers also have a countable model in addition to the standard uncountable model! Such models are called *non-standard* models and are important in the theory of mathematical logic.

As in the propositional calculus (Theorem 3.43), compactness holds.

Theorem 5.40 (Compactness) *Let U be a countable set of formulas. If all finite subsets of U are satisfiable then so is U.*

5.8 Decidability*

Alonzo Church, building on the work of Alan Turing, proved in 1936 that there is no decision procedure for validity in the predicate calculus. In this section, we will prove a version of Church's Theorem. Then we will survey theorems that show that certain classes of formulas, which are defined by restricting the syntactical structure of a formula, do have decision procedures.

Undecidability of the predicate calculus

To show that a class C of problems is undecidable you can either show it directly, or—what is usually easier—you can *reduce* a known class C' of undecidable problems to C. A reduction is an algorithm that produces for every problem $P' \in C'$, a problem $P \in C$ such that an answer to P given by a decision procedure is also an answer to P'. Since C' is assumed undecidable, this proves that C is also undecidable.

It is undecidable whether a Turing machine will halt if started on a blank tape (Minsky 1967, Section 8.3.3), so we prove Church's Theorem by giving an algorithm which produces a formula S_M in the predicate calculus for every Turing machine M, such that S_M is valid iff M halts on a blank tape. To simplify the proof, we work with two-register machines rather than directly with Turing machines.

Definition 5.41 A *two-register machine* M consists of two registers x and y which can hold natural numbers, and a program $P = \{L_0, \ldots, L_n\}$ which is a list of instructions. L_n is the instruction `halt`, and for $0 \leq i < n$, L_i is one of:

- `r := r + 1`, for `r` $\in$ {x,y}.
- `if r = 0 then goto Lj else r := r - 1`, for `r` $\in$ {x,y}, $0 \leq j \leq n$.

An execution sequence of M is a sequence of states $s_k = (L_i, x, y)$, where L_i is the current instruction, and x, y are the contents of x and y. s_{k+1} is obtained from s_k by executing L_i. The initial state is $s_0 = (L_0, m, 0)$ for some m. If for some k, $s_k = (L_n, x, y)$, the computation of M has halted and M has computed $y = f(m)$. □

Theorem 5.42 *Given a Turing machine M that computes a function f, a two-register machine can be constructed to compute the same function f.*

Proof: Minsky (1967, Section 14.1), Hopcroft & Ullman (1979, Section 7.8). ∎

Two-register machines are even more impractical than Turing machines: the registers can contain extremely large natural numbers because the computation encodes the contents of the unbounded tape of a Turing machine in a single number. It may come as a surprise that such a simple model can (according to the Church-Turing thesis) compute all computable functions.

Theorem 5.43 (Church) *Validity in the predicate calculus is undecidable.*

Proof: For every two-register machine M, we construct a formula S_M such that S_M is valid iff M terminates when started in the state $(L_0, 0, 0)$:

$$S_M = \left(\bigwedge_{i=0}^{n} S_i \wedge p_0(a,a) \right) \rightarrow \exists z_1 \exists z_2 p_n(z_1, z_2).$$

S_i is defined by cases of the instruction L_i:

L_i	S_i
`x := x + 1`	$\forall x \forall y (p_i(x,y) \rightarrow p_{i+1}(s(x), y))$
`y := y + 1`	$\forall x \forall y (p_i(x,y) \rightarrow p_{i+1}(x, s(y)))$

`if x = 0`
`then goto Lj`
`else x := x - 1`
	$\forall x (p_i(a,x) \rightarrow p_j(a,x)) \wedge$
	$\forall x \forall y (p_i(s(x),y) \rightarrow p_{i+1}(x,y)))$

`if y = 0`
`then goto Lj`
`else y := y - 1`
	$\forall x (p_i(x,a) \rightarrow p_j(x,a)) \wedge$
	$\forall x \forall y (p_i(x,s(y)) \rightarrow p_{i+1}(x,y)))$

Suppose that the computation $s_0, \ldots, s_m$ of M halts and let $\mathcal{I}$ be an arbitrary interpretation for S_M. If $v_\mathcal{I}(S_i) = F$ (for $1 \leq i \leq n$) or $v_\mathcal{I}(p_0(a,a)) = F$, then trivially $v_\mathcal{I}(S_M) = T$, so we need only consider interpretations that satisfy the antecedent of S_M. We show by induction on k that $v_\mathcal{I}(\exists z_1 \exists z_2 p_k(z_1, z_2)) = T$, where p_k is the predicate associated with the label L_k in state s_k. If $k = 0$, the result follows from $p_0(a,a) \rightarrow \exists z_1 \exists z_2 p_0(z_1, z_2)$. If $k > 0$, the result follows by induction by cases according to the instruction at s_{k-1}. Let us work through the details for `x:=x+1`: $v_\mathcal{I}(\forall x \forall y (p_{k-1}(x,y) \rightarrow p_k(s(x), y))) = T$ by assumption, and by the inductive hypothesis, $v_\mathcal{I}(\exists z_1 \exists z_2 p_{k-1}(z_1, z_2)) = T$, from which $v_\mathcal{I}(\exists z_1 \exists z_2 p_k(s(z_1), z_2))) = T$ follows by reasoning in the predicate calculus. From $\exists x A(f(x)) \rightarrow \exists x' A(x')$, we can conclude $v_\mathcal{I}(\exists z_1' \exists z_2 p_k(z_1', z_2)) = T$. By induction this holds for all k. Since M halts, in the final state s_m, $L_m = L_n$ the halt instruction, so $v_\mathcal{I}(\exists z_1' \exists z_2 p_n(z_1', z_2)) = T$ and $v_\mathcal{I}(S_M) = T$. Since $\mathcal{I}$ was arbitrary, S_M is valid.

Conversely, suppose that S_M is valid, and consider the interpretation

$$\mathcal{I} = (\mathcal{N}, \{P_0, \ldots, P_n\}, \{succ\}, \{0\}),$$

where *succ* is the successor function on $\mathcal{N}$, and $(x,y) \in P_i$ iff (L_i, x, y) is reached by the register machine when started in $(L_0, 0, 0)$. We show that the antecedent of S_M is true in $\mathcal{I}$. The initial state is $(L_0, 0, 0)$, so $(a,a) \in P_0$ and $v_\mathcal{I}(p_0(a,a)) = T$. Assume as an inductive hypothesis that if the computation has reached L_i, it has done so in a computation of length $k-1$ in state $s_{k-1} = (L_i, x_i, y_i)$, so $(x_i, y_i) \in P_i$. The proof is by cases on the instruction L_i; for example, for `x:=x+1`: the computation can reach the state $s_k = (L_{i+1}, succ(x_i), y_i)$, so $v_\mathcal{I}(S_i) = T$. Since S_M is assumed valid, $v_\mathcal{I}(\exists z_1 \exists z_2 p_n(z_1, z_2)) = T$ and $v_\mathcal{I}(p_n(m_1, m_2) = T$ for some natural numbers m_1, m_2. Thus M halts and computes $m_2 = f(0)$. ∎

Church's Theorem holds even if the structure of the formulas is restricted:

- The formulas contain only binary predicate symbols, one constant and one unary function symbol. This follows from the structure of S_M in the proof.

- The formulas are written as Prolog programs. S_M is of the form of a logic program (set of clauses) and a query. With a bit more work, it can be proved (Nerode & Shore 1997, Chapter III, Section 8) that Prolog programs are undecidable, even with no limitation on the underlying implementation.
- The formulas are pure. See Mendelson (1997, Chapter 3, Section 6).

Solvable cases of the decision problem

Theorem 5.44 *There is a decision procedure for validity of the class of pure formulas in PCNF whose prefixes are of one of the following forms (where $m, n \geq 0$):*

$$\forall x_1 \cdots \forall x_n \; \exists y_1 \cdots \exists y_m$$
$$\forall x_1 \cdots \forall x_n \; \exists y \; \forall z_1 \cdots \forall z_m$$
$$\forall x_1 \cdots \forall x_n \; \exists y_1 \exists y_2 \; \forall z_1 \cdots \forall z_m.$$

Proof: Dreben & Goldfarb (1979). ∎

These classes are conveniently abbreviated $\forall^* \exists^*$, $\forall^* \exists \forall^*$, $\forall^* \exists \exists \forall^*$. Since a decision procedure for satisfiability is the dual of a decision procedure for validity, one also sees the theorem expressed as: satisfiable is decidable for the classes $\exists^* \forall^*$, $\exists^* \forall \exists^*$, $\exists^* \forall \forall \exists^*$. The theorem cannot be improved upon.

Theorem 5.45 *The classes of formulas in pure PCNF defined by the prefixes $\exists \exists \exists \forall$, $\exists \forall \exists$ have no decision procedures for validity, even if the matrices are restricted to binary predicate letters.*

Proof: Lewis (1979). ∎

Theorem 5.46 *There is a decision procedure for satisfiability of PCNF formulas A if the matrix of A is of one of the forms:*

1. *All conjunctions are* unit *conjunctions of single literals.*
2. *All conjunctions are either positive unit conjunctions (i.e. atomic formulas) or conjunctions consisting entirely of negative literals.*
3. *All atomic formulas are monadic, that is, all predicate letters are unary.*

Proof: Dreben & Goldfarb (1979). ∎

5.9 Exercises

1. Find an interpretation which falsifies $\exists x p(x) \rightarrow p(a)$.
2. Prove that (some of) the formulas in Figure 5.2 are valid, especially

$$\exists x(A(x) \rightarrow B(x)) \leftrightarrow (\forall x A(x) \rightarrow \exists x B(x)),$$
$$(\exists x A(x) \rightarrow \forall x B(x)) \rightarrow \forall x(A(x) \rightarrow B(x)),$$
$$\forall x(A(x) \vee B(x)) \rightarrow (\forall x A(x) \vee \exists x B(x)),$$
$$\forall x(A(x) \rightarrow B(x)) \rightarrow (\exists x A(x) \rightarrow \exists x B(x)).$$

3. For each formula in Figure 5.2 that is an implication, prove that the converse is not valid by giving a falsifying interpretation.
4. For each of the following formulas, either prove that it is valid or give a falsifying interpretation.

$$\exists x \forall y((p(x,y) \wedge \neg p(y,x)) \rightarrow (p(x,x) \leftrightarrow p(y,y))),$$
$$\forall x \forall y \forall z(p(x,x) \wedge (p(x,z) \rightarrow (p(x,y) \vee p(y,z)))) \rightarrow \exists y \forall z p(y,z).$$

5. Prove Theorem 5.15 on the relationship between a non-closed formula and its closure.
6. Complete the semantic tableau construction for the negation of

$$\forall x(p(x) \vee q(x)) \rightarrow (\forall x p(x) \vee \forall x q(x)).$$

7. Prove that the formula $(\forall x p(x) \rightarrow \forall x q(x)) \rightarrow \forall x(p(x) \rightarrow q(x))$ is not valid by constructing a semantic tableau for its negation.
8. Complete the proof that every Hintikka set has a model (Lemma 5.33).
9. ^{P} The implementation of the construction of semantic tableaux is difficult because of the need to ensure that variables in the formula are not accidently unified. Write an alternate implementation that represents variables as Prolog atoms `v(n)`.
10. * Prove the Löwenheim-Skolem Theorem (5.39) using the construction of semantic tableaux for infinite sets of formulas.
11. * A closed pure formula A is *n-condensable* iff every unsatisfiable conjunction of instances of the matrix of A contains an unsatisfiable subconjunction made up of n or fewer instances.
    - Let A be a PCNF formula whose matrix is a conjunction of literals. Prove that A is 2-condensable.

- Let A be a PCNF formula whose matrix is a conjunction of positive literals and disjunctions of negative literals. Prove that A is $n+1$-condensable, where n is the maximum number of literals in a conjunct.

12. * Prove Church's Theorem by reducing Post's Correspondence Problem to validity in the predicate calculus.

6 Predicate Calculus: Deductive Systems

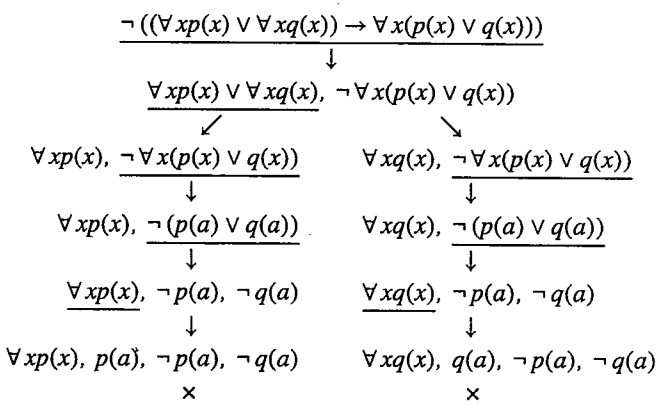
As with the propositional calculus, the Gentzen system will be presented as semantic tableaux turned upside-down so that completeness is immediate; then we will use the Gentzen system to prove completeness of a Hilbert system.

6.1 The Gentzen system $\mathcal{G}$

Here is a closed semantic tableau for the negation of the valid formula

$$(\forall x p(x) \vee \forall x q(x)) \rightarrow \forall x (p(x) \vee q(x)),$$

where we have underlined the formulas to which rules are applied and the sets of constants $C(n)$ in the labels are implicit.



If we turn the tree upside down and for every node n replace $U(n)$, the label of n, by $\bar{U}(n)$, the set of complements of the formulas in $U(n)$, we obtain a Gentzen proof for the formula. Again, the formulas that are underlined are those to which a rule is applied at each step.

$$\underline{\neg\forall x p(x),\ \neg p(a),\ p(a),\ q(a)} \qquad \underline{\neg\forall x q(x),\ \neg q(a),\ p(a),\ q(a)}$$
$$\downarrow \qquad\qquad \downarrow$$
$$\neg\forall x p(x),\ \underline{p(a),\ q(a)} \qquad \neg\forall x q(x),\ \underline{p(a),\ q(a)}$$
$$\downarrow \qquad\qquad \downarrow$$
$$\neg\forall x p(x),\ \underline{p(a) \vee q(a)} \qquad \neg\forall x q(x),\ \underline{p(a) \vee q(a)}$$
$$\downarrow \qquad\qquad \downarrow$$
$$\underline{\neg\forall x p(x)},\ \forall x(p(x) \vee q(x)) \qquad \underline{\neg\forall x q(x)},\ \forall x(p(x) \vee q(x))$$
$$\searrow \qquad\qquad \swarrow$$
$$\underline{\neg(\forall x p(x) \vee \forall x q(x)),\ \forall x(p(x) \vee q(x))}$$
$$\downarrow$$
$$(\forall x p(x) \vee \forall x q(x)) \rightarrow \forall x(p(x) \vee q(x))$$

Definition 6.1 The *Gentzen system* $\mathcal{G}$ is a deductive system: an *axiom* is any set of formulas U containing a complementary pair of literals; the rules of inference are the rules given for α- and β-formulas in Section 3.2, and the rules for γ- and δ-formulas:

$$\frac{U \cup \{\gamma, \gamma(a)\}}{U \cup \{\gamma\}} \qquad \frac{U \cup \{\delta(a)\}}{U \cup \{\delta\}},$$

provided that a does not occur in any formula of U in an application of the δ-rule. The classification of formulas is given in the following tables.

γ	$\gamma(a)$
$\exists x A(x)$	$A(a)$
$\neg\forall x A(x)$	$\neg A(a)$

δ	$\delta(a)$
$\forall x A(x)$	$A(a)$
$\neg\exists x A(x)$	$\neg A(a)$

□

The γ-rule can be read: if an existential formula and some instantiation of it are true, then the instantiation is redundant. The δ-rules formalizes the following frequently used method of mathematical reasoning:

> Let a be an *arbitrary* constant, and prove $A(a)$. Since a was arbitrary, we have proved $\forall x A(x)$.

It is essential that a not appear in some other formula in the set which would impose restrictions on the possible interpretations of a.

Example 6.2 The non-trivial proof of $\exists x \forall y p(x, y) \rightarrow \forall y \exists x p(x, y)$ in Figure 6.1 is carefully structured to apply three γ-rules which are not restricted before applying two δ-rules. By then, there exist only single occurrences of the constants a and b so that the proviso is satisfied. □

Theorem 6.3 (Soundness and completeness) *Let U be a set of formulas. There is a Gentzen proof for U if and only if there is a closed semantic tableau for $\bar{U}$.*

Proof: Exercise. ∎

$$\underline{\neg\forall y p(a,y),\ \neg p(a,b)},\ \neg p(a,a),\ \exists x p(x,b),\ p(a,b)$$
$$\downarrow$$
$$\underline{\neg\forall y p(a,y),\ \neg p(a,a)},\ \exists x p(x,b),\ p(a,b)$$
$$\downarrow$$
$$\neg\forall y p(a,y),\ \underline{\exists x p(x,b),\ p(a,b)}$$
$$\downarrow$$
$$\neg\forall y p(a,y),\ \underline{\exists x p(x,b)}$$
$$\downarrow$$
$$\underline{\neg\forall y p(a,y)},\ \forall y \exists x p(x,y)$$
$$\downarrow$$
$$\underline{\neg\exists x \forall y p(x,y)},\ \forall y \exists x p(x,y)$$
$$\downarrow$$
$$\exists x \forall y p(x,y) \rightarrow \forall y \exists x p(x,y)$$

Figure 6.1 Gentzen proof that orders applications of γ- and δ-rules

6.2 The Hilbert system $\mathcal{H}$

The Hilbert system $\mathcal{H}$ for the propositional calculus is extended to the predicate calculus by adding two axioms and a rule of inference. The proof system is for the predicate calculus with the universal quantifier only; the existential quantifier is introduced by defining $\exists x A(x)$ as $\neg\forall x \neg A(x)$.

Definition 6.4 The axioms of the Hilbert system $\mathcal{H}$ for the predicate calculus are the three axioms of the propositional calculus together with:

Axiom 4 $\vdash \forall x A(x) \rightarrow A(a)$.

Axiom 5 $\vdash \forall x(A \rightarrow B(x)) \rightarrow (A \rightarrow \forall x B(x))$,
provided that x is not free in A.

The rules inference are *MP* and *Generalization*: $\dfrac{\vdash A(a)}{\vdash \forall x A(x)}$. □

Axioms 1–3 include instances where the formulas are in the predicate calculus. Thus $\vdash \forall x p(x) \rightarrow (\exists y \exists z q(y, z, a) \rightarrow \forall x p(x))$ is justified by Axiom 1.

Note the difference between Axiom 4 and Generalization. Axiom 4 means that *any* occurrence of $\forall x A(x)$ can be replaced by $A(a)$ for any a. Generalization means that if a occurs in a formula, we may bind *all* occurrences of a with the quantifier. The justification is that since the choice of a is arbitrary, that is the same as saying that $A(x)$ is true for *all* assignments to x in an interpretation.

There is a technical problem with Generalization in the presence of assumptions. Suppose that we apply Generalization to $A(a) \vdash A(a)$ to obtain $A(a) \vdash \forall x A(x)$. In the

interpretation $(\mathcal{Z}, \{even(x)\}, \{2\})$, $A(a)$ is true but $\forall xA(x)$ is not, which means that Generalization is not sound as it transforms the logical consequence $A(a) \models A(a)$ into $A(a) \not\models \forall xA(x)$. The Generalization rule must be modified as follows.

Rule 6.5 (Generalization)

$$\frac{U \vdash A(a)}{U \vdash \forall xA(x)},$$

provided that a does not appear in U. □

With this modification, the deduction rule can be defined and the theorem proved.

Rule 6.6 (Deduction rule)

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

□

Theorem 6.7 (Deduction Theorem) *The deduction theorem is a sound derived rule.*

Proof: The proof for the propositional calculus (Theorem 3.14) must be modified to take into account the new axioms and Generalization. The modification for the additional axioms is trivial. Suppose now that $U \cup \{A\} \vdash \forall xB(x)$ at line j were deduced from $U \cup \{A\} \vdash B(a)$ at line i by Generalization.

i'.	$U \vdash A \to B(a)$	Inductive hypothesis, i
$i'+1$.	$U \vdash \forall x(A \to B)$	Generalization, i'
$i'+2$.	$U \vdash \forall x(A \to B) \to (A \to \forall xB)$	Axiom 5
$i'+3$.	$U \vdash A \to \forall xB$	MP, $i'+1$, $i'+2$

By Rule 6.5, a did not appear in $U \cup \{A\}$, justifying the use of Generalization in line $i'+1$ and Axiom 5 in line $i'+2$. ∎

The equivalence of $\mathcal{H}$ and $\mathcal{G}$ is easily shown. We now prove that any application of the γ- and δ-rules in $\mathcal{G}$ can be simulated by a proof in $\mathcal{H}$. Note that the proviso for the δ-rule justifies the use of Axiom 5.

(From now on, the notation PC is used to indicate that the inference may be justified by the axioms and rules of the propositional calculus. The reader is assumed to be sufficiently experienced in these techniques by now to fill in the details.)

Theorem 6.8 *Any use of the γ-rule can be simulated in $\mathcal{H}$.*

Proof: We show that $U \vee \neg \forall xA(x)$ can be deduced from $U \vee \neg \forall xA(x) \vee \neg A(a)$.

1.	$\vdash \forall xA(x) \to A(a)$	Axiom 4
2.	$\vdash \neg \forall xA(x) \vee A(a)$	PC 1
3.	$\vdash U \vee \neg \forall xA(x) \vee A(a)$	PC 2
4.	$\vdash U \vee \neg \forall xA(x) \vee \neg A(a)$	Assumption
5.	$\vdash U \vee \neg \forall xA(x)$	PC 3,4

∎

Theorem 6.9 *Any use of the δ-rule can be simulated in $\mathcal{H}$.*

Proof: We show that $U \vee \forall xA(x)$ can be deduced from $U \vee A(a)$.

1.	$\vdash U \vee A(a)$	Assumption
2.	$\vdash \neg U \to A(a)$	PC 1
3.	$\vdash \forall x(\neg U \to A(x))$	Gen. 2
4.	$\vdash \forall x(\neg U \to A(x)) \to (\neg U \to \forall xA(x))$	Axiom 5
5.	$\vdash \neg U \to \forall xA(x)$	PC 3,4
6.	$\vdash U \vee \forall xA(x)$	PC 5

∎

Gentzen proofs of the Hilbert axioms are left as an exercise. Generalization is a special case of the δ-rule, where the proviso on the δ-rule in the Gentzen system appears as the restriction on Generalization applied to $U \vdash A$.

The previous discussion justifies the following theorem.

Theorem 6.10 (Soundness and completeness) *The Hilbert system $\mathcal{H}$ is sound and complete.*

Axiom 4 and *MP* justify the following derived rule.

Rule 6.11 (Axiom 4)

$$\frac{U \vdash \forall xA(x)}{U \vdash A(a)}.$$

□

We now give a series of theorems and proofs in the system. The first two are elementary theorems using existential quantifiers.

Theorem 6.12 $\vdash A(a) \to \exists xA(x)$.

Proof:

1.	$\vdash \forall x \neg A(x) \to \neg A(a)$	Axiom 4
2.	$\vdash A(a) \to \neg \forall x \neg A(x)$	PC 1
3.	$\vdash A(a) \to \exists xA(x)$	Definition $\exists$

∎

Theorem 6.13 $\vdash \forall xA(x) \to \exists xA(x)$.

Proof:

1.	$\forall xA(x) \vdash \forall xA(x)$	Assumption
2.	$\forall xA(x) \vdash A(a)$	Axiom 4
3.	$\forall xA(x) \vdash A(a) \to \exists xA(x)$	Theorem 6.12
4.	$\forall xA(x) \vdash \exists xA(x)$	PC 2,3
5.	$\vdash \forall xA(x) \to \exists xA(x)$	Deduction

Theorem 6.14 $\vdash \forall x(A(x) \to B(x)) \to (\forall xA(x) \to \forall xB(x))$.

Proof:

1.	$\forall x(A(x) \to B(x)), \forall xA(x) \vdash \forall xA(x)$	Assumption
2.	$\forall x(A(x) \to B(x)), \forall xA(x) \vdash A(a)$	Axiom 4
3.	$\forall x(A(x) \to B(x)), \forall xA(x) \vdash \forall x(A(x) \to B(x))$	Assumption
4.	$\forall x(A(x) \to B(x)), \forall xA(x) \vdash A(a) \to B(a)$	Axiom 4
5.	$\forall x(A(x) \to B(x)), \forall xA(x) \vdash B(a)$	PC 2,4
6.	$\forall x(A(x) \to B(x)), \forall xA(x) \vdash \forall xB(x)$	Gen. 5
7.	$\forall x(A(x) \to B(x) \vdash \forall xA(x) \to \forall xB(x))$	Deduction
8.	$\vdash \forall x(A(x) \to B(x)) \to (\forall xA(x) \to \forall xB(x))$	Deduction

Rule 6.15 (Generalization)

$$\frac{\vdash A(x) \to B(x)}{\vdash \forall xA(x) \to \forall xB(x)}.$$

The next theorem was previously proved in the Gentzen system. Note that the proviso of Axiom 5 (x is not free in $\neg \forall y \exists xA(x, y)$) parallels the proviso in the δ-rule.

Theorem 6.16 $\vdash \exists x \forall yA(x, y) \to \forall y \exists xA(x, y)$.

Proof:

1.	$\vdash A(a, b) \to \exists xA(x, b)$	Theorem 6.12
2.	$\vdash \forall yA(a, y) \to \forall y \exists xA(x, y)$	Gen 1
3.	$\vdash \neg \forall y \exists xA(x, y) \to \neg \forall yA(a, y)$	PC 2
4.	$\vdash \forall x(\neg \forall y \exists xA(x, y) \to \neg \forall yA(x, y))$	Gen. 3
5.	$\vdash \neg \forall y \exists xA(x, y) \to \forall x \neg \forall yA(x, y)$	Axiom 5
6.	$\vdash \neg \forall x \neg \forall yA(x, y) \to \forall y \exists xA(x, y)$	PC 5
7.	$\vdash \exists x \forall yA(x, y) \to \forall y \exists xA(x, y)$	Definition of $\exists$

Theorem 6.17 $\vdash \forall x(A \to B(x)) \leftrightarrow (A \to \forall xB(x))$.

Proof:

1.	$A \to \forall xB(x) \vdash A \to \forall xB(x)$	Assumption
2.	$A \to \forall xB(x) \vdash \forall xB(x) \to B(a)$	Axiom 4
3.	$A \to \forall xB(x) \vdash A \to B(a)$	PC 1,2
4.	$A \to \forall xB(x) \vdash \forall x(A \to B(x))$	Gen. 3
5.	$\vdash (A \to \forall xB(x)) \to \forall x(A \to B(x))$	-Deduction
6.	$\vdash \forall x(A \to B(x)) \to (A \to \forall xB(x))$	Axiom 5
7.	$\vdash \forall x(A \to B(x)) \leftrightarrow (A \to \forall xB(x))$	PC 5,6

Corollary 6.18 $\vdash \exists x(A \to B(x)) \leftrightarrow (A \to \exists xB(x))$.

The name of a bound variable is not important and can be changed as convenient:

Theorem 6.19 $\vdash \forall xA(x) \leftrightarrow \forall yA(y)$.

Proof:

1.	$\vdash \forall xA(x) \to A(a)$	Axiom 4
2.	$\vdash \forall y(\forall xA(x) \to A(y))$	Gen. 1
3.	$\vdash \forall xA(x) \to \forall yA(y)$	Axiom 5
4.	$\vdash \forall yA(y) \to \forall xA(x)$	Similarly
5.	$\vdash \forall xA(x) \leftrightarrow \forall yA(y)$	PC 3,4

The following theorem shows a non-obvious relation between the quantifiers.

Theorem 6.20 $\vdash \forall x(A(x) \to B) \leftrightarrow (\exists xA(x) \to B)$.

Proof:

1.	$\forall x(A(x) \to B) \vdash \forall x(A(x) \to B)$	Assumption
2.	$\forall x(A(x) \to B) \vdash \forall x(\neg B \to \neg A(x))$	Exercise
3.	$\forall x(A(x) \to B) \vdash \neg B \to \forall x \neg A(x)$	Axiom 5
4.	$\forall x(A(x) \to B) \vdash \neg \forall x \neg A(x) \to B$	PC 3
5.	$\forall x(A(x) \to B) \vdash \exists xA(x) \to B$	Definition of "$\exists$"
6.	$\vdash \forall x(A(x) \to B) \to (\exists xA(x) \to B)$	Deduction
7.	$\exists xA(x) \to B \vdash \exists xA(x) \to B$	Assumption
8.	$\exists xA(x) \to B \vdash \neg \forall x \neg A(x) \to B$	Definition of "$\exists$"
9.	$\exists xA(x) \to B \vdash \neg B \to \forall x \neg A(x)$	PC 8
10.	$\exists xA(x) \to B \vdash \forall x(\neg B \to \neg A(x))$	Theorem 6.17
11.	$\exists xA(x) \to B \vdash \forall x(A(x) \to B))$	Exercise
12.	$\vdash \forall x(A(x) \to B) \leftrightarrow (\exists xA(x) \to B)$	PC 6,11

The C-Rule

The C-rule is a derived rule that is useful in proofs of existentially quantified formulas. The rule is the formalization of the argument: if there exists an object satisfying a certain property, let c be some instance of that object.

Definition 6.21 (C-Rule) Let U be a set of formulas and a a constant which does not appear in any formula of U or in $\exists xA(x)$.

$$\frac{U \vdash \exists xA(x)}{U \cup \{A(a)\}}.$$

Theorem 6.22 *If $U \vdash A$ using the C-Rule, then $U \vdash A$ without using the C-Rule, provided that Generalization is not used on a free variable in a formula $\exists x A(x)$ to which the C-rule has already been applied.*

Proof: See Proposition 2.10 of Mendelson (1997). ∎

We use the C-Rule to give a more intuitive proof of Theorem 6.16.

Theorem 6.23 $\vdash \exists x \forall y A(x,y) \rightarrow \forall y \exists x A(x,y)$

Proof:

1.	$\exists x \forall y A(x,y) \vdash \exists x \forall y A(x,y)$	Assumption
2.	$\exists x \forall y A(x,y) \vdash \forall y A(a,y)$	C-Rule
3.	$\exists x \forall y A(x,y) \vdash A(a,b)$	Axiom 4
4.	$\exists x \forall y A(x,y) \vdash \exists x A(a,x)$	Theorem 6.12
5.	$\exists x \forall y A(x,y) \vdash \forall y \exists x A(y,x)$	Gen. 4
6.	$\vdash \exists x \forall y A(x,y) \rightarrow \forall y \exists x A(x,y)$	Deduction

∎

6.3 Implementation^P

Let us extend the proof checker for the Hilbert system to the predicate calculus. First we need to add the two axioms. Axiom 5 requires that the quantified variable not occur as a free variable in the antecedent.

```
axiom(all(X, A1) imp A2, 4) :- instance(A1, A2, X, _).
axiom(all(X, A imp B) imp (A imp all(X, B)), 5) :-
    \+ free_in(A, X).
```

The procedure `instance(A,A1,X,C)` is similar to the one shown in the implementation of semantic tableaux (Section 5.6); the differences are explained in the comments in the source archive.

To check if a variable is free in a formula, simply traverse the formula and for every quantifier, check that the variable is different from the quantified variable.

```
free_in(all(X, A), Y) :- \+ X==Y, free_in(A, Y).
free_in(ex(X, A),  Y) :- \+ X==Y, free_in(A, Y).
free_in(A v B,     X) :- free_in(A, X); free_in(B, X).
  % Similarly for the other Boolean operators
free_in(A,         X) :- A =..  [_ | Vars], member(X, Vars).
```


An additional argument is added to the predicate `proof` to store a list of constants `Gens` to which Generalization has been applied. When the deduction rule is used, two things must be checked: that the new set of assumptions is the same as the previous one without the formula `A` and the proviso that no constant of `A` appears in `Gens`.

```
proof([Fml | Tail], Line, SoFar, Gens) :-
  Line1 is Line + 1,
  Fml = deduce(Assump, A imp B),
  nth1(L, SoFar, deduce(Previous, B)),
  member(A, Previous),
  proviso(Gens, A), !,
  delete(Previous, A, Assump),
  D is Line1 - L,
  write_proof_line(Line1, Fml, ['Deduction ', D]),
  proof(Tail, Line1, [Fml | SoFar], Gens).
```

The list `Gens` is built when the Generalization rule is used. The fourth argument of the predicate `instance(A, A1, X, C)` is the constant used in the instantiation and it is put on the list `Gens` for the recursive call.

```
proof([Fml | Tail], Line, SoFar, Gens) :-
  Line1 is Line + 1,
  Fml = deduce(_, all(X, A)),
  nth1(L, SoFar, deduce(_, A1)),
  instance(A, A1, X, C), !,
  G is Line1 - L,
  write_proof_line(Line1, Fml, ['Gen ', G]),
  proof(Tail, Line1, [Fml | SoFar], [C | Gens]).
```

To check the proviso, the list of constants is traversed and a check is made that each one does not appear (free) in `A`.

```
proviso([], _).
proviso([C|Rest], A) :- \+ free_in(A,C), proviso(Rest,A).
```

6.4 Complete and decidable theories*

Definition 6.24 Let $\mathcal{T}(U)$ be a theory. $\mathcal{T}(U)$ is *complete* if and only if for every closed formula A, $U \vdash A$ or $U \vdash \neg A$. ∎

It is important not to confuse a complete theory with the completeness of a deductive system. The latter relates the syntactic concept of proof to the semantic concept of validity: a closed formula can be proved if and only if it is valid. Completeness

of a theory looks at what can happen if the formula is not valid. Given a closed formula A, $\wedge_i U_i \rightarrow A$ is either valid (true in all interpretations), unsatisfiable (false in all interpretations) or satisfiable/falsifiable (true in some interpretations and false in others). If it is valid then $U \vdash A$, if it is unsatisfiable then $U \vdash \neg A$, but unless the theory is complete, it may be neither.

In one of the most important and surprising theorems of mathematical logic, Kurt Gödel proved that even a theory as basic as *number theory* is incomplete. Number theory is a predicate calculus with one constant symbol 0, one binary predicate symbol $=$, one unary function symbol s representing the successor function (that is $s(x) = x + 1$) and two binary function symbols $+$ and $*$. A set of axioms for number theory $\mathcal{NT}$ consists of eight axioms plus one axiom scheme for induction:

1. $\vdash x = y \rightarrow (x = z \rightarrow y = z)$
2. $\vdash x = y \rightarrow s(x) = s(y)$
3. $\vdash 0 \neq s(x)$
4. $\vdash s(x) = s(y) \rightarrow x = y$
5. $\vdash x + 0 = x$
6. $\vdash x + s(y) = s(x + y)$
7. $\vdash x * 0 = 0$
8. $\vdash x * s(y) = x * y + x$
9. For any formula $A(x)$ on the symbols of the theory,
   $\vdash A(0) \rightarrow (\forall x(A(x) \rightarrow A(s(x))) \rightarrow \forall x A(x))$.

From these axioms it is possible to prove the usual theorems of number theory, such as the distributive law: for any terms r, s, t, $r * (s + t) = r * s + r * t$. See Mendelson (1997) for details and proofs.

Theorem 6.25 (Gödel's Incompleteness Theorem) *Let $\mathcal{NT}$ be the set of axioms for number theory. If $\mathcal{T}(\mathcal{NT})$ is consistent then $\mathcal{T}(\mathcal{NT})$ is incomplete.*

If $\mathcal{T}(\mathcal{NT})$ were inconsistent, that is, a theorem and its negation were both provable, it would be even less interesting than an incomplete theory.

The detailed proof of Gödel's theorem is tedious but not too difficult. An informal justification can be found in Smullyan (1978). Here we give a sketch of the formal proof (for details see Mendelson (1997)). The idea is to create a constructive correspondence, called a *Gödel numbering*, between natural numbers and logical objects such as formulas and proofs, and then prove the following result.

Theorem 6.26 *There exists a formula $A(x, y)$ whose interpretation in number theory is: for any numbers i, j, $A(i, j)$ is true if and only if i is the Gödel number associated with some formula $B(x)$ with one free variable x, and j is the Gödel number associated with the proof of $B(i)$. Furthermore, if $A(i, j)$ is true then a proof can be constructed for these specific integers $\vdash A(i, j)$.*

Consider now the formula $C(x) = \forall y \neg A(x, y)$ which has one free variable x, and let m be its Gödel number. By the theorem, the formula $C(m) = \forall y \neg A(m, y)$ means that for no y is y the Gödel number of a proof of $C(x)$.

Theorem 6.27 (Gödel) *If $\mathcal{NT}$ is consistent then $\nvdash C(m)$ and $\nvdash \neg C(m)$.*

Proof: Suppose $\vdash C(m)$. Let n be the Gödel number of this proof. By Theorem 6.26, $A(m, n)$ is true, and furthermore $\vdash A(m, n)$. By Axiom 4 of the predicate calculus, from $\vdash C(m) = \forall y \neg A(m, y)$, we get $\vdash \neg A(m, n)$. But $\vdash A(m, n)$ and $\vdash \neg A(m, n)$ contradict consistency.

Conversely, suppose that $\vdash \neg C(m) = \neg \forall y \neg A(m, y) = \exists y A(m, y)$. Then for some n, $A(m, n)$ is true which means that n is the Gödel number of a proof of $C(m)$. But we assumed that $\neg C(m)$ is provable, contradicting consistency. ∎

Since there is no proof of $C(m)$, it is true that there is no proof of $C(m)$, that is, $C(m)$ is an example of a closed formula which is true but not provable.

Definition 6.28 Let $\mathcal{T}$ be a theory. $\mathcal{T}$ is *decidable* if and only if there is an algorithm that determines for any closed formula A, whether $A \in \mathcal{T}$ or $A \notin \mathcal{T}$. □

Number theory is not decidable.

We close this section with a sequence of theorems (without proofs) that establish relations between completeness and decidability.

Theorem 6.29 (Lindenbaum's Lemma) *Let $\mathcal{T}$ be a consistent theory. Then there is a theory $\mathcal{T}'$, such that $\mathcal{T} \subseteq \mathcal{T}'$ and $\mathcal{T}'$ is consistent and complete.*

Theorem 6.30 *Let $\mathcal{T}$ be a consistent, decidable theory. Then there is a theory $\mathcal{T}'$, such that $\mathcal{T} \subseteq \mathcal{T}'$ and $\mathcal{T}'$ is consistent, decidable and complete.*

Definition 6.31 A theory $\mathcal{T}$ is *effectively axiomatizable* if there is a set of axioms U for $\mathcal{T}$ and an algorithm such that given a closed formula A, the algorithm determines if $A \in U$ or not. □

Theorem 6.32 *Let $\mathcal{T}$ be a theory. If $\mathcal{T}$ is complete and undecidable then it is not effectively axiomatizable.*

Theorem 6.33 *Let $\mathcal{T}$ be a theory. If $\mathcal{T}$ is effectively axiomatizable and undecidable then it is incomplete.*

This extends Gödel's theorem. Since number theory is undecidable, *any* useful axiom system for the theory is necessarily incomplete.

6.5 Exercises

1. Prove in $\mathcal{G}$:
$$\vdash \forall x(p(x) \to q(x)) \to (\exists x p(x) \to \exists x q(x)),$$
$$\vdash \exists x(p(x) \to q(x)) \leftrightarrow (\forall x p(x) \to \exists x q(x)).$$

2. Prove the soundness and completeness of $\mathcal{G}$ (Theorem 6.3).

3. Prove that Axioms 4 and 5 are valid.

4. Prove the axioms of $\mathcal{H}$ in $\mathcal{G}$.

5. Prove in $\mathcal{H}$: $\vdash \forall x(p(x) \to q) \leftrightarrow \forall x(\neg q \to \neg p(x))$.

6. Prove the theorems of Exercise 1 in $\mathcal{H}$.

7. Show that unrestricted use of the C-Rule could be used to prove the non-valid formula $\forall x \exists y p(x, y) \vdash \exists y \forall x p(x, y)$.

8. Let A be a formula built from the quantifiers and the Boolean operators $\neg$, $\vee$, $\wedge$ only. A', the dual of A is obtained by exchanging $\forall$ and $\exists$ and exchanging $\vee$ and $\wedge$. Prove that $\vdash A$ iff $\vdash A'$.

9. ^{P} Implement a proof checker for $\mathcal{G}$.

10. * Prove that a formula is satisfiable iff it is satisfiable in an infinite model.

11. * Prove Lindenbaum's Lemma (Theorem 6.29).

7 Predicate Calculus: Resolution

7.1 Functions and terms

The development of the predicate calculus up to this point has been simplified in that unstructured terms—variables and constants—have been used in formulas, and interpretations are over unstructured domains. The real power of the predicate calculus comes from its ability to express logical relationships in structured domains such as numerical domains (integers and real numbers) or data structures (lists and trees).

Example 7.1 The formula $(x > y) \to ((x+1) > (y+1))$ can be written in prefix notation as $> (x, y) \to > (+(x, 1), +(y, 1))$. It is an interpreted instance of the following formula in the predicate calculus: $p(x, y) \to p(f(x, a), f(y, a))$, where $>$ is assigned to p, $+$ is assigned to f and 1 to a. The same formula can be interpreted over the domain of strings: $substr(x, y) \to substr(x \cdot 'a',\ y \cdot 'a')$, where the substring relation is assigned to p, the function $\cdot$ which concatenates a string and a character is assigned to f, and the constant character $'a'$ to a. □

We have already described the interpretation of predicate letters by relations and the interpretation of constant symbols by domain elements. What remains to do is to introduce function symbols like f into the predicate calculus, and to describe their interpretation by domain functions like $+$.

Definition 7.2 Let $\mathcal{F}$ be a countable set of function symbols. The following grammar rules define *terms*, a generalization of constants and variables. The rule for *atomic_formula* is modified to take a *term_list* as its argument.

term	::=	x	for any $x \in \mathcal{V}$
term	::=	a	for any $a \in \mathcal{A}$
term	::=	f(*term_list*)	for any $f \in \mathcal{F}$
term_list	::=	*term*	
term_list	::=	*term*, *term_list*	
atomic_formula	::=	p(*term_list*)	for any $p \in \mathcal{P}$.

As with predicate symbols, function symbols have a fixed arity or number of arguments in any formula or set of formulas. By convention, functions are denoted by $\{f, g, h\}$ with subscripts as necessary.

Example 7.3 Examples of terms are

$$a,\ \ x,\ \ f(a,x),\ \ f(g(x),y),\ \ g(f(a,g(b))),$$

and examples of atomic formulas are

$$p(a,b),\ \ p(x,f(a,x)),\ \ q(f(a,a),f(g(x),g(x))),$$

where function symbol f is of arity two and g is of arity one. □

As with formulas, terms have formation trees and theorems about terms can be proved by induction on the structure of the term.

Definition 7.4 A term or atom is *ground* iff it contains no variables. A formula is *ground* iff it contains no quantifiers and no variables. A formula A is a *ground instance* of a quantifier-free formula A iff it can be obtained from A by substituting ground terms for the (free) variables in A. □

Definition 7.5 Let U be a set of formulas such that $\{p_1, \ldots, p_k\}$ are all the predicate symbols, $\{f_1, \ldots, f_l\}$ are all the function symbols and $\{a_1, \ldots, a_m\}$ are all the constant symbols appearing in U. An *interpretation* $\mathcal{I}$ is a 4-tuple

$$(D,\ \{R_1, \ldots, R_k\},\ \{F_1, \ldots, F_l\},\ \{d_1, \ldots, d_m\}),$$

consisting of a *non-empty* domain D, an assignment of n_i-ary relations R_i on D to the n_i-ary predicate symbols p_i, an assignment of n_i-ary functions F_i on D to the n_i-ary function symbols f_i, and an assignment of elements $d_i \in D$ to the constant symbols a_i. □

Definition 7.6 Given a *ground* term t, $v_{\mathcal{I}}(t)$, the *value* of the term in the interpretation $\mathcal{I}$, is defined by induction:

$$\begin{aligned} v_{\mathcal{I}}(a_i) &= d_i \\ v_{\mathcal{I}}(f_i(t_1, \ldots, t_n)) &= F_i(v_{\mathcal{I}}(t_1), \ldots, v_{\mathcal{I}}(t_n)). \end{aligned}$$

$v_{\mathcal{I}}(A)$, the *value* of a formula, is also defined by induction. For atomic formulas:

$$v_{\mathcal{I}}(p_i(t_1, \ldots, t_n)) = T \text{ iff } (v_{\mathcal{I}}(t_1), \ldots, v_{\mathcal{I}}(t_n)) \in R_i$$

and for structured formulas as before. □

Example 7.7 We show that the formula $\forall x \forall y(p(x,y) \rightarrow p(f(x,a),f(y,a)))$ is satisfied by the interpretation $(\mathcal{Z}, \{\leq\}, \{+\}, \{1\})$. For arbitrary $m, n \in \mathcal{Z}$ assigned to x, y, $v(f(x,a)) = +(v(x), v(a)) = +(m, 1)$ and $v(f(y,a)) = +(v(y), v(a)) = +(n, 1)$, or $m+1$ and $n+1$ in infix notation. But $m \leq n$ implies $m+1 \leq n+1$; since m and n were arbitrary, the formula is true in this interpretation.

The formula is not valid since it is falsified by the interpretation $(\mathcal{Z}, \{>\}, \{*\}, \{-1\})$, because (for example) $5 > 4$ does not imply $5 * (-1) > 4 * (-1)$. □

The development of the predicate calculus with function symbols is almost the same as that of the predicate calculus with constant symbols only. In the construction of the semantic tableaux, the δ-rule requires that instantiation be with a new constant a. This assumes that there is an enumeration of all constant symbols. Here, we need an enumeration of all ground terms, but this is easily done since there is a countable number of function symbols of each arity (the superscript indicating the arity).

$$a_1,\ a_2,\ a_3,\ \ldots$$

$$f_1^1(\cdot),\ f_2^1(\cdot),\ f_3^1(\cdot),\ \ldots$$

$$f_1^2(\cdot,\cdot),\ f_2^2(\cdot,\cdot),\ f_3^2(\cdot,\cdot),\ \ldots.$$

These can be placed into a single list by diagonalization:

$$a_1,$$

$$a_2,\ f_1^1(\cdot),$$

$$a_3,\ f_2^1(\cdot),\ f_1^2(\cdot,\cdot),$$

$$a_4,\ f_3^1(\cdot),\ f_2^2(\cdot,\cdot),\ f_1^3(\cdot,\cdot,\cdot).$$

Now we can create an enumeration of all terms (Figure 7.1) by listing (without duplicates) for each n, all terms of height less than or equal to n that use the first n symbols in the list of function symbols above.

The construction of semantic tableaux and the proof of the completeness theorem for the predicate calculus (Section 5.5) is the same as before, except that it uses the enumeration of terms instead of the enumeration of constant symbols.

$n = 1$ a_1

$n = 2$ a_2

$n = 3$ $f_1^1(a_1), f_1^1(a_2), f_1^1(f_1^1(a_1)), f_1^1(f_1^1(a_2))$

$n = 4$ $a_3, f_1^1(a_3), f_1^1(f_1^1(a_3)), f_1^1(f_1^1(f_1^1(a_3))),$

$f_1^1(f_1^1(f_1^1(a_1))), f_1^1(f_1^1(f_1^1(a_2)))$

$n = 5$ $f_2^1(a_1), f_2^1(a_2), f_2^1(a_3),$

$f_1^2(a_1, a_1), f_1^2(a_1, a_2), f_1^2(a_1, a_3), \ldots.$

Figure 7.1 List of all terms

7.2 Clausal form

Recall that a formula is in CNF iff it is a conjunction of disjunctions of literals.

Definition 7.8 A formula is in *prenex conjunctive normal form (PCNF)* iff it is of the form:

$$Q_1 x_1 \cdots Q_n x_n M$$

where the Q_i are quantifiers and M is a quantifier-free formula in CNF. The sequence $Q_1 x_1 \cdots Q_n x_n$ is called the *prefix* and M is called the *matrix*. □

Definition 7.9 A *closed* formula is in *clausal form* iff it is in PCNF and its prefix consists only of *universal* quantifiers. A *clause* is a disjunction of literals. A clause C' is a *ground clause* iff it is a *ground instance* of a clause C, that is, iff it can be obtained by substituting ground terms for the variables in C. □

As in the propositional calculus we use a more concise notation. Since a formula in clausal form is closed, all variables are quantified by universal quantifiers, so the quantifiers need not be explicitly written. The matrix is written as a set of clauses, each of which may also written as a set. It is obvious that the elimination of duplicates through the use of set notation is of no importance. A further abbreviation that we will sometimes use is to omit the parentheses and commas in the argument lists of the predicates assuming that their arities are known. Instead of using set notation, each clause is written as a concatenation of its literals.

Example 7.10 The following formula is in clausal form:

$$\forall y \forall z([p(f(y)) \vee \neg p(g(z)) \vee q(z)] \wedge [\neg q(z) \vee \neg p(g(z)) \vee q(y)]).$$

It may be written in abbreviated notation in one of the following forms:

$$\{ \{p(f(y)), \neg p(g(z)), q(z)\}, \{\neg q(z), \neg p(g(z)), q(y)\} \},$$
$$\{pf(y)\ \neg pg(z)\ qz,\ \neg qz\ \neg pg(z)\ qy\}.$$

□

Theorem 7.11 (Skolem) *Let A be a closed formula. Then there exists a formula A' in clausal form such that $A \approx A'$.*

Note carefully what Skolem's theorem says. It is not true that $A \equiv A'$: A is logically equivalent to A, meaning that for all models $\mathcal{I}$, $\mathcal{I}$ is a model for A if and only if it is a model for A'. It is only true that $A \approx A'$: A is satisfiable if and only if A' is satisfiable, meaning that there *exists* a model for A if and only if there *exists* a model for A'. As we are interested in validity and satisfiability, the difference is not important.

It is straightforward to transform A into an logically equivalent PCNF formula. It is the removal of the existential quantifiers that causes the new formula to no longer be equivalent to the old one. The removal is accomplished by defining new function symbols. In $A = \forall x \exists y p(x, y)$, the quantifiers can be read: for all x, *produce* a value y associated with that x such that the predicate p is true. But our intuitive concept of a function is the same; $y = f(x)$ means that given x, f produces a value y associated with x. Thus $A' = \forall x p(x, f(x))$ expresses almost the same idea as does A.

Example 7.12 Consider the interpretation $\mathcal{I}_1 = (\mathcal{Z}, \{>\})$ for $A = \forall x \exists y p(x, y)$. Obviously, $\mathcal{I}_1 \models A$. What about the formula $A' = \forall x p(x, f(x))$? A' is not equivalent to A because there is an interpretation $\mathcal{I}_1' = (\mathcal{Z}, \{>\}, \{F(x) = x + 1\})$ such that $\mathcal{I}_1' \models A$ (ignoring the function), but $\mathcal{I}_1' \not\models A'$. However, there *is* a model for A', for example, $\mathcal{I}_1' = (\mathcal{Z}, \{>\}, \{F(x) = x - 1\})$. □

The introduction of function symbols narrows the choice of models. The relations that interpret predicate symbols are *many-many*, that is, each x may be related to several y, while functions are *many-one*, that is, each x is related (mapped) to a single y. For example, if $R = \{(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3)\}$, then when trying to satisfy A, the whole relation R can be used, but for the clausal form A', only a functional subset of R such as $\{(1, 2), (2, 3)\}$ can be used to satisfy A'.

We now give an algorithm to transform a formula A into a formula A' in clausal form and then prove that $A \approx A'$. The description of the transformation will be accompanied by a running example using the formula:

$$\forall x(p(x) \rightarrow q(x)) \rightarrow (\forall x p(x) \rightarrow \forall x q(x)).$$

Algorithm 7.13 (Skolemization)
Input: A closed formula A of the predicate calculus.
Output: A formula A' in clausal form such that $A \approx A'$.

- Rename bound variables so that no variable appears in two quantifiers.

$$\forall x(p(x) \to q(x)) \to (\forall y p(y) \to \forall z q(z)).$$

- Eliminate all binary Boolean operators other than $\vee$ and $\wedge$.

$$\neg \forall x(\neg p(x) \vee q(x)) \vee \neg \forall y p(y) \vee \forall z q(z)).$$

- Push negation operators inward, collapsing double negation, until they apply to atomic formulas only. Use the equivalences:

$$\neg \forall x A(x) \equiv \exists x \neg A(x) \qquad \text{and} \qquad \neg \exists x A(x) \equiv \forall x \neg A(x).$$

The example formula is transformed to:

$$\exists x(p(x) \wedge \neg q(x)) \vee \exists y \neg p(y) \vee \forall z q(z)).$$

- Extract quantifiers from the matrix. Repeatedly, choose a quantifier in the matrix which is not in the scope of another quantifier still in the matrix and extract it using the following equivalences which are always applicable since no variable appears in two quantifiers:

$$A \; op \; Qx B(x) \equiv Qx(A \; op \; B(x)) \quad \text{and} \quad Qx A(x) \; op \; B \equiv Qx(A(x) \; op \; B),$$

where Q is a quantifier and op is either $\vee$ or $\wedge$. In the example:

$$\exists x \exists y \forall z(\; (p(x) \wedge \neg q(x)) \; \vee \; \neg p(y) \; \vee \; q(z) \;).$$

- Use the distributive laws to transform the matrix into *CNF*.

$$\exists x \exists y \forall z(\; (p(x) \vee \neg p(y) \vee q(z)) \; \wedge \; (\neg q(x) \vee \neg p(y) \vee q(z)) \;).$$

- Let $\exists x$ be an existential quantifier in A, let $y_1, \ldots, y_n$ be the universally quantified variables *preceding* $\exists x$ and let f be a *new* n-ary function symbol. Delete $\exists x$ and replace every occurrence of x by $f(y_1, \ldots, y_n)$. If there are no universal quantifiers preceding $\exists x$, replace x by a new constant (0-ary function) symbol a. These new function symbols are called *Skolem functions*. For the example formula we have:

$$\forall z(\; (p(a) \vee \neg p(b) \vee q(z)) \; \wedge \; (\neg q(a) \vee \neg p(b) \vee q(z)) \;),$$

where a and b are the Skolem functions (constants) corresponding to the existentially quantified variables x and y, respectively.

Note that the order of extraction of outermost quantifiers from the matrix is arbitrary. A different order of extraction gives:

$$\forall z \exists x \exists y(\; (p(x) \vee \neg p(y) \vee q(z)) \; \wedge \; (\neg q(x) \vee \neg p(y) \vee q(z)) \;),$$

and its clausal form is:

$$\forall z(\; (p(f(z)) \vee \neg p(g(z)) \vee q(z)) \; \wedge \; (\neg q(f(z)) \vee \neg p(g(z)) \vee q(z)) \;),$$

where the Skolem functions f and g are unary because the single universal quantifier on z precedes the existential quantifiers.

Example 7.14 Let us follow the entire transformation on another formula.

Original formula	$\exists x \forall y p(x, y) \to \forall y \exists x p(x, y)$
Rename bound variables	$\exists x \forall y p(x, y) \to \forall w \exists z p(z, w)$
Eliminate Boolean operators	$\neg \exists x \forall y p(x, y) \vee \forall w \exists z p(z, w)$
Push negation inwards	$\forall x \exists y \neg p(x, y) \vee \forall w \exists z p(z, w)$
Extract quantifiers	$\forall x \exists y \forall w \exists z(\neg p(x, y) \vee p(z, w))$
Distribute matrix	(no change)
Replace existential quantifiers	$\forall x \forall w(\neg p(x, f(x)) \vee p(g(x, w), w))$.

f is unary because $\exists y$ is preceded by one universal quantifier $\forall x$, while g is binary because $\exists z$ is preceded by two universal quantifiers $\forall x$ and $\forall w$. □

Proof of Skolem's Theorem:

The first five transformations of the algorithm can easily be shown to preserve equivalence. Consider now the replacement of an existential quantifier by a Skolem function. Suppose that $\mathcal{I} \models \forall y_1 \cdots \forall y_n \exists x p(y_1, \ldots, y_n, x)$. We show that there exists an interpretation $\mathcal{I}'$ such that $\mathcal{I}' \models \forall y_1 \cdots \forall y_n p(y_1, \ldots, y_n, f(y_1, \ldots, y_n))$. Extend $\mathcal{I}$ to $\mathcal{I}'$ by adding the n-ary function F defined by:

> For all $\{c_1, \ldots, c_n\} \subseteq D$, let $F(c_1, \ldots, c_n) = c_{n+1}$ for some $c_{n+1} \in D$ such that $(c_1, \ldots, c_n, c_{n+1}) \in R_p$, where R_p is assigned to p in $\mathcal{I}$.

F is well-defined using the definition of the quantifiers and the fact that $\mathcal{I}$ is a model for the formula A. Also, f is new so the definition of F does not clash with any existing function definitions in $\mathcal{I}$.

To show that $\mathcal{I}' \models \forall y_1 \cdots \forall y_n p(y_1, \ldots, y_n, f(y_1, \ldots, y_n))$, let $\{c_1, \ldots, c_n\}$ be arbitrary domain elements. By construction, $f(c_1, \ldots, c_n) = c_{n+1}$ for some $c_{n+1} \in D$ and $v(p(c_1, \ldots, c_n, c_{n+1})) = T$. Since $c_1, \ldots, c_n$ were arbitrary,

$$v(\forall y_1 \cdots \forall y_n p(y_1, \ldots, y_n, f(y_1, \ldots, y_n))) = T.$$

This completes one direction of the proof of Skolem's Theorem. The proof of the converse (A is satisfiable if A' is satisfiable) is simple and is left as an exercise. ∎

In practice, it is better to use a slightly different transformation of a formula to clausal form. First push all quantifiers *inward*, then replace existential quantifiers by Skolem functions and then extract all the remaining (universal) quantifiers. This ensures that the number of universal quantifiers preceding an existential quantifier is minimal, and thus the arity of the Skolem functions is minimal.

Example 7.15 For the formula $\exists x \forall y p(x,y) \rightarrow \forall y \exists x p(x,y)$ of Example 7.14, we would obtain $\forall x \neg p(x,f(x)) \vee \forall w p(g(w),w)$ and then $\forall x \forall w(\neg p(x,f(x)) \vee p(g(w),w))$, replacing the binary function g by a unary function. □

Implementation[P]

To Skolemize a formula, we first transform it into CNF and then call the predicate `skolem(A,ListA,ListE,A1)`. `A1` is the Skolemized version of `A` and `ListA` is the list of universally quantified variables that have appeared so far. `ListE` is a list of pairs: an existentially quantified variable, and another pair that contains the Skolem function and a list of its arguments that are to be substituted for the existential variable. `gensym` is used to create new Skolem function symbols.

```
skolem(A, A2) :- cnf(A, A1), skolem(A1, [], [], A2).

skolem(all(X, A), ListA, ListE, all(X, B)) :-
  skolem(A, [X | ListA], ListE, B).
skolem(ex(X, A), ListA, ListE, B) :-
  gensym(f, Func),
  skolem(A, ListA, [(X, (Func, ListA)) | ListE], B).

skolem(A1 imp A2, _, ListE, B1 imp B2) :-
  skolem(A1, _, ListE, B1 ),
  skolem(A2, _, ListE, B2).
% Similar clauses for the other Boolean operators.
```

When an atomic proposition is encountered in the recursive traversal of the formula, the Prolog operator `=..` (read *univ*) is used to decompose the formula into a predicate symbol and a list of variables. Then, `subst_var` is called to replace existentially quantified variables by the Skolem functions, and `=..` is called again to recompose the formula.

```
skolem(A, _, ListE, B) :-
  A =..  [F | Vars],
  subst_var(Vars, Vars1, ListE),
  B =..  [F | Vars1].
```


The implementation of `subst_var` is by a simple traversal of the list of variables.

```
subst_var([], [], _).
subst_var([V | Tail], [V1 | Tail1], List) :-
  member_var((V, V1), List), !,
  subst_var(Tail, Tail1, List).
subst_var([V | Tail], [V | Tail1],  List) :-
  subst_var(Tail, Tail1, List).
```

In principle, `member` could be used on the list of pairs to obtain the substitution for the variable, but the standard procedure would unify variables. That is, if we were searching for `Y` and the the list was `[(X, (f, Z)), (Y, (g, U, W)]`, `Y` would unify with `X` instead of failing and recursing on the tail of the list. The procedure `member_var` uses the identity operator `==` to make sure that unification is not done.

```
member_var((A,Y), [(B,Y) | _]) :- A == B, !.
member_var(A,     [_ | C])     :- member_var(A, C).
```

The `cnf` predicate has to be modified to rename variables so that no two quantified variables are the same and to extract quantifiers.

```
cnf(A, A5) :-
  rename(A, A1),
  eliminate(A1, A2),
  demorgan(A2, A3),
  extract(A3, A4),
  distribute(A4, A5).
```

The modifications to `eliminate`, `demorgan` and `distribute`, and the procedure `extract` are in the source archive.

`rename` works by traversing the formula, keeping a list of variable substitutions. The call is `rename(A,List,List1,A1)`, where `A1` is `A` after the variables have been renamed, and `List` and `List1` are lists of pairs of variables.

```
rename(A, B) :- rename(A, [], _, B).
```

One the way down the recursive traverse, `List` stores all the variables that have been encountered and the new variable names. At the bottom, `List` is unified with the variable `List1` and the substitutions are made on the way up the recursive traverse. When a quantified variable is the same as one previously encountered, `copy_term` is used to create a new variable. This is a Prolog procedure that makes a copy of its first argument with fresh variable and places it in the second argument.

```
rename(all(X, A), List, List1, all(Y, A1)) :-
  member_var((X, _), List), !,
  copy_term(X, Y),
  rename(A, [(X, Y) | List], List1, A1).
```

When a quantified variable is encountered for the first time, an identity substitution is created.

```
rename(all(X, A), List, List1, all(X, A1)) :-
  rename(A, [(X, X) | List], List1, A1).
```

The clauses for ex are similar and the clauses for traversing the Boolean operators are omitted here. The clause terminating the recursion performs the subsitution on the atomic formulas.

```
rename(A, List, List, B) :-
  A =..  [F | Vars],
  subst_var(Vars, Vars1, List),
  B =..  [F | Vars1].
```

Example 7.16 If `rename` is called on the formula `all(X,p(X)) or all(X,q(X))`, the call for `all(X,p(X))` will create `[(X,X)]` and return it in `List1`; then the call for `all(X,q(X))` will create `[(X,X1),(X,X)]` and return it in `List1`. The renamed formula is `all(X,p(X)) or all(X1,q(X1))`. □

7.3 Herbrand models

When function symbols are used to form terms, the set of possible interpretations is extremely complex. In this section, we show that for sets of clauses there are canonical interpretations: if a set of clauses has a model then it has a model of this form. The section starts with a long sequence of definitions which may initially seem confusing, because in the canonical interpretations the domain elements are the syntactical terms we are trying to interpret.

Definition 7.17 Let S be a set of clauses, $\mathcal{A}$ the set of constant symbols in S and $\mathcal{F}$ the set of function symbols in S. H_S, the *Herbrand universe* of S, is defined inductively:

$$\begin{array}{ll} a_i \in H_S & \text{for } a_i \in \mathcal{A} \\ f_i(t_1, \ldots, t_n) \in H_S & \text{for } f_i \in \mathcal{F}, t_j \in H_S. \end{array}$$

As a special case, if there are no constant symbols in S, initialize the inductive definition of H_S with an arbitrary constant symbol a. □

The Herbrand universe is just the set of ground terms that can be formed from symbols in S. Obviously, if S contains a function symbol, the Herbrand universe is infinite since $f(f(\ldots(a)\ldots)) \in H_S$.

Example 7.18 Here are some examples of Herbrand universes:

$$\begin{aligned} S_1 &= \{pa\neg pbqz,\ \neg qz\neg pbqz\} \\ H_{S_1} &= \{a,\ b\} \\ S_2 &= \{\neg pxf(y),\ pwg(w)\} \\ H_{S_2} &= \{a,\ f(a),\ g(a), \\ &\quad f(f(a)),\ g(f(a)),\ f(g(a)),\ g(g(a)),\ \ldots\} \\ S_3 &= \{\neg paf(x,y),\ pbf(x,y)\} \\ H_{S_3} &= \{a,\ b,\ f(a,a),\ f(a,b),\ f(b,a),\ f(b,b), \\ &\quad f(a,f(a,a)),\ f(f(a,a),a),\ \ldots\} \end{aligned}$$

□

Definition 7.19 Let H_S be the Herbrand universe for a set of clauses S. B_S, the *Herbrand base*, is the set of ground atoms that can be formed from predicate symbols in S and terms in H_S. □

Definition 7.20 An *Herbrand interpretation* for a set of clauses S is an interpretation whose domain is the Herbrand universe for S and whose constant and function symbols are assigned 'themselves':

$$\begin{aligned} \nu(a) &= a \\ \nu(f(t_1, \ldots, t_n)) &= f(\nu(t_1), \ldots, \nu(t_n)) \end{aligned}$$

There are no restrictions on the assignments of relations over the Herbrand universe to predicates. An *Herbrand model* for a set of clauses S is an Herbrand interpretation which satisfies S. It can be identified with the subset of the Herbrand base for which $\nu(p(t_1, \ldots, t_n)) = T$. □

Example 7.21 The Herbrand base for the set of formulas S_3 from Example 7.18 is:

$$\begin{aligned} B_{S_3} = \{&paf(a,a),\ paf(a,b),\ paf(b,a),\ paf(b,b),\ \ldots,\ paf(f(a),a)),\ \ldots, \\ &pbf(a,a),\ pbf(a,b),\ pbf(b,a),\ pbf(b,b),\ \ldots\}. \end{aligned}$$

An Herbrand model for S_3 can be defined by:

$$v(paf(a,a)) = F \quad v(paf(a,b)) = F \quad v(paf(b,a)) = F \quad v(paf(b,b)) = F$$
$$v(pbf(a,a)) = T \quad v(pbf(a,b)) = T \quad v(pbf(b,a)) = T \quad v(pbf(b,b)) = T$$
$$\ldots,$$

or more simply by a subset of the Herbrand base:

$$\{pbf(a,a),\ pbf(a,b),\ pbf(b,a),\ pbf(b,b),\ \ldots\}.$$

□

Theorem 7.22 *Let S be a set of clauses. S has a model iff it has an Herbrand model.*

Proof: Let $\mathcal{I}$ be an arbitrary model for S. Define the Herbrand interpretation $\mathcal{H}_\mathcal{I}$ by the following subset of the Herbrand base:

$$\{p_i(t_1, \ldots, t_n) \mid (v(t_1), \ldots, v(t_n)) \in R_i\}$$

where R_i is the relation assigned to p_i in $\mathcal{I}$. That is, a ground atom is in the subset of the Herbrand base if its value $v_\mathcal{I}(p_i(t_1, \ldots, t_n))$ is true when interpreted in the model $\mathcal{I}$. It remains to show that $\mathcal{H}_\mathcal{I}$ is a model.

Recall that a set of clauses is a closed formula that is a conjunction of disjunctions of literals. It suffices to show that for each assignment of elements of the Herbrand universe to the variables, one literal of each disjunction is in the subset. Since $\mathcal{I}$ is a model for the set of clauses S, $v_\mathcal{I}(S) = T$ so for all assignments by $\mathcal{I}$ to the variables and for *all* clauses $C_i \in S$, $v_\mathcal{I}(C_i) = T$. Thus for all clauses $C_i \in S$, there is *some* literal D_{ij} in the clause such that $v_\mathcal{I}(D_{ij}) = T$. But, by definition of the $\mathcal{H}_\mathcal{I}$, $v_{\mathcal{H}_\mathcal{I}}(D_{ij}) = T$ iff $v_I(D_{ij}) = T$, from which follows $v_{\mathcal{H}_\mathcal{I}}(C_i) = T$ for all clauses $C_i \in S$, and $v_{\mathcal{H}_\mathcal{I}}(S) = T$. Thus $\mathcal{H}_\mathcal{I}$ is an Herbrand model for S.

The converse is trivial. ∎

It is important to note that the theorem is not true if S is an arbitrary formula which is not a set of clauses. Let $S = p(a) \wedge \exists x \neg p(x)$. Then $\langle \{0,1\}, \{\{0\}\}, \{\}, \{0\}\rangle$ is a model for S since $v(p(0)) = T, v(p(1)) = F$, but S has no Herbrand models since the only Herbrand interpretations are $\langle \{a\}, \{\{a\}\}, \{\}, \{a\}\rangle$ and $\langle \{a\}, \{\{\}\}, \{\}, \{a\}\rangle$, and neither is a model for A.

7.4 Herbrand's Theorem*

Consider a semantic tableau for an unsatisfiable set of clauses. A set of *clauses* is a universally quantified formula $A = \forall x_1 \cdots \forall x_n M(x_1, \ldots, x_n)$ whose matrix is a conjunction of disjunctions of literals. The only rules that can be used are the propositional α- and β-rules and the γ-rule for the universal quantifiers. Since the closed

tableau is finite, there will be a finite number of applications of the γ-rule. Suppose that we construct the tableau by initially applying the γ-rule repeatedly for some sequence of ground terms, and then apply the α-rule repeatedly in order to 'break up' each instantiation of the matrix M.

We obtain a node n labeled with a *finite* set of clauses. Repeated use of the β-rule on each clause (disjunction) will cause the tableau to eventually close because each leaf contains clashing literals. This sketch motivates the following theorem.

Theorem 7.23 (Herbrand's Theorem, semantic form) *A set of clauses S is unsatisfiable if and only if a* finite *set of ground instances of clauses of S is unsatisfiable.*

Since a formula is satisfiable if and only if its clausal form is satisfiable, the theorem can also be expressed as follows.

Theorem 7.24 (Herbrand's Theorem, semantic form) *A formula A is unsatisfiable if and only if a formula built from a* finite *set of ground instances of subformulas of A is unsatisfiable.*

The theorem transforms the problem of satisfiability within the predicate calculus into a problem of finding an appropriate set of ground terms and then checking satisfiability within the propositional calculus.

Since a tableau can be turned upside-down to obtain a Gentzen proof of a formula, there is a syntactic form of Herbrand's theorem.

Theorem 7.25 (Herbrand's Theorem, syntactic form) *A formula A of the predicate calculus is provable if and only if a formula build from a* finite *set of ground instances of subformulas of A is provable using only the axioms and inference rules of the propositional calculus.*

Smullyan (1995) calls this theorem the Fundamental Theorem of Quantification Theory. An alternate way to develop mathematical logic is to prove some version of Herbrand's theorem and then use it to prove completeness and compactness.

Example 7.26 The clausal form of the formula

$$\neg[\forall x(p(x) \rightarrow q(x)) \rightarrow (\forall x p(x) \rightarrow \forall x q(x))]$$

is $\{\neg p(x) \vee q(x),\ p(y),\ \neg q(z)\}$. The set of ground instances of clauses obtained by substituting a for each variable is $\{\neg p(a) \vee q(a),\ p(a),\ \neg q(a)\}$, and an application of the β-rule gives the pair of unsatisfiable sets of formulas $\{\neg p(a),\ p(a),\ \neg q(a)\}$ and $\{q(a),\ p(a),\ \neg q(a)\}$. Thus the original formula is unsatisfiable. □

Herbrand's theorem gives us the handle needed to define an efficient semi-decision procedure for validity of formulas in the predicate calculus:

- Negate the formula.
- Transform into clausal form.
- Generate a finite set of ground clauses.
- Check if the set of ground clauses is unsatisfiable.

The first two steps are trivial and the last is not difficult because any convenient decision procedure for the propositional calculus can be used by treating each distinct *ground* atomic formula as a distinct propositional letter.

Example 7.27 $p(a,f(a))$ and $\neg p(a,f(a))$ clash in any interpretation and $p(a,f(a))$ and $p(b,f(a))$ can always be given different truth values in an Herbrand interpretation. If a propositional decision procedure discovers a model for $(q \vee r) \wedge (\neg q \vee r)$, then there exists a model for:

$$(p(a,f(a)) \vee p(b,f(a))) \wedge (\neg p(a,f(a)) \vee p(b,f(a))),$$

by including in the Herbrand base ground atoms that correspond to propositional letters given the value T in the propositional model. That is, from the propositional model $v(q) = F, v(r) = T$, it follows that the subset of the Herbrand base $\{p(b,f(a))\}$ is a model for the set of ground clauses. □

Unfortunately, we still have no efficient way of generating a set of ground clauses that is likely to be unsatisfiable.

7.5 Ground resolution

We now extend the resolution procedure to the predicate calculus. This will be done in two stages: first, we define a simple, but inefficient, version called *ground resolution*, and then define substitution and unification, which are used in the *general resolution* procedure.

Rule 7.28 (Ground resolution rule) Let C_1, C_2 be *ground* clauses such that $l \in C_1$ and $l^c \in C_2$. C_1, C_2 are said to be *clashing clauses* and to *clash* on the complementary 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. □

Theorem 7.29 *The resolvent C is satisfiable if and only if the parent clauses C_1 and C_2 are (mutually) satisfiable.*

Proof: Let C_1 and C_2 be satisfiable clauses which clash on the literals l, l^c. By Theorem 7.22, they are satisfiable in an Herbrand interpretation $\mathcal{H}$. Let B be the subset of the Herbrand base that defines $\mathcal{H}$, that is,

$$B = \{p(c_1, \ldots, c_k) \mid v_H(p(c_1, \ldots, c_k)) = T\}$$

for ground terms c_i. Obviously, two complementary ground literals cannot both be elements of B. Therefore, if $l \in B$, for C_2 to be satisfied in $\mathcal{H}$ there must be some *other* literal $l' \in C_2$ such that $l' \in B$. By construction of the resolvent C using the resolution rule, $l' \in C$, so $v_{\mathcal{H}}(C) = T$, that is, $\mathcal{H}$ is a model for C. A symmetric argument holds if $l^c \in B$.

Conversely, if C is satisfiable, it is satisfiable in an Herbrand interpretation $\mathcal{H}$ defined by a subset B of the Herbrand base. So for some literal $l' \in C$, $l' \in B$. By construction, $l' \in C_1$ or $l' \in C_2$ (or both), and $l \notin C$ and $l^c \notin C$. Suppose, $l' \in C_1$. We can extend the $\mathcal{H}$ to $\mathcal{H}'$ by defining $B' = B \cup \{l^c\}$. In this interpretation, $v_{\mathcal{H}'}(l^c) = T$, so $v_{\mathcal{H}'}(C_2) = T$. Since $l' \in B \subseteq B'$, C_1 is also satisfiable in $\mathcal{H}'$, so C_1 and C_2 are mutually satisfiable in $\mathcal{H}'$. A symmetric argument holds if $l' \in C_2$. ∎

The ground resolution *procedure* is defined like the resolution procedure for the propositional calculus. Given a set of ground clauses, the resolution step is performed repeatedly. The set of ground clauses is unsatisfiable iff some sequence of resolution steps produces the empty clause. We leave it as an exercise to show that ground resolution is a sound and complete refutation procedure for the predicate calculus.

Of course, ground resolution is hardly a useful refutation procedure for the predicate calculus since the number of ground terms is both unbounded and unstructured. Since it is unbounded, we have no assurance that a refutation will be found within a given number of steps; since it is unstructured, we have no guidance on how to choose clauses to resolve. We cannot solve the first problem since there is no decision procedure for validity in the predicate calculus. However, in 1965, Robinson showed that resolution is often a practical procedure when it is done on clauses that are not ground.

7.6 Substitution

Definition 7.30 A *substitution* of terms for variables is a set

$$\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\},$$

where each x_i is a distinct variable and each t_i is a term which is not identical to the corresponding variable x_i. The *empty* substitution is defined by the empty set. □

Lower-case Greek letters $\{\lambda, \mu, \sigma, \theta\}$ will be used to denote substitutions. The empty substitution is denoted ϵ.

Definition 7.31 An *expression* is a term, a literal, a clause or a set of clauses. Let E be an expression and $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ a substitution. An *instance* $E\theta$ of E is obtained by simultaneously replacing each occurrence of x_i in E by t_i. □

Example 7.32 Here is an expression E, a substitution θ and the instance $E\theta$.

$$E = p(x) \vee q(f(y)) \qquad \theta = \{x \leftarrow y,\ y \leftarrow f(a)\} \qquad E\theta = p(y) \vee q(f(f(a))).$$

The word 'simultaneously' means that one does *not* substitute y for x in E to obtain $p(y) \vee q(f(f(a)))$ and then $f(a)$ for y to obtain $p(f(a)) \vee q(f(f(a)))$. □

Definition 7.33 Let $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ and $\sigma = \{y_1 \leftarrow s_1, \ldots, y_k \leftarrow s_k\}$ be two substitutions. Let X and Y be the sets of variables substituted for in θ and σ, respectively. $\theta\sigma$, the *composition of* θ *and* σ, is the substitution

$$\theta\sigma = \{x_i \leftarrow t_i\sigma \mid x_i \in X,\ x_i \neq t_i\sigma\} \cup \{y_j \leftarrow s_j \mid y_j \in Y,\ y_j \notin X\}.$$

In words: apply the substitution σ to the terms t_i of θ (provided that the resulting substitutions do not collapse to $x_i \leftarrow x_i$) and then append the substitutions from σ whose variables do not already appear in θ. □

Example 7.34 Let

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

and let $E = p(u, v, x, y, z)$. Then

$$\begin{aligned} \theta\sigma &= \{x \leftarrow f(g(a)),\ y \leftarrow f(a),\ u \leftarrow z,\ v \leftarrow f(f(a))\} \\ E(\theta\sigma) &= p(z, f(f(a)), f(g(a)), f(a), z), \end{aligned}$$

where we have deleted $\{z \leftarrow u\}\sigma = \{z \leftarrow z\}$ which is a vacuous substitution, and $\{y \leftarrow g(a)\}$ since the 'original' occurrences of y in E have been replaced by $f(a)$, and 'new' occurrences of y introduced by θ will already have been replaced by $g(a)$ from σ. Note that

$$\begin{aligned} E\theta &= p(u, v, f(y), f(a), u) \\ (E\theta)\sigma &= p(z, f(f(a)), f(g(a)), f(a), z), \end{aligned}$$

so that $E(\theta\sigma) = (E\theta)\sigma$. □

This equality is true in general.

Lemma 7.35 *Let E be an expression and let θ, σ be substitutions. Then $E(\theta\sigma) = (E\theta)\sigma$.*

Proof: Let E be a variable z. If z is not substituted for in θ or σ, the result is trivial. If $z = x_i$ for some $\{x_i \leftarrow t_i\}$ in θ, then $(z\theta)\sigma = t_i\sigma = z(\theta\sigma)$ by the definition of composition. If $z = y_j$ for some $\{y_j \leftarrow s_j\}$ in σ and $z \neq x_i$ for all i, then $(z\theta)\sigma = z\sigma = s_j = z(\theta\sigma)$. The result follows by induction on the structure of E. ∎

The composition of substitutions is associative.

Lemma 7.36 $\theta(\sigma\,\lambda) = (\theta\sigma)\,\lambda$.

Proof: Exercise. ∎

7.7 Unification

The two (non-ground) clauses $p(f(x), g(y))$ and $\neg p(f(f(a)), g(z))$ cannot be resolved because they do not clash. However, under the substitution

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

we obtain the two clashing clauses:

$$p(f(f(a)), g(f(g(a)))) \qquad \neg p(f(f(a)), g(f(g(a)))).$$

Obviously, this is not the only substitution that transforms these clauses into clashing ground clauses. Another, simpler, substitution is $\{x \leftarrow f(a),\ y \leftarrow a,\ z \leftarrow a\}$ giving

$$p(f(f(a)), g(a)) \qquad \neg p(f(f(a)), g(a)).$$

It is not difficult to see that once the substitution $\{x \leftarrow f(a),\ z \leftarrow y\}$ is made giving

$$p(f(f(a)), g(y)) \qquad \neg p(f(f(a)), g(y))$$

any further substitution of a ground term for y will produce clashing ground literals. However, these non-ground literals already have the form of clashing literals, and we can directly resolve the clauses without reducing them to ground clauses. The following definition formalizes these concepts.

Definition 7.37 Given a set of atoms, a *unifier* is a substitution that makes the atoms of the set identical. A *most general unifier (mgu)* is a unifier μ such that any other unifier θ can be obtained from μ by a further substitution λ such that $\theta = \mu\,\lambda$. □

Not all atoms are unifiable. It is clearly impossible to unify atoms whose predicate symbols are different (such as $p(x)$ and $q(x)$) and terms whose outer function symbols are different (such as $p(f(x))$ and $p(g(y))$). A more tricky case is shown by the atoms $p(x)$ and $p(f(x))$. Since x *occurs* within the larger term $f(x)$, any substitution—which must substitute simultaneously in both atoms—cannot unify them. It turns out that as long as these conditions do not hold the atoms will be unifiable. We now describe and prove the correctness of an algorithm for unifying atoms.

Trivially, two atoms are unifiable only if they have the same predicate letter of the same arity. Thus the unifiability of atoms is more conveniently described in terms of the unifiability of the arguments, that is, the *unifiability of a set of terms*. The set of terms to be unified will be written as a set of term equations.

Example 7.38 The unifiability of the atoms $p(f(x), g(y))$ and $p(f(f(a)), g(z))$, is expressed by the set of term equations:

$$\begin{aligned} f(x) &= f(f(a)) \\ g(y) &= g(z). \end{aligned}$$

□

Definition 7.39 A set of term equations is in *solved form* if

- All equations are of the form $x_i = t_i$ where x_i is a variable.
- Each variable x_i that appears on the left-hand side of an equation does not appear elsewhere in the set.

A set of equations in solved form defines a substitution $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$. □

The following algorithm transforms a set of unifiable term equations into a set of equations in solved form, or reports if they are not unifiable. We show that the substitution defined by the set in solved form is a most general unifier of the original set of term equations, and hence of the atoms from which the terms were taken.

Algorithm 7.40 (Unification algorithm)
Input: A set of term equations.
Output: A set of term equations in solved form or 'not unifiable'.

Perform the following transformations on the set of equations as long as any one of them is applicable:

1. Transform $t = x$, where t is not a variable, to $x = t$.
2. Erase the equation $x = x$.

3. Let $t' = t''$ be an equation where t', t'' are not variables. If the outermost function symbols of t' and t'' are not identical, terminate the algorithm with failure: the set of terms is not unifiable. Otherwise, replace the equation $f(t'_1, \ldots, t'_k) = f(t''_1, \ldots, t''_k)$ by the k equations $t'_1 = t''_1, \ldots, t'_k = t''_k$.
4. Let $x = t$ be an equation such that x has another occurrence in the set of equations. If x occurs in t, terminate the algorithm with failure: the set of terms is not unifiable. Otherwise, transform the set by replacing all occurrences of x in other equations by t.

□

Example 7.41 Consider the following set of two equations:

$$\begin{aligned} g(y) &= x \\ f(x, h(x), y) &= f(g(z), w, z). \end{aligned}$$

Apply rule 1 to the first equation and rule 3 to the second equation:

$$\begin{aligned} x &= g(y) \\ x &= g(z) \\ h(x) &= w \\ y &= z. \end{aligned}$$

Apply rule 4 on the second equation to replace the other occurrence of x by $g(z)$:

$$\begin{aligned} g(z) &= g(y) \\ x &= g(z) \\ h(g(z)) &= w \\ y &= z. \end{aligned}$$

Apply rule 3 to the first equation:

$$\begin{aligned} z &= y \\ x &= g(z) \\ h(g(z)) &= w \\ y &= z. \end{aligned}$$

Apply rule 4 on the last equation to replace y by z in the first equation and then erase the result $z = z$ using rule 2:

$$\begin{aligned} x &= g(z) \\ h(g(z)) &= w \\ y &= z. \end{aligned}$$

Finally, transform the second equation by rule 1:

$$\begin{aligned} x &= g(z) \\ w &= h(g(z)) \\ y &= z. \end{aligned}$$

This successfully terminates the algorithm. We claim that

$$\{x \leftarrow g(z),\ w \leftarrow h(g(z)),\ y \leftarrow z\}$$

is a most general unifier of the original set of equations. We leave it to the reader to check that the substitution does in fact unify the original set of term equations and further to check that the unifier:

$$\{x \leftarrow g(f(a)), y \leftarrow f(a), w \leftarrow h(g(f(a))), z \leftarrow f(a)\}$$

can be obtained by applying the mgu followed by the substitution $\{z \leftarrow f(a)\}$. □

Theorem 7.42 *The unification algorithm terminates. If the algorithm terminates with failure, there is no unifier for the set of term equations. If it terminates successfully, the resulting set of equations is in solved form and defines an mgu*

$$\mu = \{x_1 \leftarrow t_1,\ \ldots,\ x_n \leftarrow t_n\}$$

of the set of equations.

Proof: Obviously, rules 1-3 can be used only finitely many times without using rule 4. Let m be the number of distinct variables in the set of equations. Rule 4 can be used at most m times since it removes all occurrences, except one, of a variable and can never be used twice on the same variable. Thus the algorithm terminates.

The algorithm terminates with failure in rule 2 if the function symbols are distinct, and in rule 4 if a variable occurs within a term in the same equation. In both cases there can be no unifier.

We now show that if the algorithm terminates successfully, the equations define an mgu. Upon successful termination, the set of equations is in solved form since if any equation is not of the form $x = t$, rule 1 or rule 2 or rule 3 could be applied. If x occurred in some other equation then rule 4 could be applied.

Define a transformation as an *equivalence transformation* if it preserves sets of unifiers of the equations. Obviously, rules 1 and 2 are equivalence transformations. Consider now an application of rule 3 for $t' = f(t'_1, \ldots, t'_k)$ and $t'' = f(t''_1, \ldots, t''_k)$. If $t'\sigma = t''\sigma$, by the inductive definition of a term this can only be true if $t'_i\sigma = t''_i\sigma$ for all i. Conversely, if some unifier σ makes $t'_i = t''_i$ for all i, then σ is a unifier for $t' = t''$. Thus rule 3 is an equivalence transformation.

Let $t_1 = t_2$ be transformed into $u_1 = u_2$ by rule 4 on $x = t$. After applying the rule, $x = t$ remains in the set. So any unifier σ for the set must make $x\sigma = t\sigma$. Then

$$u_i\sigma = (t_i\{x \leftarrow t\})\sigma = t_i(\{x \leftarrow t\}\sigma) = t_i\sigma$$

by the associativity of substitution and by the definition of composition of substitution using the fact that $x\sigma = t\sigma$. So if σ is a unifier of $t_1 = t_2$, then $u_1\sigma = t_1\sigma = t_2\sigma = u_2\sigma$ and σ is a unifier of $u_1 = u_2$.

Finally, the substitution defined by the set is an mgu. We have just proved that the original set and the solved set of equations have the *same* set of unifiers. But the solved set itself defines a substitution (replacements of terms for variables) which is a unifier. Since the transformations were equivalence transformations, no equation can be removed from the set without destroying the property that it is a unifier. Thus any unifier for the set can only substitute more complicated terms for the same variables or substitute for other variables. That is, if μ is

$$\mu = \{x_1 \leftarrow t_1,\ \ldots,\ x_n \leftarrow t_n\},$$

any other unifier σ can be written

$$\sigma = \{x_1 \leftarrow t'_1,\ \ldots,\ x_n \leftarrow t'_n\} \ \cup\ \{y_1 \leftarrow s_1,\ \ldots,\ y_m \leftarrow s_m\}$$

which is $\sigma = \mu\lambda$ by definition of composition for some substitution λ. But a unifier μ is an mgu exactly if every unifier σ can be written as a composition of μ followed by another substitution. ▮

The algorithm is nondeterministic because we may choose to apply a rule to any equation to which it is applicable. A deterministic algorithm can be obtained by specifying the order in which to apply the rules. One such deterministic algorithm is obtained by considering the set of equations as a queue. A rule is applied to the first element of the queue and then goes to the end of the queue. If new equations are created by rule 3, they are added to the beginning of the queue.

Example 7.43 Here is the unification for Example 7.41 using this data structure:

$$\begin{array}{llllll}
\langle & g(y) = x, & f(x, h(x), y) = f(g(z), w, z) & & & \rangle \\
\langle & x = g(y), & f(x, h(x), y) = f(g(z), w, z) & & & \rangle \\
\langle & f(g(y), h(g(y)), y) = f(g(z), w, z), & x = g(y) & & & \rangle \\
\langle & g(y) = g(z), & h(g(y)) = w, & y = z, & x = g(y) & \rangle \\
\langle & y = z, & h(g(y)) = w, & y = z, & x = g(y) & \rangle \\
\langle & h(g(z)) = w, & z = z, & x = g(z), & y = z & \rangle \\
\langle & z = z, & x = g(z), & y = z, & w = h(g(z)) & \rangle \\
\langle & x = g(z), & y = z, & w = h(g(z)) & & \rangle
\end{array}$$

□

This algorithm is equivalent to the original algorithm given by Robinson. Since Robinson's algorithm appears in most other works on resolution, we will describe it here.

*Robinson's unification algorithm**

Definition 7.44 Let A and A' be two atoms with the same predicate symbols. Considering them as sequences of symbols, let k be the left-most position at which the sequences are different. The pair of terms $\{t, t'\}$ beginning at position k in A and A' is called the *disagreement set* of the two atoms. □

Algorithm 7.45 (Robinson's unification algorithm)
Input: Two atoms A and A' with the same predicate symbol.
Output: A most general unifier for A and A' or 'not unifiable'.

Initialize the algorithm by letting $A_0 = A$ and $A'_0 = A'$. For all i, assume that A_i and A'_i have already been constructed. Perform the following step.

- Let $\{t, t'\}$ be the disagreement set of A, A'. If one term is a variable x_{i+1} and the other is a term t_{i+1} such that x_{i+1} does not occur in t_{i+1}, let $\sigma_{i+1} = \{x_{i+1} \leftarrow t_{i+1}\}$ and $A_{i+1} = A_i\sigma_{i+1}, A'_{i+1} = A'_i\sigma_{i+1}$.

If it is impossible to perform the step (because both elements of the disagreement set are compound terms or because the occur check fails), the atoms are not unifiable. If after some step $A_n = A'_n$, A, A' are unifiable and the mgu is $\mu = \sigma_i \cdots \sigma_n$. □

See Lloyd (1987, Section 1.4) for the proof of the correctness of the algorithm.

Example 7.46 Let $A = p(g(y), f(x, h(x), y))$ and $A' = p(x, f(g(z), w, z))$. The initial disagreement set is $\{x, g(y)\}$. One term is a variable which does not occur in the other so $\sigma_1 = \{x \leftarrow g(y)\}$, and

$$\begin{aligned} A\sigma_1 &= p(g(y), f(g(y), h(g(y)), y)) \\ A'\sigma_1 &= p(g(y), f(g(z), w, z)). \end{aligned}$$

The next disagreement set is $\{y, z\}$ so $\sigma_2 = \{y \leftarrow z\}$, and

$$\begin{aligned} A\sigma_1\sigma_2 &= p(g(z), f(g(z), h(g(z)), z)) \\ A'\sigma_1\sigma_2 &= p(g(z), f(g(z), w, z)). \end{aligned}$$

The third disagreement set is $\{w, h(g(z))\}$ so $\sigma_3 = \{w \leftarrow h(g(z))\}$, and

$$\begin{aligned} A\sigma_1\sigma_2\sigma_3 &= p(g(z), f(g(z), h(g(z)), z)) \\ A'\sigma_1\sigma_2\sigma_3 &= p(g(z), f(g(z), h(g(z)), z)). \end{aligned}$$

Since $A\sigma_1\sigma_2\sigma_3 = A'\sigma_1\sigma_2\sigma_3$, the atoms are unifiable and the most general unifier is

$$\mu = \sigma_1\sigma_2\sigma_3 = \{x \leftarrow g(z),\ y \leftarrow z,\ w \leftarrow h(g(z))\}.$$

□

The occur check

Algorithms for unification can be extremely inefficient because the *occur check*—the check that a term to be substituted for a variable does not contain the variable—can be exponential in the size of the terms being unified.

Example 7.47 To unify:

$$\begin{aligned} x_1 &= f(x_0, x_0) \\ x_2 &= f(x_1, x_1) \\ x_3 &= f(x_2, x_2) \\ &\vdots \end{aligned}$$

we successively create the equations:

$$\begin{aligned} x_2 &= f(f(x_0, x_0), f(x_0, x_0)) \\ x_3 &= f(f(f(x_0, x_0), f(x_0, x_0)), f(f(x_0, x_0), f(x_0, x_0))) \end{aligned}$$

and so on. The check that x_i does not occur in the term in the right-hand side must look at 2^i variables. □

In the application of unification to logic programming, the occur check is simply ignored and the risk of an illegal substitution is taken.

*Implementation**

To implement the algorithm for solving a set of term equations, we maintain a list of the equations, and traverse the list, attempting to apply each of the rules to the current equation. The predicate `solve` is called with a list of equations and returns a list of substitutions written as equations $x = t$, where x is a variable and t is a term. It is convenient to maintain the list in two parts, one to the left of the current equation and one that includes the current equation as its head and the rest of the equations as its tail. The `solve` predicate will have four parameters: two for the equation list, a third for status information and the fourth will return the solved set. The initial call is:

```
solve(Eq, Subst) :-
  solve([], Eq, notmodified, Subst).
```

The status is passed down the recursive calls to `solve` and is used to terminate the recursion as necessary, for example, if either rule 3 or rule 4 fails. The equation that caused the failure is returned together with the failure indication for printing.

```
solve(_, [Current|_], failure3, [failure3, Current]) :- !.
solve([Current], _,   failure4, [failure4, Current]) :- !.
```

Each rule is applied in turn; if successful, it sets the status to `modified` as an indication to the list traversal clauses described below.

```
solve(Head, [Current | Tail], _, Result) :-
  rule1(Current, Current1), !,
  solve(Head, [Current1 | Tail], modified, Result).
solve(Head, [Current | Tail], _, Result) :-
  rule2(Current), !,
  solve(Head, Tail, modified, Result).
```

The set of equations returned by rule 3 replaces the current equation and is appended in front of the remaining equations.

```
solve(Head, [Current | Tail], _, Result) :-
  rule3(Current, NewList, Status), !,
  append(NewList, Tail, NewTail),
  solve(Head, NewTail, Status, Result).
```

When a substitution is performed in rule 4, it must be performed on *all* the equations, including those that have already been checked.

```
solve(Head, [Current | Tail], _, Result) :-
  append(Head, Tail, List),
  rule4(Current, List, NewList, Status), !,
  solve([Current], NewList, Status, Result).
```

The next three clauses traverse the list. If no equation applies, it goes on to the next one. When the end of the list is reached, another traversal is initiated if the previous one made any modifications to the list.

```
solve(Head, [Current | Rest], Mod, Result) :- !,
  append(Head, [Current], NewHead),
  solve(NewHead, Rest, Mod, Result).
solve(List, [], modified, Result) :- !,
  solve([], List, notmodified, Result).
solve(Result, [], _, Result).
```

To prevent confusion between the equality operator of the term equation and the Prolog equality operator, the former is explicitly defined using the operator `eq`. Rules 1 and 2 are straightforward.

```
rule1(T eq X, X eq T) :- nonvar(T), var(X).

rule2(X eq Y) :- X == Y.
```


Rule 3 compares the outermost functors and if they are the same, calls `new_equations` (source omitted here) to pair the subterms.

```
rule3(T1 eq T2, List, modified) :-
  T1 =..  [F | Subterms1], T2 =..  [F | Subterms2],
  new_equations(Subterms1, Subterms2, List).
rule3(T1 eq T2, [T1 eq T2], failure3) :-
  T1 =..  [F1 | _], T2 =..  [F2 | _], F1 \= F2.
```

Rule 4 reports failure if the occurs-check fails. Otherwise, it calls `subst_list` (source omitted here) to perform the substitutions. If nothing is changed, the predicate fails, initiating traversal to the next equation in the list.

```
rule4(X eq T, List, List, failure4) :-
  var(X), occur(X, T), !.
rule4(X eq T, List, NewList, modified) :-
  var(X),
  subst_list(X, T, List, NewList),
  List \== NewList.
```

`occur(X,T)` traverses the list and succeeds as soon as it finds an occurrence of the variable `X` in the term `T`. `occur_list` is used to check the list of subterms.

```
occur(X, T) :- X == T, !.
occur(X, T) :- T =..  [_ | Subterms], !,
  occur_list(X, Subterms).
occur(_, _) :- fail.

occur_list(X, [Head | _]) :- occur(X, Head), !.
occur_list(_, [])         :- !, fail.
occur_list(X, [_ | Tail]) :- occur_list(X, Tail).
```

To unify a pair of atomic proposition, simply check that the predicate symbols are identical, create a set of equations from the arguments and call `solve`.

```
unify(A1, A2, Subst) :-
  A1 =..  [Pred | Args1],
  A2 =..  [Pred | Args2],
  new_equations(Args1, Args2, Eq),
  solve(Eq, Subst).
```

7.8 General resolution

The resolution rule can be applied directly to non-ground clauses by performing unification as an integral part of the rule.

Definition 7.48 Let $L = \{l_1, \ldots, l_n\}$ be a set of literals. Then $L^c = \{l_1^c, \ldots, l_n^c\}$. □

Rule 7.49 (General resolution rule) Let C_1, C_2 be clauses *with no variables in common*. Let $L_1 = \{l_{11}, \ldots, l_{1n_1}\} \subseteq C_1$ and $L_2 = \{l_{21}, \ldots, l_{2n_2}\} \subseteq C_2$ be subsets of literals such that L_1 and L_2^c can be unified by an mgu σ. C_1 and C_2 are said to be *clashing clauses* and to *clash* on the sets of literals L_1 and L_2. C, the *resolvent* of C_1 and C_2, is the clause:

$$Res(C_1, C_2) = (C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma).$$

□

Example 7.50 Given the two clauses

$$p(f(x), g(y)) \vee q(x, y) \qquad \neg p(f(f(a)), g(z)) \vee q(f(a), g(z)),$$

an mgu for $L_1 = \{p(f(x), g(y))\}$ and $L_2^c = \{p(f(f(a)), g(z))\}$ is

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

and the clauses resolve to give $q(f(a), z) \vee q(f(a), g(z))$. □

The general resolution rule requires that the clauses have no variables in common. This is done by *standardizing apart*, which is to rename all the variables in one of the clauses before it is used in the resolution rule. Recall that variables in a clause are implicitly universally quantified so renaming does not change satisfiability.

Example 7.51 To resolve the two clauses $p(f(x))$ and $\neg p(x)$, first rename variables of the second clause $\neg p(x')$. An mgu is $\{x' \leftarrow f(x)\}$, and $p(f(x))$ and $\neg p(f(x))$ resolve to $\square$. The clauses represent the formulas $\forall x p(f(x))$ and $\forall x \neg p(x)$, and it is obvious that their conjunction $\forall x p(f(x)) \wedge \forall x \neg p(x)$ is unsatisfiable. □

The resolution rule is defined on clauses as *sets* of literals, and the mgu σ may unify sets of clashing literals. Collapsing of identical literals in a clause is called *factoring*.

Example 7.52 Let $C_1 = \{p(x), p(y)\}$ and $C_2 = \{\neg p(x), \neg p(y)\}$. First standardize apart so that $C_2' = \{\neg p(x'), \neg p(y')\}$. Let $L_1 = C_1 = \{p(x), p(y)\}$ and $L_2^c = C_2'^c = \{p(x'), p(y')\}$; these sets have an mgu $\sigma = \{y \leftarrow x, x' \leftarrow x, y' \leftarrow x\}$.

$$Res(C_1, C_2) = (\{p(x)\} - \{p(x)\}) \cup (\{\neg p(x)\} - \{\neg p(x)\}) = \square.$$

□

In this example, the empty clause cannot be obtained without factoring, but in general, we will talk about clashing literals rather than clashing sets of literals when no confusion is possible.

Algorithm 7.53 (General Resolution Procedure)
Input: A set of clauses S.
Output: The clauses are satisfiable or unsatisfiable. However, the algorithm may not terminate.

Define $S_0 = S$. Assume that S_i has been constructed. *Choose* clashing clauses $C_1, C_2 \in S_i$ and let $C = Res(C_1, C_2)$. If $C = \square$, terminate the procedure: S is unsatisfiable. Otherwise, construct $S_{i+1} = S_i \cup \{C\}$. If $S_{i+1} = S_i$ for all possible pairs of clashing clauses, terminate the procedure: S is satisfiable. □

While an unsatisfiable set of clauses will eventually produce $\square$ under a suitable systematic execution of the procedure, the existence of infinite models means that the execution of the resolution procedure on a satisfiable set of clauses may never terminate. Since we can never know if the procedure will ever terminate, general resolution is not a decision procedure.

Example 7.54 Lines 1–7 contain an unsatisfiable set of clauses. This is shown in lines 8–15 which are a refutation by the resolution procedure. Each line contains the resolvent, the mgu and the numbers of the parent clauses.

1.	$\neg p(x) \vee q(x) \vee r(x, f(x))$		
2.	$\neg p(x) \vee q(x) \vee s(f(x))$		
3.	$t(a)$		
4.	$p(a)$		
5.	$\neg r(a, y) \vee t(y)$		
6.	$\neg t(x) \vee \neg q(x)$		
7.	$\neg t(x) \vee \neg s(x)$		
8.	$\neg q(a)$	$x \leftarrow a$	3, 6
9.	$q(a) \vee s(f(a))$	$x \leftarrow a$	2, 4
10.	$s(f(a))$		8, 9
11.	$q(a) \vee r(a, f(a))$	$x \leftarrow a$	1, 4
12.	$r(a, f(a))$		8, 11
13.	$t(f(a))$	$y \leftarrow f(a)$	5, 12
14.	$\neg s(f(a))$	$x \leftarrow f(a)$	7, 13
15.	$\square$		10, 14

□

Example 7.55 Here is another example of a resolution refutation showing variable renaming and mgu's which do not produce ground clauses. The first four clauses form the set of clauses to be refuted.

1.	$\neg p(x,y) \vee p(y,x)$		
2.	$\neg p(x,y) \vee \neg p(y,z) \vee p(x,z)$		
3.	$p(x,f(x))$		
4.	$\neg p(x,x)$		
$3'$.	$p(x',f(x'))$		Rename 3
5.	$p(f(x),x)$	$\sigma_1 = \{y \leftarrow f(x), x' \leftarrow x\}$	$1, 3'$
$3''$.	$p(x'',f(x''))$		Rename 3
6.	$\neg p(f(x),z) \vee p(x,z)$	$\sigma_2 = \{y \leftarrow f(x), x'' \leftarrow x\}$	$2, 3''$
$5'''$.	$p(f(x'''),x''')$		Rename 5
7.	$p(x,x)$	$\sigma_3 = \{z \leftarrow x, x''' \leftarrow x\}$	$6, 5'''$
$4''''$.	$\neg p(x'''',x'''')$		Rename 4
8.	$\square$	$\sigma_4 = \{x'''' \leftarrow x\}$	$7, 4''''$

If we concatenate the substitutions, we get:

$$\sigma = \sigma_1\sigma_2\sigma_3\sigma_4 = \{y \leftarrow f(x), z \leftarrow x, x' \leftarrow x, x'' \leftarrow x, x''' \leftarrow x, x'''' \leftarrow x\}.$$

Restricted to the variables of the original clauses, $\sigma = \{y \leftarrow f(x), z \leftarrow x\}$. This will be of importance in the next chapter when we discuss logic programming. □

*Soundness and completeness**

The next step is to prove soundness and completeness of the general resolution procedure. There is a technical difficulty in the completeness proof. Using Herbrand's theorem and semantic trees, we can prove that there is a *ground resolution refutation* of an unsatisfiable set of clauses. However, this does not generalize into a proof for general resolution because the concept of semantic tree does not generalize—the variables give rise to a potentially infinite number of elements of the Herbrand base. The difficulty is overcome by taking a ground resolution refutation and 'lifting' it into a more abstract general refutation. This is only a theoretical issue and does not mean that resolution refutations should be constructed in this manner.

The problem is that several literals in C_1 or C_2 might collapse into one literal under the substitutions that produce the ground instances C_1' and C_2' to be resolved.

Example 7.56 The clauses

$$\begin{aligned} C_1 &= p(x) \vee p(f(y)) \vee p(f(z)) \vee q(x) \\ C_2 &= \neg p(f(u)) \vee \neg p(w) \vee r(u) \end{aligned}$$

under the substitution $\{x \leftarrow f(a),\ y \leftarrow a,\ z \leftarrow a,\ u \leftarrow a,\ w \leftarrow f(a)\}$ give the ground clauses

$$C_1' = p(f(a)) \vee q(f(a)) \qquad C_2' = \neg p(f(a)) \vee r(a),$$

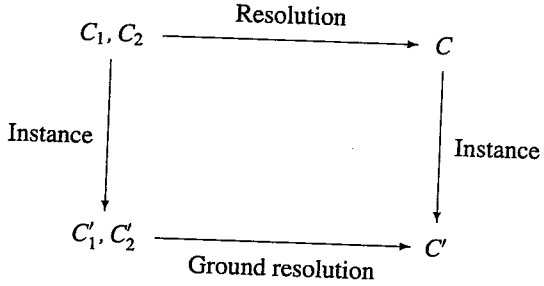
which resolve giving $C' = q(f(a)) \vee r(a)$. The lifting lemma claims that there is a clause $C = q(f(u)) \vee r(u)$ which is the resolvent of C_1 and C_2, such that C' is a ground instance of C. This can be seen by using the unification algorithm to obtain an mgu

$$\{x \leftarrow f(u),\ y \leftarrow u,\ z \leftarrow u,\ w \leftarrow f(u)\}$$

of C_1 and C_2, which then resolve giving C. □

Theorem 7.57 (Lifting Lemma) *Let C_1', C_2' be ground instances of C_1, C_2, respectively. Let C' be a ground resolvent of C_1' and C_2'. Then there is a resolvent C of C_1 and C_2 such that C' is a ground instance of C.*

The relationships among the clauses are displayed in the following diagram.



Proof: To aid in understanding the proof, the computations for Example 7.56 are shown in Figure 7.2.

First standardize apart so that the variables in C_1 are different from those in C_2. Let $l \in C_1', l^c \in C_2'$ be the clashing literals in the ground resolution. Since C_1' is an instance of C_1 and $l \in C_1'$, there must be a set of literals $L_1 \subseteq C_1$ such that l is an instance of each literal in L_1. Similarly, there is a set $L_2 \subseteq C_2$ for l^c. Let λ_1 and λ_2 mgu's for L_1 and L_2, respectively, and let $\lambda = \lambda_1 \cup \lambda_2$. λ is a substitution since L_1 and L_2 have no variables in common. $C_1\lambda$ and $C_2\lambda$ are clashing clauses on the sets of literals $L_1\lambda, L_2\lambda$; let σ be their mgu so that their resolvent is:

$$\begin{aligned} C &= ((C_1\lambda)\sigma - \{(L_1\lambda)\sigma\}) \cup ((C_2\lambda)\sigma - \{(L_2\lambda)\sigma\}) \\ &= (C_1(\lambda\sigma) - \{L_1(\lambda\sigma)\}) \cup (C_2(\lambda\sigma) - \{(L_2(\lambda\sigma)\}), \end{aligned}$$

using the associativity of substitution. C is a resolvent of C_1 and C_2 provided that $\lambda\sigma$ is an mgu of L_1 and L_2^c. But λ is already reduced to equations of the form $x \leftarrow t$ for distinct variables x and σ is constructed to be an mgu, so $\lambda\sigma$ is a reduced set of equations, all of which are necessary to unify L_1 and L_2^c. Hence $\lambda\sigma$ is an mgu.

$$
\begin{array}{lcl}
C_1 & = & \{p(x),\ p(f(y)),\ p(f(z)),\ q(x)\} \\
C_2 & = & \{\neg p(f(u)),\ \neg p(w),\ r(u)\} \\
 & & \\
\theta_1 & = & \{x \leftarrow f(a),\ y \leftarrow a,\ z \leftarrow a\} \\
\theta_2 & = & \{u \leftarrow a,\ w \leftarrow f(a)\} \\
 & & \\
C_1' & = & C_1\theta_1 = \{p(f(a)),\ q(f(a))\} \\
C_2' & = & C_2\theta_2 = \{\neg p(f(a)),\ r(a)\} \\
C' & = & Res(C_1, C_2) = \{q(f(a)),\ r(a)\} \\
 & & \\
L_1 & = & \{p(x),\ p(f(y)),\ p(f(z))\} \\
\lambda_1 & = & \{x \leftarrow f(y),\ z \leftarrow y\} \\
L_1\,\lambda_1 & = & \{p(f(y))\} \\
 & & \\
L_2 & = & \{\neg p(f(u)),\ \neg p(w)\} \\
\lambda_2 & = & \{w \leftarrow f(u)\} \\
L_2\,\lambda_2 & = & \{\neg p(f(u))\} \\
 & & \\
\lambda & = & \lambda_1 \cup \lambda_2 = \{x \leftarrow f(y), z \leftarrow y, w \leftarrow f(u)\} \\
L_1\,\lambda & = & \{p(f(y))\} \\
C_1\,\lambda & = & \{p(f(y)),\ q(f(y))\} \\
L_2\,\lambda & = & \{\neg p(f(u))\} \\
C_2\,\lambda & = & \{\neg p(f(u)),\ r(u)\} \\
 & & \\
\sigma & = & \{u \leftarrow y\} \\
C & = & Res(C_1\,\lambda, C_2\,\lambda) = \{q(f(y)),\ r(y)\},\ \text{using } \sigma \\
 & & \\
\lambda\sigma & = & \{x \leftarrow f(y),\ z \leftarrow y,\ w \leftarrow f(y),\ u \leftarrow y\} \\
C_1\,\lambda\sigma & = & \{p(f(y)),\ q(f(y))\} \\
C_2\,\lambda\sigma & = & \{\neg p(f(y)),\ r(y)\} \\
C & = & Res(C_1, C_2) = \{q(f(y)),\ r(y)\},\ \text{using } \lambda\sigma \\
 & & \\
\theta_1' & = & \{y \leftarrow a\} \\
C_1' & = & C_1\theta_1 = \{p(f(a)),\ q(f(a))\} = C_1\,\lambda\sigma\theta_1 \\
\theta_2' & = & \{y \leftarrow a\} \\
C_2' & = & C_2\theta_2 = \{\neg p(f(a)),\ r(a)\} = C_2\,\lambda\sigma\theta_2 \\
 & & \\
\theta' & = & \{y \leftarrow a\} \\
C' & = & Res(C_1', C_2') = \{q(f(a)),\ r(a)\}
\end{array}
$$

Figure 7.2 Example for the lifting lemma

Since C_1' and C_2' are ground instances of C_1 and C_2:

$$C_1' = C_1\theta_1 = C_1\,\lambda\sigma\theta_1' \qquad C_2' = C_2\theta_2 = C_2\,\lambda\sigma\theta_2'$$

for some substitutions $\theta_1, \theta_2, \theta_1', \theta_2'$. Let $\theta' = \theta_1' \cup \theta_2'$. Then $C' = C\theta'$ and C' is a ground instance of C. ∎

We now show the soundness and completeness of resolution. The reader should review the proofs in Section 2.10 for the propositional calculus as we will just give the modifications that must be made to those proofs.

Theorem 7.58 (Soundness of resolution) *If the unsatisfiable clause $\square$ is derived during the general resolution procedure then the set of clauses is unsatisfiable.*

Proof: The critical step in the proof is to show that if the parent clauses are satisfiable then so is the resolvent using the mgu σ. If the parent clauses are satisfiable, there is an interpretation $\mathcal{I}$ such that $v_{\mathcal{I}}(C_i) = T$. The clause is implicitly (universally) quantified, so by definition of $v_{\mathcal{I}}$ there are ground instances $C_i' = C_i\,\lambda_i$ such that $v_{\mathcal{I}}(C_i') = T$. Since σ is an mgu, there are substitutions θ_i such that $\lambda_i = \sigma\theta_i$. Then $C_i' = C_i\,\lambda_i = C_i(\sigma\theta_i) = (C_i\sigma)\theta_i$, which shows that $C_i\sigma$ is satisfiable in the same interpretation.

Now we can continue as in the proof in the propositional calculus. Exactly one of $l_1\sigma, l_2^c\sigma$ can be satisfiable in $\mathcal{I}$. Suppose that $v_{\mathcal{I}}(l_1\sigma) = T$. Since we have proved that $C_2\sigma$ is satisfiable, there must be a literal $l' \in C_2$ such that $l' \neq l_2^c$ and $v_{\mathcal{I}}(l'\sigma) = T$. But by the construction of the resolvent, $l' \in C$ so $v_{\mathcal{I}}(C) = T$. ∎

Theorem 7.59 (Completeness of resolution) *If a set of clauses is unsatisfiable then the empty clause $\square$ can be derived by the resolution procedure.*

Proof: The proof is by induction on the closed semantic tree for the set of clauses S. The definition of semantic tree is modified as follows:

> A node is a failure node if the (partial) interpretation defined by a branch falsifies some *ground instance* of a clause in S.

The critical step in the proof is showing that an inference node n can be associated with the resolvent of the clauses on the two failure nodes n_1, n_2 below it. Suppose that C_1, C_2 are associated with the failure nodes. Then there must be ground instances C_1', C_2' which are falsified at the nodes. By construction of the semantic tree, C_1' and C_2' are clashing clauses. Hence they can be resolved to give a clause C' which is falsified by the interpretation at n. By the Lifting Lemma, there is a clause C such that C is the resolvent of C_1' and C_2', and C' *is a ground instance of* C. Hence C is falsified at n and n (or an ancestor of n) is a failure node. ∎

Implementation^P

To use the resolution procedure to search for a refutation of a formula, first transform the formula into clausal form using the predicate `skolem` given in Section 7.2 and then convert it from a universally quantified formula to a set of sets as was done in Section 4.1 for the propositional calculus. The following predicate then searches for a resolution refutation.

```
resolution([]) :- !, fail.
resolution(S) :- member([], S), !.
resolution(S) :-
  member(C1, S), member(C2, S), C1 \== C2,
  copy_term(C2, C2_R),
  clashing(C1, L1, C2_R, L2, Subst),
  delete_lit(C1, L1, Subst, C1P),
  delete_lit(C2_R, L2, Subst, C2P),
  clause_union(C1P, C2P, Resolvent),
  \+ clashing(Resolvent, _, Resolvent, _, _),
  \+ member(Resolvent, S),
  resolution([Resolvent | S]).
```

`copy_term` is used to rename the variables of `C2`. If the clauses clash, the following predicate returns the clashing literals and the substitution which causes the clash. Similarly,

```
clashing(C1, L1, C2, neg L2, Subst) :-
  member(L1, C1), member(neg L2, C2),
  unify(L1, L2, Subst), !.
clashing(C1, neg L1, C2, L2, Subst) :-
  member(neg L1, C1), member(L2, C2),
  unify(L1, L2, Subst), !.
```

`delete_lit(Clause, Literal, Subst, Result)` deletes from `Clause` all the literals that are equal to `Literal` under the substitution `Subst` and returns the `Result`. `clause_union` takes the union of the literals in the two clauses to form the resolvent. As in previous implementations of algorithms for the predicate calculus, library predicates cannot be used as unintended unification of variables must be avoided. The details can be studied in the source archive.

This naive implementation is quite likely to start searching infinite paths. Practical resolution theorem provers are based on variants of the procedure which serve to guide the search.

7.9 Exercises

1. Transform each of the following formulas to clausal form:

$$\forall x(p(x) \rightarrow \exists y q(y)),$$
$$\forall x \forall y(\exists z p(z) \wedge \exists u(q(x,u) \rightarrow \exists v q(y,v))),$$
$$\exists x(\neg \exists y p(y) \rightarrow \exists z(q(z) \rightarrow r(x))).$$

2. For the formulas of the previous exercise, describe the Herbrand universe and the Herbrand base.

3. Prove the converse direction of Skolem's Theorem (Theorem 7.11).

4. Let $A(x_1, \ldots, x_n)$ be a formula with no quantifiers and no function symbols. Prove that $\forall x_1 \cdots \forall x_n A(x_1, \ldots, x_n)$ is satisfiable if and only if it is satisfiable in an interpretation whose domain has only one element.

5. Prove that ground resolution is sound and complete.

6. Let

$$\begin{aligned} \theta &= \{x \leftarrow f(g(y)),\ y \leftarrow u,\ z \leftarrow f(y)\}, \\ \sigma &= \{u \leftarrow y,\ y \leftarrow f(a),\ x \leftarrow g(u)\}, \\ E &= p(x, f(y), g(u), z). \end{aligned}$$

   Show that $E(\theta\sigma) = (E\theta)\sigma$.

7. Prove that the composition of substitutions is associative (Lemma 7.36).

8. Unify the following pairs of atomic formulas, if possible.

$p(a, x, f(g(y)))$	$p(y, f(z), f(z))$,
$p(x, g(f(a)), f(x))$	$p(f(a), y, y)$,
$p(x, g(f(a)), f(x))$	$p(f(y), z, y)$,
$p(a, x, f(g(y)))$	$p(z, h(z,u), f(u))$.

9. A substitution $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ is called *idempotent* iff $\theta = \theta\theta$. Let V be the set of variables occurring in the terms $\{t_1, \ldots, t_n\}$. Prove that θ is idempotent iff $V \cap \{x_1, \ldots, x_n\} = \emptyset$. Show that the mgu's produced by the unification algorithm is idempotent.

10. Prove the validity of (some of) the equivalences in Figure 5.2 by resolution refutation of their negations.

11. ^{P} Modify the program to Skolemize a formula so that operator elimination and application of the De Morgan and distributive laws are done in one pass.

12. [P] Write a program to implement Robinson's unification algorithm.

13. * A set of clauses S is called *renamable-Horn* iff there is a set of propositional letters U such that $R_U(S)$ is a set of Horn clauses. (Recall Definition 4.19 and Lemma 4.20). Prove the following theorem:

Theorem 7.60 (Lewis) *Let $S = \{C_1, \ldots, C_m\}$ be a set of clauses where $C_i = l_1^i \vee \cdots \vee l_{n_i}^i$, and let*

$$S^* = \bigcup_{i=1}^{m} \bigcup_{1 \le j < k \le n_i} (l_j^i \vee l_k^i).$$

Then S is renamable-Horn if and only if S^* is satisfiable.

8 Logic Programming

8.1 Formulas as programs

Most axioms are expressed in the form:

If premise-1 and premise-2 and ... then conclusion.

Formally, this is:

$$A = \forall x_1 \cdots \forall x_k (B_1 \wedge \cdots \wedge B_m \rightarrow B),$$

where B and B_i are atoms. In the degenerate case where there are no antecedents, $A = \forall x_1 \cdots \forall x_k B$. In clausal form, an axiom is $\neg B_1 \vee \cdots \vee \neg B_m \vee B$. To prove that a formula $G = G_1 \wedge \cdots \wedge G_l$ is a logical consequence of a set of axioms, we append $\neg G$ to the set of axioms and try to construct a refutation by resolution. $\neg G$ is called the *goal clause*.

The set of axioms is presumably chosen to be satisfiable, so we won't progress towards a refutation by resolving among the axioms; instead, we resolve the goal clause with an axiom. Looking at the form of the axioms, since $\neg G$ is a disjunction of negative literals, one of the negative literals $\neg G_i$ can only resolve with B, the *single* positive literal of the axiom. The result is a clause all of whose literals are negative:

$$\neg B_1 \vee \cdots \vee \neg B_m \vee \neg G_1 \vee \cdots \vee \neg G_{i-1} \vee \neg G_i \vee \cdots \vee \neg G_l.$$

Thus all resolvents in the refutation will contain only negative literals. Eventually, resolving with axioms having no antecedents, that is, axioms whose clausal forms consists of a single positive literal, will produce progressively shorter resolvents and finally the empty clause.

Example 8.1 Consider a fragment of the theory of strings, where there is a single binary function symbol for concatenation denoted by the infix operator $\cdot$, and three predicates *substr*, *prefix* and *suffix*. The axioms are:

1. $\forall x\ substr(x, x)$
2. $\forall x \forall y \forall z\,((substr(x, y) \wedge suffix(y, z)) \rightarrow substr(x, z))$
3. $\forall x \forall y\ suffix(x, y \cdot x)$
4. $\forall x \forall y \forall z\,((substr(x, y) \wedge prefix(y, z)) \rightarrow substr(x, z))$
5. $\forall x \forall y\ prefix(x, x \cdot y)$,

or in clausal form:

1. $substr(x, x)$
2. $\neg\, substr(x, y) \vee \neg\, suffix(y, z) \vee substr(x, z)$
3. $suffix(x, y \cdot x)$
4. $\neg\, substr(x, y) \vee \neg\, prefix(y, z) \vee substr(x, z)$
5. $prefix(x, x \cdot y)$.

Here is a refutation of $\neg\, substr(a \cdot b \cdot c,\ a \cdot a \cdot b \cdot c \cdot c)$, where the parent clauses of each resolvent are given in the right-hand column.

6.	$\neg\, substr(a \cdot b \cdot c,\ a \cdot a \cdot b \cdot c \cdot c)$	
7.	$\neg\, substr(a \cdot b \cdot c,\ y1) \vee \neg\, suffix(y1,\ a \cdot a \cdot b \cdot c \cdot c)$	6, 2
8.	$\neg\, substr(a \cdot b \cdot c,\ a \cdot b \cdot c \cdot c)$	7, 3
9.	$\neg\, substr(a \cdot b \cdot c,\ y2) \vee \neg\, prefix(y2,\ a \cdot b \cdot c \cdot c)$	8, 4
10.	$\neg\, substr(a \cdot b \cdot c,\ a \cdot b \cdot c)$	9, 5
11.	$\square$	10, 1

∎

Let us write the axioms using the Boolean operator $\leftarrow$ for reverse implication:

$$A = \forall x_1 \cdots \forall x_k (B \leftarrow B_1 \wedge \cdots \wedge B_m).$$

We can interpret the *logical formula* as a *procedure*: to compute B, compute $B_1, \cdots, B_m$.

Example 8.2 The string axioms can be written:

1. $\forall x\ substr(x, x)$
2. $\forall x \forall y \forall z\,(substr(x, z) \leftarrow (suffix(y, z) \wedge substr(x, y)))$
3. $\forall x \forall y\ suffix(x, y \cdot x)$
4. $\forall x \forall y \forall z\,(substr(x, z) \leftarrow (prefix(y, z) \wedge substr(x, y)))$
5. $\forall x \forall y\ prefix(x, x \cdot y)$

and interpreted as:

1. x is a substring of x.
2. To check if x is a substring of z,
   find a suffix y of z and check if x is a substring of y.
3. x is a suffix of $y \cdot x$.
4. To check if x is a substring of z,
   find a prefix y of z and check if x is a substring of y.
5. x is a prefix of $x \cdot y$.

∎

This is not too exciting since the computation simply checks if $substr(a{\cdot}b{\cdot}c, a{\cdot}a{\cdot}b{\cdot}c{\cdot}c)$ is true or not. Suppose, however, that instead of using resolution to determine if this *ground* goal clause is a logical consequence of the axioms, we try to determine if the formula $\exists w\ substr(w,\ a \cdot a \cdot b \cdot c \cdot c)$ is a logical consequence of the axioms. In terms of resolution we try to refute the negation

$$\neg\,(\bigwedge\{Axioms(S)\} \rightarrow \exists w\ substr(w,\ a \cdot a \cdot b \cdot c \cdot c)),$$

which transforms to

$$\bigwedge\{Axioms(S)\} \wedge \forall w \neg\ substr(w,\ a \cdot a \cdot b \cdot c \cdot c),$$

so we still have a set of clauses.

If the resolution refutation is carried out, the final step is to resolve $\neg\ substr(w, a{\cdot}b{\cdot}c)$ with the string axiom $substr(x, x)$. The resolution succeeds with the substitution $\{w \leftarrow a \cdot b \cdot c\}$ and produces the empty clause, completing the refutation. Not only have we proved that $\exists w\ substr(w,\ a \cdot a \cdot b \cdot c \cdot c)$ is a logical consequence of the axioms, but we have also *computed* a value $a \cdot b \cdot c$ for w such that $substr(w,\ a \cdot a \cdot b \cdot c \cdot c)$ is true. The axioms for strings form a *program* that computes *answers* to questions.

This program is highly nondeterministic:

- Given a goal clause such as $\neg\ substr(w,\ y1) \vee \neg\ suffix(y1,\ a \cdot a \cdot b \cdot c \cdot c)$, we can choose to resolve either of the literals in the clause with an axiom clause.
- Once we have chosen a literal such as $\neg\ substr(w,\ y1)$, we can choose to resolve it with any of the axioms whose positive literal clashes with the chosen literal.

The nondeterministic formalism becomes a practical logic programming language by specifying rules for making these choices.

Definition 8.3 A *computation rule* is a rule for choosing a literal in a goal clause to resolve with. A *search rule* is a rule for choosing a clause to resolve with the chosen literal in a goal clause. ∎

The difference between logic programming and (ordinary) algorithmic programming is that in algorithmic programming the control of the computation is *explicitly* constructed by the programmer as part of the program. This can be instantly recognized by the central place occupied by the *control structures* in a language like Pascal:

```
if X > 0 then ...  else ...
while A[I] <> Key do
for I := 1 to 10 do ...
```

In logic programming, the programmer writes declarative logic formulas that describe the relationship between the input and output, and then the compiler, that is, the resolution inference engine together with the search and computation rules, supplies a *uniform* control structure.

Obviously, no matter how efficient a logic programming compiler is, its uniform control structure can never match a control structure hand-crafted for a specific computation. Research in logic programming has explored the trade-offs between pure declarative logic versus impure procedural constructs that allow programmers to write programs whose efficiency is reasonable when compared with procedural languages. In the following section, we will study the theoretical basis of logic programming and then discuss practical logic programming languages.

8.2 SLD-resolution

Definition 8.4 A *Horn clause* is a clause $A \leftarrow B_1, \ldots, B_n$ with at most one positive literal, where the positive literal A is called the *head* and the negative literals B_i are called the *body*. A unit positive Horn clause $A\leftarrow$ is called a *fact*, and a Horn clause with no positive literals $\leftarrow B_1, \ldots, B_n$ is called a *goal clause*. A Horn clause with one positive literal and one or more negative literals is called a *program clause*. □

Definition 8.5 A set of non-goal Horn clauses whose heads have the same predicate letter is called a *procedure*. A set of procedures is called a *(logic) program*. A procedure composed of ground facts only is also called a *database*. □

Example 8.6 The following program has two procedures, one of which is a database.

1. $q(x,y) \leftarrow p(x,y)$
2. $q(x,y) \leftarrow p(x,z), q(z,y)$

3. $p(b,a)$
4. $p(c,a)$
5. $p(d,b)$
6. $p(e,b)$
7. $p(f,b)$
8. $p(h,g)$
9. $p(i,h)$
10. $p(j,h)$

□

Definition 8.7 Let P be a program and G a goal clause. A substitution θ for variables in G is called a *correct answer substitution* if $P \models \forall(\neg G\theta)$ where $\forall$ denotes the universal closure of the variables that are free in $\neg G\theta$. □

Example 8.8 Let P be the set of axioms of arithmetic, G the formula $\neg(6+y=13)$ and θ the substitution $\{y \leftarrow 7\}$. Then $\neg G$ is $6+y=13$ and $\neg G\theta$ is $6+7=13$, so θ is a correct answer substitution $P \models (6+7=13)$. □

A correct answer substitution need not be a ground substitution, that is, a substitution which makes the formula ground. If G is $\neg(x = y+13)$ then $\theta = \{y \leftarrow x-13\}$ is a correct answer substitution since

$$P \models \forall x(x = x - 13 + 13).$$

Note that the substitution θ is required since the closure of $\neg G$ is *not* a logical consequence of P:

$$P \not\models \forall x \forall y(x = y + 13).$$

In general, suppose we want to find out if $B = \exists(B_1 \wedge \ldots \wedge B_n)$ is a logical consequence of a program (set of clauses) P, where $\exists$ denotes the existential closure of the conjunction. Then $P \models B$ if and only if there is some ground substitution σ such that

$$P \models (B_1 \wedge \ldots \wedge B_n)\sigma.$$

Suppose that σ can be written as a composition of substitutions $\sigma = \theta\,\lambda$ such that

$$P \models (B_1 \wedge \ldots \wedge B_n)\theta\,\lambda$$

for *any* ground substitution λ. In fact, such a composition always exists, at worst with $\theta = \sigma$ and $\lambda = \epsilon$ (the empty substitution). It follows that

$$P \models \forall((B_1 \wedge \ldots \wedge B_n)\theta).$$

Thus 'B is a logical consequence of P' is equivalent to θ being a correct answer substitution for the *goal clause* defined by $G = \neg(B_1 \wedge \ldots \wedge B_n)$. Since G is implicitly universally quantified,

$$\forall G = \forall\neg(B_1 \wedge \ldots \wedge B_n) = \neg\exists(B_1 \wedge \ldots \wedge B_n) = \neg B.$$

Thus $P \models B$ if and only if $\models P \rightarrow B$ which is $\not\models \neg(P \rightarrow B)$ or $\not\models P \wedge \neg B$, that is $\not\models P \wedge G$. Since resolution is sound, if $P \wedge G$ leads to a refutation, then $\not\models P \wedge G$, and the substitutions produced by unification define a correct answer substitution in the Herbrand domain.

Example 8.9 Let us work through a refutation of the goal clause $\leftarrow q(y,b),\ q(b,z)$ with the program in Example 8.6. At each step we must choose a literal within the clause and a clause whose head clashes with the literal.

1. Choose $q(y,b)$ and resolve with clause 1 giving $\leftarrow p(y,b),\ q(b,z)$.
2. Choose $p(y,b)$ and resolve with clause 5 giving $\leftarrow q(b,z)$.
   This requires the substitution $\{y \leftarrow d\}$.
3. There is only one literal to choose and we resolve it with clause 1 giving $\leftarrow p(b,z)$.

4. There is only one literal to choose and we resolve it with clause 3 giving $\square$. This requires the substitution $\{z \leftarrow a\}$.

Since the goal clause is $\neg q(y,b) \vee \neg q(b,z)$, it follows that a correct answer substitution is $\{y \leftarrow d, z \leftarrow a\}$ and $P \models q(d,b) \wedge q(b,a)$. From elementary predicate calculus it follows that $P \models \exists y \exists z (q(y,b) \wedge q(b,z))$. ▯

Definition 8.10 (SLD-resolution) Let P be a set of program clauses, R a computation rule and G a goal clause. A *derivation by SLD-resolution* is defined as a sequence of resolution steps between goal clauses and the program clauses. The first goal clause G_0 is G. Assume that G_i has been derived. G_{i+1} is defined by *selecting* a literal $A_i \in G_i$ according to the computation rule R, *choosing* a clause $C_i \in P$ such that the head of C_i unifies with A_i by mgu θ_i and resolving:

$$\begin{aligned} G_i &= \leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n \\ C_i &= A \leftarrow B_1, \ldots, B_k \\ A_i\theta_i &= A\theta_i \\ G_{i+1} &= \leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1}, \ldots, A_n)\theta_i. \end{aligned}$$

An *SLD-refutation* is an SLD-derivation of $\square$. ▯

Soundness and completeness

Theorem 8.11 (Soundness of SLD-resolution) *Let P be a set of program clauses, R a computation rule and G a goal clause. Suppose that there is an SLD-refutation of G. Let $\theta = \theta_1 \cdots \theta_n$ be the sequence of unifiers used in the refutation and let σ be the restriction of θ to the variables of G. Then σ is a correct answer substitution for G.*

Proof: By definition of σ, $G\theta = G\sigma$, so $P \cup \{G\sigma\} = P \cup \{G\theta\}$ which is unsatisfiable by the soundness of resolution. If $P \cup \{G\sigma\}$ is unsatisfiable, then $P \models \neg G\sigma$. Since this is true for any substitution into the free variables of $G\sigma$, $P \models \forall(\neg G\sigma)$. ▮

SLD-refutation is *not* complete for sets of clauses in general. The set of clauses:

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

is unsatisfiable, but has no SLD-resolution since even if a unit 'goal' clause such as p is derived, $\square$ can never be obtained by resolving with a 'program' clause. For Horn clauses, however, SLD-resolution is complete.

Theorem 8.12 (Completeness of SLD-resolution) *Let P be a set of program clauses, R a computation rule and G a goal clause. Let σ be a correct answer substitution. Then there is an SLD-refutation of G from P such that σ is the restriction of the sequence of unifiers $\theta = \theta_1 \cdots \theta_n$ to the variables in G.*

Proof: Lloyd (1987, Section 2.8). ▮

We will just sketch the inductive completeness proof. Consider the program P:

$$\begin{aligned} &p(a) \\ &p(f(x)) \quad \leftarrow \quad p(x). \end{aligned}$$

Obviously there is a refutation of the goal clause $\leftarrow p(a)$, and just as obviously $p(a)$ is a logical consequence of P. Given a goal clause $G_i = \leftarrow p(f(f(\cdots(a)\cdots))$, we can resolve it with the second clause to obtain $G_{i-1} = \leftarrow p(f(\cdots(a)\cdots)$, reducing the depth of the term. By induction, G_{i-1} can be refuted and $p(f(\cdots(a)\cdots)$ is a logical consequence of P. From G_{i-1} and the second clause, it follows that $p(f(f(\cdots(a)\cdots))$ is a logical consequence of P. This bottom-up inductive construction—starting from facts in the program and resolving with program clauses—defines an Herbrand interpretation. Given a ground goal clause whose atoms are in the Herbrand base of the interpretation, we can prove by induction that it has a refutation and that its negation is a logical consequence of P. To prove that a non-ground clause has a refutation, technical lemmas are needed which keep track of the unifiers. The final step is a proof that there exists a refutation regardless of the choice of computation rule.

Other behaviors of the SLD-derivation exist because of the non-determinism in the choice of literals and clauses.

Example 8.13 Returning to Example 8.6:

- Suppose that we had chosen in step 2 to resolve with clause 6 $p(e,b)$. The resolvent would be $\leftarrow q(b,z)$ and the refutation would still succeed but a *different* answer substitution $\{y \leftarrow e, z \leftarrow a\}$ would be obtained. There may be more than one correct answer substitution for a given goal clause.

- Suppose now that the computation rule is to always choose the *last* literal in a goal clause. Resolving always with clause 2 we obtain:

$$\begin{aligned} &\leftarrow q(y,b), q(b,z) \\ &\leftarrow q(y,b), p(b,z'), q(z',z) \\ &\leftarrow q(y,b), p(b,z'), p(z',z''), q(z'',z) \\ &\leftarrow q(y,b), p(b,z'), p(z',z''), p(z'',z'''), q(z''',z) \end{aligned}$$

  and so on. *Even though an answer substitution exists* for the goal clause, this specific attempt at constructing a refutation does not terminate.

- Consider the computation rule that always chooses the first literal in the goal clause. In the first step, $q(y,b)$ is chosen and resolved with clause 2 giving $\leftarrow p(y,z'), q(z',b), q(b,z)$. Choose the first literal and and resolve it with clause 6 $p(e,b)$ and then with clause 1 to obtain $\leftarrow q(b,b), q(b,z)$ and then $\leftarrow p(b,b), q(b,z)$. There remain *no* program clauses whose head unifies with $p(b,b)$. *Even though an answer substitution exists*, the refutation has failed.

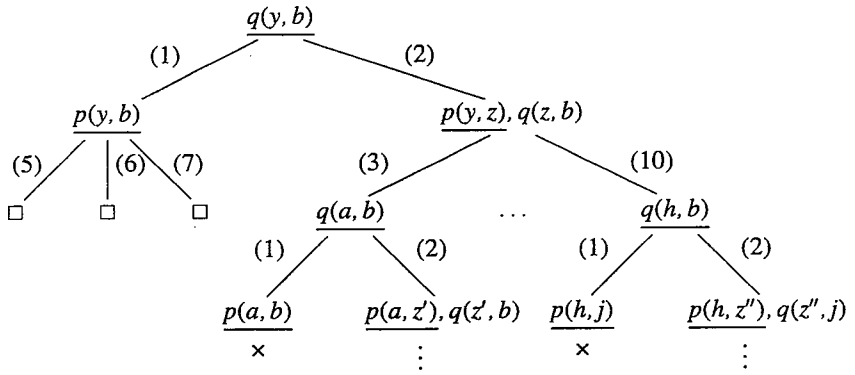


Figure 8.1 SLD-tree for selection of leftmost literal

Definition 8.14 Let P be a set of program clauses, R a computation rule and G a goal clause. All possible SLD-derivations can be displayed on an *SLD-tree*. The root is labeled with the goal clause G. Given a node n labeled with a goal clause G_n, create a child n_i for each new goal clause G_{n_i} that can be obtained by resolving *the literal chosen by* R with the head of a clause in P. □

Example 8.15 An SLD-tree for the clauses in Example 8.6 is shown in Figure 8.1. The computation rule is always to choose the *leftmost* literal of the goal clause. This is indicated by underlining the chosen literal. The number on an edge refers to the number of the program clause resolved with. □

Definition 8.16 In an SLD-tree, a branch leading to a refutation is called a *success branch*. A branch leading to a goal clause whose selected literal does not unify with any clause in the program is called a *failure branch*. A branch corresponding to a non-terminating derivation is called an *infinite branch*. □

Theorem 8.17 *Let P be a program and G be a goal clause. Then every SLD-tree for P and G has infinitely many success branches, or they all have the same finite number of success branches.*

Proof: Lloyd (1987, Section 2.10). ∎

Definition 8.18 A *search rule* is a procedure for searching an SLD-tree for a refutation. An *SLD-refutation procedure* is the SLD-resolution algorithm together with the specification of a computation rule and a search rule. □

Note carefully what Theorem 8.12 says about the choice of the computation and search rules. SLD-resolution is complete regardless of choice of the computation rule, but

the theorem only says that some refutation exists. The choice of the search rule will determine if the refutation is found or not. The trade-off is between completeness and efficiency. A *breadth-first* search of an SLD-tree, where the nodes at each depth are checked before searching deeper in the tree, is guaranteed to find a success branch if one exists. On the other hand a *depth-first* search can choose to head down a non-terminating branch. Thus the breadth-first search rule is complete in that if there is a correct answer substitution it will be found. However, the size of the tree that must be stored in a data structure grows exponentially with the depth of the search so pure breadth-first search is not a practical rule.

8.3 Prolog

Prolog was the first logic programming language and extensive implementation efforts have transformed Prolog into a practical tool for software development. This section will give an overview of the Prolog language based on the presentation of logic programming in the previous section.

The computation rule in Prolog is to choose the *leftmost* literal in the goal clause. The search rule is to choose clauses from *top to bottom* in the list of the clauses of a procedure. The notation of Prolog is different from the mathematical notation that we have been using: (a) variables begin with upper-case letters, (b) predicates and constants begin with lower-case letters, and (c) the symbol `:-` is used for ←.

Example 8.19 Let us rewrite program of Example 8.6 using the notation of Prolog. We have also replaced the arbitrary symbols by symbols that indicate the intended purpose of the program.

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).

parent(bob, allen).          parent(fred, dave).
parent(catherine, allen).    parent(harry, george).
parent(dave, bob).           parent(ida, george).
parent(ellen, bob).          parent(joe, harry).
```

The database contains facts that we are assuming to be true, such as `catherine` is a parent of `allen`. The procedure for `ancestor` gives a declarative meaning to this concept in terms of the `parent` relation, namely:

- X is an `ancestor` of Y if X is a `parent` of Y.
- X is an `ancestor` of Y if for *some* Z, X is a `parent` of Z and Z is an `ancestor` of Y.

Using the Prolog computation and search rules, the goal clause

```
:- ancestor(Y, bob), ancestor(bob, Z).
```

will succeed and return the correct answer substitution `Y=dave, Z=allen`, meaning that `dave` is an `ancestor` of `bob` who in turn is an `ancestor` of `allen`. □

The search in the proof tree is depth-first, which can lead to non-termination of the computation even if a terminating computation exists. A Prolog programmer must carefully order clauses within a procedure and literals within clauses to avoid non-termination. This choice of a search rule, however, is the reason that Prolog is extremely efficient. At run-time, essentially the only data structures that must be maintained are: (a) the current goal clause and a pointer to the current literal, (b) a pointer to the current clause within the procedure whose head unifies with the literal, (c) the current set of substitutions.

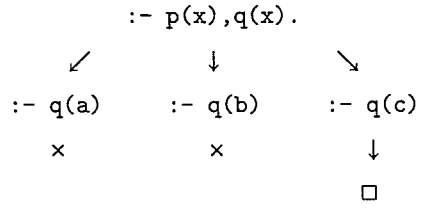
Since failure may occur at any step, a list of *backtrack points* must also be recorded. These backtrack points represent previous nodes in the SLD-tree where additional branches exist. Due to the Prolog search rule, the branches are necessarily to the right of the current branch.

Example 8.20 Consider the program consisting of only four facts and a goal clause:

```
p(a).    p(b).    p(c).    q(c).

:- p(x),q(x).
```

Here is the SLD-tree for this program:



The depth-first search attempts to resolve the first literal `p(x)` from the goal clause with `p(a)`. While this succeeds, the goal clause `q(a)` which results cannot be resolved. The search must return to its parent node in the SLD-tree and try the next clause in the procedure for p, namely, `p(b)`. Here too, the computation fails and it is only after an additional backtrack that a successful computation is completed. □

In this example, data structures must record that `p(x)` is a literal for which there are alternate clauses in the procedure that can resolve with it. Adding a recursive clause such as


```
p(x) :- r1(x,y),p(y),r2(y,z),p(z).
```

will create a potentially infinite number of backtrack points as each use of the clause will introduce two occurrences of literals with p. Since the search is depth-first, the backtrack points can be efficiently maintained on a stack.

An important concept in Prolog programming is forcing failure. This is implemented by the predicate `fail` for which no program clauses are defined. Consider the query:

```
:- ancestor(Y, bob), ancestor(bob, Z), fail.
```

Once the answer `Y=dave, Z=allen` is obtained, backtracking will be forced to produce a second answer `Y=ellen, Z=allen`. Since Prolog lacks iteration (`for`-loop), recursion and forced failure are important programming tools and a programmer used to iteration must invest some effort to learn to use them correctly.

Non-logical predicates

Non-logical predicates are predicates whose main or only purpose is the side-effects they generate. Obvious examples are the I/O predicates `get` and `put` that have no declarative meaning as logical formulas. As literals in a goal clause, they always succeed (except that `get` may fail at end of file). Their meaning is procedural: put a character on the display or get a character from the keyboard. Using this concept of non-logical predicates, Prolog systems have been augmented with interfaces to I/O peripherals and other systems services. Since use of these predicates can usually be confined to well-defined areas of the program and since their logical behavior is quite innocent, these extensions do not seriously interfere with the declarative logical structure of a Prolog program.

Prolog departs from theoretical logic programming in its treatment of numeric data types. This is a significant issue since almost all programs contain numeric calculations. As we showed in Section 6.4, it is possible to formalize arithmetic in predicate logic. However, there are two problems with this formulation. The first is that it would be unfamiliar, to say the least, to execute a query on the number of employees in a department and to receive as an answer the term $f(f(f(f(f(a)))))$ in the Herbrand domain (even if the notation were improved to $1+1+1+1+1$). The second problem is the inefficiency of resolution as a method for numeric computation. All computers contain efficient machine instructions for integer operations such as addition and subtraction, while even something as trivial as adding a constant 10 to a number would require at least ten unifications and resolutions. Computation with an axiomatic formalization of floating-point numbers would be completely impractical.

Prolog supplies a feature to allow standard arithmetic computation. The syntax is that of a predicate with an infix operator: `Result is Expression`. The following clause

retrieves the list price and discount percentage from a database and then computes the value of Price according to an arithmetic expression.

```
selling_price(Item, Price) :-
  list_price(Item, List),
  discount_percent(Item, Discount),
  Price is List - List * Discount / 100.
```

Arithmetic predicates differ from ordinary predicates. An arithmetic predicate is one-way, unlike unification. If `10 is X+Y` were a logical predicate, `X` and `Y` could be unified with say, 0 and 10, and upon backtracking with 1 and 9, and so on. However, this is illegal. In `Result is Expression`, `Expression` must evaluate to a 'ground' numeric value while `Result` must be an uninstantiated variable.

Note that arithmetic predicates are *not* assignment statements as defined in algorithmic languages. The following program is illegal because once `Price` is unified with the result of the computation `A+1`, it cannot be modified, anymore than a variable x in a logical formula can be modified once a substitution such as $\{x \leftarrow a\}$ has been applied.

```
selling_price(Item, Price) :-
  list_price(Item, List),
  discount_percent(Item, Discount),
  Price is List - List * Discount / 100,
  tax_percent(Item, Tax),
  Price is Price * (1 + Tax / 100).
```

Cuts

The most controversial modification of logic programming introduced into Prolog is the interference in the SLD-refutation procedure called the *cut*. Consider the following program for computing the factorial of a number N:

```
factorial(0,1).
factorial(N,F) :-
  N1 is N-1,
  factorial(N1, F1),
  F is N * F1.
```

This is just a translation into Prolog of the standard recursive formula for computing factorials:

$$f(0) = 1 \qquad f(n) = n \cdot f(n-1).$$

Now assume that `factorial` is called in another procedure, perhaps for checking a property of numbers that are factorials:

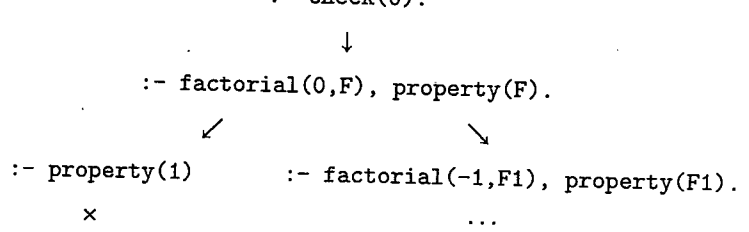

```
check(N) :- factorial(N, F), property(F).
```

If `check` is called with `N=0`, it will call `factorial(0, F)` which will compute `F=1` and call `property(1)`. Suppose that this call fails. Then the SLD-resolution procedure will backtrack, *undo* the substitution `F=1`, and try the second clause in the procedure for factorial. The recursive call `factorial(-1,F1)` will initiate a non-terminating computation. A call to `factorial` with argument 0 has only one possible answer; if we backtrack through it, the goal clause should fail.

This can be avoided by introducing a *cut*, denoted by an exclamation point, into the first clause:

```
factorial(0,1) :- !.
```

The cut prevents backtracking in this procedure. In terms of SLD-resolution, once a cut is 'executed', it cuts away a portion of the SLD-tree and prevents unwanted backtracking. In the following diagram, the rightmost branch is cut away, so that if `property(1)` fails, the backtrack point in its parent node is ignored.



In the case of the factorial procedure, there is a better solution, namely, to add a predicate to the body of the procedure to explicitly prevent the unwanted behavior:

```
factorial(0,1).
factorial(N,F) :-
  N > 0,
  N1 is N-1,
  factorial(N1, F1),
  F is N * F1.
```

Even in cases such as these, programmers often re-introduce the cut for efficiency.

Since the whole point of logic programming is to allow the user to write declarative programs and to leave the control element to the built-in logical inference machine, cuts should be avoided since they can only be understood from the procedural point of view. Nevertheless, Prolog programmers must be familiar with the cut and its uses both to avoid infinite branches and to improve efficiency.

8.4 Concurrent logic programming*

Computations which take too long on a single CPU can be speeded up by dividing the computation into tasks which are processed in parallel on multiple CPUs. Parallel computation is also used whenever a computation intrinsically involves multiple tasks, for example, in multiprogramming operating systems and real-time systems. In this context, the term concurrent computation is widely used because the multiple tasks can be executed by sharing a single CPU rather than being executed in true parallelism on multiple CPUs. The problem with parallel and concurrent computation is the difficulty of constructing and verifying algorithms that can benefit from parallelism, and of expressing these algorithms conveniently in a programming language.

Logic programs have a natural interpretation as concurrent computations. The reason is that logic programs do not express algorithms procedurally—step-by-step descriptions of how to proceed—but instead express them declaratively where the formulas have no inherent ordering. Concurrent logic programming languages specify concurrent procedural interpretations for Horn clause programs, just as Prolog specifies a sequential procedural interpretation for the clauses.

In this section we describe how a logic program can be given a parallel interpretation; then we show how modifications of unification can be used to synchronize processes and to communicate data between two processes. Finally, we present the concurrent logic programming language GHC.

And- and or-parallelism

Consider the logic program consisting of the clauses

$$\begin{aligned} p &\leftarrow r, s. \\ p &\leftarrow t. \\ q &\leftarrow u, q. \\ q &\leftarrow v. \end{aligned}$$

together with facts for r, s, t, u, v. The potential computations starting from the goal clause $\leftarrow p, q$ are displayed in the *and-or tree* in Figure 8.2. An or-node (dashed frame) is labeled with a one-literal goal clause. The dashed edges lead from an or-node to and-nodes (solid frames), which are labeled with the clauses that can unify with literal labeling the or-node. For each such clause, resolution with the literal in the or-node will cause the literals in the body to form a new goal clause. These literals become the labels of new or-nodes and solid edges lead from the and-node to the or-nodes. The success of the refutation at *any one* of the subtrees of an *or-node* implies the success of the goal clause labeling the or-node; for an *and-node* to succeed, *all* the goals labeling its children must be refuted.

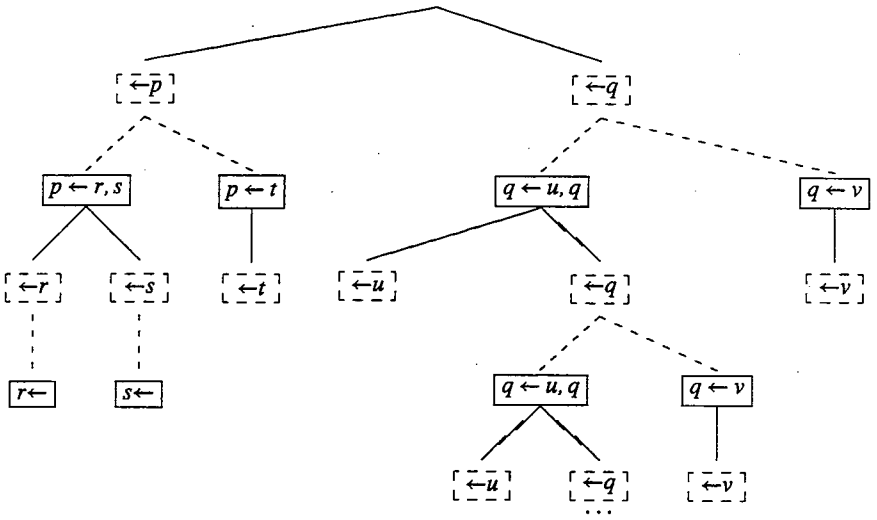


Figure 8.2 And-or tree

Example 8.21 In Figure 8.2, the root of the tree is an unlabeled and-node whose children are the two literals of the goal clause, both of which must be refuted. The or-node labeled $\leftarrow p$ has two children, one for each of the program clauses $p \leftarrow r, s$ and $p \leftarrow t$. It is sufficient that either one of these clauses lead to a refutation; in fact, both of them do so. The or-node labeled $\leftarrow q$ has two children. The right-hand one leads to a successful refutation while the left-hand contains an infinite branch. □

The computation of a Prolog program can be considered as a depth-first traversal of the tree:

- Refute all the (single-literal) goal clauses labeling or-nodes, ordered from left to right.
- Each refutation is attempted using the program clauses labeling the and-node nodes, ordered from left to right.
- If any refutation succeeds, refutations with the program clauses in the siblings of the and-node need no longer be attempted, and the goal clause labeling the or-node succeeds.
- The location of this and-node is remembered for possible backtracking.

Example 8.22 The Prolog search of the and-or tree in Figure 8.2 would find the refutation of $\leftarrow p$ from $p \leftarrow r, s$. However, a search for a refutation of $\leftarrow q$ would follow

the infinite branch starting with $q \leftarrow u, q$, and the program would not terminate. A different method of searching could discover the successful refutation from $q \leftarrow v$. □

The same and-or tree can be interpreted as a specification of a parallel computation. The goal clauses labeling or-nodes (the children of an and-node) form a set of *processes* that execute in parallel; this is called *and-parallelism* because all the processes must successfully terminate in order for the computation to succeed. Additional sets of processes are activated at each and-node (the children of an or-node) to attempt a refutation for each program clause; this is called *or-parallelism* because if any process associated with a program clause succeeds the other processes are cancelled.

Example 8.23 Let us trace a parallel computation of the and-or tree in Figure 8.2. Initially, there are two processes P_p and P_q in the attempt to refute p and q. P_p creates two new processes, $P_{r,s}$ and P_t. P_q also creates two new processes, $P_{u,q}$ and P_v. $P_{u,q}$ continues, creating two new processes, P_u and P'_q, where the prime is used to distinguish this process from the previous P_q. Then P'_q creates $P'_{u,q}$ and P'_v. The set of processes is now:

$$\{\overline{P_p},\ \overline{P_q},\ \underline{P_{r,s}},\ \underline{P_t},\ \overline{P_{u,q}},\ \underline{P_v},\ \underline{P_u},\ \overline{P'_q},\ \underline{P'_{u,q}},\ \underline{P'_v}\}.$$

The underlined processes can now be executed (they are in the ready queue in the terminology of operating systems), whereas the overlined processes are waiting for the termination of the processes they have created.

$P_{r,s}$ creates two new processes, P_r and P_s. Suppose that these processes are now scheduled for execution one after another; they resolve successfully and terminate, as does $P_{r,s}$. Since $P_{r,s}$ is the child of an or-node, its sibling P_t can be cancelled and P_p terminates. The process set is now:

$$\{\overline{P_q},\ \overline{P_{u,q}},\ \underline{P_v},\ \underline{P_u},\ \overline{P'_q},\ \underline{P'_{u,q}},\ \underline{P'_v}\}.$$

If P_v is selected for execution, it will terminate together with the entire set of processes. Alternatively, if $P'_{u,q}$ is selected, an infinite set of processes may be created. If the scheduler is *fair*, that is, if it eventually schedules every ready process to run, one of $P_v, P'_v, P''_v, \ldots$ will eventually be selected and the computation will terminate. □

Logic variables for communications

Synchronization and communication between the processes of a logic program are based on the properties of logic variables appearing in the processes. Using Prolog syntax, the goal clause `?-p(X),q(X)` can be considered as a pair of concurrent processes that share a global variable `X`. Obviously, if a ground term is assigned to `X` in `p` (say by unification with the fact `p(a)`), it will simultaneously be available to `q`. However, a logic variable can be assigned to only once, while we normally require that one

process pass a stream (sequence of values) to another. To pass a stream from process `p(X)` to process `q(X)`, `p` assigns to `X` a compound term that contains another logic variable. This additional logic variable is shared by both processes and maintains the communications channel open.

Example 8.24 Suppose that `?-p(X),q(X)` is executed for the program:

```
p([Sym|Tail]) :- gensym(a,Sym), p(Tail).
q([Head|Tail]) :- process(Head), q(Tail).
process(X) :- ...
```

where `gensym(a,Sym)` unifies `Sym` with the stream `a1`, `a2`, ... in successive calls. Resolving `p(X)` causes `X` to be unified with `[a1|Tail]` giving the goal clause:

```
?- p(Tail), q([a1|Tail]).
```

and after a resolution with the clauses for `q([a|Tail])` and `process(a)` we get

```
?- p(Tail), q(Tail).
```

which puts us right back where we started from. Thus `p` generates a stream that is processed within `q`. □

This example was rather artificial. Let us now examine a useful program.

Example 8.25 Here is a procedure that merges two *ordered* lists:

```
merge([Head1|Tail1], [Head2|Tail2], [Head1|List3]) :-
    Head1 <= Head2,
    merge(Tail1, [Head2|Tail2], List3).
merge([Head1|Tail1], [Head2|Tail2], [Head2|List3]) :-
    Head1 > Head2,
    merge([Head1|Tail1], Tail2, List3).
merge([], List, List).
merge(List, [], List).
```

and a procedure that writes a list:

```
write_list([Head|Tail] :-
    write(Head),
    write_list(Tail).
write_list([]).
```

With a sequential Prolog implementation, executing the goal clause

```
?- merge([1,3,5,7],[2,4,6,8],List), write_list(List).
```

will cause `[1,2,3,4,5,6,7,8]` to be assigned to `List` which is then displayed by `write_list`. With an and-parallel implementation, we can consider `merge` to be a producer process which creates values for the consumer process `write_list` to receive and print. As `merge` produces list elements, they can be consumed (printed) concurrently with the production of the rest of the list without waiting for the entire result list to be produced.

Initially, the first program clause can be activated and unified with the first literal of the goal clause, creating the substitution `List=[1|List3]`. This substitution is applied to `write_list` and the process `write_list([1|List3])` can now be activated, unifying `[1|List3]` with `[Head|Tail]` in the first clause. `write(Head)` can now be activated to display the element `1` without waiting for the entire list to be created.

An example of a state in the computation is:

```
merge([5,7], [6,8], [1,2,3,4 | ListN]),
write_list([4 | ListN]), write(2), write(3)
```

Four elements of the result list have been produced and passed to `write_list`, which has already created new processes to print three out of the four elements. The first process has been executed, printing `1` and terminating. The other two processes are waiting to be activated. □

Synchronization

The example shows how communication can be achieved within the framework of unification without introducing additional constructs into the programming language. There remains one problem, however. We need to devise a mechanism to synchronize the concurrent processes so that, for example, `write_list` does not attempt to write a list that has not yet been instantiated with actual values. The solution is to change the semantics of unification, so that a substitution is not allowed to substitute for a variable in a process associated with a literal of a goal clause. Instead, the process will block until the variable is instantiated with a non-variable term.

Example 8.26 In the goal clause

```
?- merge([1,3,5,7],[2,4,6,8],List), write_list(List).
```

the process `write_list` is initially blocked because unification with a program clause would substitute `[Head|Tail]` or `[]` for the variable `List` in the literal of the goal clause. However, once the substitution `List=[1|List3]` is created by the execution of the `merge` process, the process `write_list` is unblocked and `1` can be assigned to `Head`. This also unblocks the literal `write(Head)` and the first value can be printed.

In the recursive call `write_list(List3)`, unification is again trying to assign to a variable and the process is blocked. □

Committed or-parallelism

Consider a refutation of the goal clause `?-p(X)` with the program:

```
p(X) :- q(X,a), r(X).
p(X) :- q(X,b), r(X).
q(Y,Y).
r(b).
```

Or-parallelism can be used with the two clauses in the procedure for `p`. If the refutation beginning with the first clause proceeds faster than the refutation beginning with the second clause, the substitution `X=a` can be made by resolving `q(X,a)` with `q(Y,Y)`, *before* the substitution `X=b` is created in the second or-parallel refutation attempt. Since a variable can only be instantiated once, and since the substitution `X=a` will eventually cause failure, all threads of execution must maintain a history of their substitutions so that they can be rolled back. In a real program with many parallel threads, the inherent complexity and inefficiency of this requirement can outweigh any advantage of parallel computation.

A solution is to limit or-parallelism to pattern matching of the initial clause head and to resolution of literals in the clause body which do not bind variables in literals of the goal clause. Once we have *committed* to an alternative, such bindings may be done.

Example 8.27 Consider the goal clause `p(1,x)` and program:

```
p(0,X) :- q(X,a).
p(N,X) :- N<0, q(X,b).
p(N,X) :- N<0, q(X,c).
q(X,X).
```

The three clauses for `p` may be tried in parallel. The first clause is rejected because its head `p(0,X)` cannot be unified with the literal `p(1,X)`. The second and third clauses contain literals `N>0` and `N<0` which just test the value of its argument without binding any variable. Once we commit to the second clause, the parallel attempts to resolve with the first and third clauses can be terminated, and the call `q(X,b)` can bind `b` to `X` with no danger that the substitution may have to be undone. □

Most practical logic programming are *flat*, meaning that literals resolved before commitment are built-in predicates. The and-or tree collapses to a single level, because no new processes need to be created for or-parallelism.

The GHC language

As an example of a concurrent logic language we will briefly describe the *GHC (Guarded Horn Clauses)* language. A GHC program consists of a set of *guarded clauses*. A guarded clause is like an ordinary clause except that the literals in the body may be preceded by a sequence of literals called *guards*. They are separated from the body literals by a vertical bar rather than by a comma:

```
A :- G1,...,Gk | B1,...,Bn.
```

The rules for the head of the clause and the guard are the same: if an attempt is made to substitute for a variable in a literal of a goal clause, the computation is blocked until the variable is instantiated with a non-variable term. Substitutions to variables are allowed only in the body of the clause.

Example 8.28 Here is a *Prolog* program that unifies the third argument with the maximum of values in the first two arguments:

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.
```

If called with the goal clause `?-max(4,5,A)` it will return the substitution `A=5`.

However, this program is not very useful as a GHC program. If called with the goal clause `?-max(4,5,A)` it will simply block until `A` is unified with a non-variable term, because the variable cannot be assigned to from within the head of a program clause. A GHC version of the program is as follows:

```
max(X,Y,Z) :- X >= Y | Z = X.
max(X,Y,Z) :- X < Y  | Z = Y.
```

The substitution for `Z` is explicitly created in the bodies of the program clauses. The tests `X>=Y` and `X<Y` do not create such bindings so they can be placed in the guards and executed in or-parallelism. The goal clause `?-max(4,5,A)` will cause commitment to the second program clause and the assignment of `5` to `Z` and thus to `A`.

If the goal clause were `?-max(B,5,A)`, the attempt to substitute `X` for `B` causes the computation to be blocked. Such a literal would only appear in a context such as `?-generate(B),max(B,5,A)` where another process will eventually generate a binding for `B`. □

Formally, the matching of the head and the evaluation of a guard define a status for each clause in the procedure:

- The clause is a *candidate clause* if the matching succeeds and the guard resolves successfully.

- The clause is a *non-candidate clause* if the matching fails or the resolution of the guard fails (or both).
- The clause is a *suspended clause* if either the matching or the resolution of the guard would require a *variable* in a literal of the goal clause to be instantiated.

Once the status of the clause definitions in the procedure is determined, the computation proceeds as follows:

- If there is at least one candidate clause, the resolution succeeds. The computation *commits* to one of the candidate clauses, its body replaces the literal and the substitutions from the body to goal variables are done.
- If there are no candidate clauses but there are suspended clauses, the resolution is suspended.
- If all clauses are non-candidate clauses, the resolution fails.

Example 8.29 The merge program is written in GHC as follows:

```
merge([Head1|Tail1], [Head2|Tail2], Result) :-
    Head1 <= Head2 |
    Result = [Head1|List3],
    merge(Tail1, [Head2|Tail2], List3).
merge([Head1|Tail1], [Head2|Tail2], Result) :-
    Head1 > Head2 |
    Result = [Head2|List3],
    merge([Head1|Tail1], Tail2, List3).
merge([], List, Result) :- true | Result = List.
merge(List, [], Result) :- true | Result = List.
```

The tests are put in the guard and together with the matching of the heads can be performed in parallel. Once one is successful, the clause commits and the assignment to `Result` can take place.

Executing the goal clause

```
?- merge([1,3,5,7],[2,4,6,8], List),write_list(List).
```

will cause the computation to commit to the first program clause; then the substitution `List=[1|List3]` from the body can be made and the process `write_list` unblocked. □

Stream and-parallelism is a powerful technique because it makes interactive programs possible within the framework of logic programming. Consider the goal clause:

```
?- get(L1), process(L1, L2), put(L2).
```

where `get` is a process that waits for input from a keyboard, `process` is a program that takes a list of input values and produces a list of output values and `put` displays a list of values to a screen. Whenever the user inputs a value `v`, the variable `L1` is instantiated with a list containing a logic variable `[v|LL]`. `process` is now unblocked, computes the appropriate output value `w` from `v` and instantiates `L2` with `[w|LLL]`. `put` is now unblocked and displays `w`.

8.5 Constraint logic programming*

Constraint logic programming (CLP) combines the flexibility and ease of declarative programming in logic programming with the power of search techniques that were developed during research on artificial intelligence. CLP is best understood by considering two problems with logic programming in Prolog:

- Logic programming is limited to uninterpreted domains. As we saw in Section 8.3, even arithmetic cannot be done within the framework of Prolog and the nonlogical predicate `is` must be used for even the simplest computation.
- SLD-resolution (Section 8.2) searches the entire tree in a fixed order and does not use knowledge gained during the computation in order to control the search.

Let us demonstrate these difficulties and how they are avoided in CLP. The discussion is based on the classical eight-queens problem as presented in Van Hentenryck (1989). The problem is to place eight queens on a chess board such that no queen attacks another. For readers not familiar with the game: chess is played on an 8×8 matrix of cells; one queen *attacks* another if it is placed on the same row, column or diagonal. Figure 8.4 shows a solution.

The problem can be simplified by noting that since at most one queen can appear in any row or column of the matrix, it is sufficient to consider permutations of the set $\{1, 2, 3, 4, 5, 6, 7, 8\}$. The i'th element of the permutation indicates the row in the i'th column in which a queen will be placed. The solution shown in Figure 8.4 corresponds to the permutation $\{1, 5, 8, 6, 3, 7, 2, 4\}$.

Search problems of this sort can be solved in Prolog using *generate-and-test*. Simply *generate* every possible permutation of the set and *test* if it satisfies the requirements of the problem. If the test fails, backtracking will cause a new permutation to be generated.

```
queens([X1,X2,X3,X4,X5,X6,X7,X8]) :-
  permutation([X1,X2,X3,X4,X5,X6,X7,X8], [1,2,3,4,5,6,7,8]),
  safe([X1,X2,X3,X4,X5,X6,X7,X8]).
```

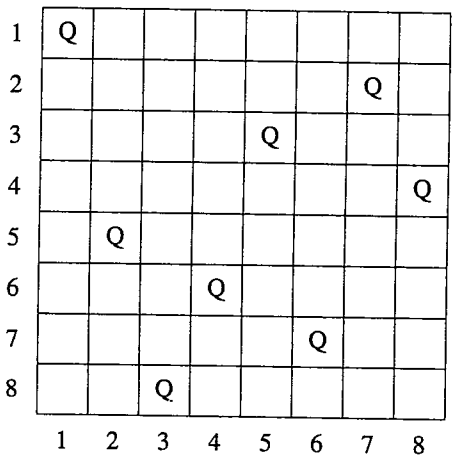



Figure 8.3 Solution of the eight-queens problem

For reference, here is a procedure to generate permutations. It selects an element of the list of numbers which will be the first element of the permutation; then it calls itself recursively to generate the rest of the permutation on the remaining numbers.

```
permutation([], []).
permutation([Head|Tail], Numbers) :-
  select(Numbers, Head, NewNumbers),
  permutation(Tail, NewNumbers).
```

`safe` checks if the permutation is a solution. It calls `noattack` successively on each element of the permutation to ensure that this placement of a queen does not attack the rest of the queens.

```
safe([]).
safe([Queen|Others]) :-
  noattack(Queen, Others),
  safe(Others).

noattack(Queen, Others) :-
  noattack(Queen, Others, 1).

noattack(_, [], _).
noattack(Queen, [Next|Others], N) :-
  Queen =\= Next - N,
  Queen =\= Next + N,
  N1 is N + 1,
  noattack(Queen, Others, N1).
```

What is wrong with this solution? First, it is extremely inefficient. The first permutation tested is $\{1,2,3,4,5,6,7,8\}$ which is clearly not safe because all the queens are on the main diagonal. Since the test fails, backtracking commences and the second permutation tested is $\{1,2,3,4,5,6,8,7\}$, which is not much better than the first one. A human trying to solve the problem would quickly note that any permutation beginning with $\{1,2,\ldots\}$ is not a solution, but the generate-and-test algorithm blindly generates all the permutations in order. In fact, this program generates 2843 permutations before finding the first solution shown in Figure 8.3!

Generate-and-test can be improved by integrating the testing with the generation. Rather than create an entire permutation and only then testing it, the permutation (placement of queens) can be build incrementally and tested after each queen has been placed. For example, if the placement $\{1,5,8,6\}$ has been determined to be safe, any attempt to place the next queen on $\{2,7\}$ is unsafe so there is no need to generate and test every permutation whose fifth element is one of those numbers.

In the following program, `queens(List,Placed,Numbers)` implements this strategy by incrementally building in `Placed` a (reversed) list of queens already placed. `Numbers` is a list of the unused elements of the permutation. `List` is used to return the result. The predicate works by selecting an unused element `Queen` from the list `Numbers`. If `noattack` succeeds, meaning that the row `Queen` is consistent with those `Placed` so far, a recursive call is made to attempt to place the next queen.

```
queens(X) :-
  queens(X, [], [1,2,3,4,5,6,7,8]).

queens([], _, []).
queens([Queen|Others], Placed, Numbers) :-
  select(Numbers, Queen, NewNumbers),
  noattack(Queen, Placed),
  queens(Others, [Queen|Placed], NewNumbers).
```

This program is an order of magnitude more efficient than the previous one: it makes only 336 selections before finding the solution. Still it is somewhat upsetting that we had to be clever and change the simplest declarative program that solves the problem.

The second problem with the Prolog solutions is that the programs waste a lot of information. For example, once a queen is placed in row 1, the entire main diagonal is taken, but this knowledge is thrown away and recreated again and again.

CLP is based on *constrain-and-generate*, which reverses the strategy of generate and test. The programmer writes constraints on the domains of the predicates, which are maintained as data structures within the compiler or interpreter. In the case of the eight-queens problem, there are two constraints: First, we know that the values of the generated configurations must be within the range of 1 to 8. Second, once a queen is placed, the inequations:


```
Queen =\= Next, Queen =\= Next - N, Queen =\= Next + N
```

constrain *future* placements.

When the eight-queens program is initially run, a domain declaration

```
domain queens(1..8).
```

constrains the values of the variables in the predicate to be integers in the stated range. Given a goal clause `queens([X1,X2,X3,X4,X5,X6,X7,X8])`, a data structure is constructed to maintain the possible values of the variables:

$$\begin{aligned} X1 &\in \{1,2,3,4,5,6,7,8\} \\ X2 &\in \{1,2,3,4,5,6,7,8\} \\ X3 &\in \{1,2,3,4,5,6,7,8\} \\ X4 &\in \{1,2,3,4,5,6,7,8\} \\ X5 &\in \{1,2,3,4,5,6,7,8\} \\ X6 &\in \{1,2,3,4,5,6,7,8\} \\ X7 &\in \{1,2,3,4,5,6,7,8\} \\ X8 &\in \{1,2,3,4,5,6,7,8\} \end{aligned}$$

Now that the values have been constrained, the next step is to generate a configuration, say, by placing a queen in row 1. Once the configuration is generated, the program returns to check the constraints. Placing even one queen significantly reduces the possible values of the remaining variables, as no other queen can be placed in the same row, column or diagonal.

$$\begin{aligned} X1 &\in \{1\} \\ X2 &\in \{3,4,5,6,7,8\} \\ X3 &\in \{2,4,5,6,7,8\} \\ X4 &\in \{2,3,5,6,7,8\} \\ X5 &\in \{2,3,4,6,7,8\} \\ X6 &\in \{2,3,4,5,7,8\} \\ X7 &\in \{2,3,4,5,6,8\} \\ X8 &\in \{2,3,4,5,6,7\} \end{aligned}$$

After constraining and generating more configurations, placing queens on rows 3, 5 and 7, the domain is reduced to:

$$\begin{aligned} X1 &\in \{1\} \\ X2 &\in \{3\} \\ X3 &\in \{5\} \\ X4 &\in \{7\} \\ X5 &\in \{2,4\} \\ X6 &\in \{4\} \\ X7 &\in \{2,6\} \\ X8 &\in \{2,4,6\} \end{aligned}$$

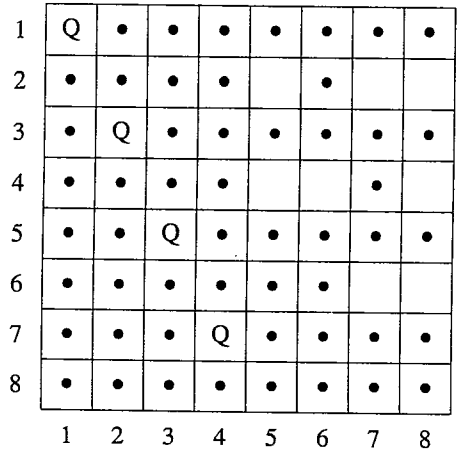


Figure 8.4 Squares that do not satisfy the constraints

as shown in Figure 8.4, where the bullets indicate locations that are in 'check'. By examining the table or the figure, we can clearly see that it is impossible to choose a configuration that contains a queen in row 8, so no solution is possible and we must backtrack. If constrain-and-generate is pursued, the choices 2, 4 and 6 for $X5$, $X6$ and $X7$ leave no choices for $X8$. The choice 4 for $X5$ immediately leaves no choice for $X6$.

By maintaining constraints on domain values, rather than throwing them away as the Prolog program did, the search for a solution is an order of magnitude shorter. Furthermore, the CLP program is even simpler than the Prolog program.

```
domain queens(1..8).
queens([]).
queens([Queen|Others]) :-
  indomain(Queen),
  noattack(Queen, Others),
  queens(Others).
```

The predicate `indomain` generates values for the variable that are within the declared domain. `noattack` is syntactically the same as before with the addition of the inequation `Queen =\= Next`. These inequations do not simply check that domain values satisfy them; instead, they are used to constrain the domain values of all the variables as shown in the example.

The maintenance and propagation of constraints as described in the eight-queens problems is the basis of the CHIP programming language which is used to solve problems in operations research such as scheduling.

An alternate view of CLP is to regard constraints as extensions of unification. Recall that we presented unification as solving a set of equations over the uninterpreted Herbrand domain. If the terms to be 'unified' are interpreted over an actual domain, algorithms for this domain can be included within the language interpreter. The most extensively developed example is the language CLP($\mathcal{R}$) for writing logic programs over the domain of real numbers (Jaffar & Maher 1994). In CLP($\mathcal{R}$), a constraint such as Ohm's Law $V = I * R$ may be written. If the constraint is executed with any two of the three variables ground, an algorithm is used to compute the third. If only one of the variables is ground, it is used to update the constraint for subsequent executions. For example, if $R = 1000.0$, then the constraint $V = I * 1000.0$ is satisfiable for $V = 500.0, I = 0.5$, but not for $V = 500.0, I = 1.0$.

CLP($\mathcal{R}$), which combines the declarative logic programming style with algorithms for computation on real numbers, has been used in applications areas such as circuit simulation and stock option analysis that were not traditional areas for logic programming.

8.6 Exercises

1. Let P be the program $p(a) \leftarrow$ and G be the goal clause $\leftarrow p(x)$. Is the identity substitution a correct answer substitution? Explain.

2. Draw in detail and to greater depth the SLD-tree in Figure 8.1.

3. Draw an SLD-tree similar to that of Figure 8.1 except that the computation rule is to select the rightmost literal in a clause.

4. Given the logic program

$$\begin{array}{lcl} p(a,b) & & \\ p(c,b) & & \\ p(x,y) & \leftarrow & p(x,z), p(z,y) \\ p(x,y) & \leftarrow & p(y,x), \end{array}$$

   and the goal clause $\leftarrow p(a,c)$, show that if any clause is omitted from the program then there is no refutation. From this prove that if a depth-first search rule is used with any fixed order of the clauses, there is no refutation no matter what computation rule is used.

5. Given the logic program

$$\begin{array}{l} p \leftarrow q(x,x) \\ q(x, f(x)), \end{array}$$

   and the goal clause $\leftarrow p$, prove that there is a refutation if and only if the occur check is omitted. Conclude that omitting the occur check invalidates the soundness of SLD-resolution.

6. Given the logic program

$$\begin{array}{lcl} p & \leftarrow & q(x,x) \\ q(x,f(x)) & \leftarrow & q(x,x), \end{array}$$

and the goal clause $\leftarrow p$, what happens if a refutation is attempted without using the occur check?

7. Write a logic program for the *Slowsort* algorithm by directly implementing the following specification of sorting: $sort(L1, L2)$ is true if $L2$ is a permutation of $L1$ and $L2$ is ordered.

8. * Suppose that the tests in the GHC program for `merge` are replaced by `true` and that the input lists are not required to be ordered. Describe the possible computations.

9. * Let `Tree` be a binary tree whose nodes are labeled with integers. Write a GHC program to sum the labels of the nodes that works by concurrent flattening the tree into a list of nodes and summing the elements of the list:

```
sum_tree(Tree, Sum) :-
  flatten(Tree, List), sum_list(List, Sum).
```

10. [P] Write a Prolog program to solve the puzzle `SEND+MORE=MONEY` that works on the same principle as the program `queens`: generate a permutation and test if it is a solution. A solution is a substitution of distinct digits for distinct letters such that the addition is correct. Instrument the program to count the number of permutations generated. (Warning: Be patient....)

11. * Simulate a constraint logic program to solve the `SEND+MORE=MONEY` puzzle. The domain of the list of variables `[S,E,N,D,M,O,R,Y,_]` is constrained to `0..9` and the four carry variables are constrained to `0..1`. How many configurations are generated?

9 Programs: Semantics and Verification

9.1 Introduction

A program is not very different from a logical formula. It is a sequence of symbols constructed according to formal syntactical rules and it has a meaning which is assigned by an interpretation of the elements of the language. In programming, the symbols are called *statements* or *commands* and the intended interpretation is an execution on a machine, rather than evaluation of a truth value. The syntax of programming languages is specified using formal systems such as BNF, but the semantics is usually informally specified.

Example 9.1 The formal BNF syntax of an `if`-statement was given in Example 1.2. Its semantics is described informally as follows:

> The Boolean expression is evaluated. If true, the statement following `then` is executed, otherwise the statement following `else` is executed.

□

If the semantics is informally defined there is no formal way of determining the validity or correctness of a program, and we are reduced to testing and debugging which are unreliable and expensive.

In this chapter, we present a formal system for specifying the semantics of a programming language and we precisely define the concept of a correct program. Then we give an axiom system that can be used to prove that a program is correct with respect to its formal specification. The next chapter presents the Z notation, which is a convenient and very expressive notation for specifying properties of a program.

There is an essential difference between program logics and the predicate calculus. In the predicate calculus, we concentrated on *arbitrary* interpretations and thus on valid formulas. In particular, the central result is the completeness theorem, which shows that the set of valid formulas equals the set of provable formulas. Most programming, however, deals with numbers or other predefined domains, such as strings or trees.

In order to concentrate on reasoning about programs, we take the theorems of these domains as axioms. For example, $(x \geq y) \rightarrow (x+1 \geq y+1)$ is a theorem of number theory and will be used as an axiom without further proof. Thus the completeness of an axiom system for verification of programs is *relative* to the domain theory. Relative completeness is not a problem in practice, for the same reason that the incompleteness of arithmetic is not a problem in practice. If you write a program to do a numerical computation, you already have the proofs to justify the computation itself; the problem is to show that the program is a correct implementation of the computation. An automated theorem prover or software for symbolic mathematics can help with the arithmetic calculations, if needed.

9.2 Semantics of programming languages

A statement in a programming language can be considered to be a function that transforms the state of a computation. If the variables `(x,y)` have the values $(8,7)$ in a state s, then the result of executing the statement `x := 2*y+1` is the state s' in which $(x,y) = (15,7)$.

Definition 9.2 If a program uses n variables (`x1`, ..., `xn`), a state s consists of an n-tuple of values $(x_1, \ldots, x_n)$, where x_i is the value of the variable `xi`. Notation: variables of the program will be indicated in typewriter font `x`, while the corresponding value of the variable will be indicated in italic font x. □

We want to be able to reason within the predicate calculus about the states of a computation, so predicates are used instead of sets of states.

Definition 9.3 Let U be the set of all n-tuples of values over some domain(s), and let $U' \subseteq U$. $P_{U'}(x_1, \ldots, x_n)$, the *characteristic predicate* of U', is defined so that $U' = \{(x_1, \ldots, x_n) \in U \mid P_{U'}(x_1, \ldots, x_n)\}$. □

Example 9.4 Let U be the set of 2-tuples over $\mathcal{Z}$ and let $U' \subseteq U$ be the 2-tuples described in the following table:

$$
\begin{array}{ccccccc}
 & & & \cdots & & & \\
\cdots & (-2,-3) & (-2,-2) & (-2,-1) & (-2,0) & (-2,1) & (-2,2) & (-2,3) \\
\cdots & (-1,-3) & (-1,-2) & (-1,-1) & (-1,0) & (-1,1) & (-1,2) & (-1,3) \\
\cdots & (0,-3) & (0,-2) & (0,-1) & (0,0) & (0,1) & (0,2) & (0,3) \\
\cdots & (1,-3) & (1,-2) & (1,-1) & (1,0) & (1,1) & (1,2) & (1,3) \\
\cdots & (2,-3) & (2,-2) & (2,-1) & (2,0) & (2,1) & (2,2) & (2,3) \\
 & & & \cdots & & & &
\end{array}
$$

The characteristic predicate of U' is $(x_1 = x_1) \wedge (x_2 \leq 3)$ or simply $x_2 \leq 3$. □

A simple program with integer variables `x1` and `x2` can be executed starting in any one of about $2^{32} \cdot 2^{32}$ states on a 32-bit machine. Without characteristic predicates like $x_2 \leq 3$ it would be extremely difficult to reason about the states of a program.

Example 9.5 Let `S` be the program statement `x := 2*y+1`. It transforms states in $\{(x,y) \mid \mathit{true}\}$ into states in $\{(x,y) \mid x = 2y+1\}$. Suppose we specify instead that the set of initial states is $\{(x,y) \mid y \leq 3\}$. Since $(y \leq 3) \rightarrow (2y+1 \leq 7)$, the final state after executing `S` is in $\{(x,y) \mid (x \leq 7) \wedge (y \leq 3)\}$. We say that `S` transforms $y \leq 3$ into $(x \leq 7) \wedge (y \leq 3)$. □

The semantics of a programming language is given by specifying how each statement in the language transforms an initial state into a final state. The same concept is also used to define the semantics of a *program*, which is composed of statements in the language.

Definition 9.6 An *assertion* is a triple $\{p\}$ `S` $\{q\}$, where `S` is a program, and p and q are formulas in the predicate calculus called the *precondition* and *postcondition*, respectively.
An assertion is true, denoted $\models \{p\}$ `S` $\{q\}$, iff: if `S` is started in a state satisfying p and if this computation of `S` terminates, then the computation terminates in a state satisfying q.
If $\models \{p\}$ `S` $\{q\}$, then `S` is said to be *partially correct with respect to p and q*. □

Assertions are also called *Hoare triples*.

Example 9.7 $\models \{y \leq 3\}$ `x := 2*y+1` $\{(x \leq 7) \wedge (y \leq 3)\}$. □

Example 9.8 $\models \{\mathit{false}\}$ `S` $\{q\}$ for any `S` and q; since there are no states satisfying *false*, the condition is vacuous. Similarly, $\models \{p\}$ `S` $\{\mathit{true}\}$ for any p and `S`. □

Weakest preconditions

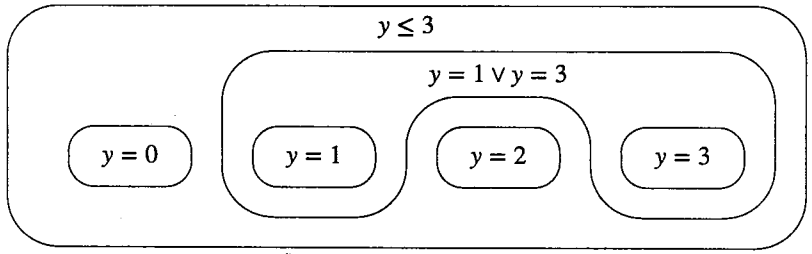
The formalization of the semantics of a language in terms of predicates is done by giving for each statement and postcondition, the minimal precondition that will guarantee the truth of the postcondition upon termination of the statement.

Example 9.9 $\models \{y \leq 3\}$ `x:=2*y+1` $\{(x \leq 7) \wedge (y \leq 3)\}$, but $y \leq 3$ is not the only precondition that will make the postcondition true. Another one is $y = 1 \vee y = 3$. □

This precondition $y = 1 \vee y = 3$ is 'less interesting' than $y \leq 3$ because it does not characterize all states from which the computation can reach a state satisfying the postcondition. We wish to choose the least restrictive precondition so that as many states as possible can be initial states in the computation.

Definition 9.10 A formula A is *weaker than* formula B if $B \rightarrow A$. Given a set of formulas $\{A_1, A_2, \ldots\}$, A_i is the *weakest* formula in the set if $A_j \rightarrow A_i$ for all j. □

Example 9.11 $y \leq 3$ is weaker than $y = 1 \vee y = 3$ because $(y = 1 \vee y = 3) \rightarrow (y \leq 3)$. Similarly, $y = 1 \vee y = 3$ is weaker than $y = 1$, and (by transitivity) $y \leq 3$ is also weaker than $y = 1$. This is demonstrated by the following Venn diagram:



□

You can always strengthen a premise and weaken a consequence, for example, if $p \rightarrow q$, then $(p \wedge r) \rightarrow q$ and $p \rightarrow (q \vee r)$. The terminology is somewhat difficult to get used to because we are used to thinking about states rather than predicates. Just remember that the *weaker* the predicate, the *more* states satisfy it.

Definition 9.12 For program S and formula q, $wp(\mathtt{S}, q)$, the *weakest precondition*[1] *of* S *and* q, is the weakest formula p such that $\models \{p\}\ \mathtt{S}\ \{q\}$. □

The proof of the following lemma is immediate from the definition of weakest.

Lemma 9.13 $\models \{p\}\ \mathtt{S}\ \{q\}$ *if and only if* $\models p \rightarrow wp(\mathtt{S}, q)$.

Example 9.14 $wp(\mathtt{x:=2*y+1}, (x \leq 7) \wedge (y \leq 3)) = y \leq 3$. Recall that $y \leq 3$ is the characteristic predicate of the set of states $U = \{(x, y) \mid y \leq 3\}$. Check that for any $(x, y) \in U$, $x' \leq 7$ holds of the values (x', y') after executing the statement, and conversely. □

The weakest precondition p depends upon both the program S and the postcondition q. If the postcondition in the example is changed to $x \leq 9$ then the weakest precondition becomes $y \leq 4$. Similarly, if S is changed to `x := y+2` without changing the postcondition, the weakest precondition becomes $y \leq 5$.

wp is a *predicate transformer* because for any given program fragment, it defines a transformation of a postcondition predicate into a precondition predicate. Initially, we

[1] The literature on semantics calls this definition the weakest *liberal* precondition *wlp*, and reserves *wp* for preconditions that ensure total correctness. Since total correctness is only surveyed in this text, we omit the distinction for conciseness.

set out to formalize a program as a transform from a given *set* of initial states to a *set* of final states. Instead we formalized a program as a transform from a postcondition *predicate* to a precondition *predicate*. This backwards formulation in terms of preconditions from postconditions is preferred since it is goal-oriented, that is, we start from the specification of the result of the entire program and work backwards to synthesize or verify the individual program statements that achieve the result.

Semantics of a fragment of Pascal

The following definitions formalize the semantics of the fragment of Pascal consisting of compositions of assignment, `if`- and `while`-statements.

Definition 9.15 $wp(\mathtt{x:=t}, p(x)) = p(x)\{x \leftarrow t\}$. □

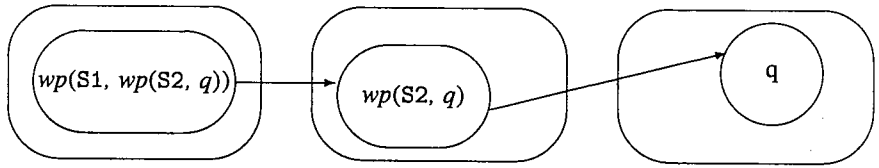
At first glance, the semantics of an assignment statement may seem to be backwards, but it must be understood in terms of a predicate transformer. If substituting t for x makes $p(x)$ true *now*, then go ahead and perform the assignment, after which $p(x)$ will be true because x has the value t.

Example 9.16 $wp(\mathtt{y:=y-1}, y \geq 0) = (y - 1 \geq 0)$, which simplifies to $y \geq 1$. □

Composition of statements takes the result of transforming the final postcondition by the second statement as the postcondition for the first statement.

Definition 9.17 $wp(\mathtt{S1;S2}, q) = wp(\mathtt{S1}, wp(\mathtt{S2}, q))$. □

The following figure illustrates the definition.



The precondition $wp(\mathtt{S2}, q)$ characterizes the set of states such that executing S2 leads to a state in which q is true. If executing S1 leads to one of these state, then S1;S2 will lead to a state whose postcondition is q.

Example 9.18

$$\begin{aligned} wp(\texttt{x:=x+1; y:=y+2},\ x<y) &= wp(\texttt{x:=x+1},\ wp(\texttt{y:=y+2},\ x<y)) \\ &= wp(\texttt{x:=x+1},\ x<y+2) \\ &= x+1<y+2 \\ &\equiv x<y+1. \end{aligned}$$

□

Example 9.19 A more complicated example is:

$$\begin{aligned} & wp(\texttt{x:=x+a; y:=y-1},\ x=(b-y)\cdot a) \\ &= wp(\texttt{x:=x+a},\ wp(\texttt{y:=y-1},\ x=(b-y)\cdot a)) \\ &= wp(\texttt{x:=x+a},\ x=(b-y+1)\cdot a) \\ &= x+a=(b-y+1)\cdot a \\ &\equiv x=(b-y)\cdot a. \end{aligned}$$

Give the precondition $x=(b-y)\cdot a$, the statement (predicate transformer) `x:=x+a; y:=y-1` does nothing! However, while the precondition is the same as the postcondition, the statements `x:=x+1;y:=y-1` have changed the relative values of x and y and this has meaning in terms of the execution of the program. □

Definition 9.20 A predicate I is an *invariant* of S iff $wp(\texttt{S}, I)=I$. □

Definition 9.21 Here are two equivalent definitions for `if`-statements:

$$wp(\texttt{if B then S1 else S2},\ q) = (B\rightarrow wp(\texttt{S1},\ q))\ \wedge\ (\neg B\rightarrow wp(\texttt{S2},\ q))$$

$$wp(\texttt{if B then S1 else S2},\ q) = (B\wedge wp(\texttt{S1},\ q))\ \vee\ (\neg B\wedge wp(\texttt{S2},\ q)).$$

□

The definition is straightforward because the predicate B partitions the set of states into two disjoint subsets, and the preconditions are then determined by the actions of each `Si` on its subset. The equivalent definition follows from the propositional equivalence $(p\rightarrow q)\wedge(\neg p\rightarrow r)\equiv(p\wedge q)\vee(\neg p\wedge r)$.

Example 9.22

$$\begin{aligned} & wp(\texttt{if y=0 then x:=0 else x:=y+1},\ x=y) \\ &= (y=0\rightarrow wp(\texttt{x:=0},\ x=y))\ \wedge\ (y\neq 0\rightarrow wp(\texttt{x:=y+1},\ x=y)) \\ &\equiv ((y=0)\rightarrow(y=0))\ \wedge\ ((y\neq 0)\rightarrow(y+1=y)) \\ &\equiv \mathit{true}\ \wedge\ ((y\neq 0)\rightarrow \mathit{false}) \\ &\equiv \neg(y\neq 0) \\ &\equiv y=0. \end{aligned}$$

□

Definition 9.23 Here are two equivalent definitions for `while`-statements:

$$wp(\texttt{while B do S},\ q) = (\neg B\rightarrow q)\ \wedge\ (B\rightarrow wp(\texttt{S; while B do S},\ q))$$

$$wp(\texttt{while B do S},\ q) = (\neg B\wedge q)\ \vee\ (B\wedge wp(\texttt{S; while B do S},\ q)).$$

□

The execution of a `while`-statement can proceed in one of two ways:

- The statement can terminate immediately because the Boolean expression evaluates to false, in which case the state does not change so the precondition is the same as the postcondition.
- The expression can evaluate to true and cause the body of the loop to be executed. Upon termination of the body, the `while`-statement again attempts to establish the postcondition.

Example 9.24 Because of the recursive definition of the weakest precondition for a `while`-statement, we cannot constructively compute it; nevertheless, an attempt to do so is informative. Let `W` be an abbreviation for `while x>0 do x:=x-1`.

$$\begin{aligned} & wp(\texttt{W},\ x=0) \\ &= (\neg(x>0)\wedge(x=0))\ \vee\ ((x>0)\wedge wp(\texttt{x:=x-1; W},\ x=0)) \\ &\equiv (x=0)\ \vee\ ((x>0)\wedge wp(\texttt{x:=x-1},\ wp(\texttt{W},\ x=0))) \\ &\equiv (x=0)\ \vee\ ((x>0)\wedge wp(\texttt{W},\ x=0)\{x\leftarrow x-1\}). \end{aligned}$$

We have to perform the substitution $\{x\leftarrow x-1\}$ on $wp(\texttt{W},\ x=0)$. But we have just computed a value for $wp(\texttt{W},\ x=0)$. Performing the substitution and simplifying gives:

$$\begin{aligned} & wp(\texttt{while x>0 do x:=x-1},\ x=0) \\ &= (x=0)\ \vee\ (x=1)\ \vee\ ((x>1)\wedge wp(\texttt{W},\ x=0)\{x\leftarrow x-1\}\{x\leftarrow x-1\}). \end{aligned}$$

Continuing the computation, we arrive at the following formula:

$$\begin{aligned} & wp(\texttt{W},\ x=0) \\ &= (x=0)\ \vee\ (x=1)\ \vee\ (x=2)\ \vee\ \cdots \\ &\equiv x\geq 0. \end{aligned}$$

The theory of fixpoints can be used to formally justify the infinite substitution, but we will not do it here. Note that if the postcondition were changed to $x=1$, the computation of the weakest precondition would give $\mathit{false}\vee\mathit{false}\vee\cdots$, showing that for no initial states can this `while`-statement terminate in a state with $x=1$. □

There are several theorems about *wp* that we will need.

Theorem 9.25 (Distribution) $\models wp(\mathrm{S}, p) \wedge wp(\mathrm{S}, q) \leftrightarrow wp(\mathrm{S}, p \wedge q)$.

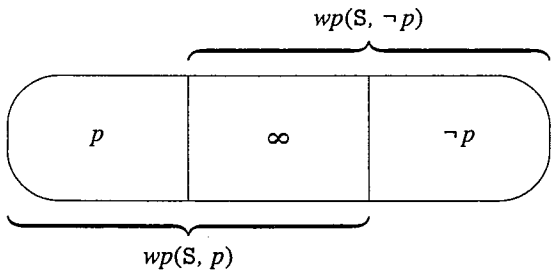
Proof: Let s be an arbitrary state in which $wp(\mathrm{S}, p) \wedge wp(\mathrm{S}, q)$ is true. Then both $wp(\mathrm{S}, p)$ and $wp(\mathrm{S}, q)$ are true in s. Executing S leads to a state s' such that p and q are both true in s'. By propositional calculus, $p \wedge q$ is true in s'. Since s was arbitrary, we have proved that

$$\{s \mid \models wp(\mathrm{S}, p) \wedge wp(\mathrm{S}, q)\} \subseteq \{s \mid \models wp(\mathrm{S}, p \wedge q)\},$$

that is, $\models wp(\mathrm{S}, p) \wedge wp(\mathrm{S}, q) \rightarrow wp(\mathrm{S}, p \wedge q)$. The converse is left as an exercise. ∎

Corollary 9.26 (Excluded miracle) $\models wp(\mathrm{S}, p) \wedge wp(\mathrm{S}, \neg p) \leftrightarrow wp(\mathrm{S}, \mathit{false})$.

According to the definition of assertion, any precondition is true in states for which the computation does not terminate. The following diagram shows how $wp(\mathrm{S}, \mathit{false})$ is the intersection (conjunction) of the weakest preconditions $wp(\mathrm{S}, p)$ and $wp(\mathrm{S}, \neg p)$. From any state, the computation is either non-terminating, or it terminates in a state where p is true, or in a state where $\neg p$ is true.



It also furnishes an informal proof of the following theorem.

Theorem 9.27 (Duality) $\models \neg wp(\mathrm{S}, \neg p) \rightarrow wp(\mathrm{S}, p)$.

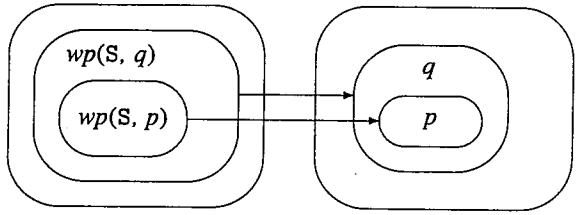
Theorem 9.28 (Monotonicity) *If* $\models p \rightarrow q$ *then* $\models wp(\mathrm{S}, p) \rightarrow wp(\mathrm{S}, q)$.

Proof:

1.	$\models wp(\mathrm{S}, p) \wedge wp(\mathrm{S}, \neg q) \rightarrow wp(\mathrm{S}, p \wedge \neg q)$	Theorem 9.25
2.	$\models p \rightarrow q$	Assumption
3.	$\models \neg (p \wedge \neg q)$	2, PC
4.	$\models wp(\mathrm{S}, p) \wedge wp(\mathrm{S}, \neg q) \rightarrow wp(\mathrm{S}, \mathit{false})$	1,3
5.	$\models wp(\mathrm{S}, \mathit{false}) \rightarrow wp(\mathrm{S}, q)$	Theorem 9.26
6.	$\models wp(\mathrm{S}, p) \wedge wp(\mathrm{S}, \neg q) \rightarrow wp(\mathrm{S}, q)$	4, 5, PC
7.	$\models wp(\mathrm{S}, p) \rightarrow \neg wp(\mathrm{S}, \neg q) \vee wp(\mathrm{S}, q)$	6, PC
8.	$\models wp(\mathrm{S}, p) \rightarrow wp(\mathrm{S}, q)$	7, Theorem 9.27

∎

A weaker formula satisfies more states:



Example 9.29 In Example 9.19 we showed that

$$wp(\texttt{x:=x+1; y:=y+2}, x < y) = x < y + 1.$$

Change the postcondition to $x < y - 2$, where $(x < y - 2) \rightarrow (x < y)$. A calculation similar to the one in the example gives

$$wp(\texttt{x:=x+1; y:=y+2}, x < y - 2) = x < y - 1,$$

and clearly $(x < y - 1) \rightarrow (x < y + 1)$. □

9.3 The deductive system $\mathcal{HL}$

A deductive system whose formulas are assertions can be used to prove properties of programs. The deductive system $\mathcal{HL}$ (*Hoare Logic*) is sound and relatively complete for proving partial correctness. If a program is partially correct, the assertion for the correctness of the program can be proved from the axioms and rules of inference. However, the completeness is *relative* to the completeness of the domain theories, because we assume that *all* true formulas in the domain(s) are axioms. In particular,

if the program operates on the domain of integers, then $\mathcal{HL}$ cannot be absolutely complete because the theory of integers is not complete.

Definition 9.30 (Deductive system $\mathcal{HL}$)

Domain axioms

Every true formula over the domain(s) of the program variables.

Assignment axiom

$$\vdash \{p(x)\{x \leftarrow t\}\}\ \texttt{x := t}\ \{p(x)\}.$$

Composition rule

$$\frac{\vdash \{p\}\ \texttt{S1}\ \{q\} \qquad \vdash \{q\}\ \texttt{S2}\ \{r\}}{\vdash \{p\}\ \texttt{S1;S2}\ \{r\}}.$$

Alternative rule

$$\frac{\vdash \{p \wedge B\}\ \texttt{S1}\ \{q\} \qquad \vdash \{p \wedge \neg B\}\ \texttt{S2}\ \{q\}}{\vdash \{p\}\ \texttt{if B then S1 else S2}\ \{q\}}.$$

Loop rule

$$\frac{\vdash \{p \wedge B\}\ \texttt{S}\ \{p\}}{\vdash \{p\}\ \texttt{while B do S}\ \{p \wedge \neg B\}}.$$

Consequence rule

$$\frac{\vdash p_1 \rightarrow p \qquad \vdash \{p\}\ \texttt{S}\ \{q\} \qquad \vdash q \rightarrow q_1}{\vdash \{p_1\}\ \texttt{S}\ \{q_1\}}.$$

□

Since $\vdash A \rightarrow A$ is a domain axiom for any A, the consequence rule can be used with just two premises either to strengthen the precondition or to weaken the postcondition.

The assignment axiom can be understood in terms of the definition of the semantics of the assignment statement in terms of weakest preconditions. The verification is done backwards: if you want to prove the assertion

$$\vdash \{p_0\}\ \texttt{x := t}\ \{p(x)\},$$

you are allowed to assume

$$\vdash \{p(x)\{x \leftarrow t\}\}\ \texttt{x := t}\ \{p(x)\}.$$

If $p_0 \rightarrow p(x)\{x \leftarrow t\}$, a formula in the domain, is true, it is a domain axiom, and a use of the consequence rule proves the assertion.

The formula p in the loop rule is called an *invariant*: it describes the behavior of a single execution of the statement S in the `while`-statement body. To prove

$$\vdash \{p_0\}\ \texttt{while B do S}\ \{q\},$$

find an invariant p. You then need to show that $p_0 \rightarrow p$ is true so that the precondition of the loop rule holds. If you can also show that $(p \wedge \neg B) \rightarrow q$ is true, then no matter how many times the loop is executed, the postcondition of the rule implies the postcondition q.

The difficulty in proving programs is to find appropriate invariants. The formula *true* (the weakest formula) is an invariant of any loop because it is true in any state reached after the computation of S. However, it is of no use in proving a program, since we will never write a loop whose whole purpose is to establish the truth of the formula *true*! On the other hand, if the formula is too strong, it will not be an invariant.

Example 9.31 $x = 5$ is too strong to be an invariant of the `while`-statement

```
while x > 0 do x := x - 1,
```

because $x = 5 \wedge x > 0$ does not imply that $x = 5$ after executing `x := x - 1`. However, $x \geq 0$ is an invariant because $x \geq 0 \wedge x > 0$ does imply $x \geq 0$ after executing the loop body. The loop rule enables us to conclude that if the loop terminates, then $x \geq 0 \wedge \neg (x > 0)$ holds upon termination. This can be simplified to $x = 0$ by reasoning within the domain and using the consequence rule. □

9.4 Program verification

Let us use $\mathcal{HL}$ to proving the partial correctness of the following program P.

```
{true}
x := 0;
{x = 0}
y := b;
{x = 0 ∧ y = b}
while y <> 0 do
    {x = (b − y) · a}
    begin x := x + a; y := y - 1 end;
{x = a · b}
```

We have *annotated* P with formulas between statements. Given

$$\{p_1\}\ \texttt{S1}\ \{p_2\}\ \texttt{S2}\ \cdots\ \{p_n\}\ \texttt{Sn}\ \{p_{n+1}\},$$

we prove $\{p_i\}$ `Si` $\{p_{i+1}\}$ for all i, and then conclude $\{p_1\}$ `S1 ; ··· ; Sn` $\{p_{n+1}\}$ by repeated application of the composition rule. See Apt & Olderog (1991, Section 3.4) for a formal proof that $\mathcal{HL}$ with annotations is equivalent to $\mathcal{HL}$ without them.

Theorem 9.32 $\vdash \{true\}$ `P` $\{x = a \cdot b\}$.

Proof: From the assignment axiom we have $\{0 = 0\}$ `x:=0` $\{x = 0\}$, and from the consequence rule with premise $true \to (0 = 0)$, we have $\{true\}$ `x:=0` $\{x = 0\}$. The proof of $\{x = 0\}$ `y:=b` $\{(x = 0) \wedge (y = b)\}$ is similar.

Let p be the formula $x = (b - y) \cdot a$. We showed that $\{p\}$ `x:=x+a; y:=y-1` $\{p\}$ (Example 9.19), that is, p is an invariant of the loop body. By the consequence rule, the precondition can be strengthened to

$$\{p \wedge y \neq 0\}\ \texttt{x:=x+a; y:=y-1}\ \{p\}.$$

But this is the premise needed for the loop rule, and the conclusion is:

$\{p\}$
```
while y<>0 do
  begin x:=x+a; y:=y-1 end
```
$\{p \wedge \neg (y \neq 0)\}$

The postcondition of the loop can be written $p \wedge (y = 0)$; using domain axioms and the consequence rule, we can deduce $x = a \cdot b$, the postcondition of `P`. ∎

In practice, the domain axioms and the consequence rule are used implicitly, as we are interested in the program-specific aspects of the proofs. Furthermore, the application of all the structural rules is purely mechanical; the only aspects of the proof requiring intelligence are the choice of the loop invariants and deduction within the domain.

*Total correctness**

We have proved only partial correctness. If the initial value of `b` is negative, the program will not terminate. However, if the precondition is strengthened to $b \geq 0$, the program is in fact totally correct. We will not give the extension needed to make $\mathcal{HL}$ a deductive system for total correctness (Apt & Olderog 1991, Section 3.3); instead, we demonstrate the method on the program `P`.

Strengthening the precondition will obviously not invalidate the proof of partial correctness, since a stronger precondition simply selects a subset of the set of states for which the computation is correct. Thus all that is needed is to prove that the program terminates, and the only statement that can possibly not terminate is the `while`-statement. To show termination, we search for a numeric function whose value *decreases* with every execution of the loop, and whose value has an invariant lower bound. The loop must eventually terminate because there cannot be an infinite decreasing sequence greater than the lower bound.

In `P`, y itself is a numeric function which decreases with each execution of the loop, and $y \geq 0$ can be added as a lower bound to the invariant. By the precondition $b \geq 0$ it follows that $y \geq 0$ after the two initial assignments, and it is easy to see that the addition of $y \geq 0$ to the invariant does not change the invariance of the formula:

$\{(x = (b - y) \cdot a \wedge y \geq 0 \wedge y \neq 0\}$
`x:=x+a; y:=y-1`
$\{(x = (b - y) \cdot a \wedge y \geq 0.\}$

Since y is decreasing and yet bounded from below by $y \geq 0$, the loop must terminate and the program is totally correct.

This method is applicable to any program for which we can find a *well-founded set*: an ordered, decreasing set with a lower bound.

Example 9.33 Let `Str` be a variable of type string and consider a program of the form:

```
Str := { Some initial value };
while Str <> '' do
  Str := Func(Str)
```

The set of strings under lexicographic order is well-founded because it has the empty string as a lower bound. If `Func` returns a string whose lexicographic value is less that that of `Str` (say it returns `"Hello work"` when called with `"Hello world"`), eventually the loop must terminate. If the program can be proved partially correct, it will also be totally correct. □

9.5 Program synthesis

Assertions may also be used in the *synthesis* of programs: the construction of a program directly from a formal specification. The emphasis is on finding invariants of loops, because the other aspects of program proving are purely mechanical. Starting from the assertion with the pre- and postconditions of the entire program, invariants are hypothesized as modifications of the postcondition and the program is constructed to maintain the truth of the invariant. We demonstrate the method by developing two different programs for finding the integer square root of a non-negative integer:

$$\{0 \leq a\}\ \texttt{S}\ \{0 \leq x^2 \leq a < (x+1)^2\}.$$

Solution 1

A loop is used to calculate values of x until the postcondition holds. Suppose we let the first part of the postcondition be the invariant and try to establish the second part upon termination of the loop. This gives the following program outline, where ? and `B(x,a)` represent expressions that must be determined.

$\{0 \le a\}$
`x := ?;`
`while B(x,a) do`
  $\{0 \le x^2 \le a\}$
  `x := ?;`
$\{0 \le x^2 \le a < (x+1)^2\}$.

Let p denote the loop invariant $0 \le x^2 \le a$. Given the precondition $0 \le a$, p can be established if the first statement is `x:=0`. The postcondition of the `while`-statement is $p \wedge \neg B(x,a)$, so `B(x,a)` should be chosen to imply the postcondition of the program. This is easy to do if $\neg B(x,a)$ is $a < (x+1)^2$, that is, if `B(x,a)` is `(x+1)*(x+1)<=a`. Since the loop is to terminate when x is large enough, a loop body consisting of a simple increment of x should suffice. Here is the resulting program:

$\{0 \le a\}$
`x := 0;`
`while (x+1)*(x+1) <= a do`
  $\{0 \le x^2 \le a\}$
  `x := x + 1;`
$\{0 \le x^2 \le a < (x+1)^2\}$.

We must check the loop invariant $\{p \wedge B\}$ `S` $\{p\}$, in this case,

$$\{0 \le x^2 \le a \wedge (x+1)^2 \le a\} \texttt{ x:=x+1 } \{0 \le x^2 \le a\}.$$

By the semantics of the assignment statement,

$$\{0 \le (x+1)^2 \le a\} \texttt{ x:=x+1 } \{0 \le x^2 \le a\},$$

but

$$(0 \le x^2 \le a \wedge (x+1)^2 \le a) \rightarrow (0 \le (x+1)^2 \le a),$$

so the invariant follows by the consequence rule.

Solution 2

Suppose that instead of deleting part of the postcondition to obtain an invariant, we introduce a new variable y to bound x from above:

$$0 \le x^2 \le a < y^2.$$

The loop can be terminated when `y=x+1`. For the invariant to be true, y must always be greater than x. Furthermore, there is no point in having y greater than `a+1` so we initialize `y:=a+1` and add $x < y \le a+1$ to the invariant, giving

$$\{(0 \le x^2 \le a < y^2) \wedge (x < y \le a+1)\}$$

as the candidate for an invariant. The program outline is:

$\{0 \le a\}$
`x := 0; y := a+1;`
`while y <> x+1 do`
  $\{(0 \le x^2 \le a < y^2) \wedge (x < y \le a+1)\}$
  `?;`
$\{0 \le x^2 \le a < (x+1)^2\}$.

Before continuing with the synthesis, let us try an example.

Example 9.34 Suppose that $a = 14$. Initially, $x = 0$ and $y = 15$. The loop must terminate when $x = 3$ and $y = x+1 = 4$, so $0 \le 9 \le 14 < 16$. We can either increase x or decrease y while maintaining the invariant $0 \le x^2 \le a < y^2$. Rather than increment or decrement the variables by 1, let us take the midpoint $(x+y)/2 = (0+15)/2 = 7$ (using integer division with truncation) and assign it to either x or y, as appropriate, to narrow the range. In this case, $a = 14 < 49 = 7 \cdot 7$, so assigning 7 to y (but not to x) will maintain the invariant. On the next iteration, $(x+y)/2 = (0+7)/2 = 3$ and $3 \cdot 3 = 9 < 14 = a$, so assigning 3 to x will maintain the invariant. After two more iterations, $x = 3$, $y = 4 = x+1$ and the loop terminates. □

Here is an outline for the loop body annotated with the following assertions: the invariant $\{p \wedge B\}$ `S1` $\{p\}$ that must be proved and additional assertions that follow from the assignment axioms.

$\{p \wedge (y \ne x+1)\}$
`z := (x+y) div 2;`
$\{p \wedge (y \ne x+1) \wedge (\lfloor z = (x+y)/2 \rfloor)\}$
`if Cond(x,y,z) then`
  $\{p\{x \leftarrow z\}\}$
  `x := z`
`else`
  $\{p\{y \leftarrow z\}\}$
  `y := z`
$\{p\}$

where z is a new variable and `Cond(x,y,z)` is a Boolean expression, chosen so that

$$[p \wedge (y \ne x+1) \wedge (z = \lfloor (x+y)/2 \rfloor) \wedge \mathit{Cond}(x,y,z)] \rightarrow p\{x \leftarrow z\},$$

and

$$[p \wedge (y \neq x+1) \wedge (z = \lfloor (x+y)/2 \rfloor) \wedge \neg Cond(x,y,z)] \rightarrow p\{y \leftarrow z\}.$$

The conclusions can be written as

$$p\{x \leftarrow z\} \equiv (0 \leq z^2 \leq a < y^2) \wedge (z < y \leq a+1),$$

and

$$p\{y \leftarrow z\} \equiv (0 \leq x^2 \leq a < z^2) \wedge (x < z \leq a+1).$$

The first conjunct of the premise is p and its first conjunct is $0 \leq x^2 \leq a < y^2$. Clearly, the first conjuncts of $p\{x \leftarrow z\}$ and $p\{y \leftarrow z\}$ will be true if `Cond(x,y,z)` is chosen to be `z*z <= a`.

What about the second conjuncts of $p\{x \leftarrow z\}$ and $p\{y \leftarrow z\}$? From $x < y \leq a+1$, the second conjunct of p, and $z = \lfloor (x+y)/2 \rfloor$, $z < y \leq a+1$ follows. Similarly, from $x < y \leq a+1$ and $z = \lfloor (x+y)/2 \rfloor$, $x < z \leq a+1$ follows, provided that $y \neq x+1$. But that formula is also a premise.

Here is the final program:

```
{0 ≤ a}
x := 0; y := a+1;
while y <> x+1 do
    {0 ≤ x^2 ≤ a < y^2  ∧ x < y ≤ a+1}
    z := (x+y) div 2;
    if z*z <= a then x := z else y := z
{0 ≤ x^2 ≤ a < (x+1)^2}.
```

While it may seem strange to develop the proof of a program concurrently with the program itself, this method is recommended because it ensures that bugs are not built into the program. Instead provably correct program fragments are combined and expanded until the complete program has been constructed.

9.6 Soundness and completeness of $\mathcal{HL}$

We start with definitions and lemmas which will be used in the proofs. The fragment of Pascal is extended with two statements `skip` and `abort`, whose semantics are defined as follows.

Definition 9.35 $wp(\texttt{skip}, p) = p$ and $wp(\texttt{abort}, p) = false$. □

Definition 9.36

$\texttt{W} = \texttt{while B do S}$

$\texttt{W}^0 = \texttt{if B then abort else skip}$

$\texttt{W}^{k+1} = \texttt{if B then S;W}^k \texttt{ else skip}.$ □

W is just an abbreviation for the statement `while B do S`. The inductive definition will be used to prove that an execution of W is equivalent to $\texttt{W}^k$ for some k.

Lemma 9.37 $wp(\texttt{W}^0, p) \equiv \neg B \wedge (\neg B \rightarrow p)$.

Proof:

$wp(\texttt{W}^0, p)$	=
$wp(\texttt{if B then abort else skip}, p)$	=
$(B \rightarrow wp(\texttt{abort}, p)) \wedge (\neg B \rightarrow wp(\texttt{skip}, p))$	=
$(B \rightarrow false) \wedge (\neg B \rightarrow p)$	≡
$(\neg B \vee false) \wedge (\neg B \rightarrow p)$	≡
$\neg B \wedge (\neg B \rightarrow p)$.	

Lemma 9.38 $\bigvee_{k=0}^{\infty} wp(\texttt{W}^k, p) \rightarrow wp(\texttt{W}, p)$.

Proof: We show by induction that for each k, $wp(\texttt{W}^k, p) \rightarrow wp(\texttt{W}, p)$.

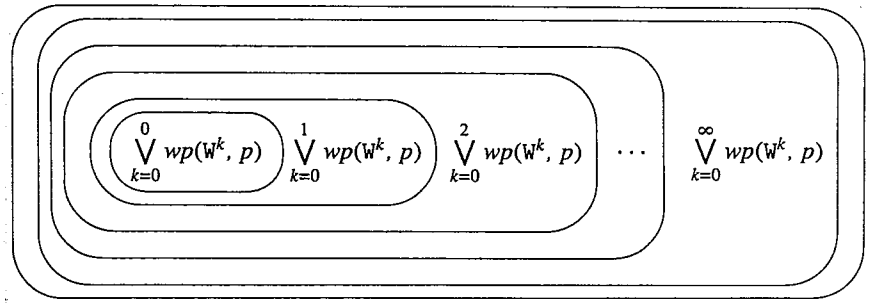
For $k = 0$:

1.	$wp(\texttt{W}^0, p) \rightarrow \neg B \wedge (\neg B \rightarrow p)$	Lemma 9.37
2.	$wp(\texttt{W}^0, p) \rightarrow \neg B \wedge p$	1, PC
3.	$wp(\texttt{W}^0, p) \rightarrow (\neg B \wedge p) \vee (B \wedge wp(\texttt{S;W}, p))$	2, PC
4.	$wp(\texttt{W}^0, p) \rightarrow wp(\texttt{W}, p)$	3, Def. 9.23

For $k > 0$:

1.	$wp(\texttt{W}^{k+1}, p) = wp(\texttt{if B then S;W}^k \texttt{ else skip}, p)$	Def. 9.36
2.	$wp(\texttt{W}^{k+1}, p) \equiv (B \rightarrow wp(\texttt{S;W}^k, p)) \wedge (\neg B \rightarrow wp(\texttt{skip}, p))$	Def. 9.21
3.	$wp(\texttt{W}^{k+1}, p) \equiv (B \rightarrow wp(\texttt{S}, wp(\texttt{W}^k, p))) \wedge (\neg B \rightarrow wp(\texttt{skip}, p))$	Def. 9.17
4.	$wp(\texttt{W}^{k+1}, p) \equiv (B \rightarrow wp(\texttt{S}, wp(\texttt{W}^k, p))) \wedge (\neg B \rightarrow p)$	Def. 9.35
5.	$wp(\texttt{W}^{k+1}, p) \rightarrow (B \rightarrow wp(\texttt{S}, wp(\texttt{W}, p))) \wedge (\neg B \rightarrow p)$	Ind. hyp.
6.	$wp(\texttt{W}^{k+1}, p) \rightarrow (B \rightarrow wp(\texttt{S;W}, p)) \wedge (\neg B \rightarrow p)$	Def. 9.17
7.	$wp(\texttt{W}^{k+1}, p) \rightarrow wp(\texttt{W}, p)$	Def. 9.23

As k increases, more and more states are included in $\bigvee_{i=0}^{k} wp(\texttt{W}^i, p)$:



Theorem 9.39 (Soundness of $\mathcal{HL}$) *If* $\vdash_{HL} \{p\}\ \mathtt{S}\ \{q\}$ *then* $\models \{p\}\ \mathtt{S}\ \{q\}$.

Proof: The proof is by induction on the length of the $\mathcal{HL}$ proof. By assumption, the domain axioms are true, and the use of the consequence rule can be justified by the soundness of MP in the predicate calculus.

By Lemma 9.13, $\models \{p\}\ \mathtt{S}\ \{q\}$ iff $\models p \rightarrow wp(\mathtt{S}, q)$, so it is sufficient to prove $\models p \rightarrow wp(\mathtt{S}, q)$. The soundness of the Assignment Axioms is immediate by Definition 9.15. Suppose that the composition rule is used. By the inductive hypothesis, we can assume that $\models p \rightarrow wp(\mathtt{S1}, q)$ and $\models q \rightarrow wp(\mathtt{S2}, r)$. From the second assumption and monotonicity (Theorem 9.28),

$$\models wp(\mathtt{S1}, q) \rightarrow wp(\mathtt{S1}, wp(\mathtt{S2}, r)).$$

By the rule of consequence and the first assumption, $\models p \rightarrow wp(\mathtt{S1}, wp(\mathtt{S2}, r))$. By the definition of wp for a compound statement this is $\models p \rightarrow wp(\mathtt{S1;S2}, r)$.

We leave the proof of the soundness of the alternative rule as an exercise. For the loop rule, by (structural) induction we assume that $\models (p \wedge B) \rightarrow wp(\mathtt{S}, p)$ and show $\models p \rightarrow wp(\mathtt{W}, p \wedge \neg B)$. We will prove by numerical induction that $\models p \rightarrow wp(\mathtt{W}^k, p \wedge \neg B)$, for all k. For $k = 0$, the proof of $\models wp(\mathtt{W}^0, p \wedge \neg B) = wp(\mathtt{W}, p \wedge \neg B)$ is the same as the proof of the base case in Lemma 9.38. The inductive step is proved as follows:

1.	$\models p \rightarrow (\neg B \rightarrow (p \wedge \neg B))$	PC
2.	$\models p \rightarrow (\neg B \rightarrow wp(\mathtt{skip}, p \wedge \neg B))$	Def. 9.35
3.	$\models (p \wedge B) \rightarrow wp(\mathtt{S}, p)$	Structural ind. hyp.
4.	$\models p \rightarrow wp(\mathtt{W}^k, p \wedge \neg B)$	Numerical ind. hyp.
5.	$\models (p \wedge B) \rightarrow wp(\mathtt{S}, wp(\mathtt{W}^k, p \wedge \neg B))$	3, 4, Monotonicity
6.	$\models (p \wedge B) \rightarrow wp(\mathtt{S;W}^k, p \wedge \neg B)$	5, Composition
7.	$\models p \rightarrow (B \rightarrow wp(\mathtt{S;W}^k, p \wedge \neg B))$	6, PC
8.	$\models p \rightarrow wp(\mathtt{if\ B\ then\ S;W}^k\ \mathtt{else\ skip}, p \wedge \neg B)$	2, 7, Def. 9.21
9.	$\models p \rightarrow wp(\mathtt{W}^{k+1}, p \wedge \neg B)$	Def. 9.36

By (infinite) disjunction, $\models p \rightarrow \bigvee_{k=0}^{\infty} wp(\mathtt{W}^k, p \wedge \neg B)$, and $\models p \rightarrow wp(\mathtt{W}, p \wedge \neg B)$ follows by Lemma 9.38. ∎

Theorem 9.40 (Completeness of $\mathcal{HL}$) *If* $\models \{p\}\ \mathtt{S}\ \{q\}$, *then* $\vdash_{HL} \{p\}\ \mathtt{S}\ \{q\}$.

Proof: We have to show that if $\models p \rightarrow wp(\mathtt{S}, q)$, then $\vdash_{HL} \{p\}\ \mathtt{S}\ \{q\}$. The proof is by structural induction on S. Note that $p \rightarrow wp(\mathtt{S}, q)$ is just a formula of the domain, so $\vdash p \rightarrow wp(\mathtt{S}, q)$ follows by the domain axioms.

Case 1: Assignment statement, `x:=t`.
$\vdash \{q\{x \leftarrow t\}\}\ \mathtt{x:=t}\ \{q\}$ is an axiom, so $\vdash \{wp(\mathtt{x:=t}, q)\}\ \mathtt{x:=t}\ \{q\}$ by Definition 9.15. By assumption, $\vdash p \rightarrow wp(\mathtt{x:=t}, q)$, so by the consequence rule, $\vdash \{p\}\ \mathtt{x:=t}\ \{q\}$.

Case 2: Composition, `S1;S2`.
By assumption, $\models p \rightarrow wp(\mathtt{S1;S2}, q)$ which is equivalent to $\models p \rightarrow wp(\mathtt{S1}, wp(\mathtt{S2}, q))$ by Definition 9.17, so by the inductive hypothesis, $\vdash \{p\}\ \mathtt{S1}\ \{wp(\mathtt{S2}, q)\}$. Obviously, $\models wp(\mathtt{S2}, q) \rightarrow wp(\mathtt{S2}, q)$, so again by the inductive hypothesis (with $wp(\mathtt{S2}, q)$ as p), $\vdash \{wp(\mathtt{S2}, q)\}\ \mathtt{S2}\ \{q\}$. An application of the composition rule gives $\vdash \{p\}\ \mathtt{S1;S2}\ \{q\}$.

Case 3: `if`-statement. Exercise.

Case 4: `while`-statement, `W = while B do S`.

1.	$\models wp(\mathtt{W}, q) \wedge B \rightarrow wp(\mathtt{S;W}, q)$	Def. 9.23
2.	$\models wp(\mathtt{W}, q) \wedge B \rightarrow wp(\mathtt{S}, wp(\mathtt{W}, q))$	Def. 9.17
3.	$\vdash \{wp(\mathtt{W}, q) \wedge B\}\ \mathtt{S}\ \{wp(\mathtt{W}, q)\}$	Inductive hypothesis
4.	$\vdash \{wp(\mathtt{W}, q)\}\ \mathtt{W}\ \{wp(\mathtt{W}, q) \wedge \neg B\}$	Loop rule
5.	$\vdash (wp(\mathtt{W}, q) \wedge \neg B) \rightarrow q$	Def. 9.23, Domain axiom
6.	$\vdash \{wp(\mathtt{W}, q)\}\ \mathtt{W}\ \{q\}$	4, 5, Consequence rule
7.	$\vdash p \rightarrow wp(\mathtt{W}, q)$	Assumption, Domain axiom
8.	$\vdash \{p\}\ \mathtt{W}\ \{q\}$	Consequence rule

∎

9.7 Exercises

1. What is $wp(\mathtt{S}, \mathit{true})$ for any statement S?

2. Let S1 be `x:=x+y` and S2 be `y:=x*y`. What is $wp(\mathtt{S1;S2}, x < y)$?

3. Prove $\models wp(\mathtt{S}, p \wedge q) \rightarrow wp(\mathtt{S}, p) \wedge wp(\mathtt{S}, q)$, (the converse direction of Theorem 9.25).

4. Prove that

   $wp(\mathtt{if\ B\ then\ begin\ S1;S3\ end\ else\ begin\ S2;S3\ end}, q) =$
   $wp(\mathtt{if\ B\ then\ S1\ else\ S2;\ S3}, q)$.

5. * Suppose that $wp(\mathtt{S}, q)$ is defined as the weakest formula p that ensures *total* correctness of S, that is, if S is started in a state in which p is true, then it *will* terminate in a state in which q is true. Show that under this definition $\models \neg wp(\mathtt{S}, \neg q) \equiv wp(\mathtt{S}, q)$ and $\models wp(\mathtt{S}, p) \vee wp(\mathtt{S}, q) \equiv wp(\mathtt{S}, p \vee q)$.

6. Complete the proofs of the soundness and completeness of $\mathcal{HL}$ for the alternative rule (Theorems 9.39 and 9.40).

7. Prove the partial correctness of the following program.

$\{a \geq 0\}$
```
x := 0; y := 1;
while y <= a do
  begin
    x := x + 1;
    y := y + 2*x + 1
  end
```
$\{0 \leq x^2 \leq a < (x+1)^2\}$

8. Prove the partial correctness of the following program.

$\{a > 0 \wedge b > 0\}$
```
x := a; y := b;
while x <> y do
  if x > y
    then x := x-y
    else y := y-x
```
$\{x = \gcd(a, b)\}$

9. Prove the partial correctness of the following program.

$\{a > 0 \wedge b > 0\}$
```
x := a; y := b;
while x <> y do
  begin
    while x > y then x := x-y;
    while y > x then y := y-x
  end
```
$\{x = \gcd(a, b)\}$

10. Prove the partial correctness of the following program.

$\{a \geq 0 \wedge b \geq 0\}$
```
x := a; y := b; z := 1;
while y <> 0 do
  if odd(y)
    then begin y := y - 1; z := x*z end
    else begin x := x*x; y := y div 2 end
```
$\{z = a^b\}$

10 Programs: Formal specification with Z

Suppose that you are asked to develop a program. The first thing that you must do is to write a *specification* of the program: what the program must do. Formally, a specification is a precondition and a postcondition, and the correctness of the program is defined relative to a specific precondition and postcondition. In practice, predicate calculus and number theory as we have presented them are inconvenient for writing specifications, and other notations have been devised.

Z is a set of notations for structuring specifications: a individual component of the specification is written as a *schema* that contains declarations of variables and formulas that constrain the values of the variables during the execution of the program. Set theory is used in the high-level specification to avoid premature choice of an implementation. The schemata are combined using a *schema calculus* to produce the full specification of the program. A set of schemata can be refined to supply implementation details; the refinement must then be proved to be equivalent to the high-level schemata.

We begin this chapter with a simple example. Then we present an overview of the Z notation and concepts, followed by a more extensive example.

10.1 Case study: a traffic signal

A traffic signal is defined by the status of each light: off or on. We begin the specification by defining a data type. In Z, as in Pascal, you can define a data type by enumerating the possible values:

$$STATUS ::= off \mid on.$$

A traffic signal is composed of three colored lights, each of which has a status. We define the schema *Signal* which declares three variables of type STATUS and an invariant. Declarations appear above the short dividing line and predicates appear below it. The invariant specifies that the red right is on if and only if the green light is off.

Signal

$$red, yellow, green : STATUS$$

$$red = on \Leftrightarrow green = off$$

Once a state has been described, schemata may be constructed to describe operations on states. (Note that different countries use different conventions for the traffic lights, so the following specification may not hold where you live.) The first such schema *WarnGo* specifies an operation that can happen only if the red light is on: the yellow light is (also) turned on to signal that the green will shortly appear. The declarative part of the schema contains two occurrences of the schema *Signal*, one to describe the state before the operation is carried out, and a second, *decorated* by a prime, to describe the state after the operation.

WarnGo

$$Signal$$
$$Signal'$$

$$red = on \land yellow = off \land green = off$$
$$red' = on \land yellow' = on \land green' = off$$

There are two predicates: the first one is the *precondition* and specifies what must be true in order to carry out the operation, and the second one, the *postcondition* which contains the decorated variables, specifies what must be true after the operation.

Since an undecorated schema and its primed version occur so often, there is an abbreviation for it: $\Delta Signal$. It is formally defined as:

ΔS

$$S$$
$$S'$$

Using the abbreviation, *WarnGo* becomes:

WarnGo

$$\Delta Signal$$

$$red = on \land yellow = off \land green = off$$
$$red' = on \land yellow' = on \land green' = off$$

We can now write the schemata for the other operations of the traffic signal.

Go

$$\Delta Signal$$

$$red = on \land yellow = on \land green = off$$
$$red' = off \land yellow' = off \land green' = on$$

WarnStop

$$\Delta Signal$$

$$red = off \land yellow = off \land green = on$$
$$red' = off \land yellow' = on \land green' = on$$

Stop

$$\Delta Signal$$

$$red = off \land yellow = on \land green = on$$
$$red' = on \land yellow' = off \land green' = off$$

The schema for a complete traffic signal is obtained by combining the above schemata in a formula of the schema calculus. A traffic signal consists of a *Signal*, together with any of the (mutually exclusive) operations defined above.

$$TrafficSignal \mathrel{\widehat{=}} Signal \land (WarnGo \lor Go \lor WarnStop \lor Stop).$$

Finally, a schema specifying an initial condition must be conjoined to *TrafficSignal*.

InitRed

$$Signal$$

$$red = on$$
$$yellow = off$$
$$green = off$$

The specification of *Signal* contains the invariant $red = on \Leftrightarrow green = off$, and it must be proved that the invariant holds at any state of the program. The proof is by induction. The invariant is trivially true in *InitRed* and each schema comprising *TrafficSignal* must be checked for the induction step. In fact this is trivial because each postcondition explicitly specifies different values for *red* and *green*.

More complex specifications can be built from this elementary one. A road intersection has independent traffic signals for each direction.

IntersectionSignals

$EastWest, NorthSouth : TrafficSignal$

$EastWest.green \neq NorthSouth.green$

The invariant of this schema specifies a safety property of the intersection, namely, that the green lights of intersecting roads not be on simultaneously. Obviously, to prove this invariant, an appropriate initial condition must be given.

InitIntersection

IntersectionSignals

$EastWest.red = on$
$EastWest.yellow = off$
$EastWest.green = off$
$NorthSouth.red = off$
$NorthSouth.yellow = off$
$NorthSouth.green = on$

Unfortunately, this is not sufficient to ensure that the invariant is maintained, because transitions of *EastWest* and *NorthSouth* can be taken independently. We leave it to the reader to extend the specification so that the invariant is maintained.

For such a simple system, a finite automaton would be more concise and easier to grasp. The true value of Z is when the state transitions involve complex manipulation of values of the variables in the states.

10.2 The Z notation

The Z notation consists of three components:

Z language An extension of the language of the predicate calculus.

Mathematical tool-kit Notations for expressing operations on sets, relations, functions, numbers, sequences and bags.

Schema calculus Rules for combining individual schemata into a full specification.

In this section we survey each of these components.

The Z language

The Z language is basically the language of the predicate calculus with typed variables. Instead of the untyped notation $\forall x p(x)$, you write $\forall x : \mathbb{N} \bullet p(x)$ to indicate that the values of x range only over some type, in this case the natural numbers.

There is a predefined type in Z, the integers, denoted $\mathbb{Z}$; the natural numbers, denoted $\mathbb{N}$, and the positive numbers, denoted $\mathbb{N}_1$, are of type $\mathbb{Z}$ with their values appropriately constrained. The simplest way to define your own type is with a *basic type definition*, which is just a set of identifiers in brackets:

$[NAMES, IDNUMBERS]$.

Basic type declarations simply state that there are types denoted by the identifiers without further specifying the structure of the type. If the set of values of the type is important, you can explicitly denote them using a *free type definition* such as the definition of *STATUS* given above. In the next section, we will show how free type definitions can be used to define recursive types such as lists and trees. Complex types can be defined whose values are sets, sequences, etc. of existing types.

A *schema* consists of declarations of typed variables and formulas called predicates. There is an implicit conjunction between the rows of predicates. Z uses the symbols $\Rightarrow$ and $\Leftrightarrow$ where we used $\rightarrow$ and $\leftrightarrow$, respectively. The notation for predicates is richer than that used in logic, in particular, a quantified formula consists of a schema followed by a formula. The predicates of the schema serve to constrain the quantified values.

Example 10.1 The following Z predicate

$$\forall n : \mathbb{N}, c : COLOR \mid n \bmod 3 = 0 \bullet c = red$$

is read: 'for all n of type natural and c of type *COLOR*, if n is a multiple of 3, then c is red'. In the predicate calculus, this would be written

$$\forall n \forall c((Integer(n) \wedge Color(c) \wedge (n \bmod 3 = 0)) \rightarrow (c = red))$$

under the appropriate interpretations of *Integer* and *Color*. If existential quantifiers are used:

$$\exists n : \mathbb{N}, c : COLOR \mid n \bmod 3 = 0 \bullet c = red,$$

the predicate is read: 'there exists n of type natural and c of type *COLOR* such that n is a multiple of 3 and c is red', which is

$$\exists n \exists c(Integer(n) \wedge Color(c) \wedge (n \bmod 3 = 0) \wedge (c = red))$$

in the predicate calculus. □

The mathematical tool-kit

The mathematical tool-kit of Z provides dozens of symbols; refer to a Z textbook for a complete list. The integer type $\mathbb{Z}$ is predefined in Z, as are the arithmetical operators on the integers. Other Z types are defined as described above, or constructed from defined types.

Sets

The most important construct is the *set*. Given the type

$$COLORS ::= red \mid yellow \mid green,$$

sets can be constructed by listing their elements:

$$\{green, red\},\ \{yellow\},\ \{yellow, red, green\},$$

or by *set comprehension*: $\{ c : COLORS \bullet c \neq yellow \}$. Set comprehension must be used when the set is infinite:

$$even == \{ n : \mathbb{Z} \bullet (\exists m : \mathbb{Z} \bullet n = 2 * m) \}.$$

The symbol == denotes that *even* is defined as a name for this set.

Here are some of the set operators:

$\in$	Element	$\notin$	Not an element
$\subseteq$	Subset	$\subset$	Proper subset
$\mathbb{P}$	Powerset	$\mathbb{F}$	Finite powerset
$\cup$	Union	$\cap$	Intersection
$\setminus$	Difference	#	Cardinality

The finite powerset is the set of all finite subsets of a set. $\mathbb{F} S$ is different from $\mathbb{P} S$ if S is an infinite set like $\mathbb{N}$. $\#S$ returns a natural number which is the cardinality of S.

Z also has symbols for specifying *bags* (also know as *multisets*), which are sets that allow repetitions of elements.

Relations

Z specifications make extensive use of relations on sets. For example, an abstract specification of a database is a relation between elements of type *KEY* and elements of type *RECORD*, denoted $KEY \leftrightarrow RECORD$. Recall that a relation is just a set of ordered pairs, so it is possible that several records have the same key, or that a record has more than one key. The individual elements of a relation can be denoted using the 'maplet' symbol $\mapsto$.

Example 10.2 On my cellular phone, each of the keys 2 through 9 can be used to input one of a set of numbers. Let us define the type

$$LETTER ::= a \mid b \mid \cdots \mid y \mid z.$$

The set $\{2,3,4,5,6,7,8,9\}$ is denoted in Z by the range $2\,..\,9$. Now the relation $2\,..\,9 \leftrightarrow LETTER$ can be defined by:

$$\begin{aligned} PHONEKEYS == \{ & \\ & 2 \mapsto a, 2 \mapsto b, 2 \mapsto c, \quad 3 \mapsto d, 3 \mapsto e, 3 \mapsto f, \\ & 4 \mapsto g, 4 \mapsto h, 4 \mapsto i, \quad 5 \mapsto j, 5 \mapsto k, 5 \mapsto l, \\ & 6 \mapsto m, 6 \mapsto n, 6 \mapsto o, \quad 7 \mapsto p, 7 \mapsto q, 7 \mapsto r, 7 \mapsto s, \\ & 8 \mapsto t, 8 \mapsto u, 8 \mapsto v, \quad 9 \mapsto w, 9 \mapsto x, 9 \mapsto y, 9 \mapsto z \\ \}. & \end{aligned}$$

□

The Z notation contains symbols for many useful operations on relations:

dom	Domain	ran	Range
$\lhd$	Domain restriction	$\rhd$	Range restriction
$\lhd\!\!\!\!-$	Domain anti-restriction	$\rhd\!\!\!\!-$	Range anti-restriction
$\oplus$	Overriding	$(\!\mid \ldots \mid\!)$	Relational image

We demonstrate the meaning of these symbols on the relation of Example 10.2.

Example 10.3 Let *PK* be an abbreviation for *PHONEKEYS*.

$$\begin{aligned} \mathrm{dom}\, PK &= 2\,..\,9 \\ \mathrm{ran}\, PK &= LETTERS \\ PK(\!\mid \{3,5\} \mid\!) &= \{d,e,f,j,k,l\} \\ \{2,3\} \lhd PK &= \{2 \mapsto a, 2 \mapsto b, 2 \mapsto c, 3 \mapsto d, 3 \mapsto e, 3 \mapsto f\} \\ PK \rhd \{l,m,n\} &= \{5 \mapsto l, 6 \mapsto m, 7 \mapsto n\} \\ 3\,..\,8 \lhd\!\!\!\!- PK &= \{2 \mapsto a, 2 \mapsto b, 2 \mapsto c, 9 \mapsto w, 9 \mapsto x, 9 \mapsto y, 9 \mapsto z\} \\ PK \rhd\!\!\!\!- \{d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v\} & \\ &= \{2 \mapsto a, 2 \mapsto b, 2 \mapsto c, 9 \mapsto w, 9 \mapsto x, 9 \mapsto y, 9 \mapsto z\}. \end{aligned}$$

Overriding is used to replace elements of the relation. The following operation:

$$PK \oplus \{9 \mapsto a, 9 \mapsto e, 9 \mapsto i, 9 \mapsto o, 9 \mapsto u\}$$

would cause 9 to be an alternate key for vowels, though {w,x,y,z} would no longer be in the range of *PK*.

□

Functions

Z contains symbols for functions with various properties as shown in the following table. (Review Appendix A.4 for the definitions of the properties.)

$\nrightarrow$	Partial	$\rightarrow$	Total
$\rightarrowtail$	Partial injection	$\rightarrowtail$	Total injection
$\twoheadrightarrow$	Partial surjection	$\twoheadrightarrow$	Total surjection
$\rightarrowtail$	Bijection		

Sequences

A sequence is an ordered set whose length may change during the computation. We describe the use of sequences through an example.

Example 10.4 My cellular phone has a cache of the last eight phone numbers dialled. The specification starts with a basic type definition for phone numbers:

$[PHONENUMBER]$

an axiomatic definition for the constant *cachesize* that specifies its invariant:

$$cachesize : \mathbb{N}$$
$$cachesize = 8$$

and a schema which specifies that the cache is a sequence of length at most *cachesize*:

Cache
$$Numbers : \mathrm{seq}\ PHONENUMBER$$
$$\#Numbers \leq cachesize$$

Initially, the cache is empty.

InitCache
$$Cache$$
$$Numbers = \langle\rangle$$

There are two operations that add a phone number to the cache: *Add* if the cache is not full and *Replace* if the cache is full, in which case, the oldest number is discarded. Each operation takes a phone number *Num*?, where the decoration ? indicates that *Num* is a variable whose value is input to the program. *front* returns the sequence obtained by deleting the last element and $\frown$ concatenates two sequences.

Add
$$\Delta Cache$$
$$Num? : PHONENUMBER$$
$$\#Numbers < cachesize$$
$$Numbers' = \langle Num\rangle \frown Numbers$$

Replace
$$\Delta Cache$$
$$Num? : PHONENUMBER$$
$$\#Numbers = cachesize$$
$$Numbers' = \langle Num\rangle \frown (front\ Numbers)$$

To dial one of the numbers, the up-button must be pressed at least one and at most *cachesize* times. Note that the sequence *Numbers* is actually a function from the positive integers to the elements of the sequence, so function application can be used. $\Xi Cache$ specifies that the values of the variables in *Cache* are *not* modified by this operation, and the decoration ! specifies that *DialCommand*! is a variable whose value is output by the operation.

Dial
$$\Xi Cache$$
$$Up? : \mathbb{N}_1$$
$$DialCommand! : PHONENUMBER$$
$$Up? \leq cachesize$$
$$DialCommand! = Numbers(Up?)$$

The specification for the cache is:

$$PhoneCache \mathrel{\widehat{=}} InitCache \wedge (Add \vee Replace) \wedge Dial.$$

□

Let us now prove that the invariant of the example is maintained, using *laws* of the Z notation for sequences taken from Spivey (1989, Section 4.5).

Theorem 10.5 *$\#Numbers \leq cachesize$ is invariant.*

Proof: $\langle\rangle$ is an abbreviation for the empty set, so $\#\langle\rangle = 0$ and the invariant holds for *InitCache*. For *Add*, $Numbers' = \langle Num\rangle \frown Numbers$, so $\#Numbers' = 1 + \#Numbers$ by the law $\#(s \frown t) = \#s + \#t$. But the precondition is $\#Numbers < cachesize$, so $\#Numbers' \leq cachesize$. For *Replace*, we have $\#front\ s = \#s - 1$, because the definition of *front s* is $(1\,..\,\#s-1) \lhd s$. Thus, $\#Numbers' = 1 + \#Numbers - 1 = \#Numbers$, which equals *cachesize* by the precondition. Finally, since *Dial* does not modify *Cache*, it obviously maintains the invariant. ∎

The schema calculus

Any of the logical operators can be applied to schemata. We have already used the schema calculus to create a specification from the conjunction and disjunction of sev-

eral schema. The meaning of a logical expression on a pair of schemata is: create a new schema whose declarations are the union of the two schemata and whose predicates are formed by applying the operator to the predicates (which are implicitly conjoined) of the two schemata. For details of these and other operations of the schema calculus, see any of the Z textbooks in the references.

10.3 Case study: semantic tableaux

In this section we give a Z specification of the algorithm for the construction of a semantic tableau in the propositional calculus.

A formula is defined by a free type definition. Atomic propositions are of the form $p(0), p(1), \ldots$, and formulas are defined recursively, using identifiers for the Boolean operators so as not to confuse the notation for a formula with the Z notation.

$$
\begin{aligned}
FORMULA ::= & \; p\langle\langle \mathbb{N} \rangle\rangle \\
| \; & neg\langle\langle FORMULA \rangle\rangle \\
| \; & and\langle\langle FORMULA \times FORMULA \rangle\rangle \\
| \; & or\langle\langle FORMULA \times FORMULA \rangle\rangle \\
| \; & imp\langle\langle FORMULA \times FORMULA \rangle\rangle \\
| \; & eqv\langle\langle FORMULA \times FORMULA \rangle\rangle .
\end{aligned}
$$

The α-β identification of formulas is given as a pair of functions; the result of the function application is the pair of subformulas. The function is partial, because not every formula can be decomposed as an α-formula (for example, a β-formula is not an α-formula), and similarly for β-formulas.

$$
\begin{array}{|l}
alpha : FORMULA \pfun FORMULA \times FORMULA \\
\hline
alpha == \{F \pfun (F1, F2) \mid \\
\quad (F = neg(neg(A)) \Rightarrow F1 = A \land F2 = A) \land \\
\quad (F = and(A1, A2) \Rightarrow F1 = A1 \land F2 = A2) \land \\
\quad (F = neg(or(F1, F2)) \Rightarrow F1 = neg(A1) \land F2 = neg(A2)) \land \\
\quad (F = neg(imp(F1, F2)) \Rightarrow F1 = A1 \land F2 = neg(A2)) \land \\
\quad (F = eqv(F1, F2) \Rightarrow F1 = imp(A1, A2) \land F2 = imp(A2, A1)) \\
\}
\end{array}
$$

$$
\begin{array}{|l}
beta : FORMULA \pfun FORMULA \times FORMULA \\
\hline
beta == \{F \pfun (F1, F2) \mid \\
\quad (F = or(A1, A2) \Rightarrow F1 = A1 \land F2 = A2) \land \\
\quad (F = neg(and(F1, F2)) \Rightarrow F1 = neg(A1) \land F2 = neg(A2)) \land \\
\quad (F = imp(F1, F2) \Rightarrow F1 = neg(A1) \land F2 = A2) \land \\
\quad (F = neg(eqv(F1, F2)) \Rightarrow F1 = neg(imp(A1, A2)) \land \\
\qquad\qquad F2 = neg(imp(A2, A1))) \\
\}
\end{array}
$$

Example 10.6 The negation of the first axiom of the propositional calculus is given by the formula $neg(\, imp(\, p(0), imp(p(1), p(0))\,)\,)$. Applying the function *alpha* gives $(\, p(0),\ neg(\, imp(p(1), p(0))\,)\,)$. The formula is not in the domain of *beta*. □

A tableau is a finite set of *NODE*s, each of which is either *nil* or is an ordered 4-tuple containing a finite list of formulas, two children and a mark.

$$MARK ::= unmarked \mid nonleaf \mid closed$$

$$NODE ::= nil \mid n\langle\langle \mathbb{F}\, FORMULA \times NODE \times NODE \times MARK \rangle\rangle$$

Tableau

$$T : \mathbb{F}\, NODE$$

The following schema specifies the application of an α-rule. We give the schema followed by a line-by-line explanation.

AlphaRule

$$
\begin{array}{l}
\Delta T : Tableau \\
\Delta N : NODE \\
\hline
N \in T \\
N.Mark = unmarked \\
\exists F : FORMULA \mid F \in N.Label \bullet \\
\quad F \in \mathrm{dom}\, alpha \land \\
\quad \exists N1 : NODE \bullet \\
\qquad N1 = n((N.Label \setminus F) \cup \{F1, F2\}, nil, nil, unmarked) \land \\
\qquad N'.Left = N1 \land N'.Mark = nonleaf \land \\
\qquad T' = T \cup \{N1\} \\
)
\end{array}
$$

- The operation performed on (and modifies) a tableau T and a node N.
- N is an unmarked element in the tableau.

- There exists a formula F in the label of the N which is in the domain of the function *alpha*.
- There exists (must be created) a node $N1$, which is unmarked, has no children and whose label is the same as the label of N except that F is removed and the two subformulas are added.
- N is marked as not a leaf and $N1$ becomes its left child. $N1$ is added to T.

The β-rule is similar.

BetaRule

$$
\begin{array}{l}
\Delta T : Tableau \\
\Delta N : NODE \\
\hline
N \in T \\
N.Mark = unmarked \\
\exists F : FORMULA \mid F \in N.Label \bullet \\
\quad F \in \mathrm{dom}\, beta \land \\
\quad \exists N1, N2 : NODE \bullet \\
\qquad N1 = n((N.Label \setminus F) \cup \{F1\}, nil, nil, unmarked) \land \\
\qquad N2 = n((N.Label \setminus F) \cup \{F2\}, nil, nil, unmarked) \land \\
\qquad N'.Left = N1 \land N'.Right = N2 \land N'.Mark = nonleaf \land \\
\qquad T' = T \cup \{N1, N2\} \\
)
\end{array}
$$

To specify the termination of a path, we first define an abbreviation *clashing* for sets of clashing formulas:

$$
\begin{array}{l}
clashing : \mathbb{F}\, FORMULA \\
\hline
\{\, Fset : \mathbb{F}\, FORMULA \bullet \\
\quad \exists F1, F2 : FORMULA \mid F1, F2 \in Fset \bullet \\
\qquad F1 = neg(F2) \lor F2 = neg(F1) \\
\}
\end{array}
$$

Now we can specify a closed node as one with a clashing pair of formulas.

ClosedNode

$$
\begin{array}{l}
\Delta T : Tableau \\
\Delta N : NODE \\
\hline
N \in T \\
N.Mark = unmarked \\
N.Label \in clashing \\
N'.Mark = closed
\end{array}
$$

The type *REPORT* is used to report the outcome of the tableau construction:

$$REPORT ::= satisfiable \mid unsatisfiable.$$

A tableau for a formula is open (and satisfiable) iff it has a node whose label is a non-clashing set of literals. A literal is a formula to which neither *alpha* nor *beta* apply.

OpenTableau

$$
\begin{array}{l}
\Xi T : Tableau \\
R! : Report \\
\hline
\exists N : NODE \mid N \in T \bullet \\
\quad N.Mark = unmarked \land N.Label \notin clashing \land \\
\quad \forall F : FORMULA \mid F \in N.Label \bullet \\
\qquad \mathrm{dom}\, alpha(F) = \emptyset \land \mathrm{dom}\, beta(F) = \emptyset \\
R! = satisfiable
\end{array}
$$

A tableau for a formula is closed (and unsatisfiable) iff all leaves are closed.

ClosedTableau

$$
\begin{array}{l}
\Xi T : Tableau \\
R! : Report \\
\hline
\forall N : NODE \mid N \in T \bullet N.Mark = nonleaf \lor N.Mark = closed \\
R! := unsatisfiable
\end{array}
$$

The initial tableau is created from an input formula.

InitTableau

$$
\begin{array}{l}
T : Tableau \\
F? : FORMULA \\
\hline
T = \{n(\{F\}, nil, nil, unmarked)\}
\end{array}
$$

The complete specification is:

$$
\begin{array}{l}
TableauAlgorithm \mathrel{\widehat{=}} InitTableau \land \\
\quad (AlphaRule \lor BetaRule \lor \\
\quad OpenTableau \lor ClosedNode \lor ClosedTableau).
\end{array}
$$

The Z notation has been used to specify large hardware and software systems. It is sufficiently flexible that it can be used on different types of systems, yet it is a formal system that can be used for deduction of properties of a specification.

10.4 Exercises

1. Prove $Leaves = \mathrm{dom}\, Label$ for the α- and β-rules.

2. A sequence can also be described as a function from $1..n$ to the set of elements. If s is a sequence then si is the function application returning the i'th element of the sequence. Show that the sequence operations can be defined as follows:

$$
\begin{aligned}
&heads = s1,\\
&lasts = s(\#s),\\
&tails = \{n : \mathbb{Z} \mid n \in 2..\#s \bullet n - 1 \mapsto sn\},\\
&s \frown t = s \cup \{n : \mathbb{Z} \mid n \in 2..\#t \bullet (n + \#s) \mapsto tn\},\\
&fronts = (1..\#s - 1) \lhd s.
\end{aligned}
$$

11 Temporal Logic: Formulas, Models, Tableaux

11.1 Introduction

The predicate calculus is adequate both for theoretical mathematics and for practical applications. Nevertheless, it is often useful to define systems of logic that are more convenient for specific tasks. This chapter studies a system for reasoning about time called *temporal logic*. Temporal logic is applicable in computer science, because the behavior of both hardware and software components is a function of time, unlike mathematical expressions such as $1 + 1 = 2$ whose behavior (truth value) is static.

Example 11.1 Here are some examples of specifications of hardware and software components, where words denoting temporal concepts have been italicized.

Flip-flop *After* the reset-line is asserted, the zero-line is asserted. The output lines *maintain* their values, *until* the set-line is asserted; *then* they are complemented.

File server If a request is made to print a file, *eventually* the file will be printed.

Operating system The system will *always* run. The system will *never* crash.

□

These properties can easily be expressed in predicate logic, for example, the file server property can be specified as:

$$\forall f\ \forall t_1 (\ RequestPrint(f, t_1) \rightarrow \exists t_2\ ((t_2 \geq t_1) \wedge PrintedAt(f, t_2))\ ).$$

In temporal logic, new operators are introduced that enable the time variables and their relationships such as $t_2 \geq t_1$ to be implicitly indicated.

Definition 11.2 There are two unary prefix temporal operators: *always*, denoted $\Box$, *eventually*, denoted $\Diamond$.

□

Informally, $\Box$ is the universal operator 'for *any* time t in the future', while $\Diamond$ is the existential operator 'for *some* time t in the future'. The operators compose, in the sense that $\Box\Diamond p$, means $\forall t_1 \exists t_2((t_2 \geq t_1) \wedge p)$, and not just $\forall t_1 \exists t_2 p$. In temporal logic, the above specification becomes:

$$\forall f \, \Box(\, \mathit{RequestPrint}(f) \rightarrow \Diamond \mathit{PrintedAt}(f)\,),$$

which is much more concise. Reasoning with a temporal formula is much easier than with its translation into the predicate calculus, because the relationships among the times are implicit.

In the predicate calculus, all theories use the same five logical axioms and two rules of inference, which have proved to be sound and complete for the interpretations used in mathematics. Alternative axiomatizations that change the set of deducible formulas are not important (with the exception of intuitionistic logic), but in temporal logic, different choices of operators and axioms lead to different models of time, several of which are useful. In these chapters, we limit our discussion to propositional temporal logics with a specific selection of operators, and the tableau construction and axiomatization apply only to one specific model of time. In Section 12.3 we survey alternatives to this logic that have been developed and applied.

There are closely related systems of logics called *modal logics* that, along with temporal logics, date back to Greek philosophy. Modal logics express the distinction between what is *necessarily* true and what is *possibly* true. For example, '7 is a prime number' is necessarily true because, under the definitions of the concepts in the statement, the statement is true always and everywhere. In contrast, the statement 'This country is ruled by a king' is only possibly true, because its truth changes from place to place and from time to time. These vague concepts proved difficult to formalize and an acceptable formal semantics for modal logic was first given by S. Kripke in 1965. The connection between modal logic and temporal logic is immediate by defining 'always' to be 'necessarily' and 'eventually' to be 'possibly'.

11.2 Syntax and semantics

Propositional temporal logic (PTL) extends the propositional calculus with two unary temporal operators $\Box$ and $\Diamond$. (In the next section, we will add a third operator.) Their precedence is the same as the other unary operator, negation. The following are syntactically correct formulas in PTL:

$$p \wedge q, \quad \Box p, \quad \Diamond(p \wedge q) \rightarrow \Diamond p, \quad \Box\Box p \leftrightarrow \Box p, \quad \Diamond\Box p \leftrightarrow \Box\Diamond p, \quad \neg\Diamond p \wedge \Box\neg q.$$

The semantics of PTL gives an interpretation, not only for the propositional letters, but also for the underlying representation of time. This is done by defining a set of states, each of which contains an interpretation for the propositional letters appearing

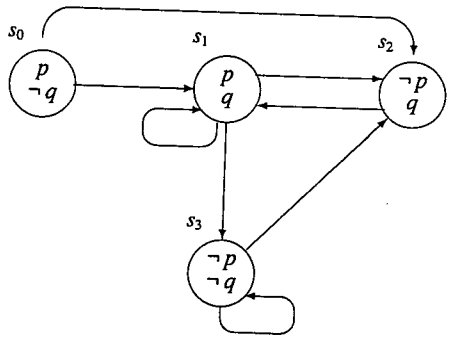


Figure 11.1 PTL interpretation

in the formula. Passage of time is modeled as a transition from one state to another. For now we allow arbitrary transitions between states.

A PTL interpretation can be displayed as a *state transition diagram* (Figure 11.1), where the circles denote the states, the arrows denote possible transitions between states and each state is labeled with a set of literals, such that p and $\neg p$ do not appear in the same label. Each state is a propositional interpretation, which is defined by assigning *true* to the literals labeling the state.

Example 11.3 We give an informal demonstration how the truth value of the temporal formula $A = \Box p \vee \Box q$ is determined for each state s in Figure 11.1.

- A is true in s_0. The states accessible from s_0 are s_1 and s_2. q is true in both states, so $\Box q$ is true in s_0, as is $\Box p \vee \Box q$.
- A is false in s_1. s_3 is accessible from s_1 and neither p nor q is true in s_3. Therefore, neither $\Box p$ nor $\Box q$ can be true in s_1, so $\Box p \vee \Box q$ is false in s_1.
- A is true in s_2. The only state accessible from s_2 is s_1, and both p and q are true in s_1. It is immediate that $\Box p \vee \Box q$ is also true.
- A is false in s_3. s_3 is accessible from itself, but neither p nor q is true in s_3.

□

Definition 11.4 An *interpretation* $\mathcal{I}$ for a formula A in PTL is a pair (S, ρ), where $S = \{s_1, \ldots, s_n\}$ is a set of states each of which is an assignment of truth values to the atomic propositions in A, and ρ is a binary relation on states.

Notation: $s_i(p) = T$ means that the atom p is assigned true by the state s_i. $(s_1, s_2) \in \rho$ will usually be written $s_2 \in \rho(s_1)$.

$v_{\mathcal{I},s}(A)$, the *value of A in s*, is defined by induction.

- If A is $p \in \mathcal{P}$ and $s = s_i$, then $v_{\mathcal{I},s}(A) = s_i(p)$.
- If A is $\neg A'$ then $v_{\mathcal{I},s}(A) = T$ iff $v_{\mathcal{I},s}(A') = F$.
- If A is $A' \vee A''$ then $v_{\mathcal{I},s}(A) = T$ iff $v_{\mathcal{I},s}(A') = T$ or $v_{\mathcal{I},s}(A'') = T$, and similarly for the other Boolean operators.
- If A is $\Box A'$ then $v_{\mathcal{I},s}(A) = T$ iff $v_{\mathcal{I},s'}(A') = T$ for *all* states $s' \in \rho(s)$.
- If A is $\Diamond A'$ then $v_{\mathcal{I},s}(A) = T$ iff $v_{\mathcal{I},s'}(A') = T$ for *some* state $s' \in \rho(s)$.

Notation: We denote $v_{\mathcal{I},s}(A)$ by $v_s(A)$ when $\mathcal{I}$ is clear from the context. ▯

The definition of semantic properties is more complex than in the predicate calculus, because we have both interpretations and states in the interpretation.

Definition 11.5 A formula A in PTL is *satisfiable* iff there is an interpretation $\mathcal{I} = (S, \rho)$ for A and a state $s \in S$ such that $v_s(A) = T$ (notation: $\mathcal{I}, s \models A$, or $s \models A$ if $\mathcal{I}$ is understood). If $\mathcal{I}, s \models A$ for some $s \in S$, then $\mathcal{I}$ is called a *model* for A (notation: $\mathcal{I} \models A$). A formula A in PTL is *valid* (notation: $\models A$) iff for all interpretations $\mathcal{I}$ for A and for all states $s \in S$, $\mathcal{I}, s \models A$. ▯

Example 11.6 The analysis we did for the formula $A = \Box p \vee \Box q$ and the interpretation $\mathcal{I}$ in Figure 11.1 can be repeated using the formal definition of interpretation. $\rho(s_0) = \{s_1, s_2\}$, and since $s_1 \models q$ and $s_2 \models q$, we conclude that $\mathcal{I}, s_0 \models \Box q$. Then $\mathcal{I}, s_0 \models \Box p \vee \Box q$ by the interpretation of disjunction. $\mathcal{I}$ is a model for A, $\mathcal{I} \models A$, but A is not valid because $\mathcal{I}, s_1 \not\models A$. ▯

Note that any valid formula of the propositional calculus is a valid formula of PTL. Furthermore, the formula $\Box p \rightarrow (\Box q \rightarrow \Box p)$ is also valid. While not a formula of the propositional calculus, it is a *substitution instance* of a propositional formula obtained by substituting PTL formulas uniformly for propositional letters.

Theorem 11.7 *Every substitution instance of a valid propositional formula is valid.*

Proof: Exercise. ▮

There are other formulas of PTL that are valid because of properties of temporal logic and not just as instances of propositional validities. We will prove the validity of two formulas directly from the semantic definition. The first establishes a duality between $\Box$ and $\Diamond$; the second is the distribution of $\Box$ over $\rightarrow$, similar to the distribution of $\forall$ over $\rightarrow$.

Theorem 11.8 (Duality) $\models \Box p \leftrightarrow \neg \Diamond \neg p$.

Proof: Let $\mathcal{I} = (S, \rho)$ be an arbitrary interpretation for the formula and let $s \in S$ be an arbitrary state. Assume that $s \models \Box p$, and suppose that $s \models \Diamond \neg p$. Then there exists a state $s' \in \rho(s)$ such that $s' \models \neg p$. Since $s \models \Box p$, for all states $t \in \rho(s)$, $t \models p$, in particular, $s' \models p$, contradicting $s' \models \neg p$, so $s \models \neg \Diamond \neg p$. Since $\mathcal{I}$ and s were arbitrary we have proved that $\models \Box p \rightarrow \neg \Diamond \neg p$. We leave the converse as an exercise. ▮

Theorem 11.9 $\models \Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$.

Proof: Suppose that the formula could be falsified. Then there would be an interpretation $\mathcal{I} = (S, \rho)$ and a state $s \in S$ such that $s \models \Box(p \rightarrow q)$ and $s \models \Box p$, but $s \models \neg \Box q$. By Theorem 11.8, the last assumption is equivalent to $s \models \Diamond \neg q$, so there exists a state $s' \in \rho(s)$ such that $s' \models \neg q$. But by the first two assumptions, $s' \models p \rightarrow q$ and $s' \models p$, so by the semantics of $\rightarrow$, $s' \models q$ which is a contradiction. ▮

11.3 Models of time

Different temporal logics can be obtained by placing different restrictions on the transition relation ρ. For each restriction, we will give a formula that characterizes interpretations with that restriction. The correspondences between the formulas and the restrictions are intuitively simple; we defer the precise formulation of the theorems and their proofs to a separate subsection which can be skipped.

Consider the formula $\Diamond$*running*. Obviously, if a program is running *now*, then there is an accessible state (namely, *now*) in which the program is running. Thus it is reasonable to require that for all s, $s \in \rho(s)$, that is, that the relation be *reflexive*.

Theorem 11.10 *An interpretation with a reflexive relation is characterized by the formula* $\Box A \rightarrow A$ *(or by the formula* $A \rightarrow \Diamond A$*).*

When the relation is intended to denote the passage of time or the execution of a computer program, it is natural to require that ρ be *transitive*:

$$s_2 \in \rho(s_1) \wedge s_3 \in \rho(s_2) \rightarrow s_3 \in \rho(s_1).$$

If the relation were not transitive, s_3 would not be in the 'future' of s_1.

Example 11.11 In Figure 11.1, ρ is not transitive since $s_1 \in \rho(s_2)$ and $s_3 \in \rho(s_1)$ but $s_3 \notin \rho(s_2)$. This leads to the anomalous situation where $s_2 \models \Box p$ but $s_2 \not\models \Box\Box p$, ▯

Theorem 11.12 *An interpretation with a transitive relation is characterized by the formula* $\Box A \rightarrow \Box\Box A$ *(or by the formula* $\Diamond\Diamond A \rightarrow \Diamond A$*).*

If the relation is also reflexive, then $\models \Box A \leftrightarrow \Box\Box A$ and $\models \Diamond A \leftrightarrow \Diamond\Diamond A$.

Definition 11.13 If ρ is defined so that every state has exactly one immediate successor state (other than itself), the logic is called *linear-time* temporal logic, otherwise, it is called *branching-time* temporal logic. □

It might appear that it is necessary to use branching-time temporal logic to reason about programs since many statements will have several possible successors; furthermore, in a concurrent system, any statement is a possible successor of a statement in another process. However, linear-time temporal logic is also used because we are interested in properties that hold in *every* possible computation, where each possible computation is a linear sequence of states.

The execution of a program is usually considered as a sequence of *discrete* steps, where each step consists of the execution of an instruction of the CPU, or a sequence of transitions of a circuit upon clock pulses. Thus it makes sense to express the concept of the 'next' instant in time. The definition of an interpretation must be changed in order to distinguish between a 'next' state after a single step and a 'future' state attained after one or more steps.

Definition 11.14 The unary prefix temporal operator *next* is denoted $\bigcirc$.

An interpretation for A is a pair (S, τ), where S is a set of states each of which is an assignment of truth values to the atomic propositions in A, and τ a binary relation on states. Let ρ be τ^*, the reflexive transitive closure of τ (see Section A.4). The definition of $v_{\mathcal{I},s}$ is as in Definition 11.4, extended for $\bigcirc A'$ as follows:

- If A is $\bigcirc A'$ then $v_{\mathcal{I},s}(A) = T$ iff $v_{\mathcal{I},s'}(A') = T$ for some $s' \in \tau(s)$. If τ linear, then there is only one such state and we can write $s' = \tau(s)$.

□

The following theorem follows directly from the definitions.

Theorem 11.15 $\models \Box p \rightarrow \bigcirc p$ *and* $\models \bigcirc p \rightarrow \Diamond p$.

In linear time, the next operator is its own dual.

Theorem 11.16 *A interpretation with a linear relation is characterized by the formula* $\bigcirc A \leftrightarrow \neg \bigcirc \neg A$.

The next operator will be used in the inductive construction of a semantic tableau.

Theorem 11.17 *Let $\mathcal{I}$ be a linear, reflexive, transitive interpretation. Then* $\mathcal{I} \models \Box p \leftrightarrow p \wedge \bigcirc\Box p$ *and* $\mathcal{I} \models \Diamond p \leftrightarrow p \vee \bigcirc\Diamond p$.

Proof: Let $\mathcal{I}$ be an arbitrary linear, reflexive, transitive interpretation, let s be an arbitrary state in S and assume that $s \models \Box p$. Since $\mathcal{I}$ is reflexive, $s \models p$. If $s \not\models \bigcirc\Box p$, then for $s' = \tau(s)$, $s' \not\models \Box p$. There exists $s'' \in \rho(s')$ such that $s'' \not\models p$. But $s' \in \rho(s)$ follows from $s' = \tau(s)$, so by transitivity, $s'' \in \rho(s)$, contradicting $s \models \Box p$.

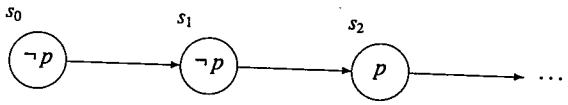
Conversely, assume that $s \models p \wedge \bigcirc\Box p$ and let $s' \in \rho(s)$ be arbitrary. Since ρ is the reflexive transitive closure of τ, there exists a sequence of states

$$s = s_0,\ s_1 = \tau(s_0),\ s_2 = \tau(s_1),\ \ldots,\ s' = s_k = \tau(s_{k-1}).$$

We prove by induction on k that $s' \models p$, so $s \models \Box p$ since s' was arbitrary. For $k = 0$, the result follows from the assumption $s \models p$. By the semantics of the next operator, $s_0 \models \bigcirc\Box p$ implies $s_1 \models \Box p$ and $s_1 \models p \wedge \bigcirc\Box p$, so by the inductive hypothesis $s' = s_k \models p$.

$\mathcal{I} \models \Diamond p \leftrightarrow p \vee \bigcirc\Diamond p$ follows by duality. ∎

We limit our discussion to interpretations with transition relations that are discrete, reflexive, transitive and linear. To simplify the display of these relations, we will just draw the immediate successor τ of a state, and infer the reflexive transitive closure ρ. In the following diagram, $\{s_0, s_2\} \subseteq \rho(s_0)$, even though no explicit arrows are drawn for these transitions.



The following definition will be convenient in the construction of semantic tableaux.

Definition 11.18 A formula of the form $\bigcirc A$ or $\neg \bigcirc A$ is called a *next* formula. A formula of the form $\Diamond A$ or $\neg \Box A$ is called a *future* formula. □

*Proofs of the correspondences**

The following definition enables us to talk about the structure (states and relations) of an entire class of interpretations by abstracting away the propositional interpretations.

Definition 11.19 A *frame* $\mathcal{F}$ is a pair $(\mathcal{W}, \rho)$, where $\mathcal{W}$ is a set of states and ρ is a binary relation on states. An interpretation $\mathcal{I} = (S, \rho)$ is *based on* a frame $\mathcal{F} = (\mathcal{W}, \rho)$ iff there is a one-to-one mapping from S onto $\mathcal{W}$.

A PTL formula A *characterizes* a class of frames iff for every $\mathcal{F}_i$ in the class and for every $\mathcal{I}$ based on $\mathcal{F}_i$, $\mathcal{I} \models A$, and conversely, if $\mathcal{I} \models A$, then $\mathcal{I}$ is based on some $\mathcal{F}_i$ in the class. □

A frame is obtained from an interpretation by ignoring the truth assignments in the states, or conversely, a interpretation is obtained from a frame by associating an assignment with each state.

Theorems 11.10, 11.12 and 11.16 are more precisely stated as follows: the formulas $\Box A \to A$, $\Box A \to \Box\Box A$ and $\bigcirc A \leftrightarrow \neg \bigcirc \neg A$ characterize the sets of reflexive, transitive, and linear *frames*, respectively. In one direction, the proofs are easy: if the restriction holds, the formula is true. In the other direction, we show that if the restriction does not hold, then it is possible to *construct* a specific interpretation in which the formula is not true.

Proof of 11.10: Let $\mathcal{F}_i$ be a reflexive frame, let $\mathcal{I}$ be an arbitrary interpretation based on $\mathcal{F}_i$, and suppose that $\mathcal{I} \not\models \Box A \to A$. Then there is a state $s \in S$ such that $s \models \Box A$ and $s \not\models A$. For any state $s' \in \rho(s)$, $s' \models A$. By reflexivity, $s \in \rho(s)$, so $s \models A$, a contradiction.

Conversely, suppose that $\mathcal{F}_i$ is not reflexive, and let $s \in S$ be a state such that $s \notin \rho(s)$. Let $\mathcal{I}$ be an interpretation based on $\mathcal{F}_i$ such that $v_s(p) = F$ and $v_{s'}(p) = T$ for all $s' \in \rho(s)$. These assignments are well-defined since $s \notin \rho(s)$. Then $s \not\models \Box p \to p$. If $\rho(s)$ is empty, assign F to $v_s(p)$ and $\Box p$ is vacuously true in s, so again $s \not\models \Box p \to p$. ∎

Proof of 11.12: Let $\mathcal{F}_i$ be a transitive frame, let $\mathcal{I}$ be an arbitrary interpretation based on $\mathcal{F}_i$, and suppose that $\mathcal{I} \not\models \Box A \to \Box\Box A$. There is an $s \in S$ such that $s \models \Box A$ and $s \not\models \Box\Box A$. Let $s' \in \rho(s)$ be such that $s' \not\models \Box A$, and let $s'' \in \rho(s')$ be such that $s'' \not\models A$. But $s \models \Box A$, and by transitivity, $s'' \in \rho(s)$, so $s'' \models A$, a contradiction.

Conversely, suppose that $\mathcal{F}_i$ is not transitive, and let $s, s', s'' \in S$ be states such that $s' \in \rho(s)$, $s'' \in \rho(s')$, but $s'' \notin \rho(s)$. Let $\mathcal{I}$ be an interpretation based on $\mathcal{F}_i$ which assigns T to p in all states in $\rho(s)$ and F to p in s'', which is well-defined since $s'' \notin \rho(s)$. Then $s \models \Box p$, but $s \not\models \Box\Box p$. If there are only two states, s'' need not be distinct from s. A one state frame is necessarily transitive, possibly vacuously if the relation is empty. ∎

We leave the proof of Theorem 11.16 as an exercise.

11.4 Semantic tableaux

The method of semantic tableaux can be used to obtain a decision procedure for satisfiability in PTL. We add the following rules to the α- and β-rules for the propositional calculus.

α	α_1	α_2
$\Box A$	A	$\bigcirc\Box A$
$\neg\Diamond A$	$\neg A$	$\neg\bigcirc\Diamond A$

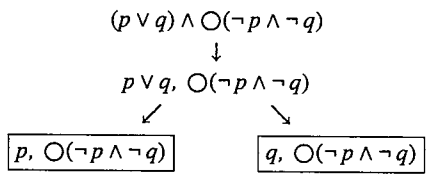
β	β_1	β_2
$\Diamond A$	A	$\bigcirc\Diamond A$
$\neg\Box A$	$\neg A$	$\neg\bigcirc\Box A$

X	X_1
$\bigcirc A$	A
$\neg\bigcirc A$	$\neg A$

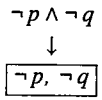
The X-rule is obvious from the definition of the interpretation of $\bigcirc$. The α- and β-rules are based on Theorem 11.17:

- If $\Box A$ is true in a state s, then A is true in s *and* A must continue to be true in *all* states accessible from the *next* state s'.
- If $\Diamond A$ is true in a state s, then either A is true in s *or* A will eventually become true in *some* state accessible from the *next* state s'.

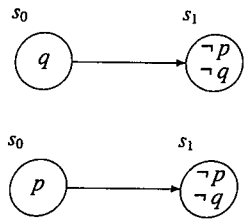
While the α- and β-rules are just the rules of the propositional calculus applied to temporal formulas, the X-rule has a different status. Consider the tableau obtained for the formula $A = (p \vee q) \wedge \bigcirc(\neg p \wedge \neg q)$ after applying the α- and β-rules:



In the initial state s_0 in any model for A, $s_0 \models p$ or $s_0 \models q$. The box around this tableau node indicates that the node defines a state by listing the literals that must be assigned T. But to satisfy the formula A in PTL, something must also be true about the next state s_1. It is the X-rule which causes new states to be created in the tableau. On both branches, the new node has the formula $\neg p \wedge \neg q$; an application of the propositional α-rule gives $\{\neg p, \neg q\}$ as the label of the next node and hence the new state s_1.

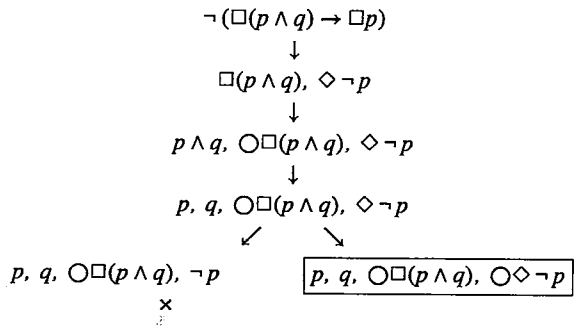


Note that the literals in s_0 are *not* carried over in the application of the X-rule. From the tableau construction, we see that any model for A must contain one of the following structures:

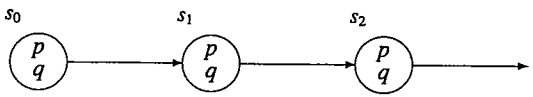


As in the propositional calculus, these are not interpretations, because we have not specified the value of the second literal in either of the possible states s_0. However, these structures are *Hintikka structures* which can be extended to interpretations by specifying the values of all atoms in each state.

Consider now the formula $A = \neg(\Box(p \wedge q) \rightarrow \Box p)$ which is the negation of a valid formula. Here is a semantic tableau, where we have implicitly changed $\neg\Box$ to $\Diamond\neg$ for clarity.



The left-hand branch closes, and the right-hand leaf defines a state s_0 in which p and q must be true. When X-rules are applied to this node, a node is created labeled by the set of formulas $\Box(p \wedge q)$, $\Diamond\neg p$. But this is the same set of formulas that labels the second node in the tableau. It is clear that the continuation of the construction will create an infinite structure:

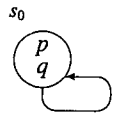


Something is wrong since A is unsatisfiable and its tableau should close! This structure is a Hintikka structure in propositional terms (it does not contain clashing literals and for every α-, β- and X-formula the Hintikka conditions hold), but it cannot be extended to model for A since the future subformula $\Diamond\neg p$ is not *fulfilled*, that is, the structure promises to eventually produce a state in which $\neg p$ is true but defers forever the creation of such a state. We will have to find a condition that ensures that future formulas can be fulfilled.

Finite presentation of an interpretation

Computations can be non-terminating; in particular computations of *reactive systems* like operating systems and control systems are supposed to be non-terminating. PTL can express properties of non-terminating computations: while formulas like $\Box\bigcirc p$ can only be satisfied in infinite models, there are only a finite number of distinct states in a PTL interpretation. To see this, note that all formulas appearing in a semantic tableau are either subformulas of the formula at the root, or $\neg A$, $\bigcirc A$, or $\neg\bigcirc A$, for A a subformula.

An interpretation for a PTL formula can be *finitely presented* in a directed graph by reusing existing states instead of creating new ones. For example, the infinite structure above can be finitely presented as follows:



However, as the example shows, not every finitely presented structure obtained from the tableau construction is a model.

Construction of semantic tableaux in PTL

We now formally describe the construction of semantic tableaux, the creation of structures from the tableaux and an algorithm to check if the structure contains a model.

Algorithm 11.20 (Construction of a semantic tableau)
Input: A PTL formula A.
Output: A semantic tableau $\mathcal{T}$ for A.

Each node of $\mathcal{T}$ is labeled with a set of formulas. Initially, $\mathcal{T}$ consists of a single node, the root, labeled with the singleton set $\{A\}$. The tableau is built inductively as follows. *Choose* an unmarked leaf l on $\mathcal{T}$. l is labeled with a set of formulas $U(l)$.

- If $U(l)$ is a set of literals, check if there is a complementary pair of literals $\{p, \neg p\} \in U(l)$. If so, mark the leaf *closed* $\times$. If not, mark the leaf *open* $\odot$.
- If $U(l)$ is not a set of literals, *choose* $A \in U(l)$ which is *not* a next formula.
  - If the formula is an α-formula, create a new node l' as a child of l and label l' with
    $$U(l') = (U(l) - \{A\}) \cup \{\alpha_1, \alpha_2\}.$$
    (In the case that A is $\neg\neg A_1$, there is no α_2.)
  - If the formula is a β-formula, create two new nodes l' and l'' as children of l. Label l' with
    $$U(l') = (U(l) - \{A\}) \cup \{\beta_1\},$$
    and label l'' with
    $$U(l'') = (U(l) - \{A\}) \cup \{\beta_2\}.$$
- If $U(l)$ consists only of literals and next formulas, let
  $$\{\bigcirc A_1, \ldots, \bigcirc A_m, \neg\bigcirc A_{m+1}, \ldots, \neg\bigcirc A_n\}$$

be the set of next formulas in $U(l)$. Create a new node l' as a child of l and label l' with

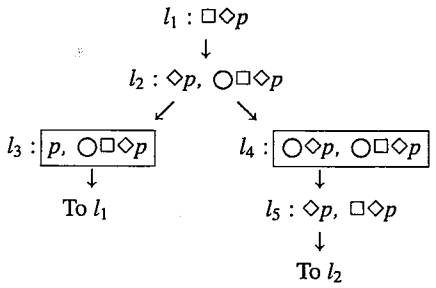
$$U(l') = \{A_1, \ldots, A_m, \neg A_{m+1}, \ldots, \neg A_n\}.$$

If $U(l') = U(l'')$ for l'' an ancestor of l, do not create l'; instead connect l to l''.

The construction terminates when every leaf is marked × or ⊙. □

Definition 11.21 A tableau whose construction has terminated is called a *completed tableau*. A completed tableau is *closed* if all leaves are marked closed. Otherwise, it is *open*. □

Example 11.22 Here is a completed open semantic tableau with *no* leaves.



□

Theorem 11.23 *The construction of a semantic tableau terminates.*

Proof: Exercise. ∎

The next step is to construct a structure from a completed tableau and to prove that the conditions for a Hintikka structure hold.

Definition 11.24 A *structure* is a triple $\mathcal{H} = (S, \mathcal{U}, \tau)$ where $S = \{s_1, \ldots, s_n\}$ is a set of states, $\mathcal{U} = \{U_1, \ldots, U_n\}$ is a set of sets of formulas, one set associated with each state, and τ is a binary relation on states. $\rho = \tau^*$ is the reflexive transitive closure of τ. Notation: $\mathcal{H} = (S, \mathcal{U}, \rho)$ may be used to emphasize the role of ρ. □

To construct a structure from a completed tableau, take the X-nodes as states, and define $s' \in \tau(s)$ if there is a path in the tableau from s to s' that does not pass through another state. The set of formulas associated with s' is the union of the labels of the nodes on such a path.

Definition 11.25 A *state path* is a path $(l_1, l_2, \ldots, l_{k-1}, l_k)$ in the tableau, such that l_1 is the root or an X-node, l_k is an X-node, and none of $\{l_2, \ldots, l_{k-1}\}$ is an X-node. It is possible that $l_1 = l_k$. □

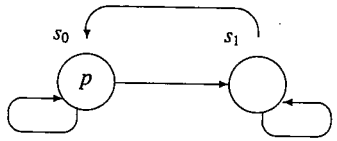
Definition 11.26 Let $\mathcal{T}$ be an open semantic tableau. The structure $\mathcal{H}$ constructed from $\mathcal{T}$ is:

- S is the set of X-nodes.
- Let $s_i = l_k$ be a state and let $l = \{l_1, l_2, \ldots, l_k = s_i\}$ be a state path terminating in l_k. Then $U_i^l = U(l_2) \cup \cdots \cup U(l_k)$. If l_1 is the root, add $U(l_1)$ to the union. $U_i = \cup_l U_i^l$.
- $s' \in \tau(s)$ iff $\{s = l_1, l_2, \ldots, l_{k-1}, l_k = s'\}$ is a state path.

□

It is possible to obtain several disconnected structures when creating a structure for the tableau for a formula such as $\Diamond p \vee \Diamond q$, but this is no problem as the formula can be satisfiable if and only if at least one of the structures leads to a model.

Example 11.27 Here is the structure constructed from the semantic tableau in Example 11.22, where s_0 is l_3 and s_1 is l_4. In the diagram, we have explicitly labeled each state s_i only with the literals in U_i.



□

Example 11.28 Let $A = \Box(\Diamond(p \wedge q) \wedge \Diamond(\neg p \wedge q) \wedge \Diamond(p \wedge \neg q))$. The construction of the tableau for A is left as an exercise. The structure obtained from the tableau is shown in Figure 11.2. □

Definition 11.29 Let $\mathcal{H} = (S, \mathcal{U}, \tau)$ be a structure and A be a formula. $\mathcal{H}$ is a *Hintikka structure* iff for all U_i:

1. For all propositional symbols p, either $p \notin U_i$ or $\neg p \notin U_i$.
2. If $A \in U_i$ is an α-formula, then $\alpha_1 \in U_i$ *and* $\alpha_2 \in U_i$.
3. If $A \in U_i$ is an β-formula, then $\beta_1 \in U_i$ *or* $\beta_2 \in U_i$.
4. If $A = \bigcirc A' \in U_i$ is an X-formula, then for all $s_j \in \tau(s_i)$, $A' \in U_j$, and similarly for $\neg \bigcirc A'$.

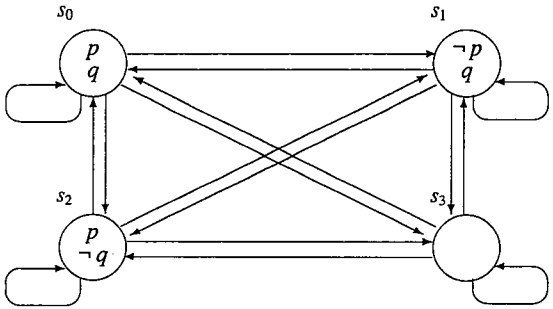


Figure 11.2 Structure for $\Box(\Diamond(p \wedge q) \wedge \Diamond(\neg p \wedge q) \wedge \Diamond(p \wedge \neg q))$

$\mathcal{H}$ is a Hintikka structure *for* A iff $\mathcal{H}$ is a Hintikka structure, $A \in s_0$ and the only propositional symbols appearing in $\mathcal{U}$ are those in A. □

Theorem 11.30 *The structure created in Definition 11.26 is a Hintikka structure.*

Proof: The structure is created from an open tableau, so condition (1) holds. α- and β-rules are applied *before* the X-rule, so the state path from one state (or the root) to another contains all the formulas required by conditions (2) and (3). When the X-rule is applied, for any $\bigcirc A$, A will appear in the label of the next node (and similarly for $\neg \bigcirc A$), and hence in every state at the end of a state path from the node. ▮

Definition 11.31 Let $\mathcal{H}$ be a Hintikka structure. $\mathcal{H}$ is a *linear* Hintikka structure iff τ is a function, that is, if for each s_i there is at most one $s_j \in \tau(s_i)$. □

Lemma 11.32 *Any unbounded path through a Hintikka structure is a linear Hintikka structure.*

Proof: Clearly, the path is linear and conditions (1–3) hold because they already held in the (non-linear) Hintikka structure. Suppose that $\bigcirc A$ occurs in some U_i. By construction, A occurs in all successor states, in particular, in the one chosen in the construction of the unbounded path. ▮

Definition 11.33 Let $\mathcal{H}$ be a Hintikka structure. $\mathcal{H}$ is a *fulfilling* Hintikka structure iff for all s_i and for all future formulas $A = \Diamond A'$: if $A \in U_i$, then for some $s_j \in \rho(s_i)$, $A' \in U_j$. The state s_j is said to *fulfil* A. □

A similar requirement must imposed on future formulas of the form $A = \neg \Box A'$, which must be fulfilled by states containing $\neg A'$. In this section, we will discuss only $\Diamond A'$ to simplify the presentation.

Theorem 11.34 (Hintikka's Lemma for PTL) *Let $\mathcal{H}$ be a linear fulfilling Hintikka structure for A. Then A is satisfiable.*

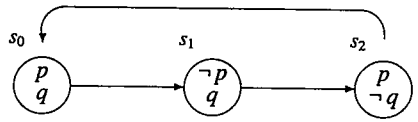
Proof: Extend $\mathcal{H}$ to an interpretation by defining for all propositional letters p in A:

$$\begin{aligned} \nu_i(p) &= T \quad \text{if } p \in U_i \text{ or } \neg p \notin U_i \\ \nu_i(p) &= F \quad \text{if } \neg p \in U_i. \end{aligned}$$

For propositional formulas the induction is the same as that used to prove Hintikka's Lemma for the propositional calculus. By condition 4 of the Hintikka structure, it follows that next formulas are satisfied, and for future formulas, the result follows by the requirement that $\mathcal{H}$ be fulfilling.

Let $A = \Box A' \in U_i$ and let $s_j \in \rho(s_i)$ be arbitrary. We must show that $s_j \models A'$. There is a sequence of states $(s_i = s_1, \ldots, s_n = s_j)$ such that for all $1 \leq k < n$, $s_{k+1} \in \tau(s_k)$. We will show by induction that for all k, $\nu_k(A') = T$ and the theorem follows when $k = n$. The induction will actually show that for all k, $\Box A' \in U_k$, from which $\nu_k(A') = T$ follows by condition 2 of the definition of Hintikka structures. $\Box A' \in U_1$ by assumption, proving the base case. By condition 2, if $\Box A' \in U_k$ then $\bigcirc \Box A' \in U_k$, so by condition 4, $\Box A' \in U_{k+1}$, proving the induction. ▮

Here is a linear fulfilling Hintikka structure constructed from the structure in Figure 11.2.



There is one link missing in order to obtain a decision procedure for satisfiability in PTL, namely, an algorithm that takes an arbitrary Hintikka structure, and decides if it contains a path that is a linear fulfilling Hintikka structure.

*Fulfillment of future formulas**

We begin with some definitions from graph theory. The concepts should be familiar, though it is worthwhile giving formal definitions.

Definition 11.35 A *graph* $G = (V, E)$ consists of a set of *vertices* $V = \{v_1, \ldots, v_n\}$ and a set of *edges* $E = \{e_1, \ldots, e_m\}$, which are pairs of vertices $e_k = \{v_i, v_j\} \subseteq V$. In a *directed* graph, each edge is an ordered pair, $e_k = (v_i, v_j)$. A *path* from v to v', denoted $v \rightsquigarrow v'$, is a sequence of edges such that

$$e_1 = (v = v_{i_1}, v_{i_2}), \ \ldots, \ e_l = (v_{i_{l-1}}, v_{i_l} = v').$$

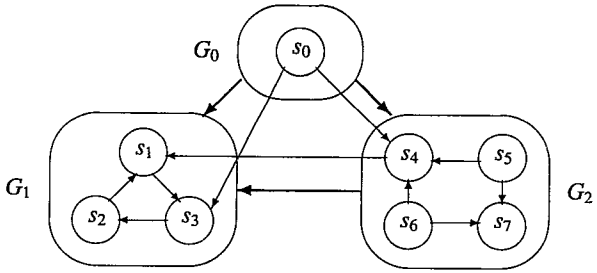


Figure 11.3 Component graph

A subgraph $G' = (V', E')$ of a directed graph $G = (V, E)$ is a graph such that $V' \subseteq V$ and $E' \subseteq E$. Note that from the definition of a graph, $e = (v_i, v_j) \in E'$ implies $\{v_i, v_j\} \subseteq V'$. □

Definition 11.36 A *strongly connected component (SCC)* $G' = (V', E')$ in a directed graph G is a subgraph such that $v_i \rightsquigarrow v_j$ for all $v_i, v_j \in V'$. A *maximal strongly connected component (MSCC)* is a SCC not properly contained in another. A *transient SCC* is a MSCC consisting of a single vertex. A *terminal SCC* is a MSCC with no outgoing edges. □

Definition 11.37 A directed graph G can be represented as a *component graph* G' which is a directed graph whose vertices are the MSCCs of G and whose edges are edges of G pointing from a vertex of one MSCC to a vertex of another MSCC. □

See Even (1979, Section 3.4) for an algorithm that constructs the component graph of a directed graph and a proof of the following theorem.

Theorem 11.38 *The component graph is acyclic.*

Example 11.39 Figure 11.3 shows a directed graph (circles and thin arrows) and its component graph (ovals and thick arrows). G_0 is transient and G_1 is terminal. □

Suppose that we have a Hintikka structure and a future formula in a *terminal* MSCC (such as G_1 in the figure). Then if the formula is going to be fulfilled at all, it will be fulfilled within the terminal MSCC because there are no other accessible nodes to which the fulfillment can be deferred. If a future formula is in a non-terminal MSCC (such as G_2), it can either be fulfilled within its own MSCC, or the fulfillment can be deferred to an accessible MSCC (in this case G_1). This suggests an algorithm for checking fulfillment: start at terminal MSCCs and work backwards.

Definition 11.40 Let $\mathcal{H} = (S, \mathcal{U}, \tau)$ be a Hintikka structure. $\mathcal{H}$ can be considered a graph $G = (V, E)$, where V is S and $(s_i, s_j) \in E$ iff $s_j \in \tau(s_i)$. We simplify the notation and write $A \in \nu$ for $A \in U_i$ when $\nu = s_i$.

Let $G = (V, E)$ be a SCC of $\mathcal{H}$. G is *self-fulfilling* iff for all $\nu \in V$ and for all future formulas $\Diamond A \in \nu$, $A \in \nu'$ for some $\nu' \in V$. □

Lemma 11.41 *Let $G = (V, E) \subseteq G' = (V', E')$ be SCCs of a Hintikka structure. If G is self-fulfilling, then so is G'. Hence, any self-fulfilling SCC is a subset of a self-fulfilling MSCC.*

Proof: Let $\Diamond A \in \nu' \in V' - V$. By definition of a Hintikka structure, either $A \in \nu'$ or $\bigcirc\Diamond A \in \nu'$. If $A \in \nu'$, then A is fulfilled in G'; otherwise, $\Diamond A \in \nu''$ for every $\nu'' \in \tau(\nu')$. By induction on the number of vertices in $V' - V$, either A is fulfilled in $V' - V$ or $\Diamond A \in \nu$ for some ν in V. But G is self-fulfilling, so $\Diamond A$ is fulfilled in some state $\nu_A \in V \subseteq V'$. Since G' is an SCC, $\nu' \rightsquigarrow \nu_A$ and A is fulfilled in G'. ∎

Lemma 11.42 *Let $G = (V, E)$ be an MSCC of $\mathcal{H}$ and let $\Diamond A \in \nu \in V$ be a future formula. If G is not self-fulfilling, $\Diamond A$ can only be fulfilled by some ν' in an MSCC G', such that $G \rightsquigarrow G'$ in the component graph. Hence, if G terminal, then $\Diamond A$ cannot be fulfilled.*

Proof: By construction. ∎

Algorithm 11.43 (Construction of a fulfilling Hintikka structure)
Input: A Hintikka structure $\mathcal{H}$.
Output: A fulfilling Hintikka structure within $\mathcal{H}$, or a report that no such structure exists.

Construct the component graph H of $\mathcal{H}$. Since H is acyclic (Theorem 11.38), there must be a terminal MSCC G. Check if G is self-fulfilling. If not, delete G and all its incoming edges from H. Repeat until the component graph is empty in which case no fulfilling structure exists, or until every terminal MSCC is self-fulfilling. □

Theorem 11.44 *The algorithm terminates with a non-empty graph iff there is a linear fulfilling Hintikka structure in $\mathcal{H}$.*

Proof: Let $G_1 \rightsquigarrow \cdots \rightsquigarrow G_n$ be a maximal path in the non-empty component graph, and let $v_i \rightsquigarrow v_{i+1}$ for $v_i \in G_i, v_{i+1} \in G_{i+1}$ be the edges in the underlying graph corresponding to the edges in the component graph.

Construct a path in $\mathcal{H}$ by following the edges in the component graph and replacing every component G_i by a path segment as follows, where $v_1^i \rightsquigarrow \cdots \rightsquigarrow v_{k_i}^i$ is a path through all the vertices in G_i.

- Replace a transient component by the single vertex v_1^i.

- Replace a terminal component by the closure

$$\nu_i \rightsquigarrow \cdots \rightsquigarrow (\nu_1^i \rightsquigarrow \cdots \rightsquigarrow \nu_{k_i}^i)^*.$$

- Replace a non-transient, non-terminal component by

$$\nu_i \rightsquigarrow \cdots \rightsquigarrow \nu_1^i \rightsquigarrow \cdots \nu_{k_i}^i \rightsquigarrow \nu_1^i \rightsquigarrow \cdots \nu_{k_i}^i \rightsquigarrow \cdots \rightsquigarrow \nu_{i+1}.$$

We leave it as an exercise to prove that the path is a linear fulfilling Hintikka structure.

Conversely, let $\mathcal{H}' = (s_1, \ldots, \ldots)$ be a linear fulfilling Hintikka structure in $\mathcal{H}$. Since $\mathcal{H}$ is finite, some suffix of $\mathcal{H}'$ must be composed of states which repeat infinitely often. These states must be contained within a self-fulfilling SCC G. By Lemma 11.41, G is contained in a self-fulfilling MSCC. ∎

Example 11.45 There are two maximal paths in the component graph in Figure 11.3: $G_0 \rightsquigarrow G_1$ and $G_0 \rightsquigarrow G_2 \rightsquigarrow G_1$. The paths constructed in the underlying graphs are: $s_0 \rightsquigarrow (s_3 \rightsquigarrow s_2 \rightsquigarrow s_1)^*$ and

$$s_0 \rightsquigarrow s_4 \rightsquigarrow s_5 \rightsquigarrow s_7 \rightsquigarrow s_6 \rightsquigarrow s_4 \rightsquigarrow s_5 \rightsquigarrow s_7 \rightsquigarrow s_6 \rightsquigarrow s_4 \rightsquigarrow (s_1 \rightsquigarrow s_2 \rightsquigarrow s_3)^*,$$

respectively. □

Theorem 11.46 *The method of semantic tableaux is a decision procedure for satisfiability in PTL.*

Proof: Construct a semantic tableau for a formula A. If it closes, A is unsatisfiable. If not, construct the Hintikka structure from the tableau. Apply Algorithm 11.43 to construct a fulfilling Hintikka structure. If the resulting graph is empty, A is unsatisfiable. Otherwise, apply the construction in Theorem 11.44 to construct a linear, fulfilling Hintikka structure; by Theorem 11.34, a model can be constructed for A. ∎

Corollary 11.47 (Finite model property) *A formula in PTL is satisfiable iff it is satisfiable in a finitely-presented model.*

Proof: By construction. ∎

11.5 Implementation of semantic tableaux^P

The implementation of the construction of semantic tableaux is quite lengthy and we give just an overview here, in particular we omit the graph algorithms which are not within the scope of this book. The complete source code can be found in the source archive. Before reading this section, you should review the program for the construction of systematic tableaux in the propositional calculus.

First we declare the temporal operators:


```
:- op(610, fy, @).                /* next */
:- op(610, fy, #).                /* always */
:- op(610, fy, <>).               /* eventually */
```

The decision procedure for satisfiability is implemented in four stages:

- `extend_tableau` performs the tableau construction until it terminates.
- `check_tableau` decides if the tableau is opened, closed or contains cycles.
- `create_states` constructs the state diagram from the tableau.
- `check_fulfillment` constructs the component graph and checks fulfillment.

Each node of the tableau contains five fields, three fields as before: the list of formulas and the links to the left and right children. The fourth is a node number that is generated by `get_num` when a new node is created. The final field is the ancestor path; it is used to check if a new state should be created or if a node should be connected to an ancestor. α- and β-nodes add 'themselves' to the ancestor path by appending the term `pt(Fmls,N)` to `Path`, where `Fmls` is the label and `N` is the node number.

```
extend_tableau(t(Fmls, Left, empty, N, Path)) :-
  alpha_rule(Fmls, Fmls1), !,
  get_num(N1),
  Left = t(Fmls1, _, _, N1, [pt(Fmls,N)|Path]),
  extend_tableau(Left).

extend_tableau(t(Fmls, Left, Right, N, Path)) :-
  beta_rule(Fmls, Fmls1, Fmls2), !,
  get_num(N1), get_num(N2),
  Left = t(Fmls1, _, _, N1, [pt(Fmls,N)|Path]),
  Right = t(Fmls2, _, _, N2, [pt(Fmls,N)|Path]),
  extend_tableau(Left),
  extend_tableau(Right).
```

`next_rule` is called to get the formulas in a node created by the X-rule, but before the node is created, `search` (source omitted) is called to search the `Path` for a node with the same set of formulas in its label. If successful, it returns the node number `N`; this branch of the tableau is terminated and marked `connect(N)`. Otherwise, a new node is created.

```
extend_tableau(t(Fmls, connect(N), empty, _, Path)) :-
  next_rule(Fmls, Fmls1),
  search(Path, Fmls1, N), !.
```

```
extend_tableau(t(Fmls, Left, empty, N, Path)) :-
  next_rule(Fmls, Fmls1), !,
  get_num(N1),
  Left = t(Fmls1, _, _, N1, [pt(Fmls,N)|Path]),
  extend_tableau(Left).
```

The following clauses are added to the procedures for `alpha` and `beta` to implement the rules for the temporal operators.

```
alpha(#A, A, @ #A).
alpha(- <>A, -A, - @ <>A).

beta(<>A, A, @ <>A).
beta(- #A, -A, - @ #A).
```

For an X-node, only next formulas are used to construct the label of the child.

```
next_rule([], []).
next_rule([@A | Tail], [A | Tail1]) :- !,
  next_rule(Tail, Tail1).
next_rule([- @A | Tail], [-A | Tail1]) :- !,
  next_rule(Tail, Tail1).
next_rule([_ | Tail], Tail1) :-
  next_rule(Tail, Tail1).
```

After the tableau construction terminates, `check_tableau` is called. It traverses the tableau down to the leaves and then returns up the tree to compute the status of the tableau: if there is one open branch, the tableau is open and satisfiable; if all branches are closed, the tableau is closed and unsatisfiable; otherwise, there is a cycle in the tableau and the structure must be checked for fulfillment.

`create_states` takes a tableau and constructs the structure: states, state paths which are transitions, and state labels which are the union of the labels on the state paths. It returns a list of terms `st(Fmls, N)`, where `Fmls` is the state label and `N` the node number of the state, and a list of terms `tau(From, To)`, where `From` and `To` are the node numbers of states.

These lists are the input to `component_graph`, which returns a list of MSCCs (a MSCC is a list of its states) and a list of edges of the form `e(From,To)`, where `From` and `To` are MSCCs. `fulfil1` selects a MSCC, `S`, with no outgoing edges and calls `self-fulfil` to check if `S` is self-fulfilling. If successful, it returns `ok(S)`, if not, it deletes `S` from the list of MSCCs, adds `notok(S,Result)` to the list of results and calls itself recursively. `Result` is the future formula that could not be fulfilled in `S`. `self-fulfil` checks each future formula `<>F` (or `-#F`) to see if `F` (or `-F`) occurs in some state in the SCC.


```
fulfil1(SCCs, Edges, States, [ok(S)]) :-
  member(S, SCCs),
  \+ member(e(S, _), Edges),
  self_fulfil(S, States, Result),
  Result == ok, !.

fulfil1(SCCs, Edges, States, [notok(S, Result)|Result1]) :-
  member(S, SCCs),
  \+ member(e(S, _), Edges),
  self_fulfil(S, States, Result),
  delete(SCCs, S, SCCs1),
  delete(Edges, e(_,S), Edges1),
  fulfil1(SCCs1, Edges1, States, Result1).
fulfil1(_, _, _, []).
```

11.6 Exercises

1. Prove that every substitution instance of a valid propositional formula is valid (Theorem 11.7).
2. Prove $\models \neg \Diamond \neg p \rightarrow \Box p$ (the converse direction of Theorem 11.8).
3. Prove that a linear interpretation is characterized by $\bigcirc p \leftrightarrow \neg \bigcirc \neg p$ (Theorem 11.16).
4. * Identify the property of a relation characterized by $p \rightarrow \Box\Diamond p$. Identify the property of a relation characterized by $\Diamond p \rightarrow \Box\Diamond p$.
5. Show that in an interpretation with a reflexive transitive relation, any formula (without $\bigcirc$) is equivalent to one whose only temporal operators are $\Box$, $\Diamond$, $\Diamond\Box$, $\Box\Diamond$, $\Diamond\Box\Diamond$ and $\Box\Diamond\Box$. If the relation is also characterized by the formula $\Diamond p \rightarrow \Box\Diamond p$, any formula is equivalent to one with a single temporal operator.
6. Construct a semantic tableau for $\Box(\Diamond(p \wedge q) \wedge \Diamond(\neg p \wedge q) \wedge \Diamond(p \wedge \neg q))$ from Example 11.28.
7. Prove that the construction of a semantic tableau terminates (Theorem 11.23).
8. Prove that the construction of the path in the proof of Theorem 11.44 gives a linear fulfilling Hintikka structure.
9. Give a complete list of the elements of U_0 and U_1 in Figure 11.2.
10. Construct a tableau and find a model for the negation of $\Box\Diamond p \rightarrow \Diamond\Box p$.
11. ^{P} Extend the implementation of semantic tableaux to include the precedence operator $\mathcal{U}^r$ from Section 12.3.

12 Temporal Logic: Deduction and Applications

12.1 The deductive system $\mathcal{L}$

We define a deductive system $\mathcal{L}$ for linear-time propositional temporal logic. The primitive operators of $\mathcal{L}$ are $\Box$ and $\bigcirc$, while $\Diamond$ is defined as the dual of $\Box$.

Definition 12.1 Let A and B be any formulas of PTL. The axioms of $\mathcal{L}$ are:

0.	PC	Any substitution instance of a valid propositional formula.
1.	Distribution of $\Box$ over $\rightarrow$	$\vdash \Box(A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B)$.
2.	Distribution of $\bigcirc$ over $\rightarrow$	$\vdash \bigcirc(A \rightarrow B) \rightarrow (\bigcirc A \rightarrow \bigcirc B)$.
3.	Expansion of $\Box$	$\vdash \Box A \rightarrow (A \wedge \bigcirc A \wedge \bigcirc\Box A)$.
4.	Induction	$\vdash \Box(A \rightarrow \bigcirc A) \rightarrow (A \rightarrow \Box A)$.
5.	Linearity	$\vdash \bigcirc A \leftrightarrow \neg \bigcirc \neg A$.

The rules of inference are *Modus Ponens* and *Generalization*: $\dfrac{\vdash A}{\vdash \Box A}$. □

In a proof by induction, the inductive step is $A \rightarrow \bigcirc A$, that is, we assume that A is true 'today' and prove that A is true 'tomorrow'. If this inductive step is always true, $\Box(A \rightarrow \bigcirc A)$, then $A \rightarrow \Box A$ by the induction axiom. Thus, if A is true 'today' (the base case), then A is always true.

We now proceed to prove theorems in $\mathcal{L}$. In order to concentrate on aspects of temporal reasoning, several shortcuts will be taken in the proofs. We will omit detailed justifications of deductions in the propositional calculus and just write PC. We will use some derived rules that are easy to justify:

$$\frac{\vdash A \rightarrow B}{\vdash \Box A \rightarrow \Box B} \qquad \frac{\vdash A \rightarrow B}{\vdash \bigcirc A \rightarrow \bigcirc B} \qquad \frac{\vdash A \rightarrow \bigcirc A}{\vdash A \rightarrow \Box A}.$$

We will call uses of the first two rules Generalization, and the third Induction.

(To make the formulas easy to read, they will be stated and proved for propositional symbols p and q, though the intention is that they hold for arbitrary PTL formulas A and B.)

Theorem 12.2 (Transitivity) $\vdash \Box\Box p \leftrightarrow \Box p$

Proof:

1.	$\vdash \Box\Box p \rightarrow \Box p$	Expansion
2.	$\vdash \Box p \rightarrow \bigcirc\Box p$	Expansion
3.	$\vdash \Box p \rightarrow \Box\Box p$	2, Induction
4.	$\vdash \Box\Box p \leftrightarrow \Box p$	1, 3, PC

Theorem 12.3 (Distribution) $\vdash \bigcirc(p \wedge q) \leftrightarrow (\bigcirc p \wedge \bigcirc q)$.

Proof:

1.	$\vdash \bigcirc(p \wedge q) \rightarrow \bigcirc p$	Generalization
2.	$\vdash \bigcirc(p \wedge q) \rightarrow \bigcirc q$	Generalization
3.	$\vdash \bigcirc(p \wedge q) \rightarrow (\bigcirc p \wedge \bigcirc q)$	1, 2, PC
4.	$\vdash \bigcirc(p \rightarrow \neg q) \rightarrow (\bigcirc p \rightarrow \bigcirc \neg q)$	Generalization
5.	$\vdash \neg(\bigcirc p \rightarrow \bigcirc \neg q) \rightarrow \neg \bigcirc(p \rightarrow \neg q)$	4, PC
6.	$\vdash \neg \bigcirc p \vee \bigcirc \neg q \vee \neg \bigcirc(p \rightarrow \neg q)$	5, PC
7.	$\vdash \neg \bigcirc p \vee \neg \bigcirc q \vee \bigcirc \neg (p \rightarrow \neg q)$	6, Linearity
8.	$\vdash (\bigcirc p \wedge \bigcirc q) \rightarrow \bigcirc(p \wedge q)$	7, PC
9.	$\vdash \bigcirc(p \wedge q) \leftrightarrow (\bigcirc p \wedge \bigcirc q)$	3, 8, PC

Theorem 12.4 (Distribution) $\vdash \Box(p \wedge q) \rightarrow (\Box p \wedge \Box q)$.

Proof: As for Theorem 12.3. (We will prove the converse below.)

The following theorem is the converse to the Expansion Axiom.

Theorem 12.5 (Contraction) $\vdash p \wedge \bigcirc\Box p \rightarrow \Box p$.

Proof:

1.	$\vdash \Box p \rightarrow p \wedge \bigcirc\Box p$	Expansion
2.	$\vdash \bigcirc\Box p \rightarrow \bigcirc(p \wedge \bigcirc\Box p)$	1, Generalization
3.	$\vdash p \wedge \bigcirc\Box p \rightarrow \bigcirc(p \wedge \bigcirc\Box p)$	2, PC
4.	$\vdash p \wedge \bigcirc\Box p \rightarrow \Box(p \wedge \bigcirc\Box p)$	3, Induction
5.	$\vdash p \wedge \bigcirc\Box p \rightarrow \Box p$	4, Distribution (12.4), PC

Now we can prove the converse of Theorem 12.4. The structure of the proof is typical of inductive proofs in $\mathcal{L}$.

Theorem 12.6 (Distribution) $\vdash (\Box p \wedge \Box q) \rightarrow \Box(p \wedge q)$.

Proof: Let $r = \Box p \wedge \Box q \wedge \neg \Box(p \wedge q)$ and prove that r is invariant.

1.	$\vdash r \rightarrow (p \wedge \bigcirc\Box p) \wedge (q \wedge \bigcirc\Box q) \wedge (\neg(p \wedge q) \vee \neg \bigcirc\Box(p \wedge q))$	Expansion Contraction (12.5), PC
2.	$\vdash r \rightarrow (p \wedge \bigcirc\Box p) \wedge (q \wedge \bigcirc\Box q) \wedge \neg \bigcirc\Box(p \wedge q)$	1, PC
3.	$\vdash r \rightarrow \bigcirc\Box p \wedge \bigcirc\Box q \wedge \neg \bigcirc\Box(p \wedge q)$	2, PC
4.	$\vdash r \rightarrow \bigcirc\Box p \wedge \bigcirc\Box q \wedge \bigcirc \neg \Box(p \wedge q)$	Linearity
5.	$\vdash r \rightarrow \bigcirc r$	4, Distribution (12.3)
6.	$\vdash r \rightarrow \Box r$	5, Induction

The proof of the theorem continues as follows:

7.	$\vdash r \rightarrow \Box p \wedge \Box q$	Def. of r, PC
8.	$\vdash r \rightarrow p \wedge q$	7, Expansion
9.	$\vdash \Box r \rightarrow \Box(p \wedge q)$	8, Generalization
10.	$\vdash r \rightarrow \Box(p \wedge q)$	6, 9, PC
11.	$\vdash r \rightarrow \neg \Box(p \wedge q)$	Def. of r, PC
12.	$\vdash r \rightarrow \mathit{false}$	10, 11, PC
13.	$\vdash \Box p \wedge \Box q \wedge \neg \Box(p \wedge q) \rightarrow \mathit{false}$	Def. of r
14.	$\vdash \Box p \wedge \Box q \rightarrow \Box(p \wedge q)$	PC

$\Box$ distributes over disjunction only in one direction. $\bigcirc$ distributes over disjunction in both directions because it is self-dual.

Theorem 12.7 (Distribution)

(a) $\vdash (\Box p \vee \Box q) \rightarrow \Box(p \vee q)$, (b) $\vdash \bigcirc(p \vee q) \leftrightarrow (\bigcirc p \vee \bigcirc q)$.

Proof: Exercise.

Theorem 12.8 (Exchange) $\vdash \Box \bigcirc p \leftrightarrow \bigcirc\Box p$.

Proof:

1.	$\vdash \Box p \rightarrow \bigcirc p$	Expansion
2.	$\vdash \Box\Box p \rightarrow \Box \bigcirc p$	1, Generalization
3.	$\vdash \Box p \rightarrow \Box \bigcirc p$	2, Transitivity (12.2)
4.	$\vdash \Box p \rightarrow p$	Expansion
5.	$\vdash \Box p \rightarrow p \wedge \Box \bigcirc p$	3, 4, PC
6.	$\vdash \bigcirc\Box p \rightarrow \bigcirc(p \wedge \Box \bigcirc p)$	5, Generalization
7.	$\vdash \bigcirc\Box p \rightarrow \bigcirc p \wedge \bigcirc\Box \bigcirc p$	6, Distribution (12.3)
8.	$\vdash \bigcirc\Box p \rightarrow \Box \bigcirc p$	7, Contraction (12.5)

9.	$\vdash \Box \bigcirc p \to \bigcirc p \wedge \bigcirc \Box \bigcirc p$	Expansion
10.	$\vdash p \wedge \Box \bigcirc p \to \bigcirc p \wedge \bigcirc \Box \bigcirc p$	9, PC
11.	$\vdash p \wedge \Box \bigcirc p \to \bigcirc (p \wedge \Box \bigcirc p)$	10, Distribution (12.3)
12.	$\vdash p \wedge \Box \bigcirc p \to \Box (p \wedge \Box \bigcirc p)$	11, Induction
13.	$\vdash p \wedge \Box \bigcirc p \to \Box p$	12, Distribution (12.4), PC
14.	$\vdash \bigcirc (p \wedge \Box \bigcirc p) \to \bigcirc \Box p$	13, Generalization
15.	$\vdash \bigcirc p \wedge \bigcirc \Box \bigcirc p \to \bigcirc \Box p$	14, Distribution (12.3)
16.	$\vdash \Box \bigcirc p \to \bigcirc \Box p$	15, Contraction (12.5)
17.	$\vdash \Box \bigcirc p \leftrightarrow \bigcirc \Box p$	8, 16, PC

From the duality between $\Box$ and $\Diamond$, we easily get theorems with future formulas.

Theorem 12.9

(a) $\vdash p \to \Diamond p$, (b) $\vdash \bigcirc p \to \Diamond p$, (c) $\vdash \Box p \to \Diamond p$.

Proof:

1.	$\vdash \Box \neg p \to \neg p$	Expansion
2.	$\vdash \neg \neg p \to \neg \Box \neg p$	1, PC
3.	$\vdash p \to \neg \Box \neg p$	2, PC
4.	$\vdash p \to \Diamond p$	3, Definition
5.	$\vdash \Box \neg p \to \bigcirc \neg p$	Expansion
6.	$\vdash \neg \bigcirc \neg p \to \neg \Box \neg p$	5, PC
7.	$\vdash \bigcirc p \to \neg \Box \neg p$	6, Linearity
8.	$\vdash \bigcirc p \to \Diamond p$	7, Definition
9.	$\vdash \Box p \to \bigcirc p$	Expansion
10.	$\vdash \bigcirc p \to \Diamond p$	(b)
11.	$\vdash \Box p \to \Diamond p$	9, 10, PC

Theorem 12.10

(a) $\vdash \Diamond (p \vee q) \leftrightarrow (\Diamond p \vee \Diamond q)$ (b) $\vdash \Diamond (p \wedge q) \to (\Diamond p \wedge \Diamond q)$

(c) $\vdash \Diamond p \leftrightarrow p \vee \bigcirc \Diamond p$ (d) $\vdash \Diamond \bigcirc p \leftrightarrow \bigcirc \Diamond p$

(e) $\vdash \Box (p \to q) \to (\Diamond p \to \Diamond q)$ (f) $\vdash \Diamond \Diamond p \leftrightarrow \Diamond p$

Proof: Exercise.

From Theorem 12.10(e), we obtain a generalization rule for $\Diamond$:

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

The following theorem shows that sequences of temporal operators not containing $\bigcirc$ collapse very quickly, for example: $\Box\Box\Diamond\Box\Diamond\Diamond\Box\Box p \leftrightarrow \Diamond\Box p$.

Theorem 12.11 (a) $\vdash \Box\Diamond\Box p \leftrightarrow \Diamond\Box p$, (b) $\vdash \Diamond\Box\Diamond p \leftrightarrow \Box\Diamond p$.

Proof:

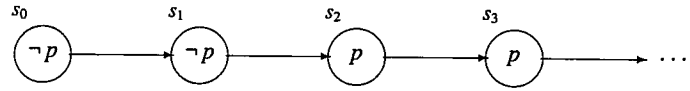
1.	$\vdash \Box\Diamond\Box p \to \Diamond\Box p$	Expansion
2.	$\vdash \Box p \to \bigcirc\Box p$	Expansion
3.	$\vdash \Diamond\Box p \to \Diamond \bigcirc \Box p$	2, Generalization
4.	$\vdash \Diamond\Box p \to \bigcirc\Diamond\Box p$	3, Exchange (12.10(d))
5.	$\vdash \Diamond\Box p \to \Box\Diamond\Box p$	4, Induction
6.	$\vdash \Box\Diamond\Box p \leftrightarrow \Diamond\Box p$	1, 5, PC
7.	$\vdash \Diamond\Box\Diamond p \leftrightarrow \Box\Diamond p$	Exercise

$\Box$ and $\Diamond$ commute in only one direction.

Theorem 12.12 $\vdash \Diamond\Box p \to \Box\Diamond p$.

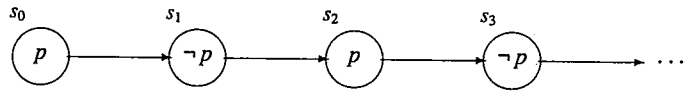
Proof: Exercise.

Example 12.13 Here is a state transition diagram, where the ellipsis indicates that $s_i(p) = T$ for $i \geq 4$.



Both $\Diamond\Box p$ and $\Box\Diamond p$ are true.

In the following diagram, $s_i(p) = T$ for even i and $s_i = F$ for odd i.



$\Box\Diamond p$ is true, since for any state s_i, $s_{2i} \models p$. Obviously, $\Diamond\Box p$ is false in all states of the diagram, because for any s_i, $s_i \models \neg p$ if i is odd and $s_{i+1} \models \neg p$ if i is even. □

The following theorem characterizes linear-time temporal logic without using $\bigcirc$. We sketch the proof (which is similar to the proof of Theorem 12.6) and leave the details to the reader.

Theorem 12.14 $\vdash \Box((p \vee \Box q) \wedge (\Box p \vee q)) \leftrightarrow (\Box p \vee \Box q)$.

Proof: Denote the formula $(p \vee \Box q) \wedge (\Box p \vee q)$ by r, so that the formula to be proved is $\Box r \leftrightarrow (\Box p \vee \Box q)$, and denote $\Box r \wedge \neg \Box p \wedge \neg \Box q$ by s. Show that s is inductive and then show that $\vdash \Box r \to (\Box p \vee \Box q)$. The other direction of the equivalence is easy.

12.2 Soundness and completeness of $\mathcal{L}^*$

Theorem 12.15 (Soundness of $\mathcal{L}$) *Let A be a formula of PTL. If $\vdash_{\mathcal{L}} A$ then $\models A$.*

Lemma 12.16 $\models \Box(A \rightarrow \bigcirc A) \rightarrow (A \rightarrow \Box A)$.

Proof: Suppose that the formula is not valid. Then there exists an interpretation $\mathcal{I}$ and a state s such that

$$s \models \Box(A \rightarrow \bigcirc A) \wedge A \wedge \neg \Box A.$$

Since $s \models \neg \Box A$, there exists a state $s' \in \rho(s)$ such that $s' \models \neg A$. By definition of ρ as τ^* there exists a sequence of states $(s = s_1, \ldots, s_n = s')$, such that $s_{i+1} \in \tau(s_i)$. By induction we show that for all i, $s_i \models A$, in particular, $s_n = s' \models A$ which is a contradiction. The base case, $s_1 = s \models A$ holds by assumption. Suppose that $s_i = s \models A$. But $s \models \Box(A \rightarrow \bigcirc A)$ by assumption and $s_i \in \rho(s)$, so $s_i \models A \rightarrow \bigcirc A$, $s_i \models \bigcirc A$ and $s_{i+1} \models A$, since $s_{i+1} \in \tau(s)$. ∎

Lemma 12.17 *If $\models A$, then $\models \Box A$.*

Proof: Let $\mathcal{I}$ be an arbitrary interpretation for A and s an arbitrary state in $\mathcal{I}$. We need to show that $s \models \Box A$. This is true if for all $s' \in \rho(s)$, $s' \models A$. But A is valid which means that it is true in every state including s'. ∎

Proof of soundness: By induction of the length of the proof in $\mathcal{L}$. The valid formulas of the propositional calculus are sound by definition, and we showed the soundness of MP. The soundness of Axiom 1 was shown in Theorem 11.9 and that of Axiom 5 follows from Theorem 11.16. Lemma 12.16 proves the soundness of Axiom 4 and Lemma 12.17 proves the soundness of Generalization. We leave the soundness of Axioms 2 and 3 as an exercise. ∎

Theorem 12.18 (Completeness of $\mathcal{L}$) *Let A be a formula of PTL. If $\models A$ then $\vdash_{\mathcal{L}} A$.*

Proof: If A is valid, the construction of a semantic tableau for $\neg A$ will fail, either because it closes or because all its terminal MSCCs are non-fulfilling. We show by induction that a disjunction of the negations of formulas labeling a node of the semantic tableau is provable in $\mathcal{L}$.

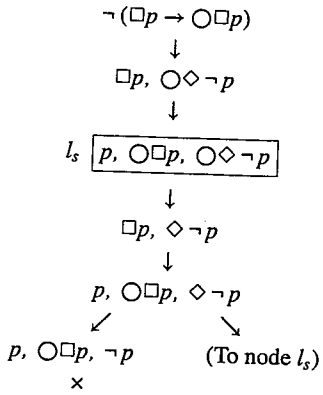
The base case of the leaves and the inductive steps where α- and β-rules are used follow from propositional reasoning together with the Expansion axiom. Suppose that the X-rule is used in the construction of the semantic tableau:

$$\begin{array}{c} \bigcirc A_1, \ldots, \bigcirc A_n,\ B_1, \ldots, B_k \\ \downarrow \\ A_1, \ldots, A_n \end{array}$$

where we assume that negations are pushed inwards as justified by the Linearity axiom. The following deduction proves the formula associated with the parent node:

1.	$\vdash \neg A_1 \vee \cdots \vee \neg A_n$	Inductive hypothesis
2.	$\vdash \Box(\neg A_1 \vee \cdots \vee \neg A_n)$	1, Generalization
3.	$\vdash \bigcirc(\neg A_1 \vee \cdots \vee \neg A_n)$	2, Expansion
4.	$\vdash \bigcirc \neg A_1 \vee \cdots \vee \bigcirc \neg A_n$	3, Distribution (12.7)
5.	$\vdash \neg \bigcirc A_1 \vee \cdots \vee \neg \bigcirc A_n$	4, Linearity
6.	$\vdash \neg \bigcirc A_1 \vee \cdots \vee \neg \bigcirc A_n \vee \neg B_1 \vee \cdots \vee \neg B_k$	5, PC

There remains the second base case of a node that is part of a non-fulfilling MSCC. We will demonstrate the technique of the proof with an example, proving $\Box p \rightarrow \bigcirc \Box p$. Here is a semantic tableau for the negation of the formula:



The crucial part of the proof is to define the *invariant* of the loop, that is, a formula A such that $\vdash A \rightarrow \bigcirc A$. The invariant will be the conjunction of the formulas A_i, where $\bigcirc A_i$ are the next formulas in the states of the SCC, as these represent what must be true from one state to the next. In the example, for invariant is $\Box p \wedge \Diamond \neg p$.

1.	$\vdash (\Box p \wedge \Diamond \neg p) \rightarrow (p \wedge \bigcirc \Box p) \wedge (\neg p \vee \bigcirc \Diamond \neg p)$	Expansion
2.	$\vdash (\Box p \wedge \Diamond \neg p) \rightarrow (p \wedge \bigcirc \Box p \wedge \bigcirc \Diamond \neg p)$	1, PC
2.	$\vdash (\Box p \wedge \Diamond \neg p) \rightarrow (\bigcirc \Box p \wedge \bigcirc \Diamond \neg p)$	2, PC
4.	$\vdash (\Box p \wedge \Diamond \neg p) \rightarrow \bigcirc(\Box p \wedge \Diamond \neg p)$	3, Distribution (12.3)
5.	$\vdash (\Box p \wedge \Diamond \neg p) \rightarrow \Box(\Box p \wedge \Diamond \neg p)$	4, Induction

From the closed left-hand branch it is immediate that $\vdash \neg p \vee \neg \bigcirc \Box p \vee \neg \neg p$. Continuing the proof:

6.	$\vdash \neg p \vee \neg \bigcirc \Box p \vee \neg \neg p$	Left-hand branch
7.	$\vdash (p \wedge \bigcirc \Box p) \rightarrow \neg \neg p$	6, PC
8.	$\vdash \Box p \rightarrow \neg \neg p$	7, Contraction
9.	$\vdash (\Box p \wedge \Diamond \neg p) \rightarrow \neg \neg p$	8, PC
10.	$\vdash \Box(\Box p \wedge \Diamond \neg p) \rightarrow \Box \neg \neg p$	9, Generalization
11.	$\vdash (\Box p \wedge \Diamond \neg p) \rightarrow \Box \neg \neg p$	5, 10, PC
12.	$\vdash (p \wedge \bigcirc \Box p \wedge \Diamond \neg p) \rightarrow \Box \neg \neg p$	11, Expansion
13.	$\vdash (p \wedge \bigcirc \Box p \wedge \Diamond \neg p) \rightarrow \neg \Diamond \neg p$	12, Duality
14.	$\vdash \neg p \vee \neg \bigcirc \Box p \vee \neg \Diamond \neg p$	13, PC

Line 14 is the disjunction of the complements of the formulas at the β-node. The proof for the node's parent and its grandparent l_s follow simply from the proof schemes for α- and X-nodes. ∎

This method may not yield the shortest possible proof of the formula, but it shows that *systematic* proof that can be discovered by constructing a semantic tableau.

12.3 Other temporal logics*

Precedence operators

In specifying systems, there is a need to express a requirement of the form: 'p happens before q'. The formula $p \rightarrow \Diamond q$ does *not* express this concept, as it is true even if p is false in the current state. To express *precedence* properties, a binary temporal operator must be used. There are two possibilities: the first is *waiting for*, denoted $p \mathcal{W} q$, which is true if p is true while it waits for q to become true. There is a stronger operator, the *until* operator, denoted $p \mathcal{U} q$, which ensures that q will eventually become true.

Definition 12.19
$s_0 \models A \mathcal{W} B$ iff for all i, $s_i \models A$ *or* there exists some n, such that $s_n \models B$ and for all i such that $0 \leq i < n$, $s_i \models A$.
$s_0 \models A \mathcal{U} B$ iff there exists some n such that $s_n \models B$ *and* for all $0 \leq i < n$, $s_i \models A$. □

The relationship between the operators is given by the following equivalences:

Theorem 12.20

$$\begin{aligned} \models A \mathcal{U} q &\leftrightarrow (A \mathcal{W} q) \wedge \Diamond q. \\ \models A \mathcal{W} q &\leftrightarrow (A \mathcal{U} q) \vee \Box A. \\ \models \Diamond A &\leftrightarrow \mathit{true}\, \mathcal{U} A. \\ \models \Box A &\leftrightarrow A \mathcal{W} \mathit{false}. \end{aligned}$$

Proof: Exercise. ∎

Either of these binary temporal operators can be taken as the sole primitive operator together with $\bigcirc$. $\mathcal{W}$ and $\mathcal{U}$ can be defined in terms of each other and the unary operators can be defined as from the binary ones. A complete axiom system is obtained by adding the following two axioms to the axioms of Definition 12.1:

6. Expansion $\vdash A \mathcal{U} B \leftrightarrow (B \vee (A \wedge \bigcirc(A \mathcal{U} B)))$.
7. Eventuality $\vdash A \mathcal{U} B \rightarrow \Diamond B$.

Axiom 6 is the β-rule for the until operator: either B is true 'today', or A is true and $A \mathcal{U} B$ will be true 'tomorrow'. The construction of the semantic tableau and the proof of completeness are very similar to those for the simpler version of temporal logic.

Past operators

The temporal operators that we defined cannot express requirements on what must have happened since an event occurred. The *since* operator, denoted $p \mathcal{S} q$, and its weak version *back-to*, denoted $p \mathcal{B} q$, are defined as follows:

Definition 12.21
$s_k \models A \mathcal{S} B$ iff for some $0 \leq i \leq k$, $s_i \models B$, *and*, for all $0 \leq i < j \leq k$, $s_j \models A$.
$s_k \models A \mathcal{B} B$ iff *if* $s_i \models B$ for some $0 \leq i \leq k$, *then* for all $0 \leq i < j \leq k$, $s_j \models A$. □

The weak operator does not require that B actually have been true in the past.

Unary operators can be defined from the binary ones: the existential operator *once* is denoted $\Diamond\!\!\!-p$ and is defined by $\mathit{true}\, \mathcal{S} p$, and the universal operator *so-far* can be defined from it by $\Box\!\!\!-p = \neg \Diamond\!\!\!- \neg p$.

The past version of the next operator is *previously*, denoted $\ominus p$, which is true in a state s_i if p was true in s_{i-1}. In s_0, $\ominus p$ is false since there is no previous state. There is a weak version of *previously* that is vacuously true in s_0.

The past operators do not introduce any difficulties in the theoretical treatment of temporal logic. Unlike the future operators which can be non-fulfilling, at any point in the computation there is only a finite number of previous states in which the formula can be true.

Branching-time operators

Branching-time logic can be formalized by introducing compound operators that quantify over the paths as well as over the states along the path (or paths) selected by the quantifier:

$\forall \Box$	for all paths π, at all states $s \in \pi$,
$\exists \Box$	for some path π, at all states $s \in \pi$,
$\forall \Diamond$	for all paths π, at some state $s \in \pi$,
$\exists \Diamond$	for some path π, at some state $s \in \pi$.

The most widely used branching-time logic, CTL, uses a different notation: A and E for $\forall$ and $\exists$, and G and F for $\Box$ and $\Diamond$.

Example 12.22 In the structure in Figure 11.1, we have $s_2 \models \exists \Box p$ and $s_2 \models \exists \Diamond \neg p$ since a path exists such that p is true at every state in the path, and *another* path exists such that $\neg p$ is true at some state in the path. Clearly, $\not\models \forall \Box p$. □

Branching-time temporal logic is presented in detail in Huth & Ryan (2000).

12.4 Specification and verification of programs*

Temporal logic is used in the specification and verification of computer systems with dynamic behavior such as hardware and operating systems. In this section we give a deductive proof of a concurrent program and in the next section we show how the same program can be proved using tableau techniques.

Specification of concurrent programs

Definition 12.23 A concurrent program is a set of subprograms $\{p_1, p_2, \ldots, p_n\}$ called *processes* that can be executed in parallel. The *variables* of the program are denoted $\{v_1, v_2, \ldots, v_m\}$. The *statements* of process i are labeled $\{l_{i1}, l_{i2}, \ldots, l_{ik_i}\}$. A *state* of a computation s consists of $(v_{1s}, v_{2s}, \ldots, v_{ms})$, where v_{js} is the value of the j-th variable in the state, and $(l_{1j_s}, l_{2j_s}, \ldots, l_{nj_s},)$, where l_{ij_s} denotes that the i-th process is at location j_s. □

The following informal definition of a computation can be formalized as we did for sequential programs in Chapter 9.

Definition 12.24 A *transition* from state s to state $s+1$ is done by selecting one of the processes i and executing the statement labeled l_{is}. State $s+1$ the same as s except for $(v_{1s}, \ldots, v_{ms})$ which may be changed by an assignment statement, and $l_{i(s+1)}$ which will be $l_{is}+1$ unless changed by an `if-` or `while`-statement. A computation is sequence of states $(s_0, \ldots)$, such that s_{i+1} follows from s_i by one of the transitions. □

This computational model is called *interleaving of atomic instructions* because it allows arbitrary interleaving of execution sequences of the concurrent processes, but each instruction of a process is atomic, that is, it is executed indivisibly. Given the statement `X := 1` in one process and `X := 2` in another, the result of the interleaved computation is that `X` has either the value 1 or the value 2, not some other value such as 3.

To formalize the temporal properties of a concurrent program, progress axioms are defined for each statement. We will assume that the sets of symbols used for labels

in each process are disjoint, so that we can use the label as a proposition with the intended meaning that the program counter for that process is *at* that label.

Definition 12.25

Statement	Progress axioms
`li: v := expression`	$\vdash l_i \rightarrow \Diamond l_{i+1}$

`li: if B then`
`lt: S1`
`else`
`lf: S2`
	$\vdash (l_i \wedge \Box B) \rightarrow \Diamond l_t$
	$\vdash (l_i \wedge \Box \neg B) \rightarrow \Diamond l_f$

`li: while B do`
`lt: S1;`
`lf: S2`
	$\vdash (l_i \wedge \Box B) \rightarrow \Diamond l_t$
	$\vdash (l_i \wedge \Box \neg B) \rightarrow \Diamond l_f$

□

In the `if-` and `while`-statements, it is always true that $l_i \rightarrow \Diamond(l_t \vee l_f)$, but without more information, you cannot know which branch will be taken. $(l_i \wedge B) \rightarrow \Diamond l_t$ does *not* hold, because by the time that this transition is taken, another process could have modified a global variable falsifying B. Only if B is held true or false indefinitely can we specify which branch will be taken.

Be careful to distinguish between $\Box\Diamond p \wedge \Box\Diamond q$ and $\Diamond\Box p \wedge \Box\Diamond q$. The first formula specifies that p and q are both true *infinitely often*, but p may 'choose' to be true exactly when q is false and conversely as shown in Figure 12.1. The second formula states that eventually p will *remain* true, and thus will be true at every subsequent true occurrence of q (Figure 12.2). The difference between $\Diamond\Box p$ and $\Box\Diamond p$ is important in the definition of the fairness of constructs such as semaphores (Manna & Pnueli 1992,1995).

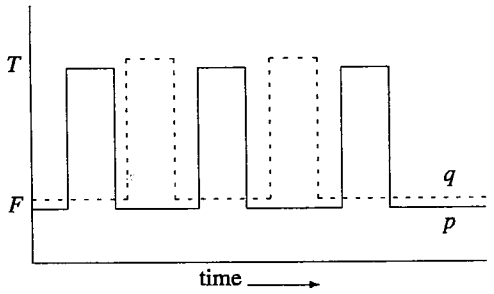


Figure 12.1 Timing diagram for $\Box\Diamond p \wedge \Box\Diamond q$

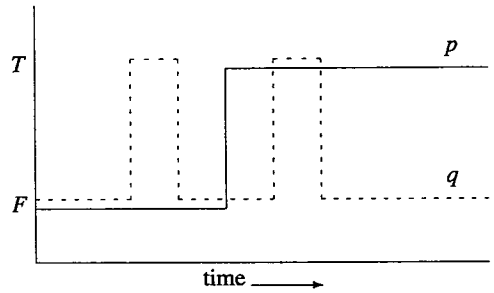


Figure 12.2 Timing diagram for $\Diamond\Box p \wedge \Box\Diamond q$

Verification of concurrent programs

In this section we prove the correctness of Peterson's algorithm for solving the *critical section problem*. The description of the problem is:

> There are two processes P1 and P2. Each process consists of a *critical section* and a *non-critical section*. A process may stay indefinitely in its non-critical section, or it may request to enter its critical section. A process that has entered its critical section will eventually leave it. The solution must satisfy two properties:
>
> **Mutual exclusion** It is forbidden for the two processes to be in their critical sections simultaneously.
>
> **Liveness** If a process attempts to enter its critical section, it will eventually succeed.

Figure 12.3 shows Peterson's algorithm for solving the mutual exclusion problem, where each process is written in Pascal syntax and the statements are labeled. (To simplify the proof, the two assignment statements are assumed to execute atomically.)

P1 requests entry to the critical section by setting variable C1 to True. It then waits until P2 is not in its critical section by testing variable C2. Since this part of the algorithm is symmetric, the variable Last is used to break ties. Its value indicates which process was the *last* one to attempt to enter its critical section. If both processes attempt to enter their critical sections simultaneously, the tie is broken in favor of the process which first attempted. To prove the correctness of Peterson's algorithm we must prove the following two formulas:

Mutual exclusion	$\Box\neg(CS1 \wedge CS2)$,
Liveness:	$\Box(Set1 \rightarrow \Diamond CS1) \wedge \Box(Set2 \rightarrow \Diamond CS2)$.

It is not easy to be convinced of the correctness of the algorithm just by examining the code. Let us start the formal proof with a few elementary lemmas.

Lemma 12.26
(a) $\vdash \Box((Last = 1) \vee (Last = 2))$.
(b) $\vdash \Box(C1 \leftrightarrow (Test1 \vee CS1))$.
(c) $\vdash \Box(C2 \leftrightarrow (Test2 \vee CS2))$.

Proof: The formulas are simple invariants which are proved by induction. We give one partial proof and leave the complete proofs as exercises. (b) is true initially, because $C1$ is false as is $Test1 \vee CS1$ (since we are *at* NC1). Suppose that (b) is true; we have to check that each of the ten statements preserve the truth of the formula. For example, if the next statement to be executed is Set1, then $C1$ becomes true as does $Test1$, so the truth of the equivalence is preserved. ∎

```
program Peterson;
var C1, C2: Boolean := False;
    Last:   1..2 := 1;

process P1;
begin
  while true do
  begin
    NC1:    Non_critical_section_1;
    Set1:   C1 := True; Last := 1;
    Test1:  while C2 and (Last = 1) do { nothing };
    CS1:    Critical_section_1;
    Reset1: C1 := False;
  end;
end;

process P2;
begin
  while true do
  begin
    NC2:    Non_critical_section_2;
    Set2:   C2 := True; Last := 2;
    Test2:  while C1 and (Last = 2) do { nothing };
    CS2:    Critical_section_2;
    Reset2: C2 := False;
  end;
end;
```

Figure 12.3 Peterson's algorithm for mutual exclusion

Lemma 12.27
(a) $\vdash \Box(\Box\neg C1 \vee \Diamond(Last = 1))$.
(b) $\vdash \Box(\Box\neg C2 \vee \Diamond(Last = 2))$.

Proof: Let us prove (b) as the proof for (a) is symmetrical:

1.	$\vdash \Box NC2 \vee \Diamond Set2$	Def. of non-critical section
2.	$\vdash \Box NC2 \rightarrow \Box\neg(Test2 \vee CS2)$	Progress
3.	$\vdash \Box\neg C2 \vee \Diamond Set2$	1, 2, Lemma 12.26(c)
4.	$\vdash Set2 \rightarrow \Diamond(Test2 \wedge (Last = 2))$	Progress
5.	$\vdash \Box\neg C2 \vee \Diamond(Last = 2)$	3, 4, Theorem 12.10(e,b), PC
6.	$\vdash \Box(\Box\neg C2 \vee \Diamond(Last = 2))$	5, Generalization

∎

We now prove $Test1 \rightarrow \Diamond CS1$ from which the liveness of P1 follows by the progress axioms and Generalization; the proof of the liveness of P2 is symmetric. To assist in understanding the formal proof, it is followed by a line-by-line commentary.

Theorem 12.28 $\vdash Test1 \rightarrow \Diamond CS1$

Proof:

1.	$\vdash \Box Test1 \rightarrow \Diamond C2$	Progress
2.	$\vdash \Box Test1 \rightarrow \Diamond(Last = 2)$	1, Lemma 12.27(b)
3.	$\vdash \Box Test1 \rightarrow \Diamond\Box(Last = 2))$	2, Lemma 12.26(a), Progress
4.	$\vdash \Box Test1 \rightarrow \Diamond(Last = 1)$	Progress
5.	$\vdash \Box\Box Test1 \rightarrow \Box\Diamond(Last = 1)$	5, Generalization
6.	$\vdash \Box Test1 \rightarrow \Box\Diamond(Last = 1)$	6, Transitivity (11.12)
7.	$\vdash \Box Test1 \rightarrow \Box\Diamond((Last = 1) \wedge (Last = 2))$	3, 6, Theorem 12.12
8.	$\vdash \Box Test1 \rightarrow false$	7, PC
9.	$\vdash Test1 \rightarrow \Diamond CS1$	8, Progress

Commentary

1. If P1 attempts to execute the `while`-loop and remains at `Test1`, then C2 contained true.
2. From the Lemma and duality $\vdash \Diamond C2 \leftrightarrow \neg\Box\neg C2$.
3. If $\Box Test1$, `Set1` which sets `Last` to 1 can never be executed, so if $Last = 2$ it remains so.
4. As in line 1.
5–7. Temporal logic reasoning.
8. From $(Last = 1) \rightarrow \neg(Last = 2)$.
9. If P1 can't stay forever at `Test1`, it must eventually be at `CS1`.

∎

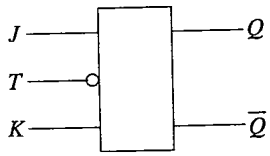
To prove mutual exclusion, we must prove that $\neg(at(CS1) \wedge at(CS2))$ is inductive. Unfortunately, it is not. We leave it as an exercise to prove that

$$[(Test1 \wedge CS2) \rightarrow (C2 \wedge Last = 1)] \wedge [(Test2 \wedge CS1) \rightarrow (C1 \wedge Last = 2)]$$

is inductive, and then to prove mutual exclusion.

Hardware specification

Temporal logic can be used to specify and verify hardware systems. Consider a clocked J-K flip-flop:



State changes occur on falling edges of the clock T, denoted by the proposition $T\downarrow$. The permissible state changes are specified by the following formulas:

$$T\downarrow \wedge (J=0) \wedge (K=1) \rightarrow \bigcirc(Q=0)$$
$$T\downarrow \wedge (J=1) \wedge (K=0) \rightarrow \bigcirc(Q=1)$$
$$T\downarrow \wedge (J=1) \wedge (K=1) \rightarrow (Q=v \rightarrow \bigcirc(Q=1-v))$$
$$\Box(Q=v \leftrightarrow \bar{Q}=1-v).$$

The first two formulas describe how the state of the flip-flop can be set by setting the values of the J and K lines and pulsing the clock. The third formula specifies that pulsing the clock with $J=K=1$ causes the value of Q to flip. Finally, the value of $\bar{Q}$ is always the complement of the value of Q. The requirement that the flip-flop not change state between pulses can be expressed using the binary operator *waiting-for*: $((Q=v) \rightarrow \bigcirc(Q=v))\ \mathcal{W}\ (T\downarrow)$.

12.5 Model checking*

The deductive proof of the correctness of a concurrent program is quite complex and demands a degree of ingenuity. For finite-state programs like Peterson's algorithm, there are only a finite number of states in which the program can be and the computations (paths) can be finitely presented, so it is feasible to actually construct an automaton describing all of the states and to check the automaton in order to prove that a property holds. This method, called *model checking*, is easily automated, and is often preferred to deductive methods when appropriate.

A state in Peterson's algorithm consists of the instruction counters of both processes, and the current values of `C1`, `C2` and `Last`. There are at most $5\cdot5\cdot2\cdot2\cdot2 = 200$ different states that can appear in any computation. We can construct a finite automaton whose transitions are the legal transitions that the program can take. For example, from state (`Test1`,`Test2`,`True`,`True`,1), the legal transitions are for process `P1` to return to the same state or for process `P2` to move to state (`Test1`,`CS2`,`True`,`True`,1).

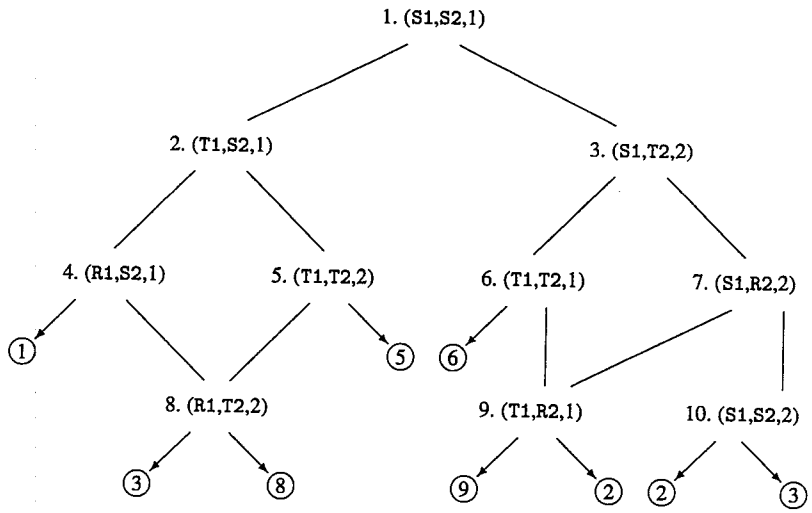


Figure 12.4 State automaton for Peterson's algorithm

In fact, we can significantly reduce the number of states. First, the values of `C1` and `C2` need not be included in the state, since by Theorem 12.26 (a,b), their values can be deduced from the instruction counters of the processes. A second simplification is to remove the statements `NC1`, `CS1`, `NC2` and `CS2`. It may seem strange to remove the statements which are the *raison d'être* of the algorithm, but they do not change or test the value of any variable, so they cannot affect the correctness of the algorithm. The mutual exclusion condition now becomes $\Box\neg(Reset1 \wedge Reset2)$, and the liveness conditions $\Box(Seti \rightarrow \Diamond Reseti)$. With these simplification, there are at most $3\cdot3\cdot2 = 18$ states in the automaton.

Algorithm 12.29 (Construction of a state automaton)
Input: A concurrent program.
Output: A state automaton for the program.

Start with the initial state. Choose a state whose transitions have not be created. For each process, create the new state that results from the execution of the current statement of the process, and create a transition (labeled by the process) to the new state. If the new state already exists, create a transition to the existing state. Terminate when all states have transitions. □

Example 12.30 Figure 12.4 gives the state automaton for Peterson's algorithm. The statement names are abbreviated and left (right) arrows are implicitly labeled `P1` (`P2`). Note that only ten of the eighteen possible states occur in any computation. □

A quick check of the automaton shows that it contains neither (R1,R2,1) nor (R1,R2,2). We can immediately conclude that mutual exclusion is preserved, because any state that could possibly be reached appears in the automaton.

Implementation

The software archive contains a program that creates the state automaton for Peterson's algorithm. The predicate `state` has six arguments: the first five are the location counters of `P1` and `P2`, the values of `C1`, `C2` and `Last`. (Since the computer is doing the work, we give also the values of `C1` and `C2`.) A database of states is maintained so that a new state can be identified with an existing identical state. For this purpose, the sixth argument of `state` is the identification number of the state. For each statement in the program, predicates `next1` and `next2` describe the state transition if `P1` and `P2`, respectively, are executed. Here is the procedure for `next1`; the code for `next2` is similar.

```
next1(state(set1,   P2,  _, C2,    _, N),
      state(test1,  P2,  1, C2,    1, N)).
next1(state(test1,  P2, C1,  0, Last, N),
      state(reset1, P2, C1,  0, Last, N)).


next1(state(test1,  P2, C1, C2,    2, N),
      state(reset1, P2, C1, C2,    2, N)).
next1(state(reset1, P2,  _, C2, Last, N),
      state(set1,   P2,  0, C2, Last, N)).
```

When a new state is created it is `assert`'ed into the database and the program recursively attempts to find a state accessible to the new state.

We leave it as a project to extend the program so that it can automatically create the transition database from a program.

Model checking tableaux

What about liveness properties like $\Box(Set1 \rightarrow \Diamond Reset1)$? Recall that we checked the satisfiability of a PTL formula by constructing a semantic tableau and checking that all future formulas are fulfilled. Here we do the same, except that we cannot consider all possible structures, but only structures that are consistent with state automaton of the algorithm. This is done by simultaneously building both the state automaton and the semantic tableau.

One technical detail before we give the construction: rather than build the tableau *rule by rule*, we build the tableau *state by state* by implicitly performing all α- and β-rules

and explicitly constructing the states from X-rules and state paths. In the presence of β-formulas, of course, a state in the tableau will have more than one successor.

Algorithm 12.31 (Model checking tableau)
Input: A finite-state program P and a PTL formula A.
Output: A model checking tableau for A.

Let P be a finite-state program and A a PTL formula. A node consists of a pair (s_i, U_i) where s_i is a state in the computation of P and U_i is a set of PTL formulas. The initial node is $(s_0, \neg A)$. Choose a node (s_i, U_i) and create as its children all nodes (s_j, U_j) where s_j is a legal successor of s_i in P and U_j is a successor of U_i according to the tableau rules. If (s_j, U_j) is not consistent, do not create the node. If (s_j, U_j) is the same as an existing node, connect (s_i, U_i) to the existing node. Terminate when no new nodes can be produced. □

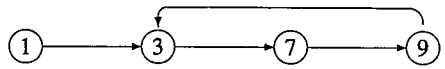
Let us construct the tableau for Peterson's algorithm and $\Box(Set1 \rightarrow \Diamond Reset1)$. The negation of the formula can be written $\Diamond(Set1 \wedge \Box \neg Reset1)$, which we will abbreviate by $\Diamond(S1 \wedge \Box \neg R1)$. By the β-rule, the initial state in the *semantic* tableau will have two successors, one with $S1 \wedge \Box \neg R1$ and one with $\bigcirc\Diamond(S1 \wedge \Box \neg R1)$. Because of the α-rules, the state with $S1 \wedge \Box \neg R1$ will also contain $S1$, $\Box \neg R1$, $\neg R1$ and $\bigcirc\Box \neg R1$. Thus the model checking tableau will have two initial states:

$$((\mathrm{S1}, \mathrm{S2}, 1), \{S1 \wedge \Box \neg R1, S1, \Box \neg R1, \neg R1, \bigcirc\Box \neg R1\})$$
$$((\mathrm{S1}, \mathrm{S2}, 1), \{\bigcirc\Diamond(S1 \wedge \Box \neg R1)\}).$$

Figure 12.5 shows the model checking tableau constructed from the first one, where, to avoid cluttering the diagram, the formulas that are created by the α-rules are not explicitly written. The transitions marked × are those that cannot be taken because they lead to inconsistent nodes. For example, executing `Test1` from the states $(T1, S2, 1)$ or $(T1, T2, 2)$ leads to a state with the instruction counter of `P1` is at `R1`, which is clearly inconsistent with $\Box \neg R1$.

The start of the tableau for the second initial state is shown in Figure 12.6. When the construction is continued, nodes 12 and 14 containing $S1 \wedge \Box \neg R1$ can be connected to nodes in the tableau in Figure 12.5, while nodes containing $\bigcirc\Diamond(S1 \wedge \Box \neg R1)$ will give rise to more nodes containing the same formula.

So, does the construction prove that $\Diamond(S1 \wedge \Box \neg R1)$ can be satisfied in a computation of Peterson's algorithm? The tableau does not close, so we have to check if the future formula is fulfilled in a MSCC. In fact it is; the following MSCC from Figure 12.5 fulfills $\Diamond(S1 \wedge \Box \neg R1)$:



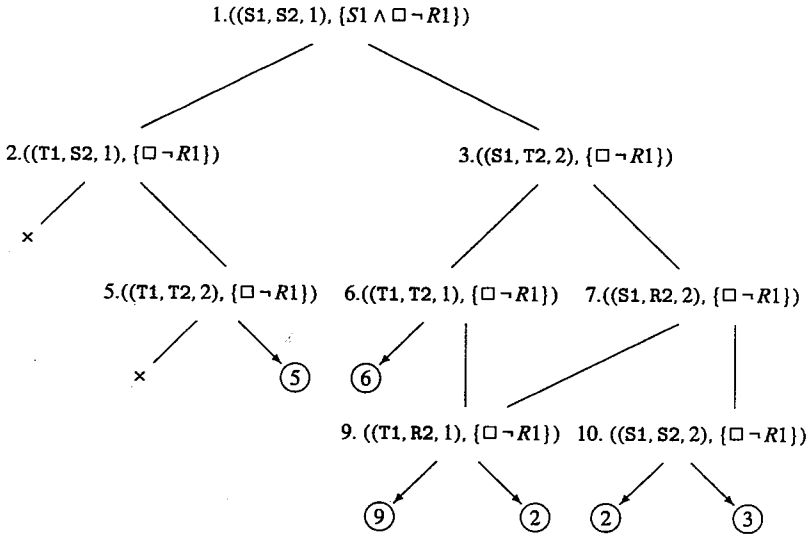


Figure 12.5 Model checking tableau for Peterson's algorithm, first part

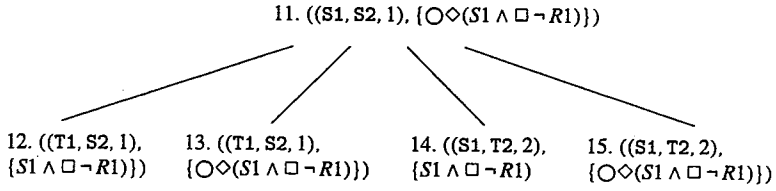


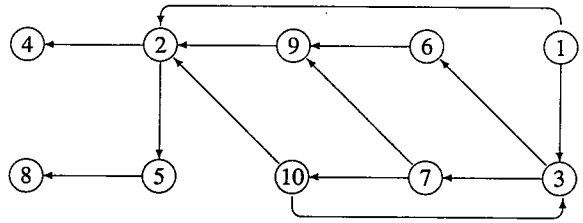
Figure 12.6 Model checking tableau for Peterson's algorithm, second part

so we seem to have a counterexample to the liveness property. Something must be wrong because we proved the liveness property deductively in a previous section.

In the deductive proof, we used progress axioms such as $Set1 \rightarrow \Diamond Test1$ which ensured the fairness of the computation. The model checking algorithm must be extended, not only to check for an open tableau and fulfilling of future formulas, but for fair computations. The 'fulfilling' path that we found is not fair because P1 is never allowed to execute a statement; instead, it remains forever at Set1. We refer you to Manna & Pnueli (1992,1995) for details on including fairness in the model checking algorithm.

Model checking in branching time

In a branching-time temporal logic, model checking is easier because we can work directly from the state automaton. For example, we can show that $\forall \Diamond R1$, that is, on any path, you can go somewhere where P1 is in its critical section. This is checked by incrementally building the set of states that satisfy $\forall \Diamond R1$, starting with the set of states that satisfy $R1$, $\{4, 8\}$ in Figure 12.4, and then working backwards. State 5 is included because all its successor states are in the set $\{8\} \subseteq \{4, 8\}$, then state 2 is included because $\{4, 5\} \subseteq \{4, 8, 5\}$. The set of states that results from this algorithm is shown below:



We can see that any *fair* path must eventually lead to states $\{4, 8\}$. The fairness restriction rejects the path $1 \rightsquigarrow (3 \rightsquigarrow 7 \rightsquigarrow 10)^*$.

*Symbolic model checking**

The size of a state automaton or model checking tableau depends on the size of the domain of the variables in the program. Peterson's algorithm has a single two-valued variable Last, but many algorithms and hardware devices have 32-bit variables, causing the number of states to be explosively large. In *symbolic model checking*, the states are not represented individually; rather, the entire automaton or tableau is represented by a propositional formula. By storing the formula as a BDD and using the BDD algorithms to manipulate it, the tableau construction is efficient and practical for extremely large numbers of states. In this section, we demonstrate the basic ideas involved by

describing how a semantic tableau for a temporal formula can be implemented using BDDs.

Consider the construction of a semantic tableau for the formula $A = \Box\Diamond p \vee \Diamond\Box\neg p$. A semantic tableau is, in effect, a search for a satisfying interpretation. Just by examining A, we see that there are four possible interpretations, depending on the assignment of *true* or *false* to its two subformulas; obviously, if both are assigned *false*, the formula is *false*, otherwise, it is *true*. However, the subformulas are not simply propositions that can arbitrarily be assigned a truth value. To satisfy A, the decompositions of the two subformulas into

$$\begin{aligned} A_1 &= (\Box\Diamond p \leftrightarrow \Diamond p \wedge \bigcirc\Box\Diamond p), \\ A_2 &= (\Diamond\Box\neg p \leftrightarrow \Box\neg p \vee \bigcirc\Diamond\Box\neg p), \end{aligned}$$

must also be taken into account because the assignments to $\Box\Diamond p$ and $\Diamond\Box\neg p$ are not independent. These, in turn, have their own decompositions that must be taken into account:

$$\begin{aligned} A_3 &= (\Diamond p \leftrightarrow p \vee \bigcirc\Diamond p), \\ A_4 &= (\Box\neg p \leftrightarrow \neg p \wedge \bigcirc\Box\neg p). \end{aligned}$$

We now have a set of subformulas that must be assigned truth values:

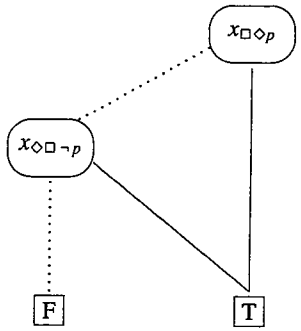
$$\Box\Diamond p,\ \Diamond\Box\neg p,\ \Diamond p,\ \Box\neg p,\ p,\ \neg p,\ \bigcirc\Box\Diamond p,\ \bigcirc\Diamond\Box\neg p,\ \bigcirc\Box\neg p.$$

An assignment must satisfy A as well as the four decompositions A_1, A_2, A_3, A_4.

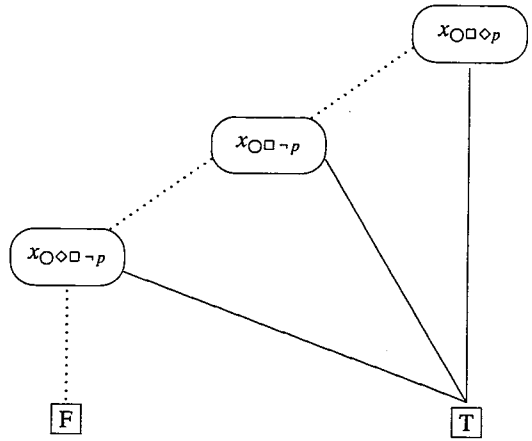
The set of subformulas can be divided into two subsets: the first six are subformulas that must be assigned truth values in the current (first) state. The other three subformulas will be assigned truth values in the next state. In fact, only five assignments are needed in the current state, since the assignment must be consistent, so an assignment to p determines the assignment to $\neg p$. The tableau construction requires us to create new states for each of these $2^5 = 32$ assignments by partially evaluating $A' = A \wedge A_1 \wedge A_2 \wedge A_3 \wedge A_4$, which will result in formulas to be satisfiable in the next states (after removing the $\bigcirc$ operator). A will be satisfiable if *there exists* an assignment for which the partial evaluation of A' is satisfiable in the next state.

By representing each subformula as a separate atomic proposition, the satisfiability of A is reduced to the satisfiability of $A'' = \exists x_{\Box\Diamond p}\, \exists x_{\Diamond p}\, \exists x_p\, \exists x_{\Diamond\Box\neg p}\, \exists x_{\Box\neg p}\, A$, where the existential operators are propositional quantification over the atomic propositions representing the subformulas. If we simplify A'' using the decomposition, we get the formula $\bigcirc\Box\Diamond p \vee \bigcirc\Diamond\Box\neg p \vee \bigcirc\Box\neg p$ from which we know that $\Box\Diamond p \vee \Diamond\Box\neg p \vee \Box\neg p$ must label the next state in the semantic tableau.

It seems that not much has been accomplished, trading a large set of subformulas for a complicated propositional formula. But we know of a generally efficient representation for propositional formulas, namely, ordered BDDs. Here is the obvious representation of A as a BDD:



Similarly, BDDs can be constructed for each of $A_1, \ldots, A_4$ and using the apply operation, A' can be easily computed. Recall now the definition of propositional quantification (Definition 4.63), and its computation using restriction (Theorem 4.64): $\exists p A = A|_{p=F} \vee A|_{p=T}$. The BDD for the formula A'' that specifies what must be true in the next state can be computed using repeated quantification on the BDD for A'. The result will be the BDD for $\bigcirc\Box\Diamond p \vee \bigcirc\Diamond\Box\neg p \vee \bigcirc\Box\neg p$:



By removing the next operators, we get a BDD for the formula that must be satisfied by all successor states. The tableau is obtained by performing the disjunction operation accumulatively on the BDDs for each state that it is constructed. An additional algorithm must be used to ensure that the structure is fulfilling. These algorithms are beyond the scope of this book.

Implementation

The source archive contains a Prolog program that constructs a BDD from a temporal formula, and constructs the BDD of the structure using BDDs for the inductive formu-

las. A database is constructed which assigns to each atomic and temporal subformula a global number representing an element of a sequence of propositional letters. Then the algorithms from Section 4.3 can be applied. The propositional numbers are translated back to subformulas for output.

The only non-trivial part of the implementation is removing the next operator from the BDD of the new state. It is not sufficient simply to remove the operator from the formulas on the nodes of the BDD, because the ordering of the propositional letters in the resultant BDD may not be consistent with the original ordering of the non-next formulas. The simplest solution is to transform the BDD to a formula and then back to a BDD; a second transformation will use the original ordering in the database.

Model checkers

Programs have been built that implement symbolic algorithms for model checking in both branching-time (SMV) and linear-time (TLV). The programs are easy to use: a simple programming language is used to specify the system which is compiled into a finite automaton, and the property is entered in temporal logic. The construction of the formulas, states and BDDs is normally hidden from the user. Symbolic model checkers are not just research tools; they have been used to verify temporal properties of extremely large systems.

12.6 Exercises

1. Prove $\vdash \Box(p \wedge q) \rightarrow (\Box p \wedge \Box q)$ and $\vdash \bigcirc(p \vee q) \leftrightarrow (\bigcirc p \vee \bigcirc q)$ (Theorem 12.7).
2. Prove the future formulas in Theorem 12.10.
3. Prove that Axioms 2, 3 and 6 are valid.
4. Prove $\vdash \Diamond\Box\Diamond p \leftrightarrow \Box\Diamond p$ (Theorem 12.11) and $\vdash \Diamond\Box p \rightarrow \Box\Diamond p$ (Theorem 12.12).
5. Prove $\vdash \Box(\Box\Diamond p \rightarrow \Diamond q) \leftrightarrow (\Box\Diamond q \vee \Diamond\Box \neg p)$.
6. Fill in the details of the proof of $\vdash \Box((p \vee \Box q) \wedge (\Box p \vee q)) \leftrightarrow (\Box p \vee \Box q)$ (Theorem 12.14).
7. * Prove the properties of the precedence operators in Theorem 12.20.
8. * Show that the branching time operators are not independent:

$$\models \exists \Diamond p \leftrightarrow \neg \forall \Box \neg p \qquad\qquad \models \forall \Diamond p \leftrightarrow \neg \exists \Box \neg p.$$

9. * Show that $\models \ominus\Diamond p \leftrightarrow \Diamond\ominus p$ and $\models (\Box p \vee \Box q) \leftrightarrow \Box(\boxminus p \vee \boxminus q)$.

10. Prove the invariants of Peterson's algorithm (Theorem 12.26).
11. Prove that the following formula is inductive:

$$[(Test1 \wedge CS2) \rightarrow (C2 \wedge Last = 1)] \wedge [(Test2 \wedge CS1) \rightarrow (C1 \wedge Last = 2)].$$

12. * Prove the correctness of Dekker's algorithm for mutual exclusion (see Ben-Ari (1990)).
13. * An asynchronous J-K flip-flop is like the clocked J-K flip-flop except that it has two additional input signals P and C (representing *Preset* and *Clear*). Write temporal logic formulas to formalize the following informal specification of their behavior.
    - If $P = C = 1$, the behavior is like a clocked J-K flip-flop.
    - $P = C = 0$ is forbidden.
    - $P = 0, C = 1$ causes $Q = 1$.
    - $P = 1, C = 0$ causes $Q = 0$.
14. * Construct the model checking tableau from the state

$$((S1, S2, 1), \{\bigcirc\Diamond(S1 \wedge \Box \neg R1)\}).$$

A Set Theory

If mathematical logic is the formal study of reasoning about mathematical objects, set theory is the formal study of the basic objects themselves. Because of its importance in the study of the foundations of mathematics, set theory is often presented in logic textbooks, especially the more advanced ones. In this book, we use results of elementary set theory, as summarized in this appendix. For an elementary, but detailed, development of set theory see Velleman (1994).

A.1 Finite and infinite sets

We assume the concepts of set and element as undefined.

Definition A.1 A *set* is composed of *elements*. Notation: $a \in S$, a is an element of set S, and $a \notin S$, a is *not* an element of S. The set with no elements is called the *empty set*, denoted $\emptyset$. □

S, T and U will be used to denote sets.

Specific sets will be defined in one of two ways. Either we may explicitly write the elements comprising the set, or we may use *set comprehension*, describing the set as all possible elements which satisfy a condition.

Example A.2 The following examples show how to describe sets:

- The set of colors of a traffic light is $\{red, yellow, green\}$. Braces are used to denote a set.
- The set of atomic elements is $\{hydrogen, helium, lithium, \ldots\}$. If a set is large and it is clearly understood what its elements are, ellipsis '...' is used to indicate the elements not explicitly listed.
- The set of *integers* $\mathcal{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. If the number of elements of a set is infinite, ellipsis is necessary to indicate elements missing from the explicit list.

- The set of *natural numbers* $\mathcal{N} = \{0, 1, 2, \ldots\} = \mathcal{N} = \{n \mid n \in \mathcal{Z} \text{ and } n \geq 0\}$, using set comprehension. The notation is read: $\mathcal{N}$ is the set of all n such that n is an integer and $n \geq 0$.
- The set of even natural numbers $\mathcal{E}v = \{n \mid n \in \mathcal{N} \text{ and } n \bmod 2 = 0\}$.
- The set of prime numbers is $\mathcal{P}r = \{n \mid P(n)\}$, where $P(n)$ is the condition: $n \in \mathcal{N}$, $n \geq 2$ and no positive integers except 1 and n divide n.

□

Note that there is no meaning to order or repetition in the definition of a set, since an element is either in a given set or it is not in the set: $\{3, 2, 1, 1, 2, 3\} = \{1, 2, 3\}$.

A.2 Set operators

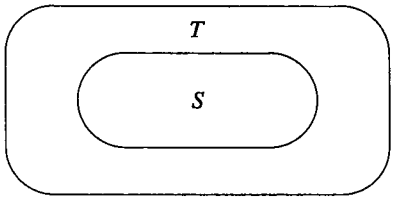
Definition A.3 S is a *subset* of T, denoted $S \subseteq T$, iff every element of S is an element of T. S is a *proper subset* of T, denoted $S \subset T$, iff $S \subseteq T$ and $S \neq T$. □

Example A.4 $\mathcal{N} \subset \mathcal{Z}$, $\mathcal{E}v \subset \mathcal{N}$, $\{red, green\} \subset \{red, yellow, green\}$. □

Theorem A.5 $\emptyset \subseteq T$.

The intuition behind $\emptyset \subseteq T$ is as follows. To prove $S \subseteq T$, take each element of S and ensure that it is also an element of T. Since there are no elements in $\emptyset$, the statement is vacuously true.

The relationships among several sets can be graphically shown by the use of *Venn diagrams*. These are closed curves drawn in the plane and labeled with the name of a set. A point is in the set if it is within the interior of the curve. In the following diagram, since every point within S is within T, S is a subset of T.



Theorem A.6 *The subset property is transitive:*

If $S \subseteq T$ and $T \subseteq U$ then $S \subseteq U$.

If $S \subset T$ and $T \subseteq U$ then $S \subset U$.

If $S \subseteq T$ and $T \subset U$ then $S \subset U$.

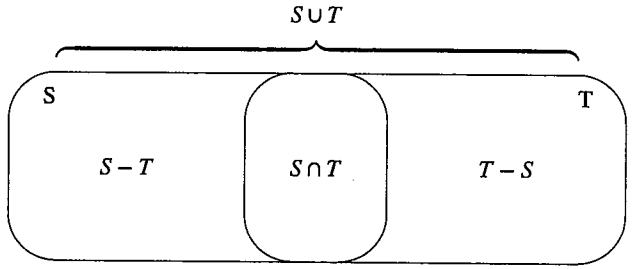
If $S \subset T$ and $T \subset U$ then $S \subset U$.

Definition A.7 The set operators are:

- $S \cup T$, the *union* of S and T, is the set consisting of those elements which are elements of *either S or T*.
- $S \cap T$, the *intersection* of S and T, is the set consisting of those elements which are elements of *both S and T*. If $S \cap T = 0$ then S and T are *disjoint*.
- $S - T$, the *difference* of S and T, is the set consisting of those elements which are elements of S but not of T.
- If S is understood as a universal set, $\bar{T}$, the *complement* of T, is $S - T$.

□

The following Venn diagram illustrates these concepts.



Example A.8 Here are some examples of operations on sets:

$$\begin{aligned}
\{red, yellow\} \cup \{red, green\} &= \{red, yellow, green\} \\
\{red, yellow\} \cap \{red, green\} &= \{red\} \\
\{red, yellow\} - \{red, green\} &= \{yellow\} \\
\mathcal{P}r \cap \mathcal{E}v &= \{2\} \\
\mathcal{P}r \cap \mathcal{N} &= \mathcal{P}r \\
\mathcal{P}r \cup \mathcal{N} &= \mathcal{N}.
\end{aligned}$$

□

Theorem A.9 *The operators $\cup$ and $\cap$ are commutative, associative and distributive:*

$$\begin{aligned} S \cup T &= T \cup S \\ S \cap T &= T \cap S \\ (S \cup T) \cup U &= S \cup (T \cup U) \\ (S \cap T) \cap U &= S \cap (T \cap U) \\ S \cup (T \cap U) &= (S \cup T) \cap (S \cup U) \\ S \cap (T \cup U) &= (S \cap T) \cup (S \cap U). \end{aligned}$$

Theorem A.10 *Properties of the set operators:*

$S = T$ iff $S \subseteq T$ and $T \subseteq S$.

$T = (T - S) \cup (S \cap T)$.

If $S \subseteq T$ then $S \cap T = S$, $S \cup T = T$ and $S - T = \emptyset$.

If S and T are disjoint then $S - T = S$.

$S \cup \emptyset = S$, $S \cap \emptyset = \emptyset$, $S - \emptyset = S$.

A.3 Ordered sets

If we impose an order on the elements of a set, we obtain an ordered set, denoted $(a_1, \ldots, a_n)$ using parentheses rather than braces.

Definition A.11 A finite ordered set of n elements is called an *n-tuple* or a *(finite) sequence of length n*. A 2-tuple is called a *pair*, a 3-tuple is called a *triple* and a 4-tuple is called a *quadruple*. A infinite ordered set is called an *infinite sequence*. □

Example A.12 Examples of ordered sets:

- A triple: *(red, yellow, green)*.
- A different triple: *(red, green, yellow)*.
- A 1-tuple: *(red)*, which is not the same as the element *red*.
- A triple with repeated elements: *(red, green, green)*.
- An infinite sequence: $(1, 2, 2, 3, 3, 3, 4, 4, 4, 4, \ldots)$.
- A pair whose elements are themselves sets: $(\mathcal{P}r, \mathcal{N})$.

□

Definition A.13 $S \times T$, the *Cartesian product* of S and T, is the set of all ordered pairs whose first element is from S and whose second element is from T. In general, given sets $S_1, \ldots, S_n$, the Cartesian product, $S_1 \times \cdots \times S_n$, is the set of ordered n-tuples whose i-th element is in S_i. If all the sets S_i are the same set S, the notation S^n is used. □

Example A.14 $\mathcal{N} \times \mathcal{N} = \mathcal{N}^2$ is the set of all ordered pairs of natural numbers. $\mathcal{N} \times \{red, yellow, green\}$ is the set of all pairs whose first element is a number and whose second is a color. This could be used to represent the color of a traffic light at points of time. □

A.4 Relations and functions

Definition A.15 An *n-ary relation* $\mathcal{R}$ is a subset of $S_1 \times \cdots \times S_n$. $\mathcal{R}$ is said to be a relation *on* $S_1 \times \cdots \times S_n$. □

A relation over a set S is just a subset of S.

Example A.16 Here are examples of relations over $\mathcal{N}^n$ for $n = 1, 2, 2, 3, 4$.

- The set of prime numbers $\mathcal{P}r = \{2, 3, 5, 7, 11, \ldots\}$.
- The set of pairs (x, y) such that y is the square of x:

$$\mathcal{S}q = \{(1, 1), (2, 4), (3, 9), (4, 16), \ldots\}.$$

- The set of pairs (x, y) such that x and y are relatively prime (that is, x and y have no common factor except 1):

$$\mathcal{R}p = \{(2, 3), (2, 5), (2, 7), (2, 9), \ldots, (3, 2), (3, 4), \ldots\}.$$

- The set of triples (x, y, z) such that $x^2 + y^2 = z^2$:

$$\mathcal{R}t = \{(3, 4, 5), (6, 8, 10), \ldots\}.$$

- The set of quadruples (x, y, z, n), $n > 2$ and $x^n + y^n = z^n$.
  $\mathcal{F} = \emptyset$ by Fermat's Last Theorem.

□

Definition A.17 Let R be a binary relation on S^2. R is *reflexive* iff $R(x, x)$ for all $x \in S$. R is *symmetric* iff $R(x_1, x_2)$ implies $R(x_2, x_1)$. R is *transitive* iff $R(x_1, x_2)$ and $R(x_2, x_3)$ imply $R(x_1, x_3)$.

R^*, the *reflexive transitive closure*, is defined as follows:

- If $R(x_1, x_2)$ then $R^*(x_1, x_2)$.
- $R^*(x_i, x_i)$ for all $x_i \in S$.
- $R^*(x_1, x_2)$ and $R^*(x_2, x_3)$ imply $R^*(x_1, x_3)$.

□

Example A.18 Let C be the relation on the set of ordered pairs of strings (s_1, s_2) such that $s_1 = s_2$, $s_1 = c \cdot s_2$, or $s_1 = s_2 \cdot c$, for some c in the underlying character set. Then C^* is the substring relation between strings. □

The relation Sq is special in that given the first argument x, there is at most one element y such that $Sq(x, y)$.

Definition A.19 Let $\mathcal{F}$ be a relation on $S_1 \times \cdots \times S_n$. $\mathcal{F}$ is a *function* iff for every ordered $n-1$-tuple $(x_1, \ldots, x_{n-1}) \in S_1 \times \cdots \times S_{n-1}$, there is at most one $x_n \in S_n$, such that $\mathcal{F}(x_1, \ldots, x_n)$. The notation $x_n = \mathcal{F}(x_1, \ldots, x_{n-1})$ is used.

The *domain* of $\mathcal{F}$ is the set of all $(x_1, \ldots, x_{n-1}) \in S_1 \times \cdots \times S_{n-1}$, for which (exactly one) $x_n = \mathcal{F}(x_1, \ldots, x_{n-1})$ exists. The *range* of $\mathcal{F}$ is the set of all $x_n \in S_n$ such that $x_n = \mathcal{F}(x_1, \ldots, x_{n-1})$ for at least one $(x_1, \ldots, x_{n-1})$. □

Definition A.20 $\mathcal{F}$ is *total* if the domain of $\mathcal{F}$ is (all of) $S_1 \times \cdots \times S_{n-1}$; otherwise, $\mathcal{F}$ is *partial*. $\mathcal{F}$ is *injective* or *one-to-one* iff $(x_1, \ldots, x_{n-1}) \neq (y_1, \ldots, y_{n-1})$ implies that $\mathcal{F}(x_1, \ldots, x_{n-1}) \neq \mathcal{F}(y_1, \ldots, y_{n-1})$. $\mathcal{F}$ is *surjective* or *onto* iff its range is (all of) S_n. $\mathcal{F}$ is *bijective* iff it is both injective and surjective. □

Example A.21 Sq is a total function on $\mathcal{N}^2$. Its domain is all of $\mathcal{N}$, but its range is only the subset of $\mathcal{N}$ consisting of all squares. Therefore Sq is not surjective and thus not bijective. The function is injective, because given an element in its range, there is exactly one (positive) square root, symbolically, $x \neq y \rightarrow x^2 \neq y^2$, or equivalently, $x^2 = y^2 \rightarrow x = y$. □

A.5 Cardinality

Definition A.22 The *cardinality* of a set is the number of elements in the set. The cardinality of a S is *finite* iff there is an integer i such that the number of elements in S is the same that the number of elements in the set $\{1, 2, \ldots, i\}$. Otherwise the cardinality is *infinite*. The cardinality of S is *countable* if it is the same as the cardinality of $\mathcal{N}$. Otherwise the cardinality is *uncountable*. □

How can we show that the number of elements in a set is countable? We must find a function that maps each number in $\mathcal{N}$ to an element of the set such that all elements of the set are covered by exactly one member of the mapping. Actually, this is the same method we use to decide how many elements there are in a finite set, namely by mapping a initial subset $\{1, \ldots, n\}$ of $\mathcal{N}$ to members of the set $\{a_1, \ldots, a_n\}$.

The theory of infinite sets is non-intuitive because an infinite set can have the same cardinality as a proper ('smaller') subset. For example, $\mathcal{E}v$, the set of even natural numbers is countable because it has the same number of elements as $\mathcal{N}$, but $\mathcal{E}v \subset \mathcal{N}$ because $\mathcal{N}$ also contains odd numbers. Simply map each $i \in \mathcal{N}$ to $2i$ and you will never run out of even numbers:

$$0 \mapsto 0,\ 1 \mapsto 2,\ 2 \mapsto 4,\ 3 \mapsto 6,\ \cdots.$$

Theorem A.23 (Cantor) *The set of real numbers is uncountable.*

To prove the theorem, assume that there exists a function that maps an element of $\mathcal{N}$ to each real number and obtain a contradiction by constructing a real number that is not in the list of matched real numbers.

Definition A.24 $\mathcal{P}(S)$, the *powerset* of S, is the set of all subsets of S. □

Example A.25 Here is the powerset of the finite set $\{red, yellow, green\}$:

$$\begin{aligned}\mathcal{P}(\{red, yellow, green\}) = \{ & \\ & \{red, yellow, green\}, \{red, yellow\}, \{red, green\}, \{yellow, green\}, \\ & \{red\}, \{yellow\}, \{green\}, \emptyset\}.\end{aligned}$$

□

Theorem A.26 *Let S be a set of cardinality N. Then the cardinality of $\mathcal{P}(S)$ is 2^N.*

Theorem A.27 $2^N \neq N$.

These theorems are easy to check for finite sets. For infinite sets, the theorems imply that there are infinitely many cardinalities. For most applications, however, finite and countably infinite sets are sufficient.

A.6 Proving properties of sets

To show that two sets are equal, use Theorem A.10 and show that each set is a subset of the other. To show that a set S is a subset of another set T, choose an *arbitrary* element $x \in S$ and show $x \in T$. This is also the way to prove a property $R(x)$ of a set S by showing that $S \subseteq T = \{x \mid R(x)\}$.

Example A.28
Theorem Let S be the set of prime numbers greater than 2. Then every element of S is odd.
Proof: Let n be an *arbitrary* element of S. If n is even, then $n = 2k$ for some k. Thus 2 is a factor of n. Since $n > 2$, this shows that n has a factor other than 1 and itself, so it cannot be a prime number. Thus the assumption that n is even is false. Since n was an arbitrary element of S, all elements of S are odd. ∎

□

Many proofs use *induction*. Given a sequence $S = (a_1, a_2, a_3, \ldots)$, we want to prove some statement about every element of S. The method of induction is:

- Prove the statement for the first element a_1. This is called the *base case*.
- Assume the statement for an arbitrary element a_i, and prove the statement for the next element a_{i+1}. This is called the *induction step* and the assumption is called the *inductive hypothesis*.

We can now conclude that the statement is true for all elements of S.

Example A.29
Theorem Every non-zero even number in $\mathcal{N}$ is the sum of two odd numbers.
Proof: The base case is trivial because $2 = 1 + 1$. The i-th element is $2i$ and by the inductive hypothesis it is the sum of two odd numbers $2i = (2j + 1) + (2k + 1)$. We have to show that the $i + 1$-st element $2(i + 1)$ is the sum of two odd numbers:

$$\begin{aligned} 2(i+1) &= 2i+2 \\ &= (2j+1)+(2k+1)+2 \\ &= (2j+1)+(2k+3) \\ &= (2j+1)+(2(k+1)+1). \end{aligned}$$

By the induction rule, we can now conclude that every element of the set is the sum of two odd numbers. ∎

□

The induction rule can be generalized to any mathematical structure which can be ordered—larger structures constructed out of smaller structures. The outline of the induction is still the same. Prove the base case for the smallest, indivisible structures, and then prove the induction step assuming an inductive hypothesis.

B Further Reading

Elementary logic

Students who find this book too challenging will benefit from studying Velleman (1994) who shows how to use formal deduction rules to construct natural-language proofs as they are used in mathematics textbooks. The book also contains a thorough grounding in naive set theory. Gries & Schneider (1993) integrates mathematical logic into a introductory course on discrete mathematics. Smullyan's puzzle books, starting with Smullyan (1978), are not only fascinating and entertaining, but can also help you understand advanced concepts in logic.

Mathematical logic

An excellent graduate text to follow this book is Nerode & Shore (1997). The book has an extensive bibliography as well as a modern discussion of the history of logic. A more theoretical approach to logic can be found in the classical textbook of Mendelson (1997), which is now in its fourth edition. Monk (1976) is an advanced graduate textbook with parts on recursion theory and model theory.

Smullyan (1995) was the direct inspiration for this book. It is a crystal-clear presentation of mathematical logic that uses tableaux as the unifying concept. For many years the book was out of print, but fortunately it has recently been reprinted. A modern development of logic based on analytic tableaux can be found in Fitting (1996).

Huth & Ryan (2000) is a logic textbook for computer science students that emphasizes natural deduction and modal, temporal and intuitionistic logics.

For more on ordered binary decision diagrams see Bryant (1986). His survey paper (Bryant 1992) contains an extensive bibliography.

Complexity

Harel (1985) is an informal introduction to decision procedures and complexity. Urquhart (1995) is a survey of complexity issues in the propositional calculus. The presentation of Tseitin's clauses is from Galil (1977). Dreben & Goldfarb (1979) and Lewis (1979)

are advanced monographs on the decidable and undecidable classes of predicate calculus formulas. The complexity of the predicate calculus is discussed in Statman (1978).

Resolution and logic programming

Lloyd (1987) is an excellent text on the theory of logic programming. Our presentation of the unification algorithm is taken from Martelli & Montanari (1982). Loveland (1978) is the classical textbook on resolution in automatic theorem proving. Wos (1996) is a modern book on automatic theorem proving. Prolog language texts are Clocksin & Mellish (1987) and Sterling & Shapiro (1994). Shapiro (1989) compares concurrent logic programming languages and contains an extensive bibliography. For constraint logic programming, see Van Hentenryck (1989) and Jaffar & Maher (1994).

Formalization of programs

Textbooks on program verification are Francez (1992) and Apt & Olderog (1991). Potter, Sinclair & Till (1996) and Diller (1994) are textbooks on Z; the reference manual is Spivey (1989).

Temporal logic

My textbook on concurrent programming (Ben-Ari 1990) contains elementary examples of the use of temporal logic in the verification of programs. Manna & Pnueli (1992,1995) is an extensive treatment of temporal logic and its use in the specification and verification of programs.

Classical modal logic is the subject of Hughes & Creswell (1981) and classical temporal logic is the subject of Rescher & Urquhart (1971). The use of semantic tableaux in temporal logic is adapted from the presentation in Ben-Ari, Manna & Pnueli (1983). For model checking see McMillan (1993) and Manna & Pnueli (1992,1995).

Bibliography

Apt, K. & Olderog, E.-R. (1991), *Verification of Sequential and Concurrent Programs*, Springer-Verlag, Berlin.

Ben-Ari, M. (1990), *Principles of Concurrent and Distributed Programming*, Prentice-Hall International, Hemel Hempstead.

Ben-Ari, M., Manna, Z. & Pnueli, A. (1983), 'The temporal logic of branching time', *Acta Informatica* **20**, 207–226.

Bryant, R. (1986), 'Graph-based algorithms for Boolean function manipulation', *IEEE Transactions on Computers* **C-35**, 677–691.

Bryant, R. (1992), 'Symbolic Boolean manipulation with ordered binary-decision diagrams', *ACM Computing Surveys* **24**, 293–318.

Clocksin, W. & Mellish, C. (1987), *Programming in PROLOG (Third Edition)*, Springer-Verlag, Berlin.

Diller, A. (1994), *Z: An Introduction to Formal Methods (Second edition)*, John Wiley & Sons, Chichester.

Dreben, B. & Goldfarb, W. (1979), *The Decision Problem: Solvable Classes of Quantificational Formulas*, Addison-Wesley, Reading, MA.

Even, S. (1979), *Graph Algorithms*, Computer Science Press, Potomac, MD.

Fitting, M. (1996), *First-order Logic and Automated Theorem Proving (Second edition)*, Springer-Verlag, New York, NY.

Francez, N. (1992), *Program Verification*, Addison-Wesley, Reading, MA.

Galil, Z. (1977), 'On the complexity of regular resolution and the davis-putnam procedure', *Theoretical Computer Science* **4**, 23–46.

Gries, D. & Schneider, F. (1993), *A Logical Approach to Discrete Math*, Springer-Verlag, New York, NY.

Harel, D. (1985), *Algorithmics: The Spirit of Computing*, Addison-Wesley, Reading, MA.

Hopcroft, J. & Ullman, J. (1979), *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA.

Hughes, G. & Creswell, M. (1981), *An Introduction to Modal Logic*, Methuen, London.

Huth, M. & Ryan, M. (2000), *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge University Press, Cambridge.

Jaffar, J. & Maher, M. (1994), 'Constraint logic programming: A survey', *Journal of Logic Programming* **19-20**, 503–581.

Lewis, H. (1979), *Unsolvable Classes of Quantificational Formulas*, Addison-Wesley, Reading, MA.

Lloyd, J. (1987), *Foundations of Logic Programming (Second edition)*, Springer-Verlag, Berlin.

Loveland, D. (1978), *Automated Theorem Proving: A Logical Basis*, North-Holland, Amsterdam.

Manna, Z. & Pnueli, A. (1992,1995), *The Temporal Logic of Reactive and Concurrent Systems. Vol. I: Specification. Vol. II: Safety*, Springer-Verlag, New York, NY.

Martelli, A. & Montanari, U. (1982), 'An efficient unification algorithm', *ACM Transactions on Programming Languages and Systems* **4**, 258–282.

McMillan, K. (1993), *Symbolic Model Checking*, Kluwer Academic Publishers, Dordrecht.

Mendelson, E. (1997), *Introduction to Mathematical Logic (Fourth edition)*, Chapman & Hall, London. Originally published by Van Nostrand Reinhold Company, 1965.

Minsky, M. (1967), *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, NJ.

Monk, J. (1976), *Mathematical Logic*, Springer-Verlag, New York, NY.

Nerode, A. & Shore, R. (1997), *Logic for Applications (Second edition)*, Springer-Verlag, New York, NY.

Potter, B., Sinclair, J. & Till, D. (1996), *An Introduction to Formal Specification and Z (Second edition)*, Prentice-Hall International, Hemel Hempstead.

Rescher, N. & Urquhart, A. (1971), *Temporal Logic*, Springer-Verlag, New York, NY.

Shapiro, E. (1989), 'The family of concurrent logic programming languages', *ACM Computing Surveys* **21**, 413–510.

Singh, S. (1997), *Fermat's Enigma : The Quest to Solve the World's Greatest Mathematical Problem*, Walker & Co., New York, NY.

Smullyan, R. (1978), *What is the Name of this Book?—The Riddle of Dracula and Other Logical Puzzles*, Prentice-Hall, Englewood Cliffs, NJ.

Smullyan, R. (1995), *First-order Logic*, Dover, New York, NY. First published in 1968 by Springer-Verlag, New York, NY.

Spivey, J. (1989), *The Z Notation: A Reference Manual*, Prentice-Hall International, Hemel Hempstead.

Statman, R. (1978), 'Bounds for proof-search and speed-up in the predicate calculus', *Annals of Mathematical Logic* **15**, 225–287.

Sterling, L. & Shapiro, E. (1994), *The Art of Prolog: Advanced Programming Techniques (Second edition)*, MIT Press, Cambridge, MA.

Urquhart, A. (1995), 'The complexity of propositional proofs', *Bulletin of Symbolic Logic* **1**, 425–465.

Van Hentenryck, P. (1989), *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, MA.

Velleman, D. (1994), *How to Prove It: A Structured Approach*, Cambridge University Press, Cambridge.

Wos, L. (1996), *The Automation of Reasoning: An Experimenter's Notebook with Otter Tutorial*, Academic Press, London.

Index of Symbols

Index