Texts and Monographs in Computer Science

Suad Alagic
Object-Oriented Database Programming
1989. XV, 320 pages, 84 illus.

Suad Alagic
Relational Database Technology
1986. XI, 259 pages, 114 illus.

Suad Alagic and Michael A. Arbib
The Design of Well-Structured and Correct Programs
1978. X, 292 pages, 68 illus.

S. Thomas Alexander
Adaptive Signal Processing: Theory and Applications
1986. IX, 179 pages, 42 illus.

Krzysztof R. Apt and Ernst-Rüdiger Olderog
Verification of Sequential and Concurrent Programs
1991. XVI, 441 pages

Michael A. Arbib, A.J. Kfoury, and Robert N. Moll
A Basis for Theoretical Computer Science
1981. VIII, 220 pages, 49 illus.

Friedrich L. Bauer and Hans Wössner
Algorithmic Language and Program Development
1982. XVI, 497 pages, 109 illus.

W. Bischofberger and G. Pomberger
Prototyping-Oriented Software Development: Concepts and Tools
1992. XI, 215 pages, 89 illus.

Ronald V. Book and Friedrich Otto
String-Rewriting Systems
1993. VII, 200 pages

Kaare Christian
A Guide to Modula-2
1986. XIX, 436 pages, 46 illus.

Edsger W. Dijkstra
Selected Writings on Computing: A Personal Perspective
1982. XVII, 362 pages, 13 illus.

(continued after index)

Logic for Applications

Anil Nerode
Richard A. Shore

With 66 Illustrations

Springer-Verlag
New York Berlin Heidelberg London Paris
Tokyo Hong Kong Barcelona Budapest

Anil Nerode
Department of Mathematics
Cornell University
White Hall
Ithaca, NY 14853-7901
USA

Richard A. Shore
Department of Mathematics
Cornell University
White Hall
Ithaca, NY 14853-7901
USA

Series Editors:
David Gries
Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853-7501
USA

Fred B. Schneider
Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853-7501
USA

Library of Congress Cataloging-in-Publication Data
Nerode, Anil
Logic for applications / Anil Nerode, Richard A. Shore.
p. cm. — (Texts and monographs in computer science)
Includes bibliographical references and indexes.
ISBN 0-387-94129-0. — ISBN 3-540-94129-0
1. Logic programming. 2. Logic, Symbolic and mathematical.
I. Shore, Richard A., 1946- . II. Title. III. Series.
QA76.63.N45 1993
005.1—dc20 93-27846

Printed on acid-free paper.

Production managed by Hal Henglein; manufacturing supervised by Genieve Shaw.
Photocomposed copy produced from the authors' AMSTeX files.
Printed and bound by Hamilton Printing Co., Castleton, NY.
Printed in the United States of America.

9 8 7 6 5 4 3 2 1

ISBN 0-387-94129-0 Springer-Verlag New York Berlin Heidelberg
ISBN 3-540-94129-0 Springer-Verlag Berlin Heidelberg New York

Preface

In writing this book, our goal was to produce a text suitable for a first course in mathematical logic more attuned than the traditional textbooks to the recent dramatic growth in the applications of logic to computer science. Thus our choice of topics has been heavily influenced by such applications. Of course, we cover the basic traditional topics — syntax, semantics, soundness, completeness and compactness — as well as a few more advanced results such as the theorems of Skolem–Löwenheim and Herbrand. Much of our book, however, deals with other less traditional topics. Resolution theorem proving plays a major role in our treatment of logic, especially in its application to Logic Programming and PROLOG. We deal extensively with the mathematical foundations of all three of these subjects. In addition, we include two chapters on nonclassical logics — modal and intuitionistic — that are becoming increasingly important in computer science. We develop the basic material on the syntax and semantics (via Kripke frames) for each of these logics. In both cases, our approach to formal proofs, soundness and completeness uses modifications of the same tableau method introduced for classical logic. We indicate how it can easily be adapted to various other special types of modal logics. A number of more advanced topics (including nonmonotonic logic) are also briefly introduced both in the nonclassical logic chapters and in the material on Logic Programming and PROLOG.

The intended audience for this text consists of upper level undergraduate and beginning graduate students of mathematics or computer science. We assume a basic background in abstract reasoning as would be provided by any beginning course in algebra or theoretical computer science, as well as the usual familiarity with informal mathematical notation and argument as would be used in any such course.

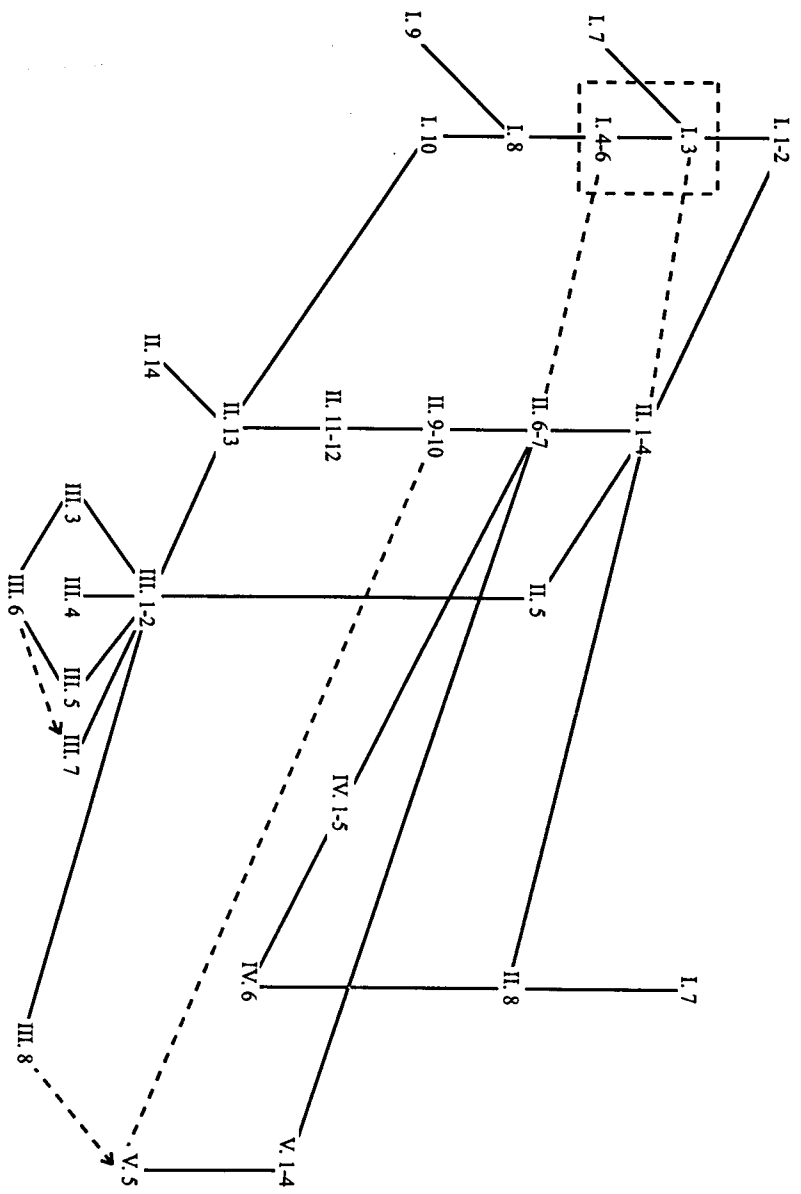
If taught as a course for advanced undergraduates, essentially all the material in Chapters I–III, together with a reasonable amount of programming in PROLOG, can be covered in one semester with three hours of lectures a week. When teaching it in this way, we have had (and recommend) an additional weekly section devoted to homework problems and programming instruction. Alternatively, the material on resolution theorem proving and Logic Programming can be replaced by the chapters on modal and intuitionistic logic to get a rather different course. For two quarters, one can simply add on one of the nonclassical logics to the first suggested semester course. We have deliberately made these two chapters entirely independent of one another so as to afford a choice. There is, however, much similarity

in the developments and, if both are covered, the corresponding sections of the second chapter can be covered more quickly. At the graduate level, essentially the whole book can be covered in a semester.

The text develops propositional logic in its entirety before proceeding to predicate logic. However, depending on the background of the class and the predilections of the instructor, it is possible to combine the treatments. Indeed, for a graduate course or with students with some previous exposure to propositional logic and truth tables, this may well be better. To follow such a path, begin with I.1–2 and then move on to II.1–4. If planning to introduce PROLOG and develop the foundations of Logic Programming, II.5 can be used to explain the syntax and semantics of PROLOG. To start the students on actual programming, an informal introduction to PROLOG implementation and programming can be continued in the recitation section (parts of I.10 are relevant here). Of course, the theoretical underpinnings must wait for the formal development of resolution and unification. With or without the Logic Programming, it is now possible (based on the tableau approach) to prove the soundness and completeness theorems for predicate logic in II.6–7 and to continue through the proof of Herbrand's theorem in II.10. A Hilbert–style proof system is presented without proving any results in I.7 and II.8.

The resolution–style proof system on which PROLOG is based is more intimately tied to its development in the propositional case, as the basic theorems are proven by using Herbrand's theorem to reduce predicate logic results to corresponding ones of propositional logic. Thus it is necessary, if covering this material, to do the propositional case first. The sections needed for the treatment of PROLOG are I.8, I.10 and II.11–13. The refinements of resolution considered in I.9 are strictly optional. While linear resolution (II.14) is highly relevant to PROLOG, the completeness theorem is one of the most difficult in the text. We have therefore provided in III.1 an alternate approach to the corresponding results for Horn logic and PROLOG that requires only the basic definitions of linear resolution (I.14.1–3).

A diagram of the logical dependencies between sections of the text is given on the facing page. Unless indicated otherwise by an arrow, the order of dependency runs right to left. Dotted lines (as from the propositional to predicate sections for classical logic) indicate relevance but not strict logical dependence. We should note that the very first section, I.1, simply collects definitions and facts about orderings and trees that are needed at various points in the text. This material can be covered at the beginning or inserted as needed. In particular, König's Lemma, I.1.4, is not needed until the treatment of complete systematic tableaux in I.4. At various other points in the text, certain paragraphs or even whole sections that are not needed for the later material are marked as optional by an asterisk (*). They are also printed in smaller type and with wider margins to set them off from the main line of development. In addition, there are three important possible exceptions to the order indicated above; these are of particular relevance for those courses not including very much about PROLOG.



The first exception also needs a warning. Our basic approach to logic does not include a specialized equality predicate. Although no longer the common approach in texts on mathematical logic, this is the right way to develop the subject to do resolution theorem proving, Logic Programming and PROLOG. We have thus relegated to III.5 the analysis of equality, either as a special predicate with the privileged semantics of true equality and the corresponding logical axiom schemes, or as an ordinary one with the appropriate equality axioms added on to each system under consideration. It is, however, quite possible to cover the relevant material in III.5 up to III.5.3 and the proofs of the soundness and completeness theorems for equality interpretations described there immediately after II.7.

The second exception concerns the proof of Church's theorem on the undecidability of validity for predicate logic. In III.8 we present a proof designed to apply even to the fragment of predicate logic represented in PROLOG. In Exercise 3 of III.5, however, we indicate how the presentation can easily be modified to make no mention of PROLOG notation or procedures and so give a proof of Church's Theorem which is accessible after II.7.

Finally, the introduction to nonmonotonic logic given in III.7 up to III.7.6 can also be read independently of the material on Logic Programming and PROLOG. The rest of III.7 consists of an analysis of the stable models of Logic Programming with negation in terms of nonmonotonic logic. Other, more self–contained, applications to graphs and partial orders are given in Exercises 8–9 of III.7.

We should point out that there is a considerable overlap in the basic material for the development of modal and intuitionistic logic in Chapters IV and V. Indeed, a single unified development is possible albeit at some expense to the ease of intelligibility. We have instead written these chapters so that either may be read independently of the other. For the readers who wish to delve into both topics, we supply a comparative guide to basic notions of classical, modal and intuitionistic logic in V.6. We try there to point out the similarities and differences between these logics.

We have included a variety of problems at the end of almost every section of the text, including a fair number of programming problems in PROLOG which can be assigned either for theoretical analysis or actual implementation. In particular, there are series of problems based on a database consisting of the genealogical lists in the first few chapters of *Chronicles*. It is reproduced in Appendix B. We have included these problems to serve as paradigms for use with a similar database or to be appropriately modified to fit other situations. This is not, however, a text on PROLOG programming. When teaching this material, we always supplement it with one of the standard texts on PROLOG programming listed in the suggestions for further reading at the end of Chapter III. We used ARITY PROLOG and the printouts of program runs are from that implementation, but nothing in the text is actually tied to a particular version of the language; we use only standard syntax and discuss typical implementations.

When we (infrequently) cite results from the current literature, we attribute them as usual. However, as this is a basic textbook, we have made no attempt to attribute the standard results of the subject to their discoverers, other than when at times we name theorems according to common usage. We have, however, supplied a brief history of logic in an appendix that should give the student a feel for the development of the subject. In addition, suggestions for further reading that might be useful for either students or teachers using this text are given at the end of each chapter. Finally, a fairly extensive bibliography of related material, arranged by subject, is given at the end of the book.

Portions of this book appeared in various formats over many years. Very early versions of the material on classical logic appeared in lecture notes by Nerode which were distributed for courses at Cornell years ago. This material was also independently reworked by George Metakides and appeared as lecture notes in English with Nerode and in Greek as his lectures at the University of Patras. Nerode [1990, 4.2] and [1991, 4.4] contain preliminary versions of our treatment of intuitionistic and modal logic based on the tableau method which were presented in lectures at Montecatini Terme and Marktoberdorf in 1988 and 1989, respectively. Our approach to resolution was also influenced by courses on program verification given by Richard Platek at Cornell. More current versions of the material have been read and used over the past five years by a number of teachers in both mathematics and computer science departments and we have benefited considerably from their comments and suggestions. We should mention Uri Abraham (Mathematics, Ben Gurion University, Israel), John Crossley (Mathematics and Computer Science, Monash University, Australia), George Metakides (University of Patras, Greece and Information Technologies Research, EEC), Dexter Kozen (Computer Science, Cornell) and Jacob Plotkin (Mathematics, Michigan State University). Warren Goldfarb (Philosophy, Harvard) helped us avoid a number of pitfalls in the historical appendix. Particularly extensive (and highly beneficial) comments were received from Wiktor Marek (Computer Science, University of Kentucky), George Odifreddi (Computer Science, University of Turin, Italy) and Robert Soare (Mathematics and Computer Science, University of Chicago) who used several versions of the text in their courses. We also owe a debt to our graduate students who have served as assistants for our logic courses over the past few years and have made many corrections and suggestions: Jennifer Davoren, Steven Kautz, James Lipton, Sherry Marcus and Duminda Wijesekera.

We gratefully acknowledge the financial support over the past few years of the NSF under grants DMS–8601048 and DMS–8902797, the ARO under grants DAAG29–85–C–0018 and DAAL03–91–C–0027 through the Mathematical Sciences Institute at Cornell University, and IBM for an equipment grant through Project Ezra at Cornell University. We would also like to thank Arletta Havlik and Graeme Bailey for their help with the TEXing of the text, and Geraldine Brady, Jennifer Davoren and George Odifreddi for their help in proofreading.

Finally, in appreciation of their continuing support, we dedicate this book to our wives, Sally and Naomi.

Cornell University
Ithaca, NY
December, 1992

Anil Nerode
Richard A. Shore

Contents

Introduction

In 1920 logic was mostly a philosopher's garden. There were also a few mathematicians there, cultivating the logical roots of the mathematical tree. Today, Recursion Theory, Set Theory, Model Theory and Proof Theory, logic's major subdisciplines, have become full–fledged branches of mathematics. Since the 1970s, the winds of change have been blowing new seeds into the logic garden from computer science, AI, and linguistics. These winds have also uncovered a new topography with many prominences and depths, fertile soil for new logical subjects. These days, if you survey international meetings in computer science and linguistics, you will find that the language of mathematical logic is a lingua franca, that methods of mathematical logic are ubiquitous and that understanding new logics and finding feasible algorithms for implementing their inference procedures plays a central role in many disciplines. The emerging areas with an important logic component include imperative, declarative and functional programming; verification of programs; interactive, concurrent, distributed, fault tolerant and real time computing; knowledge–based systems; deductive databases; and VLSI design. Various types of logic are now also playing key roles in the modeling of reasoning in special fields from law to medicine.

These applications have widened the horizons of logical research to encompass problems and ideas that were not even considered when logic was motivated only by questions from mathematics and philosophy. Applied logic is now as much a reality as is applied mathematics, with a similarly broad, overlapping but somewhat different area of application. This situation has arisen because of the needs for automated inference in critical, real time, and large database information processing applications throughout business, government, science, and technology. Mathematical logic, coupled with some of its applications, should be as easily available to college and university students as is applied mathematics. It may well be as important to the future of many previously qualitative disciplines as ordinary applied mathematics has been to the traditionally quantitative ones.

This book is a rigorous elementary introduction to classical predicate logic emphasizing that deduction is a form of computation. We cover the standard topics of soundness, completeness and compactness: our proof methods produce only valid results, all valid sentences are provable and, if a fact is a logical consequence of an infinite set of axioms, it is actually a consequence of finitely many of them. The need for soundness seems obvious but, as we shall see in our discussion of PROLOG, even this requirement of simple correctness is often sacrificed on the altar of efficiency in

actual implementations. Completeness, on the other hand, is a remarkable result connecting proofs and validity. We can prescribe an effective proof procedure that precisely captures the semantics of first order logic. A valid sentence, i.e., one true for every interpretation of the relations used to state it, always has a proof in a particular formal system and there is an algorithm to find such a proof. Compactness also has surprising applications that deduce results about infinite structures from results about finite ones. To cite just one example, it implies that every planar map is colorable with four colors as every finite planar map is so colorable. We also prove that validity is undecidable: no single algorithm can decide if any given sentence is valid. Thus although we can, using a particular algorithm, search for a proof of a given sentence φ and be assured of finding one if φ is valid, we cannot know in general whether we are searching in vain.

Our treatment begins in Chapter I with the syntax and semantics of classical propositional logic, that is the logic of compound sentences formed with connectives such as "and", "or", "if" and "not" but without consideration of the quantifiers "for all" and "there exists". We present a traditional approach to syntax in terms of strings of symbols as well as one based on tree structures. As trees have become basic objects in many computer science areas, the latter approach may well be more accessible (or at least familiar) to many students. Either approach can be adopted. We then introduce the semantic tableau proof method developed by Beth (*Foundations of Mathematics* [1959, 3.2]) and Smullyan (*First order Logic* [1968, 3.2]) for propositional logic. We have found over the years that the tableaux method is the easiest for students to learn, use and remember. This method seeks to find a proof of a sentence φ by discovering that a systematic search for a counterexample to φ fails in a finite amount of time. The procedure brings out the unadorned reasons for completeness by directly analyzing the subformulas of the formula φ for which a proof is being attempted. It presents the systematic search as a tree–constructing algorithm. The goal of the algorithm is to produce a finite tree beginning with "φ is false" with a contradiction on every branch. Such a tree shows that every analysis of "φ is false" leads to a contradiction. We call this a tableau proof of φ. Employing a systematic search for tableau proofs, we prove the soundness, completeness and compactness theorems.

We then develop the resolution method of theorem proving introduced by J. A. Robinson [1965, 5.7]. This method has played a crucial role in the development of automated reasoning and theorem proving. After again establishing soundness and completeness, we specialize this method to Horn clauses to develop the mathematical foundations of Logic Programming and PROLOG (still at the propositional level). Logic Programming is a general abstract approach to programming as logical deduction in a restricted setting. PROLOG is a type of programming language designed to implement this idea that computations are deductions.

In Chapter II we introduce the rest of predicate logic (functions and relations; variables and quantifiers) with explanations of its syntax and

semantics. We present a tableau style proof system for predicate logic and prove its soundness and completeness. Our approach naturally leads to Herbrand's theorem which, in a certain sense, reduces predicate logic to propositional logic. Then, following Robinson, we add to resolution the pattern–matching algorithm, called unification, which is originally due to Herbrand. This produces Robinson's system of deduction for predicate logic; it was the first complete redesign of logical inference for the purpose of mechanization of inference on digital computers. It is really better carried out by machines than by hand. Robinson's work made automation of reasoning on digital computers a major area of research. Many of his ideas and much of his terminology have persisted to the present day.

Chapter III is devoted to the specialization of resolution to Horn clauses, a special class of predicate logic formulas that are the domain of Logic Programming and PROLOG. The predicate version of Logic Programming has applications to expert systems, intelligent databases and AI among many others. Logic Programming has a very active research community and has become a separate discipline. In addition to restricting its attention to a limited class of formulas, Logic Programming and PROLOG make various changes in proof procedures to attain computational efficiency. We cover the mathematical foundations of Horn clause logic and then of PROLOG: syntax, semantics, soundness and completeness. We also touch on proofs of termination for PROLOG programs. As an example of current trends, we give an introductory account of the so–called "general logic programs". We present views of implementation and semantics for negation in this setting in terms of both negation as failure and stable models. This area is still in considerable flux. It is one example of the larger evolving subject of nonmonotonic reasoning. Unlike the classical situation, in nonmonotonic logic the addition of new premises may force the withdrawal of conclusions deduced from the previous ones. We include a brief introduction to this area in III.7. We are not programmers, however, and do not attempt to really cover PROLOG programming beyond what is needed to illustrate the underlying logical and mathematical ideas. (References to basic books on PROLOG programming are included in the bibliography.) We do, however, deal with theoretical computability by PROLOG programs as our route to undecidability.

Standard proofs of undecidability for a theory come down to showing how to represent each effectively computable function by a logical formula so that computing the values of the function amounts to deducing instances of that formula. The noncomputability of specific functions such as the halting problem (deciding if a given program halts on a given input) are then translated into the impossibility of deciding the provability of given formulas. In this way, we prove the undecidability of PROLOG and of Horn clause logic (and so *a fortiori* of all of predicate logic) by showing that Horn clause programs and even standard implementations of PROLOG compute all effectively computable functions. As a definition of an algorithm for an effective computation, we use the model of computation given by programs on register machines. Thus we simulate each register machine program for

computing a recursive function by a PROLOG program computing a coded version of that same function. As it is known that all other models of computation can be simulated by register machines, this suffices to get the desired results on computability and undecidability.

For the final chapters, we turn to some nonclassical logics that are becoming increasingly important in understanding and modeling computation and in verifying programs. "Nonclassical" has a technical meaning: the truth of a composite sentence may not depend solely on the truth of its parts and, indeed, even the truth of simple statements may depend on context, time, beliefs, etc. Although this attitude is not the traditional one in mathematics, it reflects many real life situations as well as many important problems in computer science. The truth of an implication often has temporal components. Usually sentences are evaluated within some context. If our knowledge or beliefs change, so may our evaluation of the truth of some sentence. The analysis of programs depends on the states of knowledge of the computer over time, on what may happen and on what must happen. We touch briefly on one form of such logic (nonmonotonic logic in which later information may invalidate earlier conclusions) in Chapter III. The last two chapters are devoted to a systematic study of two such logics: modal and intuitionistic.

Intuitionism incorporates a constructive view of mathematics into the underlying logic. We can claim that we have a proof of A or B only if we have a proof of one of them. We can claim to have a proof of "there exists an x with property P" only if we can actually exhibit an object c and a proof that c has property P. Modal logic attempts to capture notions of necessity and possibility to serve as a basis for the analyses of systems with temporal, dynamic or belief–based components. We describe the semantics of both of these logics in terms of Kripke frames. These are sets of classical models together with a partial ordering or some other relation on the models; it is this relation that embodies the nonclassical aspects of Kripke semantics. We then formulate tableau systems that generalize the classical ones and faithfully reflect the semantics expressed by Kripke frames. Once again, soundness and completeness play a central role in our exposition. The two logics are presented independently but a comparative guide is supplied in V.6.

For good or ill, the philosophical tenets of intuitionism play no role here, nor do the philosophers' analyses of time and necessity. Rather, we explain Kripke frames as a way of modeling the notion of a consequence of partial information and modal operators as simply expressing relations among sets of models. This explanation fits the prospective use of intuitionistic logic, as Scott has suggested, as a language for Scott domains and information systems, or for Horn clause logic, which is a subtler use of both classical and intuitionistic logic. It also fits the applications of modal logic to program analysis and verification, as initiated by Hoare with dynamic logic and continued by many in the field.

Finally, we believe that knowing the historical context in which mathematical logic and its applications have developed is important for a full understanding and appreciation of the subject. Thus we supply in an appendix a brief historical view of the origins and development of logic from the Greeks to the middle of the twentieth century. Parts of this survey may be fully appreciated only after reading the text (especially the first two chapters) but it can be profitably consulted before, after or while reading the book. It is intended only as a tourist brochure, a guide to the terrain. To supplement this guide, we have included a fairly extensive bibliography of historical references and sources for additional information on many topics in logic, including several not covered in the text. When possible, we have confined our suggestions to historical material, handbooks, surveys and basic texts at a level suitable for a reader who has finished this book. Some newer subjects, however, also require references to the current literature. This bibliography is arranged as several (partially annotated) bibliographies on individual subjects. References are made accordingly. Thus, for example, Thomas [1939, 1.1] refers to the item *Selections Illustrating the History of Greek Mathematics with an English Translation* by Ivor Thomas published in 1939 which is listed in Bibliography 1.1, Sourcebooks for the History of Mathematics. We have also included at the end of each chapter suggestions for further reading that are keyed to these bibliographies.

I Propositional Logic

1. Orders and Trees

Before starting on the basic material of this book, we introduce a general representation scheme which is one of the most important types of structures in logic and computer science: Trees. We expect that most readers will be familiar with this type of structure at least informally. A tree is something that looks like the following:

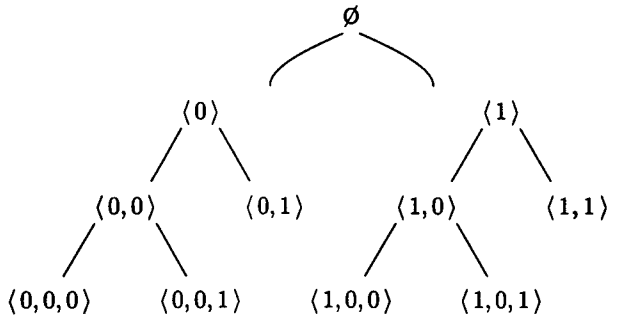


FIGURE 1

It has nodes (in this example, binary sequences) arranged in a partial order (extension as sequences means lower down in the picture of the tree). There is typically a single node (the empty set, $\emptyset$) at the top (above all the others) which is called the root of the tree. (We will draw our trees growing downwards to conform to common practice but the ordering on the tree will be arranged to mirror the situation given by extension of sequences. Thus the root will be the smallest (first) element in the ordering and the nodes will get larger as we travel down the tree.) A node on a tree may have one or more incomparable immediate successors. If it has more than one, we say the tree branches at that node. If each node has at most n immediate successors, the tree is n–ary or n–branching. (The one in the picture is a 2–ary, or as we usually say, a *binary* tree.) A terminal node, that is one with no successors, is called a leaf. Other terminology such as a path on a tree or the levels of a tree should have intuitively clear meanings. For those who wish to be formal, we give precise definitions. This material can be omitted for now and referred to as needed later. In particular König's lemma (Theorem 1.4) is not needed until §4.

Definition 1.1:

(i) A *partial order* is a set S with a binary relation called "less than", and written $<$, on S which is *transitive* and *irreflexive*:

$$x < y \text{ and } y < z \Rightarrow x < z \text{ and}$$
$$x \text{ is not less than } x \text{ for any } x.$$

(ii) The partial order $<$ is a *linear order* (or simply an *order*) if it also satisfies the *trichotomy law*:

$$x < y \text{ or } x = y \text{ or } y < x.$$

(iii) A linear order is *well ordered* if it has no infinite descending chain, i.e., there is no set of elements $x_0, x_1, \ldots$ of S such that

$$\ldots < x_2 < x_1 < x_0.$$

(iv) We use the usual notational conventions for orderings:

$$x \leq y \Leftrightarrow x < y \text{ or } x = y.$$
$$x > y \Leftrightarrow y < x.$$

Note that antisymmetry for partial orderings, $x < y \Rightarrow$ it is not the case that $y < x$, follows immediately from Definition 1.1 (i). (We follow standard mathematical practice in using "$\Rightarrow$" and "$\Leftrightarrow$" as abbreviations for "implies" and "if and only if", respectively.)

Definition 1.2: A *tree* is a set T (whose elements are called *nodes*) partially ordered by $<_T$, with a unique least element called the *root*, in which the predecessors of every node are well ordered by $<_T$.

A *path* on a tree T is a maximal linearly ordered subset of T.

Definition 1.3:

(i) The *levels* of T are defined by induction. The 0^{th} level of T consists precisely of the root of T. The $k+1^{\text{st}}$ level of T consists of the immediate successors of the nodes on the k^{th} level of T.

(ii) The *depth* of a tree T is the maximum n such that there is a node of level n in T. If there are nodes of level n for every natural number n we say the depth of T is infinite or ω.

(iii) If each node has at most n immediate successors, the tree is *n-ary* or *n-branching*. If each node has finitely many immediate successors, we say that the tree is *finitely branching*. A node with no successors is called a *leaf* or a *terminal node*.

We will only consider trees of depth at most ω, i.e., every node on every tree we ever consider will be on level n of the tree for some natural number n (and as such will have precisely n many predecessors in the tree order).

The crucial fact about finitely branching trees is König's lemma:

Theorem 1.4 (König's Lemma): *If a finitely branching tree T is infinite, it has an infinite path.*

Proof: Suppose T is infinite. We define the sequence of elements x_0, $x_1, \ldots$, constituting a path P on T by induction (or recursion). The first element x_0 of P is, of course, the root of T. It has infinitely many successors in T by the assumption that T is infinite. Suppose that we have defined the first n elements of P to be $x_0, x_1, \ldots, x_{n-1}$ on levels $0, 1, \ldots, n-1$ of T respectively so that each x_i has infinitely many successors in T. By hypothesis, x_{n-1} has only finitely many immediate successors. As it has infinitely many successors all together, (at least) one of its immediate successors, say y, also has infinitely many successors. We now set $x_n = y$. x_n is on level n of T and has infinitely many successors in T and so we may continue our definition of P. □

König's lemma is a version of the compactness theorem in propositional logic and in topology. See §6 (and especially Theorem 6.13 and Exercise 6.11) for the former and Exercises 10 and 11 for its relations to the latter.

Frequently it is just the shape of the tree that is important and not the nodes themselves. To facilitate talking about the arrangement of different materials into the same shape and to allow the same component to be used at different places in this assemblage, we will talk about labeled trees. We attach labels to the nodes of the tree. Again the picture should be clear.

Definition 1.5: A *labeled tree* T is a tree T with a function (the labeling function) which associates some object with every node. This object is called the *label* of the node.

In fact, after the first exposure or two to labeled trees we will at times simply drop the word. We will draw our trees already labeled and will let the concerned reader adjust the formalities.

Another way of putting more structure on a tree is by adding a linear ordering on the entire tree. Consider the case of the standard binary tree of finite sequences of 0's and 1's: The underlying set is the set $S = \{0,1\}^*$ of such sequences. We think of a binary sequence σ of length n as a map from $\{0, 1, \ldots, n-1\}$ into $\{0,1\}$. We use "$\hat{}$" to denote *concatenation*. Thus, for example, $\langle 0,1\rangle\hat{}0$ is $\langle 0,1,0\rangle$ while $\langle 0,1,0\rangle\hat{}\langle 0,1\rangle$ is $\langle 0,1,0,0,1\rangle$. (Note that we frequently abuse notation by identifying 0 with $\langle 0\rangle$ and 1 with $\langle 1\rangle$ in such situations.) The tree ordering $<_S$ is given by extension as functions $\sigma < \tau \Leftrightarrow \sigma \subset \tau$. The additional linear order usually associated with this tree is the *lexicographic ordering on sequences*: For two sequences σ and τ we say that $\sigma <_L \tau$ if $\sigma \subset \tau$ or if $\sigma(n)$, the n^{th} entry in σ, is less than $\tau(n)$ where n is the first entry at which the sequences differ (otherwise, as one can easily see, $\tau <_L \sigma$ or $\sigma = \tau$). The same procedure can be applied to any tree to produce a linear order of all the nodes. We begin by defining a linear order on each level of the tree. This order is usually called $<_L$ and described as a *left to right ordering* for the obvious pictorial reason.

(This corresponds to ordering the binary strings of each fixed length by $\sigma <_L \tau$ if, at the first place σ and τ differ, σ is 0 and τ is 1. We then say that σ is *to the left of* τ.) The left–right orderings of each level are then extended to a linear ordering (also designated $<_L$) of all the nodes of the tree: Given two nodes x and y, we say that $x <_L y$ if $x <_T y$. If x and y are incomparable in the tree ordering, we find the largest predecessors x' and y' of x and y respectively which are on the same level of T. We then order x and y in the same way that x' and y' were ordered by $<_L$ on their own level: $x <_L y$ iff $x' <_L y'$. Any such total ordering of the nodes of a tree is also referred to as the *lexicographic ordering of the nodes*.

Exercises

1. Give an example of a finitely branching tree which is not n–branching for any n.
2. Give an example of an infinite tree of depth 3.
3. Prove that the notion of the level of a node in a tree is well defined, i.e., every node in a tree T is on exactly one level.
4. Prove that every node of a tree other than the root has exactly one immediate predecessor.
5. Let T be a tree. We say that two nodes x and y of T are *adjacent* if one is an immediate predecessor of the other, i.e., $x <_T y$ or $y <_T x$ and there is no node strictly between them. Prove that there is no sequence of nodes $x_1, \ldots, x_n$ $(n > 3)$ such that each x_i is adjacent to x_{i+1}, $x_1 = x_n$ but there are no other duplications on the list. (In graph theoretic terms (see Exercise 6.8) this says that if we define the edges of a graph to be the adjacent nodes of a tree, then the graph is acyclic.) Hint: Use Exercise 3.
6. Prove that a linear order $<$ on S is well ordered iff every subset of S has a least element.
7. Prove that the lexicographic ordering $<_L$ of pairs from the natural numbers $\mathbb{N}$ is well ordered.
8. a) Prove that the lexicographic ordering $<_L$ of n–tuples of natural numbers is well ordered for each n.

   b) Prove that the lexicographic ordering of the set of all n–tuples (with $n < m$ for any $m \in \mathbb{N}$) is well ordered.
9. Consider the set of all finite sequences of natural numbers. Define an ordering $<$ as follows: $\sigma < \tau$ iff either σ is shorter than τ or, if not, $\sigma <_L \tau$. Prove that $<$ is a well ordering.

The next two exercises are for those readers familiar with the topological notions of product topologies and compactness.

10. Show that König's lemma for binary trees is equivalent to the compactness of the topological space $C = \{0,1\}^\omega$ where $\{0,1\}$ is given the discrete topology and C the product topology.
11. Show that König's lemma for all finitely branching trees is equivalent to the compactness of all spaces $\prod X_i$ for every sequence of finite sets X_i, $i \in \mathbb{N}$, where each X_i has the discrete topology.

2. Propositions, Connectives and Truth Tables

Propositions are just statements and propositional logic describes and studies the ways in which statements are combined to form other statements. This is what is called the syntactic part of logic, the one which deals with statements as just strings of symbols. We will also be concerned with ascribing meaning to the symbols in various ways. This part of language is called semantics and a major theme in the development of logic is the relationship between these two aspects of language. The analysis of the internal structure of statements is left to a later time and a subject called predicate logic. For now we will consider some of the ways in which one builds statements from other statements in English. The construction procedures we consider will be the ones basic to mathematical texts. We call the operations that combine propositions to form new ones *connectives*.

The connectives one finds most frequently in a mathematical text are "or", "and", "not", "implies" and "if and only if". The meaning given to them by the working mathematician does not precisely reflect their meaning in everyday discourse; it has been changed slightly so as to become entirely unambiguous. They should be thought of simply as part of the jargon of mathematics.

We introduce formal symbols for these connectives as follows:

$\vee$	for	"or"	*(disjunction)*
$\wedge$	for	"and"	*(conjunction)*
$\neg$	for	"not"	*(negation)*
$\rightarrow$	for	"implies"	*(conditional)*
$\leftrightarrow$	for	"if and only if"	*(biconditional).*

Before making the meaning of these connectives precise, we describe how they are used to form statements of propositional logic. The description of the syntax of any language begins with its alphabet. The *language of propositional logic* consists of the following symbols:

(i) *Connectives*: $\vee$, $\wedge$, $\neg$, $\rightarrow$, $\leftrightarrow$

(ii) *Parentheses*:) , (

(iii) *Propositional Letters*: A, A_1, A_2, $\ldots$, B, B_1, B_2, $\ldots$, $\ldots$.

Once the symbols of our language are specified, we can describe the statements of the language of propositions. The definition that follows selects out certain strings of symbols from the language and calls them propositions. It is an *inductive* definition which describes the "shortest" statements first and then describes how to build longer statements from shorter ones in accordance with certain definite rules.

Definition 2.1: (Propositions)

(i) Propositional letters are propositions.

(ii) If α and β are propositions, then $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\neg\alpha)$, $(\alpha \rightarrow \beta)$ and $(\alpha \leftrightarrow \beta)$ are propositions.

(iii) A string of symbols is a proposition if and only if it can be obtained by starting with propositional letters (i) and repeatedly applying (ii).

For example, $(A \vee B)$, C, $((A \wedge B) \rightarrow C)$, $(\neg(A \wedge B) \rightarrow C)$ are all propositions while $A \wedge \neg$, $(A \vee B$, $(\wedge \rightarrow A)$ are not. We will return to these examples in 2.3 below. The importance of clause (iii) in the above definition is that it provides us with the basis for applying induction directly to propositions. We will also return to this subject after first introducing another approach to describing propositions.

Labeled (binary) trees provide us with an important way of representing propositions. It is not hard to see that each proposition φ can be represented as a finite labeled binary tree T. The leaves of T are labeled with propositional letters. If any nonterminal node of T is labeled with a proposition α, its immediate successors are labeled with propositions (one or two) which can be combined using one of the connectives to form α. The left to right ordering on the immediate successors of α is given by the syntactic position of the component propositions. This procedure can be carried out so that the original proposition φ is the label of the root of the tree T. One could take such labeled trees to define propositions and ignore the inductive definition given above. We instead offer a precise translation between these notions:

Definition 2.2: A *formation tree* is a finite tree T of binary sequences (with root $\emptyset$ and a left to right ordering given by the ordinary lexicographic ordering of sequences) whose nodes are all labeled with propositions. The labeling satisfies the following conditions:

(i) The leaves are labeled with propositional letters.

(ii) If a node σ is labeled with a proposition of the form $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \rightarrow \beta)$ or $(\alpha \leftrightarrow \beta)$, its immediate successors, $\sigma^\frown 0$ and $\sigma^\frown 1$, are labeled with α and β (in that order).

(iii) If a node σ is labeled with a proposition of the form $(\neg\alpha)$, its unique immediate successor, $\sigma^\frown 0$, is labeled with α.

The formation tree T is *associated* with the proposition with which its root is labeled.

Example 2.3: We can depict the formation trees associated with the correctly formed propositions listed above by inserting the appropriate labels for nodes on the proper trees:

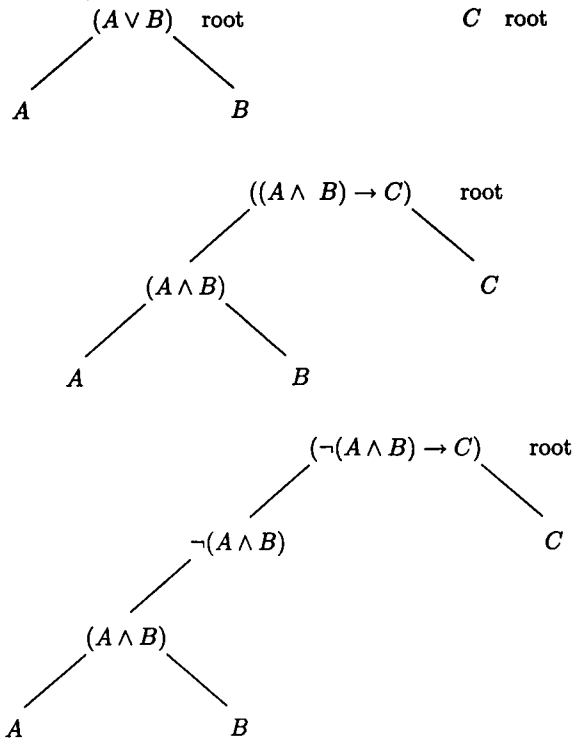


FIGURE 2

Note that in these examples we talked of *the* formation tree associated with a given proposition. Behind this usage stands a theorem that there is, in fact, a unique formation tree associated with each proposition. The method of proof for such a theorem is induction. The definition of propositions is an inductive one: the propositional letters are the base case and the formation rules given by the various connectives constitute the inductive step. Clause (iii) of the definition says that all propositions are included in this process. Corresponding to this type of definition we have both other definitions and proofs by induction. Indeed, induction is the primary method for dealing with most of the notions we will consider. Thus, for example, to define a formation tree associated with each proposition it suffices to define one for each propositional letter and to specify how to define one for a proposition constructed from others via each of the connectives in terms of the ones defined for its constituent propositions. The

corresponding method of proof by induction establishes that some property P (such as there being at most one formation tree associated with each proposition) holds for every proposition by first showing that it holds for each propositional letter (the base case) and then showing that, if it holds for propositions α and β, then it holds for each of the propositions constructed from α and β via the five basic connectives. This method is really nothing more than the usual procedure of induction on natural numbers. A translation into that terminology is supplied in Exercise 15.

Theorem 2.4: *Each proposition has a unique formation tree associated with it.*

Proof: We first show by induction that each proposition α has a formation tree associated with it. The base case is that α is a propositional letter, say A. In this case the tree consisting precisely of $\emptyset$ (its root and only node) labeled with A is the desired tree. For the induction step consider the proposition $\alpha \to \beta$. By induction there are formation trees T_α and T_β associated with α and β respectively. The desired formation tree $T_{(\alpha \to \beta)}$ for $(\alpha \to \beta)$ has as its root $\emptyset$ labeled with $(\alpha \to \beta)$. Below this root we attach copies of T_α (on the left) and T_β (on the right). This can be described formally as letting the other nodes of $T_{(\alpha \to \beta)}$ be all sequences $0^\frown\sigma$ for every σ on T_α and $1^\frown\tau$ for every τ on T_β. The labels are the same as they were in the original trees. As both T_α and T_β were formation trees, it is clear that $T_{(\alpha \to \beta)}$ is one as well. (The labeling of nodes is inherited from the original trees except for that of the root by $(\alpha \to \beta)$. This labeling of the root is acceptable as the root's immediate successors are, by definition, labeled with α and β respectively.) The cases for the other binary connectives are handled in exactly the same way. For $(\neg\alpha)$ we simply add on the nodes $0^\frown\sigma$ for σ in T_α (with the same label as σ) to the root $\emptyset$ which we label with $(\neg\alpha)$. Note that if the formation trees for α and β have depth n and m respectively, the tree for any of the propositions built by applying one of the basic connectives has depth $\max\{n, m\} + 1$.

We next claim that there is at most one formation tree associated with each proposition. For the propositional letters, this claim is clear. As the root of the tree $\emptyset$ must be labeled by the propositional letter if the tree is associated with it, the definition requires that the root be a leaf, i.e., it is the entire tree. Consider now the inductive case for $\alpha \to \beta$. If T is a formation tree associated with $(\alpha \to \beta)$ then its root $\emptyset$ must be labeled with $(\alpha \to \beta)$ by definition. Again, by the definition of a formation tree, $\emptyset$ must have two immediate successors 0 and 1 which must be labeled with α and β respectively. Every node on T below $i = 0$ or 1 must be of the form $0^\frown\sigma$ or $1^\frown\sigma$ respectively for some binary sequence σ. For $i = 0, 1$ let $T_i = \{\sigma \mid i^\frown\sigma \in T\}$ have the standard ordering and be labeled as in T. It is clear that T_0 is a formation tree for α and T_1 for β. They are unique by induction and so T has been uniquely determined as required. The other connectives are handled similarly. □

This theorem corresponds to what is often called the *unique readability* of propositions: There is only one way to parse a proposition into its component parts all the way down to the propositional letters. Along these lines, we will, in informal usage, abuse our notation by omitting parentheses whenever no confusion can arise. Thus, for example, we will write $\neg\alpha$ for $(\neg\alpha)$ and $\alpha \to \beta$ for $(\alpha \to \beta)$. Formally, unique readability gives us another way to define functions on the propositions and prove facts about them: induction on formation trees. Typically, we induct on the depth of the formation tree associated with a proposition. The advantage of using trees is that, if one defines a function on formation trees, one automatically has one on the associated propositions. If instead, one defined an operation directly on propositions by induction, one would not know that there is only one way of analyzing a given proposition inductively so as to guarantee that the operation is well defined. This is precisely unique readability. We will see some examples of such procedures in the next section. For now, we just note that the theorem allows us to define the depth of a proposition. We can also use it to pick out the propositional letters that are "relevant" to a given proposition:

Definition 2.5:

(i) The *depth* of a proposition is the depth of the associated formation tree.

(ii) The *support* of a proposition is the set of propositional letters that occur as labels of the leaves of the associated formation tree. (That this notion corresponds to the ones which occur syntactically in the proposition is proven by another induction argument. See Exercise 16.)

***Closure operations and inductive definitions:**

Another approach to the type of inductive definition given may clarify the role of (iii) in guaranteeing that only those expressions generated by (i) and (ii) are propositions. We begin with the (algebraic) notion of closure. A set S is *closed* under a single (for example n–ary) operation $f(s_1, ..., s_n)$ iff for every $s_1, ..., s_n \in S$, $f(s_1, \ldots, s_n) \in S$. The *closure* of a set S under (all) the operations in a set T is the smallest set C such that

1) $S \subseteq C$ and
2) if $f \in T$ is n–ary and $s_1, \ldots, s_n \in C$ then $f(s_1, \ldots, s_n) \in C$.

To see that there is a smallest such set consider the set

$$C = \cap\{D \mid S \subseteq D \ \&\ D \text{ is closed under the operations of } T\}.$$

Of course $S \subseteq C$. Now show that C is closed under the operations of T. It is then clear that C is the smallest such set as it is contained in every set $D \supseteq S$ which is closed under the operations of T. We could now define the set of propositions as the closure of the set of propositional letters (i) under the operations $\wedge$, $\vee$, $\neg$, $\to$ and $\leftrightarrow$ as listed in (ii).

Turning now to semantics, we take the view that the meaning of a propositional letter is simply its truth value, that is, its truth or falsity. (Remember that we are postponing the analysis of the internal structure of propositions to the next chapter.) Each proposition will then have a unique *truth value* (T, for true or F, for false). The truth value of a compound proposition is determined from the truth values of its parts in accordance with the following *truth tables*:

Definition 2.6: (*Truth tables*):

α	β	$(\alpha \vee \beta)$
T	T	T
T	F	T
F	T	T
F	F	F

α	β	$(\alpha \wedge \beta)$
T	T	T
T	F	F
F	T	F
F	F	F

α	β	$(\alpha \rightarrow \beta)$
T	T	T
T	F	F
F	T	T
F	F	T

α	β	$(\alpha \leftrightarrow \beta)$
T	T	T
T	F	F
F	T	F
F	F	T

α	$\neg\alpha$
T	F
F	T

FIGURE 3

As pointed out earlier, the meaning of these connectives as specified by these truth tables is not exactly the same as in ordinary English. For $\vee$, the meaning is that of the inclusive or: $\alpha \vee \beta$ is true if either or both of α and β are true. The meaning of $\rightarrow$ is further removed from that in colloquial usage. In mathematics, $(\alpha \rightarrow \beta)$ is asserted to be false only when α is true and β is false. It is asserted to be true in all other cases.

The formal assignment of truth values to propositions based on those of the propositional letters will be given in the next section. Intuitively, it should be clear from the inductive definition of propositions how, given any proposition whatsoever, we can construct a truth table for it by considering it as being built up step by step starting from propositional letters. For example Figure 4 is the truth table for $((A \wedge B) \rightarrow C)$.

The eight combinations of truth values for A, B, C ($2^3 = 8$) can be thought of as all possible states of the world as far as any proposition in which only the propositional letters A, B, C appear is concerned. The column for $(A \wedge B)$ is auxiliary and could be eliminated. The result would be the *abbreviated truth table* for $((A \wedge B) \rightarrow C)$. If the convention implicit in

A	B	C	$(A \wedge B)$	$((A \wedge B) \rightarrow C)$
T	T	T	T	T
T	T	F	T	F
T	F	T	F	T
T	F	F	F	T
F	T	T	F	T
F	T	F	F	T
F	F	T	F	T
F	F	F	F	T

FIGURE 4

the above table for systematically listing the (eight) possible combinations of truth values for the propositional letters A, B and C is observed, then it is clear that to any proposition there corresponds a unique abbreviated truth table.

A priori, we might have begun with some other list of basic connectives. In general, an *n-ary connective* is any function σ which assigns a proposition $\sigma(A_1, \ldots, A_n)$ to every n-tuple of propositions $A_1, \ldots, A_n$. So $\neg$ is 1-ary (*unary*), while $\wedge$ and $\vee$ are 2-ary (or *binary*). An n-ary connective is *truth functional* if the truth value for $\sigma(A_1, \ldots, A_n)$ is uniquely determined by the truth values for $A_1, \ldots, A_n$. Our five connectives are truth functional since their meaning was defined by truth tables. On the other hand a connective like "because" is not. For let A symbolize "I had prune juice for breakfast" and B "there was an earthquake at noon". Even in the event that both A and B have truth values T it is at least debatable whether (B because A) should have truth value T. The debate might be more or less heated in other cases depending on the content of A and B. An n-ary connective which is truth functional can be completely described by means of a truth table. Here each b_i, $1 \leq i \leq 2^n$ is either T or F:

A_1	A_2	$\cdots$	A_n	$\sigma(A_1, \ldots, A_k)$
T	T	$\cdots$	T	b_1
T	T	$\cdots$	F	b_2
.	.	$\cdots$	.	.
.	.	$\cdots$	.	.
F	F	$\cdots$	F	.

FIGURE 5

Conversely, two distinct abbreviated truth tables (with the conventional listing of truth values for $A_1, \ldots, A_n$) correspond to distinct truth functional connectives. By counting we see that there are 2^{2^n} distinct n–ary truth functional connectives. (So there are $12 = 16 - 4$ binary connectives which we are not using.)

Definition 2.7: A set S of truth functional connectives is *adequate* if, given any truth functional connective σ, we can find a proposition built up from the connectives in S with the same abbreviated truth table as σ.

Theorem 2.8 (Adequacy): $\{\neg, \wedge, \vee\}$ *is adequate.*

Proof: Let $A_1, \ldots, A_k$ be distinct propositional letters and let a_{ij} denote the entry (T or F) corresponding to the i^{th} row and j^{th} column of the truth table for $\sigma(A_1, \ldots, A_k)$. Suppose that at least one T appears in the last column.

A_1	$\cdots$	A_j	$\cdots$	A_k	$\cdots$	$\sigma(A_1, \ldots, A_k)$
						b_1
						b_2
						.
						.
		a_{ij}				b_i

FIGURE 6

For any proposition α, let α^T be α and α^F be $(\neg\alpha)$. For the i^{th} row denote the *conjunction* $(A_1^{a_{i1}} \wedge \ldots \wedge A_k^{a_{ik}})$ by a_i. Let $i_1, \ldots, i_m$ be the rows with a T in the last column. The desired proposition is the *disjunction* $(a_{i_1} \vee \ldots \vee a_{i_m})$. The proof that this proposition has the given truth table is left as Exercises 14. (Note that we abused our notation by leaving out a lot of parentheses in the interest of readability. The convention is that of *right associativity*, that is, $A \wedge B \wedge C$ is an abbreviation for $(A \wedge (B \wedge C))$.) We also indicate a disjunction over a set of propositions with the usual set–theoretic terminology. Thus the disjunction just constructed would be written as $\bigvee\{a_i : b_i = T\}$. □

Example 2.9: The procedure given in the above proof can be illustrated by constructing a proposition built using only $\wedge$, $\vee$ and $\neg$ which has the truth table given in Figure 7.

	A	B	C	?
1	T	T	T	T
2	T	T	F	F
3	T	F	T	F
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	T

FIGURE 7

We begin by looking only at rows with a T in the last column. For each such row we find a proposition which is true for that row and false for every other row. The proposition we want is the disjunction of the propositions we obtain for all relevant rows (rows 1, 5, 8 in this case). For any particular row, the proposition true for only that row is obtained by taking the conjunction of the letters having a T in that row and the negations of letters having an F on that row. In this case row 1 gives $(A \wedge B \wedge C)$ (we abuse notation again!); row 5 gives $((\neg A) \wedge B \wedge C)$; and row 8 gives $((\neg A) \wedge (\neg B) \wedge (\neg C))$. Thus the proposition $(A \wedge B \wedge C) \vee ((\neg A) \wedge B \wedge C) \vee ((\neg A) \wedge (\neg B) \wedge (\neg C))$ has the given truth table.

Clearly, given any proposition α we can construct its truth table and then follow the above procedure to find another proposition which has the same truth table and is a disjunction of conjunctions of propositional letters and their negations. A proposition of this form which has the same (abbreviated) truth table as α is called a *disjunctive normal form* (DNF) of α. There is also a *conjunctive normal form* (CNF) equivalent of α which is presented in Exercise 3.3. Another method of finding DNF and CNF equivalents of α is presented at the end of the exercises for §4.

Remark 2.10: The above procedure does not tell us what to do in case the last column consists entirely of F's. See Exercise 13.

Corollary 2.11: $\{\neg, \vee\}$ *is adequate.*

Proof: We can easily check that $(A_1 \wedge A_2)$ has the same truth table as $\neg((\neg(A_1)) \vee (\neg(A_2)))$. Thus given any proposition α we can find a DNF of α and then eliminate any use of $\wedge$ by this substitution. The resulting proposition will still have the same truth table. □

The sets $\{\neg, \wedge\}$ and $\{\neg, \rightarrow\}$ are also shown to be adequate in the exercises. If a set is not adequate, how do you prove that? (See Exercises 10.)

Remark 2.12: By the adequacy theorems (Theorem 2.8 and Corollary 2.11) we could, in theory, get by with just the connectives $\neg$, $\vee$ and $\wedge$ or even just $\neg$ and $\vee$. The induction clause in the definition of propositions and many related definitions and proofs (such as those involving tableaux in sections 4, 5 and 6) could then be considerably shortened. We will, however, leave the list of connectives as it is but will generally explicitly deal with only a couple of cases in any particular proof and leave the rest as exercises.

Exercises

1. Which of the following expressions are official (that is, unabbreviated) propositions of propositional logic based on the propositional letters A, B, C, D, ...?
   a) $((\neg(A \vee B)) \wedge C)$
   b) $(A \wedge B) \vee C$
   c) $A \rightarrow (B \wedge C)$
   d) $((A \leftrightarrow B) \rightarrow (\neg A))$
   e) $((\neg A) \rightarrow B \vee C)$
   f) $(((C \vee B) \wedge A) \leftrightarrow D)$
   g) $((\vee A) \wedge (\neg B))$
   h) $(A \wedge (B \wedge C)))$
2. Prove your answers to 1(a), (b) and (f) by either giving the step by step procedure producing the proposition in accordance with the inductive definition of propositions (you can simply draw a correctly labeled formation tree) or proving, by induction on statements, that there is some property enjoyed by all propositions but not by this expression.
3. Prove that the number of right and left parentheses are equal in every proposition.
4. Prove that the depth of a proposition is less than or equal to the number of left parentheses appearing in the proposition. (Use the official definition of proposition.)
5. Find DNF equivalents for the following propositions:
   a) $(A \rightarrow B) \rightarrow C$
   b) $(A \leftrightarrow B) \vee (\neg C)$
6. Prove that $\{\neg, \wedge\}$ is an adequate set of connectives. (Hint: Express $\vee$ in terms of $\neg$ and $\wedge$.)
7. Prove that $\{\neg, \rightarrow\}$ is an adequate set of connectives.

8. Prove that the binary connective $(\alpha \mid \beta)$ ("not both ... and") called the *Sheffer stroke* whose truth table is given by

α	β	$\alpha \mid \beta$
T	T	F
T	F	T
F	T	T
F	F	T

FIGURE 8

   is adequate. (Hint: Express $\neg$ and $\wedge$ in terms of $\mid$.)
9. Show that joint denial (neither α nor β), written as $\alpha \downarrow \beta$, is also adequate.
10. Prove that $\{\wedge, \vee\}$ is not adequate.
    Hint: Show by induction that $\neg\alpha$ is not equivalent to any statement built up from α using only $\wedge$ and $\vee$
11. Prove that $\{\vee, \rightarrow\}$ is not an adequate set of connectives.
12. Prove that $\{\vee, \rightarrow, \leftrightarrow\}$ is not an adequate set of connectives.
13. Explain how to handle the case of a column of all F's in the proof of Theorem 2.8.
14. Prove that the expressions constructed in the proof of Theorem 2.8 (including the case considered in exercise 13) have the desired truth tables.
15. We say that all propositional letters are built at stage 0. If propositions α and β have been built by level n, we say that $(\neg\alpha)$, $(\alpha \vee \beta)$, $(\alpha \wedge \beta)$, $(\alpha \rightarrow \beta)$ and $(\alpha \leftrightarrow \beta)$ have been built by level $n+1$. Clause (iii) of Definition 2.1 says that every proposition φ is built by some level n. Explain how we can rephrase proof by induction on the definition of propositions in terms of ordinary induction on the natural numbers $\mathbb{N}$.

    (Hint: Proving that all propositions have property P by induction on propositions corresponds to proving that all propositions built by level n have property P by induction on n.)
16. We say that each propositional letter A *occurs* in itself and no propositional letters other than A occur in A. The propositional letters that *occur* in $(\neg\alpha)$ are precisely the ones that occur in α. The ones that occur in $(\alpha \vee \beta)$, $(\alpha \wedge \beta)$, $(\alpha \rightarrow \beta)$ and $(\alpha \leftrightarrow \beta)$ are precisely those that occur in either α or β (or both). This notion clearly captures the idea of a syntactic occurrence of a propositional letter A in a proposition α.

    Prove that the support of a proposition α is precisely the set of propositional letters that occur in α.

3. Truth Assignments and Valuations

Our view of propositional logic is that the meaning or content of a proposition is just its truth value. Thus, the whole notion of semantics for propositional logic consists of assigning truth values to propositions. We begin with the propositional letters.

Definition 3.1: A *truth assignment* $\mathcal{A}$ is a function which assigns to each propositional letter A a unique truth value $\mathcal{A}(A) \in \{T, F\}$.

The truth values of all propositions should now be determined by the assignment to the propositional letters. The determinations are made in accordance with the truth tables for the connectives given in the last section.

Definition 3.2: A *truth valuation* $\mathcal{V}$ is a function which assigns to each proposition α a unique truth value $\mathcal{V}(\alpha)$ so that its value on a compound proposition (that is, one with a connective) is determined in accordance with the appropriate truth tables. Thus, for example, $\mathcal{V}((\neg\alpha)) = T$ iff $\mathcal{V}(\alpha) = F$ and $\mathcal{V}((\alpha \vee \beta)) = T$ iff $\mathcal{V}(\alpha) = T$ or $\mathcal{V}(\beta) = T$. We say that $\mathcal{V}$ *makes* α *true* if $\mathcal{V}(\alpha) = T$.

The basic result here is that a truth assignment to the propositional letters uniquely determines the entire truth valuation on all propositions. We analyze the situation in terms of an induction on the depth of the propositions, that is, the depth of the (unique) formation tree associated with the proposition.

Theorem 3.3: *Given a truth assignment $\mathcal{A}$ there is a unique truth valuation $\mathcal{V}$ such that $\mathcal{V}(\alpha) = \mathcal{A}(\alpha)$ for every propositional letter α.*

Proof: Given a truth assignment $\mathcal{A}$, define (by induction on the depth of the associated formation tree) a valuation $\mathcal{V}$ on all propositions by first setting $\mathcal{V}(\alpha) = \mathcal{A}(\alpha)$ for all propositional letters α. This takes care of all formation trees (propositions) of depth 0. Assuming that $\mathcal{V}$ has been defined on all propositions with depth at most n, the inductive steps are simply given by the truth tables associated with each connective. For example, suppose $T_{(\alpha\to\beta)}$ is the formation tree (of depth $n+1$) for $(\alpha \to \beta)$. (It is built from T_α and T_β (with the maximum of their depths being exactly n) as in Theorem 2.4.) $\mathcal{V}((\alpha \to \beta))$ is then defined to be F iff $\mathcal{V}(\alpha) = T$ and $\mathcal{V}(\beta) = F$. The valuation is defined on α and β by induction since they have depth at most n.

Clearly $\mathcal{V}$ has been defined so as to be a valuation and it does extend $\mathcal{A}$. It remains to show that any two valuations $\mathcal{V}_1$, $\mathcal{V}_2$ both extending $\mathcal{A}$ must coincide. We prove this by induction on the depth of propositions:

(i) $\mathcal{V}_1(\alpha) = \mathcal{V}_2(\alpha)$ for all propositional letters α (depth 0) since $\mathcal{V}_1$, $\mathcal{V}_2$ both extend $\mathcal{A}$.

(ii) Suppose $\mathcal{V}_1(\alpha) = \mathcal{V}_2(\alpha)$ for all propositions α of depth at most n and that α and β have depth at most n. Thus $\mathcal{V}_1(\alpha) = \mathcal{V}_2(\alpha)$ and $\mathcal{V}_1(\beta) = \mathcal{V}_2(\beta)$ by induction. $\mathcal{V}_1((\alpha \wedge \beta))$ and $\mathcal{V}_2((\alpha \wedge \beta))$ are then both given by the truth table for $\wedge$ and so are equal. The same argument works for all the other connectives and so $\mathcal{V}_1$ and $\mathcal{V}_2$ agree on every proposition. □

Note that, by induction again on the depth of α, the definition of $\mathcal{V}(\alpha)$ in this construction only depends on the values of $\mathcal{A}$ on the support of α (the propositional letters occurring in α). Thus the proof of the theorem actually proves:

Corollary 3.4: *If $\mathcal{V}_1$ and $\mathcal{V}_2$ are two valuations which agree on the support of α, the finite set of propositional letters used in the construction of the proposition α, then $\mathcal{V}_1(\alpha) = \mathcal{V}_2(\alpha)$.* □

Definition 3.5: A proposition σ of propositional logic is said to be *valid* if for any valuation $\mathcal{V}$, $\mathcal{V}(\sigma) = T$. Such a proposition is also called a *tautology*.

Definition 3.6: Two propositions α and β such that, for every valuation $\mathcal{V}$, $\mathcal{V}(\alpha) = \mathcal{V}(\beta)$ are called *logically equivalent*. We denote this by $\alpha \equiv \beta$.

Example 3.7:

(i) $(A \vee (\neg A))$, $(((A \to B) \to A) \to A)$ (Law of the excluded middle, Peirce's law) are tautologies. See Exercise 1.

(ii) For any proposition α and any DNF β of α, $\alpha \equiv \beta$.

(iii) We could rephrase the adequacy theorem (2.8) to say that, given any proposition α, we can find a β which uses only $\neg$, $\vee$, $\wedge$ and such that $\alpha \equiv \beta$.

Although Corollary 3.4 allows us to check whether a given proposition is a tautology or not using Definition 2.5, it also tells us that we can answer the same question by finding out whether the last column of the corresponding truth table has all T's or not. We choose not to develop these proofs by truth tables further because they do not generalize to proofs for predicate logic which we will study shortly. We close this section with some definitions and notations which will be important later and generalize nicely to the corresponding concepts in the logic of predicates.

Definition 3.8: Let Σ be a (possibly infinite) set of propositions. We say that σ *is a consequence of* Σ (and write $\Sigma \vDash \sigma$) if, for any valuation $\mathcal{V}$,

$$\mathcal{V}(\tau) = T \text{ for all } \tau \in \Sigma \quad \Rightarrow \quad \mathcal{V}(\sigma) = T.$$

Note that, if Σ is empty, $\Sigma \vDash \sigma$ (or just $\vDash \sigma$) iff σ is valid. We will also write this as $\vDash \sigma$. This definition gives a semantic notion of consequence. We will see several syntactic notions in the coming sections that correspond to different proof procedures. A major result will be the equivalence of the syntactic and semantic notions of consequence which will be embodied in the soundness and completeness theorems (§5).

Definition 3.9: We say that a valuation $\mathcal{V}$ is a *model* of Σ if $\mathcal{V}(\sigma) = T$ for every $\sigma \in \Sigma$. We denote by $\mathcal{M}(\Sigma)$ the set of all models of Σ.

Notation: Rather than writing "implies" and "if and only if" in our definitions, theorems, etc., we often use $\Rightarrow$ and $\Leftrightarrow$ instead. These are not symbols of the language of propositional logic but of the language (or metalanguage) in which we discuss propositional logic.

Proposition 3.10: *Let Σ, Σ_1, Σ_2 be sets of propositions. Let $Cn(\Sigma)$ denote the set of consequences of Σ and Taut the set of all tautologies.*

(i) $\Sigma_1 \subseteq \Sigma_2 \Rightarrow Cn(\Sigma_1) \subseteq Cn(\Sigma_2)$.

(ii) $\Sigma \subseteq Cn(\Sigma)$.

(iii) $Taut \subseteq Cn(\Sigma)$ *for all* Σ.

(iv) $Cn(\Sigma) = Cn(Cn(\Sigma))$.

(v) $\Sigma_1 \subseteq \Sigma_2 \Rightarrow \mathcal{M}(\Sigma_2) \subseteq \mathcal{M}(\Sigma_1)$.

(vi) $Cn(\Sigma) = \{\sigma \mid \mathcal{V}(\sigma) = T \text{ for all } \mathcal{V} \in \mathcal{M}(\Sigma)\}$.

(vii) $\sigma \in Cn(\{\sigma_1, \ldots \sigma_n\}) \Leftrightarrow \sigma_1 \rightarrow (\sigma_2 \ldots \rightarrow (\sigma_n \rightarrow \sigma) \ldots) \in Taut$.

We leave the proof of this proposition as Exercise 4.

The last assertion of Proposition 3.10 tells us that testing whether σ is a consequence of a finite set Σ of propositions (sometimes called "premises") with, say, n members can be done in at most 2^n steps by checking whether the proposition on the right-hand side of (vii) is a tautology. But what do we do if Σ is infinite? We had better learn how to *prove* that σ is a consequence. The first method we will consider is that of tableaux.

Exercises

1. Prove that the propositions in Example 3.7 (i) are tautologies by checking directly, using Corollary 3.4, that they are true under all valuations.
2. Prove De Morgan's laws for any propositions $\alpha_1, \ldots, \alpha_n$, i.e.,
   a) $\neg(\alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_n) \equiv \neg\alpha_1 \wedge \neg\alpha_2 \wedge \ldots \wedge \neg\alpha_n$
   b) $\neg(\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n) \equiv \neg\alpha_1 \vee \neg\alpha_2 \vee \ldots \vee \neg\alpha_n$.

   Hint: Do not write out the truth tables. Argue directly from the truth conditions for disjunctions and conjunctions.
3. A proposition is a *literal* if it is a propositional letter or its negation. A proposition α is in *conjunctive normal form* (CNF) if there are literals $\alpha_{1,1}, \ldots, \alpha_{1,n_1}, \alpha_{2,1}, \ldots, \alpha_{2,n_2}, \ldots, \alpha_{k,1}, \ldots, \alpha_{2,n_k}$ such that α is

   $(\alpha_{1,1} \vee \alpha_{1,2} \vee \ldots \vee \alpha_{1,n_1}) \wedge (\alpha_{2,1} \vee \alpha_{2,2} \vee \ldots \vee \alpha_{2,n_2}) \wedge \ldots \wedge (\alpha_{k,1} \vee \ldots \vee \alpha_{k,n_k})$.

   Prove that every proposition is equivalent to one in CNF (i.e., one that has the same truth table). (Hint: Consider a DNF (of $\neg\alpha$) and use Exercise 2.)

4. Find a CNF for each of the following propositions:
   a) $(A \wedge B \wedge C) \rightarrow D$
   b) $(A \wedge B) \rightarrow (C \vee D)$.
5. Supply the (short) proofs from the appropriate definitions for (i)–(vii) of Proposition 3.10.

4. Tableau Proofs in Propositional Calculus

We will describe a system for building proofs of propositions. The proofs will be labeled binary trees called tableaux. The labels on the trees will be *signed propositions*, that is a proposition preceded by either a T or an F (which we can think of as indicating an assumed truth value for the proposition). We call the labels of the nodes the *entries of the tableau*. Formally we will define (or describe how to build) tableaux for propositions inductively by first specifying certain (labeled binary) trees as tableaux (the so–called atomic tableaux) and then giving a development rule defining tableaux for compound propositions from tableaux for simple propositions.

The plan of the procedure is to start with some entry, i.e., some signed proposition such as $F(\neg(A \wedge (B \vee C)))$, and analyze it into its components. We will say that an entry is correct if our assumption about the truth value of the given proposition is correct. For our current example, $F(\neg(A \wedge (B \vee C)))$, this would mean that $\neg(A \wedge (B \vee C))$ is false. The guiding principle for the analysis is that, if an entry is correct, then (at least) one of the sets of entries into which we analyze it contains only correct entries. In our sample case, we would analyze $F(\neg(A \wedge (B \vee C)))$ first into $T(A \wedge (B \vee C))$. (If $\neg(A \wedge (B \vee C))$ is false, then $(A \wedge (B \vee C))$ is true.) We would then analyze $T(A \wedge (B \vee C))$ into TA and $T(B \vee C)$. (If $(A \wedge (B \vee C))$ is true then so are both A and $(B \vee C)$.) Next we would analyze $T(B \vee C)$ into either TB or TC. (If $(B \vee C)$ is true then so is one of B or C.)

The intent of the procedure, as a way of producing proofs of propositions, is to start with some signed proposition, such as $F\alpha$, as the root of our tree and to analyze it into its components in such a way as to see that any analysis leads to a contradiction. We will then conclude that we have refuted the original assumption that α is false and so have a proof of α. Suppose, for example, that we start with $F(\neg(A \wedge \neg A))$ and proceed as in the above analysis (replacing $(B \vee C)$ by $\neg A$). We reach TA and $T\neg A$ and then analyze $T\neg A$ into FA. We now have entries saying both that A is true and that it is false. This is the desired contradiction and we would conclude that we have a proof of the valid proposition $\neg(A \wedge \neg A)$.

The base case of our inductive definition of tableaux starts with the following (labeled binary) trees as the *atomic tableaux* for any propositions α and β and propositional letter A (Figure 9).

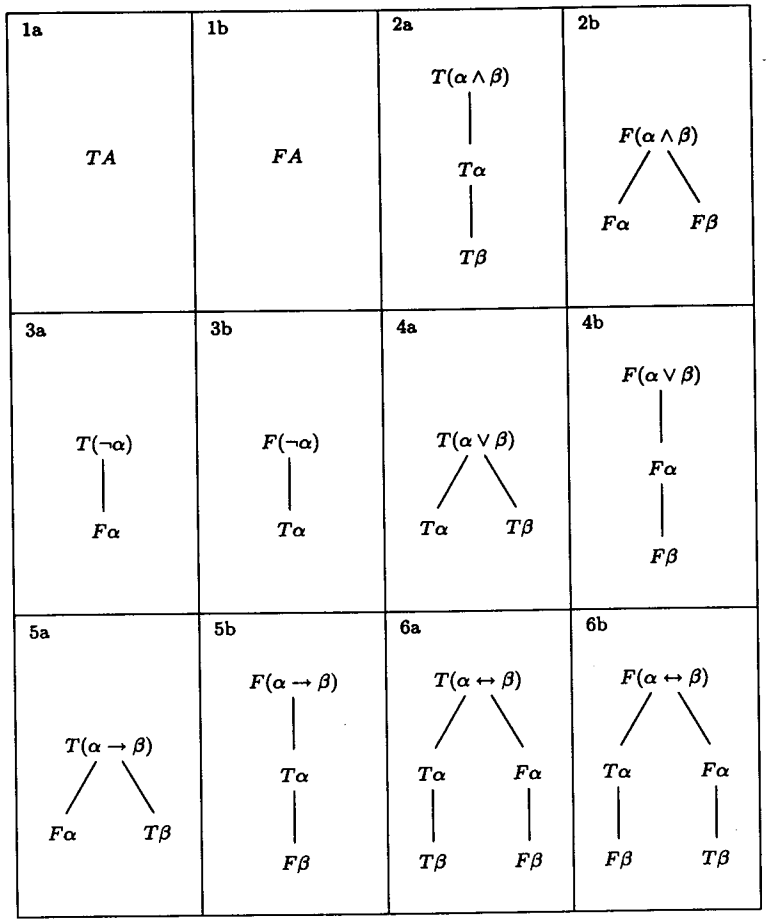


FIGURE 9

Definition 4.1 (Tableaux): A *finite tableau* is a binary tree, labeled with signed propositions called entries, which satisfies the following inductive definition:

(i) All atomic tableaux are finite tableaux.

(ii) If τ is a finite tableau, P a path on τ, E an entry of τ occurring on P and τ' is obtained from τ by adjoining the unique atomic tableau with root entry E to τ at the end of the path P then τ' is also a finite tableau.

If $\tau_0, \tau_1, ..., \tau_n, ...$ is a (finite or infinite) sequence of finite tableaux such that, for each $n \geq 0$, τ_{n+1} is constructed from τ_n by an application of (ii), then $\tau = \cup\tau_n$ is a *tableau*.

This definition describes all possible tableaux. We could get by with finite tableaux in propositional logic (see the appendix to this section) but would necessarily be driven to infinite ones in predicate logic. As they simplify some proofs even in our current situation we have introduced them here.

Each tableau is a way of analyzing a proposition. The intent is that, if it is all right to assume that all the signs on entries on a path down to some entry E in a tableau are correct, then one of the paths of the tableau that continue on through E to the next level of the tree is also correct. To see that this intention is realized, it suffices to consider the atomic tableaux. Consider for example (5a). If $\alpha \to \beta$ is true then so is one of the branches through it: α is false or β is true. Similarly for (4a), if $\alpha \vee \beta$ is true then so is one of α or β. The other atomic tableaux can be analyzed in the same way. This intuition will be developed formally in the next section as the soundness theorem for tableaux. The other major theorem about tableaux is the completeness theorem. It is connected with the idea that we can show that if α is valid, then all possible analyses of a given signed proposition $F\alpha$ lead to contradictions. This will constitute a proof α. In order to do this, we will have to develop a systematic method for generating a tableau with a given root which includes all possible procedures. First, however, some examples of tableaux.

Example 4.2: We wish to begin a tableau with the signed proposition $F\big(\big((\alpha \to \beta) \vee (\gamma \vee \delta)\big) \wedge (\alpha \vee \beta)\big)$. There is only one atomic tableau which has this entry as its root — the appropriate instance of the atomic tableau of type (2b):

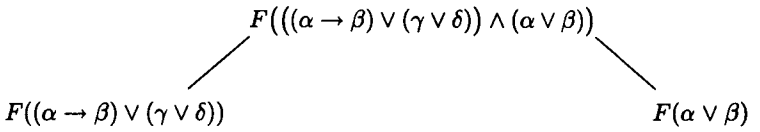


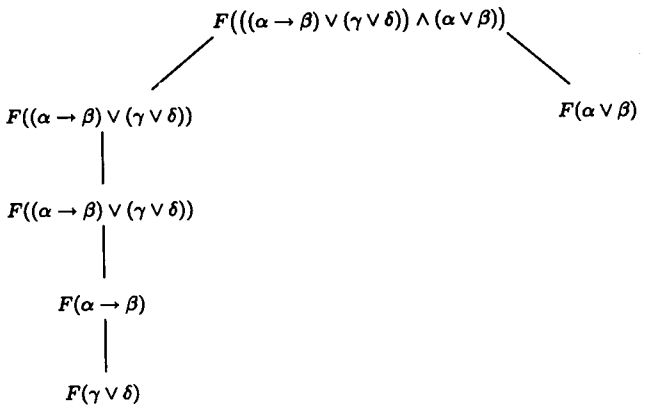
FIGURE 10

Now this tableau has two entries other than its root either of which could be chosen to use in the induction clause to build a bigger tableau. (We could legally use the root entry again but that would not be very interesting.) The two possibilities are given in Figures 11 A and B below.

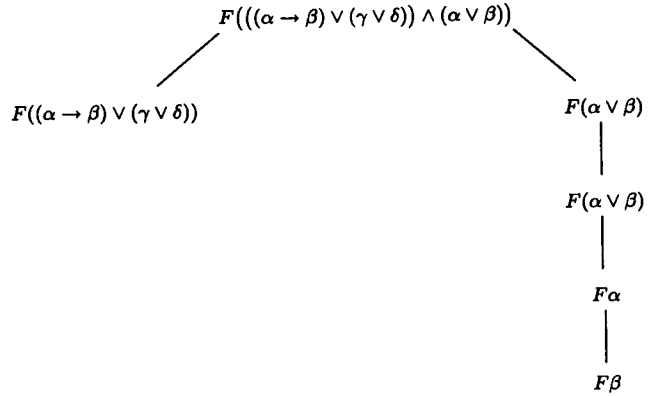
We could also do each of these steps in turn to get the tableau given in Figure 11 C.

In this last tableau we could (again ignoring duplications) choose either $F(\alpha \to \beta)$ or $F(\gamma \vee \delta)$ as the entry to d evelop. $F(\gamma \vee \delta)$ is the end of the only path in the tableau which contains either of these entries. Thus, in either case the appropriate atomic tableau would be appended to that path. Choosing $F(\alpha \to \beta)$ would give the tableau of Figure 12.

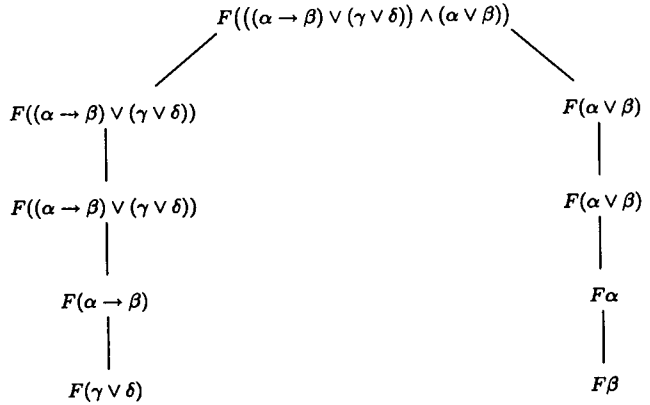
(A)



(B)



(C)



FIGURES 11 A, B, C

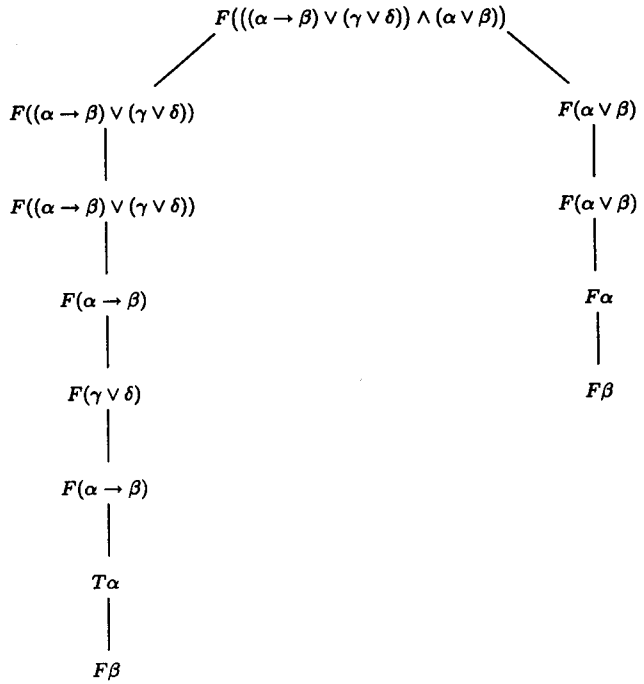


FIGURE 12

As the reader must have noticed, each time we select an entry it gets repeated at the end of the chosen path as part of the atomic tableau that we affix to the end of the path. As a notational convenience we will often omit this second occurrence when we draw tableaux although it remains part of the formal definition. (They will actually be needed when we consider the predicate calculus and so we included them in our formal definition.)

We now wish to describe those tableaux that will constitute proofs and a systematic procedure for generating them from given signed propositions. We need a number of auxiliary notions:

Definition 4.3: Let τ be a tableau, P a path on τ and E an entry occurring on P.

(i) E has been *reduced* on P if all the entries on one path through the atomic tableau with root E occur on P. (For example, TA and FA are reduced for every propositional letter A. $T\neg\alpha$ and $F\neg\alpha$ are reduced (on P) if $F\alpha$ and $T\alpha$ respectively appear on P. $T(\alpha \vee \beta)$ is reduced if either $T\alpha$ or $T\beta$ appears on P. $F(\alpha \vee \beta)$ is reduced if both $F\alpha$ and $F\beta$ appear on P.)

(ii) P is *contradictory* if, for some proposition α, $T\alpha$ and $F\alpha$ are both entries on P. P is *finished* if it is contradictory or every entry on P is reduced on P.

(iii) τ is *finished* if every path through τ is finished.

(iv) τ is *contradictory* if every path through τ is contradictory. (It is, of course, then finished as well.)

Example 4.4: Here is a finished tableau with three paths. The leftmost-path is contradictory; the other two are not.

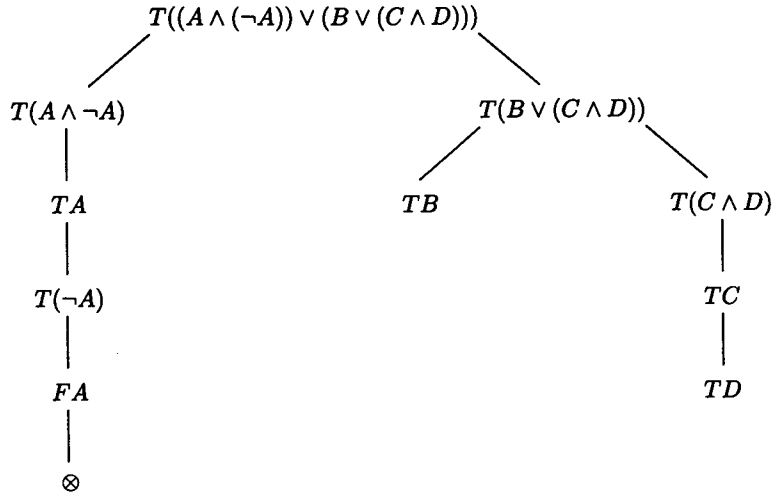


FIGURE 13

We can now define tableau proofs of α as ones that show that the assumption that α is false always leads to a contradiction:

Definition 4.5: A *tableau proof* of a proposition α is a contradictory tableau with root entry $F\alpha$. A proposition is *tableau provable*, written $\vdash \alpha$, if it has a tableau proof.

A *tableau refutation* for a proposition α is a contradictory tableau starting with $T\alpha$. A proposition is *tableau refutable* if it has a tableau refutation.

The following example is a tableau proof of an instance of Peirce's law. Remember that we don't actually recopy the entries that we are reducing. We put $\otimes$ at the end of a path to denote that it is contradictory.

Example 4.6: Peirce's Law.

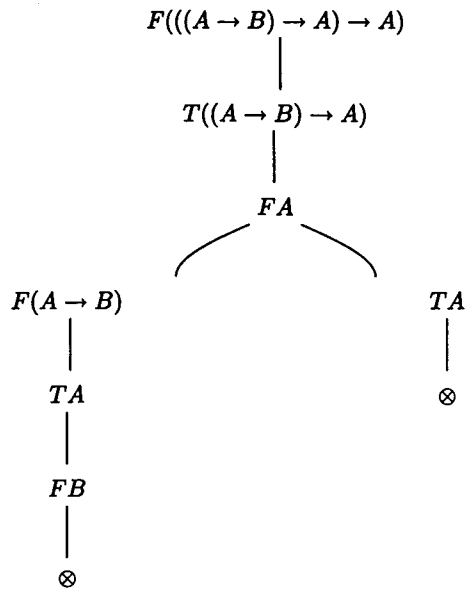


FIGURE 14

In much of what follows, for every definition or theorem dealing with a tableau proof or a logical truth (or both), there is a dual definition or theorem dealing with a tableau refutation and a logical falsehood respectively. It is left to the reader to provide these dual propositions.

The next step in producing proofs is to see that there is a finished tableau starting with any given signed proposition as root entry. We will describe a simple systematic procedure for producing such a tableau. A development that uses only finite tableaux is provided in the appendix to this section.

Definition 4.7 (Complete Systematic Tableaux)**:** Let R be a signed proposition. We define the *complete systematic tableau* (CST) with root entry R by induction. We begin the construction by letting τ_0 be the unique atomic tableau with R at its root. Assume that τ_m has been defined. Let n be the smallest level of τ_m containing an entry which is unreduced on some noncontradictory path in τ_m and let E be the leftmost such entry of level n. We now let τ_{m+1} be the tableau gotten by adjoining the unique atomic tableau with root E to the end of every noncontradictory path of τ_m on which E is unreduced. The union of the sequence τ_m is our desired complete systematic tableau.

Theorem 4.8: *Every* CST *is finished.*

Proof: Consider any entry E which occurs at some level n of the CST τ and lies on a noncontradictory path P in τ. There are at most finitely many entries on τ at or above level n. Thus, all the entries at level n or above on τ must be in place by some point of the construction. That is, there is an m_0 such that for every $m \geq m_0$, τ_m through level n is the same as τ through level n. Now, for $m \geq m_0$, the restriction of P to τ_m is a path in τ_m containing E. At each step $m \geq m_0$ in the construction of the CST we reduce the entry on the lexicographically least node labeled with an unreduced entry which is on some noncontradictory path in the tableau τ_m. If E is not already reduced on P by stage m_0, we can proceed for at most finitely many steps in this construction before E would become the lexicographically least unreduced entry. At this point in the construction we would reduce E. □

In allowing infinite tableaux, we seem to be in conflict with the intuition that proofs should be finite objects. However, by König's lemma, we can restrict our attention to finite contradictory tableaux.

Theorem 4.9: *If $\tau = \cup\tau_n$ is a contradictory tableau then for some m, τ_m is a finite contradictory tableau. Thus, in particular, if a* CST *is a proof, it is a finite tableau.*

Proof: τ is a finitely branching tree. Consider the subset of all nodes of τ with no contradiction above them. If this set is infinite, it has an infinite path by König's lemma. As this contradicts the assumption that every path in τ is contradictory, there are only finitely many such nodes. They must all appear by some level n of τ. Thus every node at level $n+1$ of τ has a contradiction above it. Once again, as τ through level $n+1$ is finite, there is an m such that τ_m is the same as τ through level $n+1$. Now every path P in τ_m is either a path in τ (ending with a leaf of level $\leq n$) or a path containing a node of level $n+1$. In the first case, P is contradictory by our assumption that τ is contradictory. In the second, P is contradictory by our choice of n and m. Thus τ_m is the desired contradictory tableau.

Note that if $\tau = \cup\tau_n$ is as in the definition of a CST and m is least such that τ_m is contradictory, then we cannot extend τ_m in the construction of τ. In this case $\tau = \tau_m$. □

In the next section, we will consider the import of this argument for the semantic as well as syntactic versions of the compactness theorem.

*Appendix: Finite tableaux suffice.

We wish to show that we can develop the results of this section without recourse to infinite tableaux. We first prove that there is a finite finished tableau with any given root. We then consider systematic procedures to generate such tableaux.

Theorem 4.10: *There is a finished finite tableau τ for each possible root entry $T\alpha$ or $F\alpha$.*

Proof: We proceed by induction on the depth of the given proposition α. If its depth is 0, i.e., α is a propositional letter, then the tableau consisting of just the signed propositional letter is finished. The point here is that signed propositional letters are themselves atomic tableaux. For the inductive case consider first case (2b): $F(\alpha \wedge \beta)$. By induction there are finished tableaux τ_α and τ_β with root entries $F\alpha$ and $F\beta$ respectively. We form the desired tableau with root $F(\alpha \wedge \beta)$ by beginning with the corresponding atomic tableau (2b) and then appending copies of τ_α and τ_β below the entries $F\alpha$ and $F\beta$ respectively. It is immediate from the definition of a finished tableau that this gives the desired result. The arguments for cases (3a), (3b), (4a) and (5a) are similar.

Next consider case (2a): $T(\alpha \wedge \beta)$. As before, we have finished tableaux τ_α and τ_β. We again begin our desired tableau τ with the appropriate atomic tableau (2a). To the end of this tableau we add a copy of τ_α to get a tableau τ' in which the only possible unreduced occurrence of an entry is that of $T\beta$ in the original atomic tableau. We now add a copy of τ_β to the end of every noncontradictory path in τ' to get our desired τ.

The cases (4b) and (5b) are similar. (6a) and (6b) are handled by applying this same procedure below each of the two distinct paths introduced by the atomic tableau to the two adjoined signed propositions which are put on these paths. □

The above proof actually gives a recursive procedure to construct a finished finite tableau with given root. The procedure is, however, somewhat complicated and hard to carry out. We now define a simpler systematic way of generating tableaux that will always produce a finished finite tableau with any given root entry. The idea is to always reduce the unreduced entry of greatest depth. We will see in the next section that this procedure will always produce a tableau proof of $F\alpha$.

Definition 4.11 (*Systematic Tableaux): We define **systematic tableaux* by induction:

(i) Every atomic tableau is a *systematic tableau.

(ii) If τ is a *systematic tableau and E is an entry of τ of maximal depth such that there is a noncontradictory path P of τ on which E is unreduced, then if τ' is the tableau gotten by adjoining the unique atomic tableau with root E to the end of every noncontradictory path P' of τ on which E is unreduced, then τ' is a *systematic tableau.

Our claim is that if we start with any *systematic tableau (and so in particular any atomic tableau) and repeatedly apply instances of the inductive step described in Definition 4.11(ii), then we eventually reach a finished tableau. Of course, once we have a finished tableau there is, by definition, no way of continuing this process. We can thus specify a sequence of steps that will eventually build a finished tableau with any given root entry.

Let τ_0 be the atomic tableau having the given entry as its root. Then, by induction, we form from τ_n the *systematic tableau, τ_{n+1}, gotten by applying the formation rule to say the (lexicographically) least E of maximal depth (as a proposition) which occurs unreduced on some noncontradictory path in τ_n.

The proof that this procedure eventually produces a finished tableau is a somewhat more complicated induction than the ones we have seen so far. We do not, however, actually need this result for later work. The simple existence of a finished tableau with any specified root entry (Theorem 4.10) suffices for the theorems of the next sections. Moreover, this procedure is not the most efficient way to produce such a finished tableau. We give examples and hints as to how one actually builds finished tableaux in the exercises.

The advantage of the *systematic tableaux approach is that the proof of termination, although somewhat complicated, is quite clean. It proceeds by induction on the pair consisting of the maximal depth of an entry in τ as described above and the number of unreduced occurrences of entries with this maximal depth. These pairs are ordered lexicographically and so progress is made in the induction when either the maximal depth of such an entry in this tableau is reduced or, failing that, when the number of unreduced occurrences of entries with the given maximal depth is reduced. (Exercise 1.7 says that the lexicographic ordering $<_L$ (used at the end of the proof of Theorem 4.12) of pairs of natural numbers is well founded, i.e., it has no infinite descending chains. Thus we may do a proof by induction on this ordering in the usual format: If, from the assumption that some property holds for every pair less than $\langle x, y\rangle$ in this ordering, we can prove that it holds for $\langle x, y\rangle$ as well, then we may conclude that it holds for every pair.)

Theorem 4.12: *There is no infinite sequence $\langle \tau_n \rangle$ of *systematic tableaux such that for each n, τ_{n+1} is gotten from τ_n by an application of the inductive clause (ii) of Definition 4.11.*

Proof: We proceed by induction on the lexicographic order of the pair $h(\tau_0) = \langle i, j\rangle$ where i is the maximal depth of any entry of τ_0 occurring unreduced on some noncontradictory path in τ_0 and j is the number of unreduced occurrences (on any noncontradictory path of τ_0) of entries of depth i. Now if $i = 0$, the only relevant entries are signed propositional letters. As all such are atomic tableaux, any occurrence of one is necessarily reduced. In this case then, there are no unreduced occurrences of entries ($j = 0$) and τ_0 itself is finished. (Remember that in this case there can be no τ_1 produced by applying clause (ii) to τ_0.) Thus, by induction, it suffices to prove that the application of clause (ii) of the definition of *systematic tableaux decreases h as we can then apply the induction hypothesis to the sequence beginning with τ_1. Consider a situation as described in Definition 4.11 (ii). Note that all occurrences of E in τ on noncontradictory paths in τ' are reduced: Each occurrence of E in τ on a noncontradictory path P in τ has been reduced by the addition of the atomic tableau with root E to end P. The only new occurrences of E in τ' are in these added atomic tableaux; these occurrences are already reduced by definition. Now, all the occurrences of entries other than E in τ' that are not ones of τ are nonroot

entries in the atomic tableau with root E. These all have depth less than that of E. Thus τ' has fewer occurrences of unreduced entries of depth that of E than τ and no new entries of greater depth. In other words, $h(\tau') <_L h(\tau)$ as required. □

The advantage of the original procedure for generating a CST is that it is not necessary to check every entry in the tableau to find the one E for which we must act. One simply checks the occurrences of entries level by level in lexicographic order until one is found which is unreduced. The *systematic procedure requires checking every entry to see that we have an unreduced one of maximal depth. The original procedure also always terminates with a finite finished tableau. Unfortunately the proof of termination for the CST procedure seems considerably more complicated than for the *systematic one. One can, however, use it in any particular example and simply notice that one has a finished tableau when it is produced.

Exercises

Give tableau proofs of each of the propositions listed in (1) – (7) below.

1. Idempotence and Commutativity of $\wedge$, $\vee$
   a) $(\alpha \vee \alpha) \leftrightarrow \alpha$
   b) $(\alpha \wedge \alpha) \leftrightarrow \alpha$
   c) $(\alpha \wedge \beta) \leftrightarrow (\beta \wedge \alpha)$
   d) $(\alpha \vee \beta) \leftrightarrow (\beta \vee \alpha)$
2. Associativity and Distributivity of $\wedge$, $\vee$
   a) $((\alpha \wedge \beta) \wedge \gamma) \leftrightarrow (\alpha \wedge (\beta \wedge \gamma))$
   b) $((\alpha \vee \beta) \vee \gamma) \leftrightarrow (\alpha \vee (\beta \vee \gamma))$
   c) $(\alpha \vee (\beta \wedge \gamma)) \leftrightarrow ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$
   d) $(\alpha \wedge (\beta \vee \gamma)) \leftrightarrow ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$
3. Pure Implication Laws
   a) $\alpha \rightarrow \alpha$
   b) $\alpha \rightarrow (\beta \rightarrow \alpha)$
   c) $(\alpha \rightarrow \beta) \rightarrow ((\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \gamma))$
   d) $(\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))$
4. Introduction and Elimination of $\wedge$
   a) $((\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \wedge \beta) \rightarrow \gamma)$
   b) $((\alpha \wedge \beta) \rightarrow \gamma) \rightarrow ((\alpha \rightarrow (\beta \rightarrow \gamma))$
5. De Morgan's Laws
   a) $\neg(\alpha \vee \beta) \leftrightarrow (\neg\alpha \wedge \neg\beta)$
   b) $\neg(\alpha \wedge \beta) \leftrightarrow (\neg\alpha \vee \neg\beta)$
6. Contrapositive
   $(\alpha \rightarrow \beta) \leftrightarrow (\neg\beta \rightarrow \neg\alpha)$

7. Double Negation
   $\alpha \leftrightarrow \neg\neg\alpha$
8. Contradiction
   $\neg(\alpha \wedge \neg\alpha)$
9. a) $(\neg\alpha \vee \beta) \rightarrow (\alpha \rightarrow \beta)$
   b) $(\alpha \rightarrow \beta) \rightarrow (\neg\alpha \vee \beta)$

Conjunctive and disjunctive normal forms

Recall from Exercise 3.3 that a conjunctive normal form (CNF) for a proposition α is a conjunct of disjuncts of literals (propositional letters or their negations) which is equivalent to α. Similarly, a disjunctive normal form (DNF) for α is a disjunct of conjuncts of literals which is equivalent to α. For any proposition α, we can find equivalent conjunctive and disjunctive normal forms by the following procedure:

(i) Eliminate all uses of $\leftrightarrow$ in the formation (tree) ofα by replacing any step going from β and γ to $\beta \leftrightarrow \gamma$ by one going to $(\beta \rightarrow \gamma) \wedge (\gamma \rightarrow \beta)$. This produces a proposition α_1 equivalent to α in which $\leftrightarrow$ does not occur.

(ii) Eliminate all uses of $\rightarrow$ in the formation of α_1 by replacing any step going to $\beta \rightarrow \gamma$ by one going to $\neg\beta \vee \gamma$. This produces an α_2 equivalent to α in which the only connectives are $\neg$, $\vee$ and $\wedge$.

(iii) Get a third equivalent proposition α_3 in which, in addition, $\neg$ appears only immediately before propositional letters by replacing in α_2 all occurrences of $\neg\neg\beta$ by β, of $\neg(\beta \vee \gamma)$ by $\neg\beta \wedge \neg\gamma$ and of $\neg(\beta \wedge \gamma)$ by $\neg\beta \vee \neg\gamma$.

(iv) Now use the associativity and distributivity laws above to get equivalents of α_3 which are either conjuncts of disjuncts of literals (CNF) or disjuncts of conjuncts of literals (DNF).

We provide an example of this procedure by finding both normal forms for the proposition $\alpha = (A \rightarrow B) \leftrightarrow \neg C$:

$$(A \rightarrow B) \leftrightarrow \neg C \qquad \text{(i)}$$
$$((A \rightarrow B) \rightarrow \neg C) \wedge (\neg C \rightarrow (A \rightarrow B)) \qquad \text{(ii)}$$
$$(\neg(\neg A \vee B) \vee \neg C) \wedge (\neg\neg C \vee (\neg A \vee B)) \qquad \text{(iii)}$$
$$((\neg\neg A \wedge \neg B) \vee \neg C) \wedge (\neg\neg C \vee (\neg A \vee B)) \qquad \text{(iii)}$$
$$((A \wedge \neg B) \vee \neg C) \wedge (C \vee (\neg A \vee B)) \qquad \text{(iii)}.$$

We can now apply step (iv) to get a CNF for α :

$$(A \vee \neg C) \wedge (\neg B \vee \neg C) \wedge (C \vee \neg A \vee B)$$

We can also use distributivity to produce a DNF for α :

$$(((A \wedge \neg B) \vee \neg C) \wedge C) \vee (((A \wedge \neg B) \vee \neg C) \wedge (\neg A \vee B))$$
$$(A \wedge \neg B \wedge C) \vee (\neg C \wedge C) \vee (A \wedge \neg B \wedge \neg A) \vee (\neg C \wedge \neg A) \vee (A \wedge \neg B \wedge B) \vee (\neg C \wedge B).$$

This last line is a DNF for α. It can, however be simplified by using some of the other rules proved above and simple truth table considerations. In particular, contradictions such as $C \wedge \neg C$ can be eliminated from disjuncts and tautologies such as $C \vee \neg C$ can be eliminated from conjuncts. Applying these procedures simplifies the DNF derived for α to the following:

$$(A \wedge \neg B \wedge C) \vee (\neg C \wedge \neg A) \vee (\neg C \wedge B).$$

10. Use the procedure described above to find CNF and DNF equivalents for the following propositions:
    a) $(A \rightarrow B) \leftrightarrow (A \rightarrow C)$
    b) $(A \leftrightarrow B) \rightarrow (C \vee D)$.
11. Use the laws provided in the above exercises to prove that each step of the above procedure produces a proposition equivalent to the original propositionα.

5. Soundness and Completeness of Tableau Proofs

We are going to prove the equivalence of the semantic notion of validity ($\models$) and the syntactic notion of provability ($\vdash$). Thus we will show that all tableau provable propositions are valid (soundness of the proof method) and that all valid propositions are tableau provable (completeness of the method).

Theorem 5.1 (Soundness): *If α is tableau provable, then α is valid, i.e.,* $\vdash \alpha \Rightarrow \models \alpha$.

Proof: We prove the contrapositive. Suppose α is not valid. By definition there is a valuation $\mathcal{V}$ assigning F to α. We say that the valuation $\mathcal{V}$ *agrees with* a signed proposition E in two situations: if E is $T\alpha$ and $\mathcal{V}(\alpha) = T$ or if E is $F\alpha$ and $\mathcal{V}(\alpha) = F$. We will show (Lemma 5.2) that if any valuation $\mathcal{V}$ agrees with the root node of a tableau, then there is a path P in the tableau such that $\mathcal{V}$ agrees with every entry on P. As no valuation can agree with any path on a contradictory tableau there can be no tableau proof of α. □

Lemma 5.2: *If $\mathcal{V}$ is a valuation which agrees with the root entry of a given tableau τ (given as in Definition 4.1 as $\cup\tau_n$), then τ has a path P every entry of which agrees with $\mathcal{V}$.*

Proof: We prove by induction that there is a sequence $\langle P_n \rangle$ such that, for every n, P_n is contained in P_{n+1} and P_n is a path through τ_n such that $\mathcal{V}$ agrees with every entry on P_n. The desired path P through τ will then simply be the union of the P_n. The base case of the induction is easily seen to be true by the assumption that $\mathcal{V}$ agrees with the root of τ. As an example, consider (6a) with root entry $T(\alpha \leftrightarrow \beta)$. If $\mathcal{V}(\alpha \leftrightarrow \beta) = T$,

then either $\mathcal{V}(\alpha) = T$ and $\mathcal{V}(\alpha) = T$ or $\mathcal{V}(\alpha) = F$ and $\mathcal{V}(\alpha) = F$ by the truth table definition for $\leftrightarrow$. We leave the verifications for the other atomic tableaux as Exercise 1.

For the induction step, suppose that we have constructed a path P_n in τ_n every entry of which agrees with $\mathcal{V}$. If τ_{n+1} is gotten from τ_n without extending P_n, then we let $P_{n+1} = P_n$. If P_n is extended in τ_{n+1}, then it is extended by adding on to its end an atomic tableau with root E for some entry E appearing on P_n. As we know by induction that $\mathcal{V}$ agrees with E, the same analysis as used in the base case shows that $\mathcal{V}$ agrees with one of the extensions of P_n to a path P_{n+1} in τ_{n+1}. □

Theorem 5.3 (Completeness): *If α is valid, then α is tableau provable, i.e., $\models \alpha \Rightarrow \vdash \alpha$. In fact, any finished tableau with root entry $F\alpha$ is a proof of α and so, in particular, the complete systematic tableaux with root $F\alpha$ is such a proof.*

The crucial idea in the proof of the completeness theorem is embodied in Lemma 5.4: We can always define a valuation which agrees with all entries on any noncontradictory path of any finished tableau.

Lemma 5.4: *Let P be a noncontradictory path of a finished tableau τ. Define a truth assignment $\mathcal{A}$ on all propositional letters A as follows:*

$\mathcal{A}(A) = T$ if TA is an entry on P.

$\mathcal{A}(A) = F$ otherwise.

If $\mathcal{V}$ is the unique valuation (Theorem 3.3) extending the truth assignment $\mathcal{A}$, then $\mathcal{V}$ agrees with all entries of P.

Proof: We proceed by induction on the depth of propositions on P.

(i) If α is a propositional letter and $T\alpha$ occurs on P, then $\mathcal{V}(\alpha) = T$ by definition and we are done. If $F\alpha$ occurs on P, then, as P is noncontradictory, $T\alpha$ does not and $\mathcal{V}(\alpha) = F$.

(ii) Suppose $T(\alpha \wedge \beta)$ occurs on the noncontradictory path P. Since τ is a finished tableau, both $T(\alpha)$ and $T(\beta)$ occur on P. By the induction hypothesis $\mathcal{V}(\alpha) = T = \mathcal{V}(\beta)$ and so $\mathcal{V}(\alpha \wedge \beta) = T$ as required.

(iii) Suppose $F(\alpha \wedge \beta)$ occurs on the noncontradictory path P. Again by the definition of a finished tableau, either $F\alpha$ or $F\beta$ must occur on P. Whichever it is, the induction hypothesis tells us that it agrees with $\mathcal{V}$ and so either $\mathcal{V}(\alpha) = F$ or $\mathcal{V}(\beta) = F$. In either case $\mathcal{V}(\alpha \wedge \beta) = F$ as required.

The remaining connectives are treated like one of these two cases depending on whether or not the corresponding atomic tableau branches. The details are left as Exercise 2. □

Proof (of Theorem 5.3): Suppose that α is valid and so $\mathcal{V}(\alpha) = T$ for every valuation $\mathcal{V}$. Consider any finished tableau τ with root $F\alpha$. (The CST with root $F\alpha$ is one by Theorem 4.8.) If τ had a noncontradictory path P there would be, by Lemma 5.5, a valuation $\mathcal{V}$ which agrees with all its entries and so in particular with $F\alpha$. This would give us a valuation with $\mathcal{V}(\alpha) = F$ contradicting the validity of α. Thus every path on τ is contradictory and τ is a tableau proof of α. □

It is clear from the proof of the completeness theorem (in fact from Lemma 5.4) that if you try to construct a tableau proof for α (i.e., one starting with $F\alpha$) and you do your best by constructing a finished tableau with root $F\alpha$ but fail to produce a proof of α (i.e., the finished tableau has at least one noncontradictory path) then the valuation defined by this noncontradictory path as in Lemma 5.4 gives us a counterexample to the assertion that α is valid. As we can always produce a finished tableau with any given root, we must, for every proposition, be able to get either a tableau proof or a counterexample to its validity!

It is this dichotomy (albeit expressed at the level of more complicated fragments of predicate logic) that forms the basis for constructive solutions to many problems. It is also the underlying rationale of PROLOG and of the implementation of other constructive theorem provers as programming languages. One starts with an assumption such as "there is no x such that $\mathcal{P}(x)$" and one either proves it true or finds a counterexample, that is, one actually produces an x such that $\mathcal{P}(x)$. We will consider these matters in II.5 and, in more detail, in Chapter III.

Exercises

1. Verify the remaining cases of atomic tableaux in Lemma 5.2.
2. Verify the cases for the remaining connectives in Lemma 5.4.

Reformulate and prove the analogs of the results of this section for tableau refutations and satisfiability:

3. If α is tableau refutable, i.e., there is a contradictory tableau with root $T\alpha$, then α is *unsatisfiable*, i.e., there is no valuation $\mathcal{V}$ such that $\mathcal{V}(\alpha) = T$.
4. If α is unsatisfiable then there is a tableau refutation of α.

6. Deductions from Premises and Compactness

Recall the treatment at the end of §3 of the consequences of a set Σ of propositions (which we called premises). A proposition σ is a consequence of Σ ($\Sigma \models \sigma$) if every valuation which is a model of Σ is also one of σ, i.e., every valuation which makes all the elements of Σ true also makes σ true. (See Definitions 3.2 and 3.8.) This notion of consequence and the associated one of a proof from given premises (which we are about to define) reflect common usage in mathematical arguments. A theorem is typically stated as an implication of the form $\alpha \rightarrow \beta$. The proof of the theorem,

however, is generally presented in a format which begins by assuming that the hypotheses (α) are true and then argues that the conclusion (β) must be true. Viewed syntactically in terms of proofs, we might describe this procedure as "assuming" α and then "deducing" β. The semantic notion of consequence captures the first view of such an argument. We now want to capture the syntactic or proof theoretic version by defining what it means to prove a proposition from a set of premises. Once we have developed the appropriate notions, a formal version of the informal mathematical method of argument described above is (easily) provided by the deduction theorem (Exercise 6). We now turn to the abstract formulation of the notions needed to express this result.

We begin our analysis with the definition of tableaux with premises from a set of sentences. It differs from the basic definition only in that we are allowed to add on entries of the form $T\alpha$ for premises α. This variation reflects the intuition that working from a set of premises means that we are assuming them to be true.

Definition 6.1 (Tableaux from Premises): Let Σ be a (possibly infinite) set of propositions. We define the *finite tableaux with premises from* Σ (or just *from* Σ for short) by induction:

(i) Every atomic tableau is a finite tableau from Σ.

(ii) If τ is a finite tableau from Σ and $\alpha \in \Sigma$, then the tableau formed by putting $T\alpha$ at the end of every noncontradictory path not containing it is also a finite tableau from Σ.

(iii) If τ is a finite tableau from Σ, P a path in τ, E an entry of τ occurring on P and τ' is obtained from τ by adjoining the unique atomic tableau with root entry E to the end of the path P, then τ' is also a finite tableau from Σ.

If $\tau_0, \tau_1, \ldots, \tau_n, \ldots$ is a (finite or infinite) sequence of finite tableaux from Σ such that, for each $n \geq 0$, τ_{n+1} is constructed from τ_n by an application of (ii) or (iii), then $\tau = \cup\tau_n$ is a *tableau from* Σ.

We can now define tableau proofs as before.

Definition 6.2: A *tableau proof of a proposition* α *from* Σ (or *with premises from* Σ) is a tableau from Σ with root entry $F\alpha$ which is contradictory, that is, one in which every path is contradictory. If there is such a proof we say that α *is provable from* Σ and write it as $\Sigma \vdash \alpha$.

Example 6.3: Figure 15 gives a tableau proof of A from the set of premises $\{\neg B, (A \vee B)\}$.

We can now mimic the development of the last section to prove the soundness and completeness theorems for deductions from premises. The only changes are in the definition of a finished tableau and the CST. A *finished tableau from* Σ is a tableau from Σ with $T\alpha$ on every noncontradictory path for every $\alpha \in \Sigma$. The idea here is again that we are incorporating the truth of the premises into the analysis.

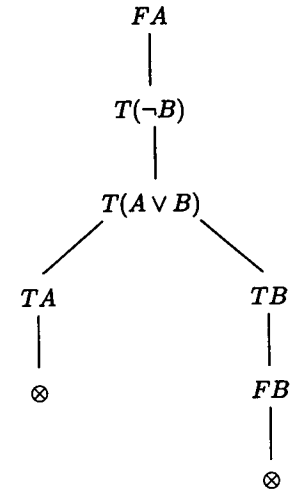


FIGURE 15

Similarly we must take steps in the construction of the CST *from* Σ to guarantee the appearance of these premises. We list the elements of Σ as α_m, $m \in \mathbb{N}$, and revise the definition of the CST by simply adding on one step to the definition of τ_{m+1}. If our new construction has produced τ_m we let τ'_{m+1} be the next tableau that would be defined by the standard CST procedure. (If that procedure would now terminate, we also terminate the current construction.) We now add on $T\alpha_m$ to the end of every noncontradictory path in τ'_{m+1} which does not already contain $T\alpha$ to form our new τ_{m+1}.

Theorem 6.4: *Every* CST *from a set of premises is finished.*

Proof: Exercise 1. □

The proofs of the soundness and completeness theorems can now be carried out as before with the caveat that we must always see to it that the propositions in Σ are true in the relevant valuations. We state the appropriate lemmas and theorems and leave most of their proofs in this setting as exercises.

Lemma 6.5: *If a valuation* $\mathcal{V}$ *makes every* $\alpha \in \Sigma$ *true and agrees with the root of a tableau* τ *from* Σ*, then there is a path in* τ *every entry of which agrees with* $\mathcal{V}$.

Proof: Exercise 2. □

Theorem 6.6 (Soundness of deductions from premises): *If there is a tableau proof of* α *from a set of premises* Σ*, then* α *is a consequence of* Σ*, i.e.,* $\Sigma \vdash \alpha \Rightarrow \Sigma \models \alpha$.

Proof: If not, there is a valuation which makes β true for every $\beta \in \Sigma$ but makes α false. Continue now as in the proof of Theorem 5.1. □

Lemma 6.7: *Let P be a noncontradictory path in a finished tableau τ from Σ. Define a valuation $\mathcal{V}$ as in Lemma 5.4. $\mathcal{V}$ then agrees with all entries on P and so in particular makes every proposition $\beta \in \Sigma$ true (as $T\beta$ must appear on P for every $\beta \in \Sigma$ by definition of a finished tableau from Σ).*

Proof: Exercise 3. □

Theorem 6.8 (Completeness of deductions from premises): *If α is a consequence of a set Σ of premises, then there is a tableau deduction of α from Σ, i.e., $\Sigma \models \alpha \Rightarrow \Sigma \vdash \alpha$.*

Proof: If $\Sigma \models \alpha$, every valuation $\mathcal{V}$ which makes every proposition in Σ true also makes α true. Consider the CST from Σ with root $F\alpha$. It is finished by Theorem 6.4. Now apply Lemma 6.7. □

Again we consider the problem of finiteness of proofs. The argument for Theorem 4.9 using König's lemma works just as before for tableaux from premises Σ even if Σ is infinite:

Theorem 6.9: *If $\tau = \cup\tau_n$ is a contradictory tableau from Σ, then, for some m, τ_m is a finite contradictory tableau from Σ. In particular, if a* CST *from Σ is a proof it is finite.*

Proof: Exercise 4. □

Thus we know that if α is provable from Σ then there is a finite tableau proof of it. This can be viewed as a syntactic version of the compactness theorem. Using the completeness and soundness theorems it can be converted into a semantic one:

Theorem 6.10 (Compactness): *α is a consequence of Σ iff α is a consequence of some finite subset of Σ.*

Proof: Exercise 5. □

We have left the indirect proof, via completeness and soundness, of the semantic version of the compactness theorem as an exercise. However, a direct proof of this result is also available. The compactness theorem is genuinely deeper than the others we have proven and deserves two proofs. An advantage of the direct approach is that the completeness theorem can be proved from the compactness theorem and without recourse to infinite tableaux. The direct approach also shows that compactness is simply a consequence of König's lemma.

Definition 6.11: A set Σ of propositions is called *satisfiable* if it has a model, i.e., there is a valuation $\mathcal{V}$ such that $\mathcal{V}(\alpha) = T$ for every $\alpha \in \Sigma$. We also say that such a valuation *satisfies* Σ.

Example 6.12:

(i) $\{A_1, A_2, (A_1 \wedge A_2), A_3, (A_1 \wedge A_3), A_4, (A_1 \wedge A_4), \ldots\}$ is a satisfiable infinite set of propositions.

(ii) $\{A_1, A_2, (A_1 \rightarrow A_3), (\neg A_3)\}$ is a finite set of propositions which is not satisfiable nor is any set containing it.

Theorem 6.13 (Compactness): *Let $\Sigma = \{\alpha_i \mid i \in \omega\}$ be an infinite set of propositions. Σ is satisfiable if and only if every finite subset Γ of Σ is satisfiable.*

Proof: Note that the "only if" direction of the theorem is trivially true; the other direction is not (not *trivially* that is). The problem is that finding different valuations which satisfy longer and longer initial segments does not necessarily mean that there is a single valuation satisfying the whole sequence. Building such a valuation is essentially an application of König's lemma.

Let $\langle C_i \mid i \in \omega \rangle$ be a list of all the propositional letters. We define a tree T whose nodes are binary sequences ordered by extension. We use $lth(\sigma)$ to denote the length of a sequence σ and set $T = \{\sigma \mid$ there is a valuation $\mathcal{V}$ such that, for $i \leq lth(\sigma)$, $\mathcal{V}(\alpha_i) = T$ and $\mathcal{V}(C_i) = T$ iff $\sigma(i) = 1\}$. What this definition says is that we put σ on the tree unless interpreting it as an assignment of truth values to the propositional letters C_i $(i \leq lth(\sigma))$ already forces one of the α_i to be false for $i \leq lth(\sigma)$.

Claim: There is an infinite path in T if and only if Σ is satisfiable.

Proof of Claim: If $\mathcal{V}$ satisfies Σ then, by definition, the set of all σ such that $\sigma(i) = 1$ iff $\mathcal{V}(C_i) = T$ is a path on T. On the other hand, suppose that $\langle \sigma_j \mid j \in \mathbb{N} \rangle$ is an infinite path on T. Let $\mathcal{V}$ be the unique valuation extending the assignment determined by the σ_j, i.e., the one for which C_i is true iff $\sigma_j(i) = 1$ for some j (or equivalently, as the σ_i are linearly ordered by extension, iff $\sigma_j(i) = 1$ for every i such that $i \leq lth(\sigma_j)$). If $\mathcal{V} \not\models \Sigma$, then there is some $\alpha_j \in \Sigma$ such that $\mathcal{V}(\alpha_j) = F$. Now by Corollary 3.4 this last fact depends on the truth values assigned by $\mathcal{V}$ to only finitely many propositional letters. Let us suppose it depends only on those C_i with $i \leq n$. It is then clear from the definition of T that no σ with length $\geq n$ can be on T at all. As there are only finitely many binary sequences σ with length $\leq n$, we have contradicted the assumption that the sequence $\langle \sigma_j \rangle$ is an infinite path on T and so $\mathcal{V} \models \Sigma$ as claimed.

The next claim is that there is, for every n, a σ of length n in T. By assumption every finite subset of Σ is satisfiable. Thus, for each n, there is a valuation $\mathcal{V}_n$ which makes α_i true for each $i \leq n$. The string σ given by $\sigma(i) = 1$ iff $\mathcal{V}_n(C_i) = T$ for $i \leq n$ is then on T by definition.

König's lemma (Theorem 1.4) now tells us that there is an infinite path in T and so Σ is satisfiable as required. □

The connection between this version of the compactness theorem for propositional logic and the compactness theorem of topology is considered in Exercises 9 and 10. Other applications of the compactness theorem can be found in Exercises 7 and 8.

*Finite tableaux suffice.

We wish to outline the development of the soundness and completeness theorems from premises based on the semantic proof of the compactness theorem without recourse to infinite tableaux. Now, if the given set of premises Σ is finite there is no trouble in proving the soundness and completeness results of §5 for proofs from Σ. One can follow the path outlined in the appendix to that section but still consider only finite tableau. One simply requires that a finished tableau have $T\alpha$ on every noncontradictory path for every $\alpha \in \Sigma$. The proofs (including that of Theorem 4.10 that there is always a finished finite tableau from Σ with any given root entry) can be carried over without difficulty.

The situation for infinite sets of premises seems more complicated. Proofs, even from infinite sets of premises, must now be finite objects. Including this restriction as part of the definition of a proof from premises can cause no difficulties in the proof of soundness since the hypothesis of the theorem starts us with such a proof. Thus the soundness theorem for finite proofs is simply a special case of Theorem 6.5.

The problem with considering only finite proofs comes in the proof of the completeness theorem. There we begin with a finished tableau τ with root $F\alpha$. If τ is contradictory, it is a proof of α. If not, we use a noncontradictory path in τ to define a valuation that agrees with Σ but makes α false. The existence of such a valuation contradicts the assumption that α is a consequence of Σ.

Now if Σ is infinite and τ being finished means that $T\alpha$ appears on every noncontradictory path on τ, then it seems as if we have to consider infinite tableaux. We can avoid this by appealing to the compactness theorem (6.13). It essentially reduces the infinite case to the finite one. Thus we can use the compactness theorem to reduce the completeness theorem for finite proofs from infinite sets of premises to the one for finite sets of premises.

Theorem 6.14 (Completeness of deductions from premises): *If α is a logical consequence of a set of premises Σ, then there is a tableau proof of α from Σ, i.e., $\Sigma \models \alpha \Rightarrow \Sigma \vdash \alpha$.*

Proof: As $\Sigma \models \alpha$, $\Sigma \cup \{\neg\alpha\}$ is not satisfiable by definition. By the compactness theorem (6.13), there is a finite $\Gamma \subset \Sigma$ such that $\Gamma \cup \{\neg\alpha\}$ is not satisfiable. Again, by definition, this means that $\Gamma \models \alpha$. The completeness theorem for finite sets of premises outlined above (see also Exercise 12) now tells us that there is a proof of α from Γ. As $\Gamma \subset \Sigma$, this proof is also one from Σ. □

Exercises

1. Prove Theorem 6.4.
2. Prove Lemma 6.5.
3. Follow the proof of Lemma 5.4 to prove Lemma 6.7.
4. Follow the proof of Theorem 4.9 to prove Theorem 6.9.
5. Deduce Theorem 6.10 from the results preceding it.
6. **Deduction Theorem**: Let Σ be a finite set of propositions and $\bigwedge\Sigma$ the conjunction of its members. Prove that for any proposition α the following are equivalent:
   (i) $\Sigma \models \alpha$.
   (ii) $\models \bigwedge\Sigma \to \alpha$.
   (iii) $\Sigma \vdash \alpha$.
   (iv) $\vdash \bigwedge\Sigma \to \alpha$.

Applications of Compactness

For problems 7 and 8, use the compactness theorem for propositional logic or König's lemma. The key point in each case is to faithfully translate the given problem into an appropriate set of propositions (or an appropriate tree). One then applies compactness or König's lemma. Finally, one must translate the result of this application back into the terms of the problem. These problems are treated in predicate logic in Exercises II.7.5.

7. A partial order has *width at most n* if every set of pairwise incomparable elements has size at most n. A chain in a partial order $<$ is simply a subset of the order which is linearly ordered by $<$. Prove that an infinite partial order of width at most 3 can be divided into three chains (not necessarily disjoint) if every finite order of width at most 3 can be so divided.

   Hint (using compactness): Let the elements of the order be $\{p_n | n \in \mathbb{N}\}$. Consider propositions Rp_ip_j, Ap_i, Bp_i and Cp_i for $i, j \in \mathbb{N}$. Think of Rp_ip_j as saying that $p_i < p_j$. Think of Ap_i as saying that p_i is in chain A and similarly for Bp_i and Cp_i. Now write down the *sets* of propositions expressing the desired conclusions: Each of A, B and C is a chain; every element is in A, B or C; the order has width 3.

   Note: Dilworth's theorem states that any partial order of width at most n can be divided into n chains. Thus Dilworth's theorem for infinite orders follows from the theorem for finite orders by compactness. As the finite case is proved by induction on the size of the given order this is a nontrivial application of compactness.

8. A *graph* G is a set of elements $\{a_0, a_1, \dots\}$ called nodes and a set of pairs of nodes $\{a_i, a_j\}$ called edges. We say that G is *n-colorable* if we can label its nodes with n colors $C_1, \dots, C_n$ so that no two nodes in a single edge of G have the same color. Suppose every finite

subgraph of G (a finite subset of the nodes and the edges between them) is 4–colorable. Prove that G is 4–colorable.

Hint (Using König's lemma): Define a tree of 4–ary sequences ordered by extension. Put a sequence σ of length $n+1$ on the tree if and only if it defines a 4–coloring of the nodes $a_0, a_1, \ldots, a_n$ by coloring a_j with color $C_{\sigma(j)}$.

Note: The four color theorem says that every planar graph is 4–colorable. By this exercise, it suffices to prove the theorem for finite graphs as a graph is planar if and only if all its finite subgraphs are planar.

Connections with topological compactness and König's Lemma

The compactness theorem for propositional logic can be connected to the topology on the set $\mathcal{T}$ of all possible truth valuations which is determined by letting the open sets be generated by those of the form $\{\mathcal{V} : (\exists \alpha \in \Sigma)(\mathcal{V} \not\models \alpha)\}$ for any set Σ of propositions.

9. Prove that the space $\mathcal{T}$ with this topology is compact.
10. Deduce the nontrivial direction of the semantic version of the compactness theorem (6.13). Hint: Prove the contrapositive from the open cover property.
11. Prove König's lemma from Theorem 6.13.

See also Exercises 1.10 and 1.11 for other connections between König's lemma and topological compactness.

Finite Sets of Premises and Tableau Proofs

12. Give a direct proof of the completeness theorem for deductions from finite sets of premises that could be used in the proof of Theorem 6.14.

7*. An Axiomatic Approach

Propositional calculus (as well as other mathematical systems) are often formulated as a collection of *axioms* and *rules of inference*. The axioms of propositional logic are certain valid propositions. A rule of inference, R, in general, "infers" a proposition α from certain n–tuples $\alpha_1, \ldots, \alpha_n$ of propositions in a way that is expected to preserve validity. Thus, for R to be an acceptable rule of inference, it must be true that, if one can use R to infer α from the valid propositions $\alpha_1, \ldots, \alpha_n$, then α must be valid as well.

We now give a brief description of one such classical formulation based on the adequate set of connectives $\{\neg, \rightarrow\}$. (For simplicity we view the other connectives as defined from $\neg$ and $\rightarrow$. This considerably reduces the number of axioms needed.)

7.1 Axioms: The axioms of our system are all propositions of the following forms:

(i) $(\alpha \rightarrow (\beta \rightarrow \alpha))$

(ii) $((\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)))$

(iii) $(\neg\beta \rightarrow \neg\alpha) \rightarrow ((\neg\beta \rightarrow \alpha) \rightarrow \beta)$

where α, β and γ can be any propositions.

The forms in this list are often called *axiom schemes*. The axioms are all instances of these schemes as α, β and γ vary over all propositions. It is easy to check that these axioms are all valid. Their choice will, in some sense, be justified a bit later. Our system has only one rule of inference called *modus ponens*.

7.2 The Rule of Inference (Modus Ponens):
From α and $\alpha \rightarrow \beta$, we can infer β. This rule is written as follows:

$$\frac{\alpha \qquad \alpha \rightarrow \beta}{\beta}\ .$$

Systems based on axioms and rules in the style of the one presented above are generally called Hilbert–style proof systems. We therefore denote provability in this system by $\vdash_H$.

Definition 7.3: Let Σ be a set of propositions.

(i) A *proof from* Σ is a finite sequence $\alpha_1, \alpha_2, \ldots \alpha_n$ such that for each $i \leq n$ either:

(1) α_i is a member of Σ;

(2) α_i is an axiom;

or

(3) α_i can be inferred from some of the previous α_j by an application of a rule of inference.

(ii) α is *provable from* Σ, $\Sigma \vdash_H \alpha$, if there is a proof $\alpha_1, \ldots, \alpha_n$ from Σ where $\alpha_n = \alpha$.

(iii) A *proof* of α is simply a proof from the empty set $\emptyset$; α is *provable* if it is provable from $\emptyset$.

Example 7.4: Here is a proof of $((\neg\beta \rightarrow \alpha) \rightarrow \beta)$ from $\Sigma = \{\neg\alpha\}$:

$\neg\alpha$	from Σ
$(\neg\alpha \rightarrow (\neg\beta \rightarrow \neg\alpha))$	axiom (i)
$(\neg\beta \rightarrow \neg\alpha)$	modus ponens
$((\neg\beta \rightarrow \neg\alpha) \rightarrow ((\neg\beta \rightarrow \alpha) \rightarrow \beta))$	axiom (iv)
$((\neg\beta \rightarrow \alpha) \rightarrow \beta)$	modus ponens.

We should note here, as we did for tableau deductions, that, although the set of premises Σ may be infinite, if α is provable from Σ then α is provable from a finite subset of Σ. Proofs are always finite!

The standard theorems are again soundness, completeness and compactness. Soundness is fairly easy to prove. One has only to check that the axioms are all valid and the rule of inference (modus ponens) preserves truth, i.e., if the premises are true for some valuation then so is the conclusion. The syntactic version of the compactness theorem is immediate in this setting as all proofs are finite. The semantic version (as stated in Theorem 6.13) remains nontrivial. Of course, the semantic proof given there also remains applicable. The theorem can also be derived from the completeness theorem for this rule–based system (which must therefore be nontrivial).

We omit the proofs of soundness and completeness for this particular system (they can be found in Mendelson [1979, 3.2] but in the next section we will consider another rule–based system and will supply the proofs of such results. For now, we simply state the theorems for the system presented here.

Theorem 7.5 (Soundness and Completeness from Premises): *α is provable from a set of propositions Σ if and only if α is a consequence of Σ, i.e.,* $\Sigma \vdash_H \alpha \Leftrightarrow \Sigma \models \alpha$.

Corollary 7.6 (Soundness and Completeness): *A proposition α is provable if and only if it is valid, i.e.,* $\vdash_H \alpha \Leftrightarrow \models \alpha$.

Remarks 7.7:

(i) *On modus ponens*: If α has a tableau proof and $\alpha \to \beta$ has a tableau proof, then α and $\alpha \to \beta$ are both valid by the soundness theorem. As modus ponens preserves validity, β is also valid. Thus by the completeness theorem for tableau proofs, β has a tableau proof. There is actually an algorithm for getting a tableau proof for β from such proofs for α and $\alpha \to \beta$. This is known as the Gentzen Hauptsatz (principle theorem) and is too long to prove here. Modus ponens is also called the cut rule and this theorem is therefore referred to as a cut elimination theorem.

(ii) *On theorems*: A *theorem* is any proposition which is provable. So any proposition which occurs as an element in a proof is a theorem. We usually think of the *conclusion* as being the last element of a proof but, any initial segment of a proof is also a proof.

(iii) *Choice of axioms*: The corollary says that the axioms are complete in the sense that we can prove any valid proposition from them by repeated applications of modus ponens. On the other hand, since the axioms are valid and modus ponens preserves validity, every theorem (i.e., every proposition provable in this system) has a tableau proof. Thus tableau proofs are sufficient and so are the axioms and rules of inference listed above. One could have more axioms (or fewer) or more (or other) rules of inference or both. Sometimes it is a matter of taste, other times a matter of expediency (e.g., what makes various proofs easier). The key point is that whatever the proof system, there is really only one set of theorems, the valid propositions.

(iv) *Efficiency*: Proving theorems efficiently from such a system of axioms and rules may be somewhat tricky since you often have to guess which axiom to use rather than having a systematic procedure as is the case for the tableaux. The source of this problem is having a plethora of axioms from which to choose. The Hilbert–style proof system presented here has many axioms and few rules. Other systems which reverse the emphasis are Gentzen systems and natural deduction systems. These are much more relevant to automatic theorem proving and, in their intuitionistic or constructivist forms, to producing systems which have the property of always being able to produce a proof or counterexample for any given proposition (as discussed at the end of §5).

8. Resolution

The proof method underlying PROLOG and most automatic theorem provers is a particularly simple and efficient system of axioms and rules called *resolution*. Like the system presented in §7, resolution has only one rule. It reduces the large amount of the guesswork involved in producing a proof by essentially eliminating all axioms. (Actually it incorporates them automatically via various formatting rules but as far as the work of producing the proof is concerned, this almost amounts to their elimination.) The resolution method, like our version of the tableau method, is a refutation procedure. That is, it tries to show that the given formula is unsatisfiable. It begins by assuming that the formula of interest is in conjunctive normal form (see Exercises 3.3 and 4.8). In typical computer science treatments this form is called *clausal form* and the associated terminology is as follows:

Definition 8.1:

(i) A *literal* ℓ is a propositional letter p or its negation $\neg p$. If ℓ is p or $\neg p$, we write $\bar{\ell}$ for $\neg p$ or p respectively. The propositional letters are also called *positive literals* and their negations *negative literals*.

(ii) A *clause* C is a finite set of literals (which you should think of as the disjunction of its elements). As we think of C as being true iff one of its elements is true, the *empty clause* $\Box$ is always false — it has no true element.

(iii) A *formula* S is a (not necessarily finite) set of clauses (which you should think of as the conjunction of its elements). As we think of a formula S as being true if every one of its elements is true, the *empty formula* $\emptyset$ is always true — it has no false element.

(iv) An *assignment* $\mathcal{A}$ is a consistent set of literals, i.e., one not containing both p and $\neg p$ for any propositional letter p. (This, of course, is just the (partial) truth assignment in which those $p \in \mathcal{A}$ are assigned T and those q with $\bar{q} \in \mathcal{A}$ are assigned F.) A *complete assignment* is one containing p or $\neg p$ for every propositional letter p. It corresponds to what we called a truth assignment in Definition 3.1.

(v) *$\mathcal{A}$ satisfies S*, $\mathcal{A} \models S$, iff $\forall C \in S(C \cap \mathcal{A} \neq \emptyset)$, i.e., the valuation induced by $\mathcal{A}$ makes every clause in S true.

(vi) A formula S is *(un)satisfiable* if there is an (no) assignment $\mathcal{A}$ which satisfies it.

Examples 8.2:

(i) p, q, r, $\neg p$, $\bar{q}(=\neg q)$, $\bar{r}$ and $\neg\bar{q}(=q)$ are literals.

(ii) $\{p, r\}$, $\{\neg q\}$, and $\{q, \neg r\}$ are clauses.

(iii) $S = \{\{p,r\}, \{q, \neg r\}, \{\neg q\}, \{\neg p, t\}, \{s, \neg t\}\}$ is a formula which, in our original notation system, would be written as $((p \vee r) \wedge (q \vee \neg r) \wedge (\neg q) \wedge (\neg p \vee t) \wedge (s \vee \neg t))$.

(iv) If $\mathcal{A}$ is given by $\{p, q, r, s, t\}$, i.e., the (partial) assignment such that $\mathcal{A}(p) = T = \mathcal{A}(q) = \mathcal{A}(r) = \mathcal{A}(s) = \mathcal{A}(t)$, then $\mathcal{A}$ is an assignment not satisfying the formula S in (iii). S is, however, satisfiable.

*PROLOG Notation:

Another way of thinking of clausal or conjunctive normal form is in terms of implications. Suppose we have a clause C whose *positive literals* (the propositional letters contained in C) are $A_1, \ldots, A_m$ and whose *negative literals* (the propositional letters p such that $\bar{p}$ (i.e., $(\neg p)$) is an element of C) are $B_1, \ldots, B_n$. The clause C is then equivalent to $A_1 \vee A_2 \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_n$. This in turn is equivalent to $B_1 \wedge B_2 \ldots \wedge B_n \rightarrow A_1 \vee \ldots \vee A_m$. If there is at most one positive literal (i.e., at most one A_i) in C then C is called a *Horn clause* (or a *program clause* if it has exactly one positive literal). If the Horn clause contains some negative literals it is a *rule*, otherwise a *fact*. A *goal clause* is one with no positive literals. It is the logic and proof theory of these clauses (which we analyze in section 10) that is the heart of PROLOG. (PROLOG is, however, not limited to propositional letters. It also allows for variables in its literals. We will elaborate on this when we deal with predicate logic in chapter II.)

The standard notations in PROLOG reverses the order used in $\rightarrow$ and instead use either $\leftarrow$ or :- which are read "if". Occurrences of the $\wedge$ symbol are replaced by commas. Thus A_1 :- $B_1, B_2, \ldots, B_n$ or $A_1 \leftarrow B_1, \ldots, B_n$ is read (and means) A_1 if (B_1 and B_2 and ... and B_n). In terms of generating deductions or writing programs one thinks of the assertion of a clause C such as A_1 :- $B_1, \ldots, B_n$, as specifying conditions under which A_1 is true. We are usually interested in establishing some result. Thus A_1 is called the *goal* of the clause C (or at times the *head* of C) and $B_1, \ldots, B_n$ the subgoals (or *body* and with this terminology the symbol " :- " is called the *neck*) of C. The idea is that C tells us that to establish A we should first establish each of $B_1, \ldots, B_n$. Along with the goal — subgoal terminology come the terms succeed and fail. One says a goal A succeeds if it is true, or more precisely from the programming point of view, if we have a proof of A. Otherwise we say the goal fails. Be warned, however, that this terminology of success and failure is (at least for now) somewhat imprecise.

It is worth noting what these views imply for the meaning of the degenerate cases of the notation :- , i.e., when $n = 0$ or $m = 0$. If $m = 0$ then :- $B_1, \ldots, B_n$ (or $\leftarrow B_1, \ldots, B_n$), called a *goal clause*, is equivalent to $\neg B_1 \vee \ldots \vee \neg B_n$, i.e., it asserts that one of the B_i fail (is false). If $n = 0$ then A_1 :- (or $A_1 \rightarrow$), called a *unit clause*, is equivalent to simply A_1, thus this notation simply says that A_1 succeeds (is true).

The *resolution rule* is much like a version of modus ponens called **cut**. Modus ponens (see §7) says that from α and $\alpha \rightarrow \beta$ one can infer β. In this format, the cut rule says that from $\alpha \vee \gamma$ and $\neg\alpha \vee \beta$ infer $\gamma \vee \beta$. Thus cut is somewhat more general than modus ponens in that it allows one to carry along the extra proposition γ. Resolution is a restricted version of cut in which α must be a literal while β and γ must be clauses.

Definition 8.3 (*Resolution*): In our current terminology, we say that, from clauses C_1 and C_2 of the form $\{\ell\} \sqcup C_1'$ and $\{\bar{\ell}\} \sqcup C_2'$, infer $C = C_1' \cup C_2'$ which is called a *resolvent* of C_1 and C_2. (Here ℓ is any literal and $\sqcup$ means that we are taking a union of disjoint sets.) We may also call C_1 and C_2 the *parent* and C their *child* and say that we *resolved on* (the literal) ℓ.

(Note that, compared to the classical form of the cut rule, the resolution rule also eliminates redundancies, i.e., letters common to C_1 and C_2. This takes the place of certain axioms in a classical proof system such as the Hilbert–style one of §7.)

Resolution is, of course, a sound rule, that is, it preserves satisfiability by any given truth assignment. If some assignment satisfies both C_1 and C_2, whatever it does for p it must satisfy one of C_1' or C_2'. (This argument is formalized in Lemma 8.12.) It can thus be used as the basis of a sound proof procedure.

Definition 8.4: A (*resolution*) *deduction* or *proof of C from a given formula S* is a finite sequence $C_1, C_2, \ldots, C_n = C$ of clauses such that each C_i is either a member of S or a resolvent of clauses C_j, C_k for $j, k < i$. If there is such a deduction, we say that C *is (resolution) provable from* S and write $S \vdash_{\mathcal{R}} C$. A deduction of $\square$ from S is called a (*resolution*) *refutation of* S. If there is such a deduction we say that S *is* (*resolution*) *refutable* and write $S \vdash_{\mathcal{R}} \square$.

Warning: A resolution refutation of S gives a proof of $\square$ from S. As $\square$ is always false, we should think of this as showing that S can never be true, i.e., S is unsatisfiable. This will be the content of the soundness theorem (Theorem 8.11).

Examples 8.5:

(i) From $\{p, r\}$ and $\{\neg q, \neg r\}$ conclude $\{p, \neg q\}$ by resolution (on r).

(ii) From $\{p, q, \neg r, s\}$ and $\{\neg p, q, r, t\}$ we could conclude either $\{q, \neg r, s, r, t\}$ or $\{p, q, s, \neg p, t\}$ by resolution (on p or r) respectively. Of course, both of these clauses are valid and are equivalent to the empty formula.

A more useful picture of a resolution proof is as a tree of deductions rather than just the sequence described above.

Definition 8.6: A *resolution tree proof of C from S* is a labeled binary tree T with the following properties:

(i) The root of T is labeled C.

(ii) The leaves of T are labeled with elements of S.

(iii) If any nonleaf node σ is labeled with C_2 and its immediate successors σ_0, σ_1 are labeled with C_0, C_1 respectively, then C_2 is a resolvent of C_0 and C_1.

Example 8.7: Here is a resolution tree refutation of the formula $S = \{\{p,r\}, \{q,\neg r\}, \{\neg q\}, \{\neg p,t\}, \{\neg s\}, \{s,\neg t\}\}$, i.e., a resolution tree proof of $\square$ from S:

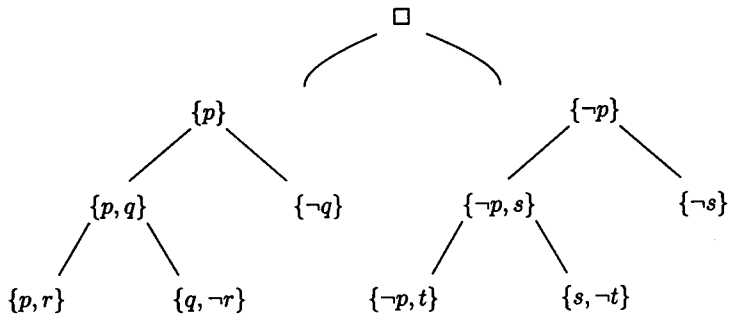


FIGURE 16

Lemma 8.8: *C has a resolution tree proof from S if and only if there is a resolution deduction of C from S.*

Proof: ($\Rightarrow$) List all the labels of the nodes σ of the tree proof of C from S in any order that reverses the $<$ ordering of the tree (so leaves are listed first and the root last). This sequence can be seen to be a resolution deduction of C from S by simply checking the definitions.

($\Leftarrow$) We proceed by induction on the length of the resolution deduction of C from S. Suppose we can get tree proofs for any deduction of length $< n$ and $C_1, \ldots, C_n$ is one of length n from S. If $C_n \in S$ there is nothing to prove. If not, then C_n is the resolvent of C_i and C_j for some i and j less than n. By induction, we have tree proofs T_i and T_j of C_i and C_j. Let T_n be the tree whose root is labeled C and to whose immediate successors we attach T_i and T_j. Again, by definition, this is the desired tree proof. $\square$

Yet another picture of resolution deduction corresponds to the inductive definition of the set of theorems or clauses provable from S.

Definition 8.9: $\mathcal{R}(S)$ is the *closure of S under resolution*, i.e., the set determined by the following inductive definition:

1) If $C \in S$, $C \in \mathcal{R}(S)$.

2) If C_1, $C_2 \in \mathcal{R}(S)$ and C is a resolvent of C_1 and C_2 then $C \in \mathcal{R}(S)$.

Proposition 8.10: *For any clause C and formula S, there is a resolution deduction of C from S iff $C \in \mathcal{R}(S)$. In particular, there is a resolution refutation of S iff $\square \in \mathcal{R}(S)$.*

Proof: Exercise 1. $\square$

The first observation to be made is that no matter how the resolution method is described, it gives a sound proof procedure.

Theorem 8.11 (Soundness of Resolution)**:** *If there is a resolution refutation of S then S is unsatisfiable.*

We first prove a lemma which is needed for the inductive step in the proof of the theorem.

Lemma 8.12: *If the formula (i.e., set of clauses) $S = \{C_1, C_2\}$ is satisfiable and C is a resolvent of C_1 and C_2, then C is satisfiable. Indeed, any assignment $\mathcal{A}$ satisfying S satisfies C.*

Proof: As C is a resolvent of C_1 and C_2, there are ℓ, C_1' and C_2' such that $C_1 = \{\ell\} \sqcup C_1'$, $C_2 = \{\bar{\ell}\} \sqcup C_2'$ and $C = C_1' \cup C_2'$. As $\mathcal{A}$ satisfies $\{C_1, C_2\}$, it satisfies (that is it contains an element of) each of C_1 and C_2. As $\mathcal{A}$ is an assignment, it cannot be the case that both $\ell \in \mathcal{A}$ and $\bar{\ell} \in \mathcal{A}$. Say $\bar{\ell} \in \mathcal{A}$. As $\mathcal{A} \models C_2$ and $\bar{\ell} \notin \mathcal{A}$, $\mathcal{A} \models C_2'$ and so $\mathcal{A} \models C$. The proof for $\ell \notin \mathcal{A}$ just replaces C_2 by C_1. $\square$

Proof of Theorem 8.11: If $C_1, \ldots, C_n$ is a resolution deduction from S then the lemma shows by induction (on n) that any assignment satisfying S satisfies every C_i. If the deduction is in fact a refutation of S then $C_n = \square$. As no assignment can satisfy $\square$, S is unsatisfiable. $\square$

Remark 8.13: The soundness theorem and its proof could just as well have been phrased directly in terms of Definitions 8.6 or 8.9. We leave these formulations as exercises 2 and 3.

Our next major goal is to prove that the resolution method is complete, i.e., if S is unsatisfiable then there is a resolution refutation of S. We will then want to consider ways of implementing a search for a refutation of S. We will first consider using the resolution method as originally presented. We then introduce more and more restrictive versions of resolution which are designed to make the search more efficient without rendering the method either unsound or incomplete. Following this line of development, we will first present a simple direct proof of the completeness of the general form of resolution given in Definition 8.3. This proof will, however, rely on the (semantic form of the) compactness theorem. We will then introduce and analyze a somewhat abstract description of unsatisfiability. It will supply us with a proof of the completeness theorem for resolution deduction

which does not rely on the compactness theorem and a new proof of the compactness theorem. That proof of completeness will be the paradigm for the completeness proofs of the restricted version of resolution presented in §9.

We begin our first path to completeness with a lemma that will allow us to eliminate literals in clauses which are resolution deducible from an unsatisfiable formula S. Repeated applications of the lemma will show that $\square$, the clause with no literals, is deducible from S.

Lemma 8.14: *For any formula T and any literal ℓ, let $T(\ell) = \{C \in \mathcal{R}(T) \mid \ell, \bar{\ell} \notin C\}$. If T is unsatisfiable, then so is $T(\ell)$.*

Proof: Assume T is unsatisfiable and suppose, for the sake of a contradiction, that $\mathcal{A}$ is any assignment which satisfies $T(\ell)$ and is defined on all the literals (of T) other than ℓ. Let $\mathcal{A}_1 = \mathcal{A} \cup \{\ell\}$ and $\mathcal{A}_2 = \mathcal{A} \cup \{\bar{\ell}\}$. As T is unsatisfiable, there are clauses C_1 and C_2 in T such that $\mathcal{A}_1 \not\models C_1$ and $\mathcal{A}_2 \not\models C_2$. Now as $\ell \in \mathcal{A}_1$ and $\mathcal{A}_1 \not\models C_1$, $\ell \notin C_1$. If $\bar{\ell}$ is also not in C_1 then $C_1 \in T(\ell)$ by definition. As this would contradict our assumption that $\mathcal{A} \models T(\ell)$, $\bar{\ell} \in C_1$. Similarly, $\ell \in C_2$. Thus we may resolve C_1 and C_2 on ℓ to get a clause D not containing ℓ and hence in $T(\ell)$. (As a resolvent of two clauses in T, D is certainly in $\mathcal{R}(T)$). Then, by our choice of $\mathcal{A}$, $\mathcal{A} \models D$. If $\mathcal{A}$ satisfies the resolvent D, however, it must satisfy one of the parents C_1 or C_2. Thus we have the desired contradiction. $\square$

Theorem 8.15 (Completeness of Resolution)**:** *If S is unsatisfiable then there is a resolution refutation of S.*

Proof: By the compactness theorem (Theorem 6.13), there is a finite subset S' of S which is unsatisfiable. As any refutation deduction from S' is one from S, we may assume that S is finite, i.e., it contains only finitely many clauses. If there are only finitely many clauses in S and each clause is finite, there are only finitely many literals, say $\ell_1, \ell_2, \ldots, \ell_n$ which are in any clause in S. For the rest of the proof we will consider only clauses and formulas based on these n literals.

We wish to consider the set of clauses $C \in \mathcal{R}(S)$ and prove that it contains $\square$. We proceed by eliminating each literal in turn by applying Lemma 8.14. We begin with $S_n = S(\ell_n) = \{C \in \mathcal{R}(S) \mid \ell_n, \bar{\ell}_n \notin C\}$. By definition, it is a collection of resolution consequences of S none of which contain ℓ_n or $\bar{\ell}_n$. By Lemma 8.14 it is unsatisfiable. Next we let $S_{n-1} = S_n(\ell_{n-1})$. It is an unsatisfiable collection of resolution consequences of S_n (and hence of S) none of which contain ℓ_{n-1}, $\bar{\ell}_{n-1}$, ℓ_n or $\bar{\ell}_n$. Continuing in this way we define $S_{n-2}, \ldots, S_0$. By repeated applications of the definitions and Lemma 8.14, we see that S_0 is an unsatisfiable set of resolution consequences of S containing no literals at all. As the only formulas with no literals are $\emptyset$ and $\{\square\}$ and $\emptyset$ is satisfiable, $\square \in S_0$. Thus $\square$ is a resolution consequence of S as required. $\square$

We now turn to a more abstract formulation of the notions and lemmas inherent in the proof of the completeness of resolution deduction. They will be needed to deal with the refinements of resolution in §9 and §10.

Definition 8.16: If S is a formula and ℓ a literal, we let

$$S^\ell = \{C - \{\bar{\ell}\} \mid C \in S \ \wedge \ \ell \notin C\}.$$

So S^ℓ consists of those clauses C of S containing neither ℓ nor $\bar{\ell}$, plus those clauses (not containing ℓ) such that $C \cup \{\bar{\ell}\} \in S$. Note that if the singleton clause $\{\bar{\ell}\}$ is in S then $\square$ is in S^ℓ.

Admittedly, this definition seems somewhat obscure at first reading. It is based on the idea that we can analyze (the satisfiability of) S by cases. S^ℓ corresponds to the result of the analysis under the assumption that ℓ is true. $S^{\bar{\ell}}$ gives the result when ℓ is assumed false. Consider, for example, the formula S under the assumption that ℓ is true. The first point here is that, if ℓ is true, then any clauses containing ℓ is satisfied since a clause is equivalent to the disjunction of its literals. As the formula S is equivalent to the conjunction of its clauses, any clause known to be true can be eliminated from S without changing its satisfiability. Thus, assuming ℓ to be true, we may omit any clause containing ℓ from S as far as satisfiability is concerned. This is precisely the point of the part of the definition of S^ℓ which restricts the clauses under consideration to those C such that $\ell \notin C$. The next point of the analysis is that, still assuming ℓ to be true, $\bar{\ell}$ can be omitted from any clause C containing it without changing the satisfiability of C. (Again C is equivalent to the disjunction of its members. If one of them is known to be false it cannot affect the satisfiability of the disjunction.) Of course, if the satisfiability of C is not affected, neither is that of the formula S containing it. This is then the point of that part of the definition of S^ℓ which says replace C by the smaller clause $C - \{\bar{\ell}\}$.

If ℓ is false, then $\bar{\ell}$ is true and the same analysis applies to $S^{\bar{\ell}}$. As one of ℓ and $\bar{\ell}$ must be true, we can argue (as we do in Lemma 8.17) that S is satisfiable if and only one of S^ℓ and $S^{\bar{\ell}}$ is satisfiable. Thus, we can reduce the satisfiability problem for S to two similar problems for formulas S^ℓ and $S^{\bar{\ell}}$ with one less propositional letter. We can then continue this procedure by considering each of the two new formulas S^ℓ and $S^{\bar{\ell}}$. In this way, we could produce a binary tree of formulas in which we would successively eliminate one literal at each level of the tree. Every path through this tree corresponds to an assignment. The branch through S^ℓ is the one that makes ℓ true. The one through $\bar{\ell}$ is the one that makes ℓ false. If every path through the tree ends with a formula containing the empty clause $\square$, we can conclude that the original formula S was unsatisfiable. On the other hand, if not all paths lead to $\square$, then, if we successively eliminate all the literals appearing in S, either there is an infinite path along which we have eliminated every literal or at least one path ends with the empty formula $\emptyset$. In either case S is satisfiable. Indeed, the appropriate path (infinite or leading to $\emptyset$) directly supplies an assignment satisfying S.

Seen in this way, the plan of the analysis is similar to that of tableau proofs beginning with $F\alpha$ for some proposition α. There too, we attempted to analyze all ways of making α false, i.e., of verifying $F\alpha$. If they all lead to contradictions ($\otimes$) we conclude that $F\alpha$ is unsatisfiable and α is valid. Here, if all paths lead to a formula containing the unsatisfiable clause $\Box$, we conclude that the formula S is unsatisfiable. On the other hand, if the tableau analysis was finished and produced a noncontradictory path, we could use that path (Lemma 5.4) to define a valuation satisfying α. In the analysis here, when we eliminate all the literals (corresponding to finishing the tableau) and are left with an infinite path or one ending with the empty formula $\emptyset$, this path itself directly supplies the assignment satisfying S.

We illustrate the construction of S^ℓ from S and the general form of this analysis by considering two examples.

Example 8.17: Let $S = \{\{p\}, \{\neg q\}, \{\neg p, \neg q\}\}$. The analysis in which we eliminate first p and then q can be represented by the following tree:

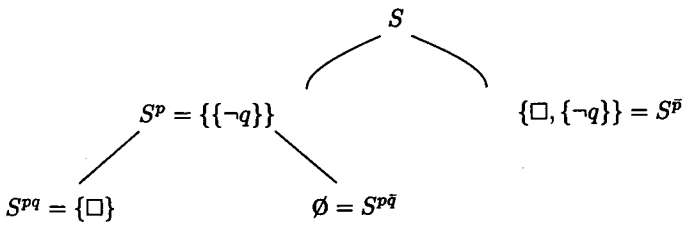


FIGURE 17

Assuming p is true, we eliminate the clause $\{p\}$ from S and the literal $\neg p$ from the clause $\{\neg p, \neg q\}$ to get S^p on the left side of the first level of the tree. Assuming that p is false, the right side ($S^{\bar{p}}$) reduces to $\{\Box, \{\neg q\}\}$ since S asserts that p is true by having $\{p\}$ as one of its clauses. At the next level, we consider q. On the left, when q is assumed true, we again get $\Box$ as S^p asserts that $\neg q$ is true. On the right, where we assume that q is false, we eliminate all clauses containing $\neg q$ to get the empty formula. Thus, we have a path ending in $\emptyset$. It supplies the assignment satisfying S: Make p true and q false.

Example 8.18: Consider the formula proven unsatisfiable in Example 8.7, $S = \{\{p, r\}, \{q, \neg r\}, \{\neg q\}, \{\neg p, t\}, \{\neg s\}, \{s, \neg t\}\}$. We begin the analysis by eliminating p. When we assume p to be true, we eliminate the clauses containing p (as they are true) and omit $\neg p$ from the others (since being false $\neg p$ cannot help to satisfy them) to get $S^p = \{\{q, \neg r\}, \{\neg q\}, \{t\}, \{\neg s\}, \{s, \neg t\}\}$. On the other hand, when we assume that p is false we eliminate clauses containing $\neg p$ and remove p from the others to get $S^{\bar{p}} = \{\{r\}, \{q, \neg r\}, \{\neg q\}, \{\neg s\}, \{s, \neg t\}\}$. Here is part of the full tree analysis:

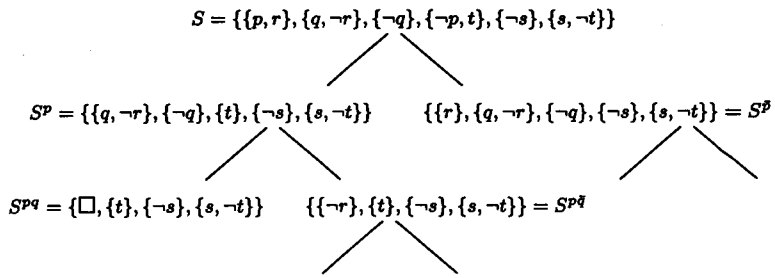


FIGURE 18

The path of the analysis through S^{pq} terminates at this point since it contains $\Box$ and so is unsatisfiable. The other paths displayed, however, continue. If continued, every path would eventually terminate with an unsatisfiable formula containing $\Box$ as a clause. This is the analog of the proof that $S \vdash_{\mathcal{R}} \Box$. We leave the completion of this analysis as Exercise 4.

We now formulate and prove the results that say that the analysis discussed above correctly captures the notion of satisfiability.

Lemma 8.19: *S is satisfiable if and only if either S^ℓ or $S^{\bar{\ell}}$ is satisfiable.* (Warning: In the "if" direction the assignments are not necessarily the same.)

Proof: ($\Rightarrow$) Suppose that $\mathcal{A} \vDash S$. If $\mathcal{A}$ were a complete assignment we could conclude that it must make one of $\ell, \bar{\ell}$ true, say ℓ. We could then show that $\mathcal{A} \vDash S^\ell$. If we do not wish to make this assumption on $\mathcal{A}$, we instead start with the fact that, by definition, one of ℓ or $\bar{\ell}$ does not belong to $\mathcal{A}$. For the sake of definiteness assume that $\bar{\ell} \notin \mathcal{A}$. We now also claim that $\mathcal{A} \vDash S^\ell$. We must show that $\mathcal{A}$ satisfies every clause in S^ℓ. Consider any $C \in S^\ell$. By the definition of S^ℓ, either $C \cup \{\bar{\ell}\} \in S$ or $C \in S$ (depending on whether or not $\bar{\ell}$ is in the clause of S which "puts" C into S^ℓ). Thus, by hypothesis, $\mathcal{A} \vDash C$ or $\mathcal{A} \vDash C \cup \{\bar{\ell}\}$. As an assignment satisfies a clause only if it contains one of its literals, there is a literal k such that either $k \in C \cap \mathcal{A}$ or $k \in (C \cup \{\bar{\ell}\}) \cap \mathcal{A}$. As $\bar{\ell} \notin \mathcal{A}$ by our assumption, in either case we must have $k \in C \cap \mathcal{A}$, i.e., $\mathcal{A} \vDash C$ as required. The case that $\ell \notin \mathcal{A}$ is handled similarly.

($\Leftarrow$) Suppose for definiteness that $\mathcal{A} \vDash S^\ell$. Now neither ℓ nor $\bar{\ell}$ appear in any clause of S^ℓ and so we may adjust $\mathcal{A}$ on ℓ as we choose without disturbing the satisfiability of S^ℓ. More precisely, if we let $\mathcal{A}' = (\mathcal{A} - \{\bar{\ell}\}) \cup \{\ell\}$ then $\mathcal{A}' \vDash S^\ell$ as well. We claim that $\mathcal{A}' \vDash S$. Consider any $C \in S$. If $\ell \in C$ then $\mathcal{A}' \vDash C$ as $\ell \in \mathcal{A}'$. If $\ell \notin C$ then $C - \{\bar{\ell}\} \in S^\ell$ by definition of S^ℓ. As $\mathcal{A} \vDash S^\ell$ there is some literal $k \in (C - \{\bar{\ell}\}) \cap \mathcal{A}$. Now $\mathcal{A}$ and $\mathcal{A}'$ differ at most at ℓ and $\bar{\ell}$. As $k \neq \ell$ or $\bar{\ell}$, we see that $k \in \mathcal{A}' \cap C$ as required. $\Box$

Corollary 8.20: *S is unsatisfiable iff both S^ℓ and $S^{\bar{\ell}}$ are.* □

This corollary, together with the unsatisfiability of □, actually characterizes the property of unsatisfiability.

Theorem 8.21: *If* UNSAT $= \{S \mid S$ *is an unsatisfiable formula}* *then* UNSAT *is the collection $\mathcal{U}$ of formulas defined inductively by the following clauses:*

(i) $\square \in S \Rightarrow S \in \mathcal{U}$

and

(ii) $S^\ell \in \mathcal{U} \wedge S^{\bar{\ell}} \in \mathcal{U} \Rightarrow S \in \mathcal{U}$.

Proof: As □ is unsatisfiable UNSAT satisfies (1). By Corollary 8.20 it also satisfies (ii). Thus $\mathcal{U} \subseteq$ UNSAT. We must show that UNSAT $\subseteq \mathcal{U}$. We prove the contrapositive by showing that if $S \notin \mathcal{U}$ then S is satisfiable. Let $\{p_i\}$ list the propositional letters such that p_i or $\bar{p}_i$ occurs in a clause of S. Define by induction the sequence $\{\ell_i\}$ such that $\ell_i = p_i$ or $\bar{p}_i$ and $S^{\ell_1,\ldots,\ell_i} \notin \mathcal{U}$. (Property (ii) guarantees that we can always find such an ℓ_i.) Now let $\mathcal{A} = \{\ell_i \mid i \in \mathbb{N}\}$. We claim that $\mathcal{A}$ satisfies S. Suppose $C \in S$. We must show that $C \cap \mathcal{A} \neq \emptyset$. As C is finite, there is an n such that for all propositional letters p_i occurring in C, $i < n$. If $C \cap \mathcal{A} = \emptyset$ then $\forall i < n$ $(\ell_i \notin C)$ and so a clause corresponding to C is passed on to each $S^{\ell_1,\ldots,\ell_i}$ for $i < n$. At each such transfer, say to $S^{\ell_1,\ldots,\ell_i}$, we remove $\bar{\ell}_i$ from the clause. As all literals in C are among the $\bar{\ell}_i$, the clause deriving from C becomes □ in $S^{\ell_1,\ldots,\ell_n}$. By our choice of the ℓ_i, $S^{\ell_1,\ldots,\ell_n} \notin \mathcal{U}$. On the other hand, any S containing □ is in $\mathcal{U}$ by clause (i) and we have our desired contradiction. □

This result is the analog of Lemma 5.4. The choice of the sequence ℓ_i corresponds to the definition of the assignment in Lemma 5.4 from the signed propositional letters appearing on the noncontradictory path on the finished tableau. As there, we are building an assignment that satisfies every entry on the path being constructed. Since we eventually reach the unsatisfiable clause □ in this construction, we have the desired contradiction. As for tableau proofs, this characterization of unsatisfiability is really the heart of the completeness proof of the resolution method.

Theorem 8.22 (Completeness of the resolution method)**:** *If S is unsatisfiable then there is a resolution refutation of S (equivalently, $\square \in \mathcal{R}(S)$).*

Proof: We proceed by induction according to the characterization of UNSAT provided by Theorem 8.21. Of course, if $\square \in S$, then $\square \in \mathcal{R}(S)$. For the inductive step, suppose that, for some ℓ and S, $\square \in \mathcal{R}(S^\ell)$ and $\square \in \mathcal{R}(S^{\bar{\ell}})$. We must show that $\square \in \mathcal{R}(S)$. By assumption, we have tree proofs T_0 and T_1 of □ from S^ℓ and $S^{\bar{\ell}}$. Consider T_0. If every leaf in T_0 is labeled with a clause in S, then T_0 is already a proof of □ from S. If not, we define a tree T_0' by changing every label C on T_0 which is above a leaf labeled with a clause not in S to $C \cup \{\bar{\ell}\}$. We claim that T_0' is a tree proof of $\{\bar{\ell}\}$ from S. Clearly, by the definition of S^ℓ, every leaf of T_0' is in S. We must now check that every nonleaf node of T_0' is labeled with a resolvent C' of its immediate successors C_0' and C_1'. Suppose they correspond to clauses C, C_0 and C_1 respectively on T_0. As T_0 is a resolution tree proof, C is a resolvent of C_0 and C_1. Note first that no resolution in T_0 is on ℓ or $\bar{\ell}$ as neither appear in any label on T_0 (by the definition of S^ℓ). Next, consider the possible forms of clauses C_0', C_1' and C' on T_0'. If, for example, both C_0 and C_1 (and hence certainly C) are above leaves labeled with clauses not in S, then $C' = C \cup \{\bar{\ell}\}$ is the resolvent of $C_0' = C_0 \cup \{\bar{\ell}\}$ and $C_1' = C_1 \cup \{\bar{\ell}\}$, as is required for T_0' to be a resolution tree proof. The other cases to consider either keep all three clauses the same in T_0' as they were in T_0 or change C and precisely one of C_0 and C_1 by adding on $\{\bar{\ell}\}$. In all these cases C' is still clearly the resolvent of C_0' and C_1' and we again verify that T_0' is a tree proof. Similarly, if we replace every label C on a node of T_1 above a leaf labeled with a clause not in S by $C \cup \{\ell\}$ we get T_1', a tree proof of $\{\ell\}$ from S (or in the case that all leaves were in S, one of □). We can now define a tree proof T of □ from S by simply attaching T_0' and T_1' to the immediate successors of the root node of T which we label with □. As □ is a resolvent of $\{\ell\}$ and $\{\bar{\ell}\}$ the resulting tree T is a proof of □ from S. □

*Compactness revisited

Of course there is no need to reprove the compactness theorem as it can be phrased solely in semantic terms. Nonetheless we offer another proof based on the characterization of UNSAT given by Theorem 8.21. It is the construction of the infinite sequence ℓ_i of literals in the proof of this theorem which corresponds to the path through the tree of assignments (given by König's lemma) in our original proof of compactness in Theorem 6.13.

Theorem 8.23 (Compactness)**:** *If S is unsatisfiable, so is some finite subset of S.*

Proof: Let $\mathcal{T} = \{S \mid \exists S_1 \subseteq S[S_1$ is finite $\wedge$ S_1 is unsatisfiable$]\}$. If we can show that $\mathcal{T}$ satisfies (i) and (ii) of Theorem 8.21 then we are done for it will then contain all unsatisfiable formulas.

(i) If $\square \in S$ then $S_1 = \{\square\} \subseteq S$ shows that $S \in \mathcal{T}$ as required.

(ii) Suppose S^ℓ, $S^{\bar{\ell}} \in \mathcal{T}$. We must show that $S \in \mathcal{T}$. By definition of $\mathcal{T}$, S^ℓ and $S^{\bar{\ell}}$, there are finite unsatisfiable formulas S_1, $S_2 \subseteq S$ such that $S_1^\ell \subseteq S^\ell$ and $S_2^{\bar{\ell}} \subseteq S^{\bar{\ell}}$. Let $S_3 = S_1 \cup S_2$. S_3 is a finite subset of S. It suffices to show that it is unsatisfiable. If not, then there would be an assignment $\mathcal{A}$ satisfying S_3. Now $\mathcal{A}$ must omit either ℓ or $\bar{\ell}$. Thus $\mathcal{A}$ would satisfy either $S_3^{\bar{\ell}}$ or S_3^ℓ respectively. As it would then satisfy $S_2^{\bar{\ell}}$ or S_1^ℓ (as $S_3 \supset S_2$, S_1), we have the desired contradiction. □

Exercises

1. Prove Proposition 8.10 by induction. (Hint: For one direction proceed by induction on the number of lines in the proof. For the other direction proceed by induction on the definition of $\mathcal{R}(S)$.)
2. Rephrase the proof of Theorem 8.11 (soundness) in terms of resolution tree proofs and an induction on their definition.
3. Do the same for the version of resolution deductions defined in terms of $\square \in \mathcal{R}(S)$.
4. Continue the analysis in Example 8.18 until every path terminates with a formula equivalent to the unsatisfiable clause $\square$.
5. Rewrite the following in both conjunctive normal and clausal form.
   a) $((A \vee B) \to (C \vee D))$
   b) $\neg(A \wedge B \wedge \neg C)$
   c) $\neg((A \wedge B) \vee (B \vee C) \vee (A \wedge C))$.
6. Which of the following clause sets are satisfiable? Give assignments satisfying them if they are. If they are not, explain why not.
   a) $\{\{A, B\}, \{\neg A, \neg B\}, \{\neg A, B\}\}$
   b) $\{\{\neg A\}, \{A, \neg B\}, \{B\}\}$
   c) $\{\{A\}, \square\}$
   d) $\{\square\}$
7. Find all resolvents for the following pairs:
   a) $\{A, B\}, \{\neg A, \neg B\}$
   b) $\{A, \neg B\}, \{B, C, D\}$
8. Find $\mathcal{R}(S)$ for the following sets S:
   a) $\{\{A, \neg B\}, \{A, B\}, \{\neg A\}\}$
   b) $\{\{A\}, \{B\}, \{A, B\}\}$
9. Find a deduction of the empty clause from
   $$\{\{A, \neg B, C\}, \{B, C\}, \{\neg A, C\}, \{B, \neg C\}, \{\neg B\}\}$$
10. Use resolution to show that each of the following is not satisfiable by any assignment.
    a) $(A \leftrightarrow (B \to C)) \wedge ((A \leftrightarrow B) \wedge (A \leftrightarrow \neg C))$
    b) $\neg(((A \to B) \to \neg B) \to \neg B)$
11. Let α be the proposition $\neg(p \vee q) \to (\neg p \wedge \neg q)$.
    a) Give a tableau proof of α.
    b) Convert $\neg\alpha$ into CNF and clausal form. (Show the steps of the conversion.)
    c) Give a resolution proof of α.
12. Do the same for the proposition $\beta = (\neg r \vee (p \wedge q)) \to ((r \to p) \wedge (r \to q))$.
13. Prove that if $S \vdash_{\mathcal{R}} C$, then $S \models C$.

14. Prove that if $S \cup \{\neg C\} \in$ UNSAT, then $S \vdash_{\mathcal{R}} C$.
15. Let $\mathcal{T}$ be defined inductively by the following clauses:
    (i) $\{\square\} \in \mathcal{T}$
    (ii) $S^{\ell}, \ S^{\bar{\ell}} \in \mathcal{T} \Rightarrow S \in \mathcal{T}$

    Prove that for every finite $S \in$ UNSAT, $S \in \mathcal{T}$ but that not every $S \in$ UNSAT is in $\mathcal{T}$. (Thus the characterization of UNSAT in Theorem 8.21 cannot be changed by replacing the base step assumption that all formulas containing $\square$ are included by the plausible alternative that just the formula consisting of $\square$ alone be included.)

9. Refining Resolution

Resolution is already a considerable improvement, for example, on the classical system of rules and axioms in §7. Resolution is intuitively more efficient because one is never tempted to ask which (of the infinitely many) axioms (of §7) should we put down next in our proof. There is only one rule for resolution. Thus, when we try to search systematically for a resolution refutation of a given (say finite) S, we need only arrange to check the application of this one rule to elements of S and previously deduced clauses. Even so, the search space can quickly become quite large. In fact, it is known that, for a certain class of theorems, the standard resolution method takes exponential time. A major concern is then developing ways to limit the search space (preferably without giving up soundness or completeness although in actual applications both are often sacrificed; more on this point later). In all honesty, we should point out that restricting the search space for proofs means that we will miss some proofs. Thus, although we search through a smaller space, the proofs we find may well be longer than those found by a wider search. Nonetheless, pruning the search tree does seem to be more efficient. (Of course we are using efficiency in a heuristic sense. SAT $= \{S \mid S$ is satisfiable$\}$ is NP complete and no system can avoid this theoretical limitation. Nonetheless, in practice smaller search spaces tend to correspond to faster run times.) We will consider just a few of the many possible strategies for directing the search for a resolution refutation.

We can consider directing the search from two viewpoints. The first is to terminate the search along paths that are unpromising. The second is to direct it by specifying the order in which we should try to go down alternative paths. Perhaps the most obvious branches to prune are those with tautologies on them: If C is a tautology then it can't be of any use in showing that S is unsatisfiable. As it is easy to check if a clause C is a tautology (just in case it contains both p and $\bar{p}$ for some propositional letter p) this is an inexpensive and useful pruning. (The cost of checking for tautologies has been absorbed by the requirement that we consider only clausal forms. Putting an arbitrary proposition into CNF can be expensive.)

Definition 9.1: *T–resolutions* are resolutions in which neither of the parent clauses is a tautology. $\mathcal{R}^T(S)$ is the closure of S under T–resolutions.

Lemma 9.2: *Any restriction of a sound method, i.e., one which allows fewer deductions than the sound method, is itself sound. In particular, as resolution is sound, so is $\mathcal{R}^T$, i.e., if $\Box \in \mathcal{R}^T(S)$, S is unsatisfiable.*

Proof: As any deduction in the restricted system is one in the original system and by soundness there is no deduction of $\Box$ in the original one, there is none in the restricted system. $\Box$

It is also not hard to see that $\mathcal{R}^T$ is complete.

Theorem 9.3 (Completeness of T–resolution)**:** *If S is unsatisfiable then $\Box \in \mathcal{R}^T(S)$.*

Proof: The proof of the completeness of resolution given in Theorem 8.22 remains correct for $\mathcal{R}^T$. The only remark needed is that if T_0 and T_1 have no tautologies on them then neither do the trees T_0' and T_1' gotten by adding $\bar{\ell}$ and ℓ respectively to the appropriate clauses. The point here is that no clause on T_0 (T_1) contains ℓ ($\bar{\ell}$) by assumption as T_0 (T_1) is a proof from S^ℓ ($S^{\bar{\ell}}$). $\Box$

Tautologies are true in every assignment and so can surely be ignored. We can considerably strengthen this semantic approach to refining resolution by fixing one assignment $\mathcal{A}$ and requiring that in every resolution one of the clauses be false in $\mathcal{A}$. (Again, if both are true in $\mathcal{A}$, so is the resolvent and we cannot hope to get unsolvability without resorting to clauses which fail in $\mathcal{A}$. Of course, this is far from a proof that we can simply ignore all such resolutions.)

Definition 9.4: Let $\mathcal{A}$ be an assignment. An *$\mathcal{A}$–resolution* is a resolution in which at least one of the parents is false in $\mathcal{A}$. $\mathcal{R}^{\mathcal{A}}$ is the closure of S under $\mathcal{A}$–resolutions. This procedure is often called *semantic resolution.*

Theorem 9.5 (Completeness of $\mathcal{A}$–resolution)**:** *For any $\mathcal{A}$ and S, if $S \in$* UNSAT *then $\Box \in \mathcal{R}^{\mathcal{A}}(S)$.*

Proof: Fix an assignment $\mathcal{A}$ and let $\mathcal{T}^{\mathcal{A}} = \{S \mid \Box \in \mathcal{R}^{\mathcal{A}}(S)\}$. We must show that UNSAT $\subseteq \mathcal{T}^{\mathcal{A}}$. By the characterization of UNSAT of Theorem 8.21 it suffices to prove that

(i) $\Box \in S \Rightarrow S \in \mathcal{T}^{\mathcal{A}}$ and

(ii) For any S and ℓ, if $S^\ell \in \mathcal{T}^{\mathcal{A}}$ and $S^{\bar{\ell}} \in \mathcal{T}^{\mathcal{A}}$ then $S \in \mathcal{T}^{\mathcal{A}}$.

(i) is immediate. For (ii) consider the $\mathcal{A}$–resolution proofs T_0 and T_1 of $\Box$ from S^ℓ and $S^{\bar{\ell}}$ respectively. We can form T_0' (T_1') as before by adding $\bar{\ell}$ (ℓ) to the appropriate clauses of T_0 (T_1). The resulting trees are of course resolution proofs of $\{\bar{\ell}\}$ and $\{\ell\}$ respectively (or perhaps of $\Box$). They

may not, however, be $\mathcal{A}$–resolutions since one of $\bar{\ell}$, ℓ may be true in $\mathcal{A}$. On the other hand, as at most one of $\ell, \bar{\ell}$ is true in $\mathcal{A}$, at least one of T_0' and T_1' is an $\mathcal{A}$–resolution proof. For definiteness say that $\ell \notin \mathcal{A}$ and so T_1' is an $\mathcal{A}$–resolution proof of $\{\ell\}$ or $\Box$ from S. In the latter case we are done. In the former, we can combine this proof of $\{\ell\}$ with T_0 to get the desired $\mathcal{A}$–resolution proof of $\Box$ as follows: To each leaf C of T_0 which is not in S attach as children $C \cup \{\bar{\ell}\}$ and $\{\ell\}$. As $\ell \notin \mathcal{A}$, this is an $\mathcal{A}$–resolution. Since $C \notin S$, $C \cup \{\bar{\ell}\}$ is in S. Thus, except for the fact that $\{\ell\}$ may not be in S, we have the desired $\mathcal{A}$–resolution proof of $\Box$ from S. We finish the construction of the required proof by attaching a copy of the tree T_1' below each leaf labeled with $\{\ell\}$. The resulting tree is now easily seen to represent an $\mathcal{A}$–resolution deduction of $\Box$ from S. Other than the resolutions of $\{\ell\}$ and nodes of the form $C \cup \{\bar{\ell}\}$ that we have just considered, all the resolutions appearing in this new proof appear in one of the $\mathcal{A}$–resolution deduction trees T_0 or T_1'. Thus every resolution appearing on the tree is an $\mathcal{A}$–resolution. $\Box$

As an example of a syntactic procedure which, to some extent at least, determines which resolutions we should try first we consider *ordered resolution.*

Definition 9.6: Assume that we have indexed all the propositional letters. We define $\mathcal{R}^<(S)$, for ordered resolution, as usual except that we only allow resolutions of $C_1 \sqcup \{p\}$ and $C_2 \sqcup \{\bar{p}\}$ when p has higher index than any propositional letter in C_1 or C_2.

Again if we try to mimic the proof of completeness given in Theorem 8.22 by simply restoring p and $\bar{p}$ to the ordered proofs T_0, T_1 of $\Box$ from S^p and $S^{\bar{p}}$, we may no longer have ordered resolutions. All we need to do, however, is reexamine our characterization of UNSAT to see that ordering can be imposed.

Theorem 9.7: UNSAT *is equal to the class of formulas $\mathcal{U}^<$ defined inductively by the following clauses:*

(i) $\Box \in S \Rightarrow S \in \mathcal{U}^<$ *and*

(ii$^<$) *If no propositional letter with index strictly smaller than that of p occurs in S, $S^p \in \mathcal{U}^<$ and $S^{\bar{p}} \in \mathcal{U}^<$ then $S \in \mathcal{U}^<$.*

Proof: As the inductive clause (ii$^<$) is weaker than (ii) of 8.21, $\mathcal{U}^<$ is surely contained in $\mathcal{U} =$ UNSAT. On the other hand, the original proof of the characterization of UNSAT (Theorem 8.21) shows that any $S \notin \mathcal{U}^<$ is satisfiable and so UNSAT is also contained in $\mathcal{U}^<$. The only point is to list the $\{p_i\}$ occurring in S in ascending order of their indices. $\Box$

The proof of completeness of resolution in Theorem 8.22 with $\mathcal{R}$ replaced by $\mathcal{R}^<$ and (ii) by (ii$^<$) now proves the completeness of ordered resolution.

Theorem 9.8 (Completeness of ordered resolution): *If S is unsatisfiable, then there is an ordered resolution refutation of S, i.e., $\square \in \mathcal{R}^<(S)$.* $\square$

Ordered resolution eliminates some of the duplications resulting from different permutations of the literals on which we resolve producing the same resolvent. It therefore reduces the number of times we derive any particular clause. There are many other versions of refutation each of which eliminates some aspect of the search space. A couple of them will be discussed in the exercises while the most powerful — linear resolution — will be considered in the next section in the setting of propositional logic and in the next chapter in the setting of full predicate logic.

Exercises

1. Let S be a finite set of clauses. Arbitrarily give each **occurrence** of a literal in the clauses of S a distinct index. A *lock resolution* is a resolution in which the literal resolved on has in each parent the lowest index of any literal in that parent. The literals in the child inherit their indices from its parents with the proviso that, if a literal appears in both parents, then in the child it has the smaller of the two possible indices. (We use superscripts to indicate the indexing.)

   Example: $C_1 = \{p^1, q^2, r^3\}$, $C_2 = \{\neg p^4, q^5\}$,
   $C_3 = \{\neg q^6\}$; $S = \{C_1, C_2, C_3\}$.

   Here we can lock resolve C_1 and C_2 to get $\{q^2, r^3\} = C_4$. C_4 can then be lock resolved against C_3 to get $\{r^3\}$. We cannot, however, lock resolve C_2 and C_3 as we would have to resolve on q and the occurrence of q in C_2 does not have the lowest index of any literal in C_2. (It has index 5 while $\neg p$ has index 4.)

   Prove that lock resolution is complete, i.e., if S is unsatisfiable then there is a lock resolution deduction of $\square$ from S. (Hint: Proceed by induction on the *excess literal parameter* = the total number of occurrences of literals in S minus the number of clauses in S.)

2. Show that lock resolution cannot be combined with the omission of tautologies to get a complete resolution system.

3. Suppose S is a set of clauses, $U \subseteq S$ and $S - U$ is satisfiable. A resolution has *support* U if not both parents are in $S - U$. Give a complete definition of a resolution of clauses with support U and the associated set $\mathcal{R}^U(S)$. Prove that $S \in$ UNSAT $\Leftrightarrow \square \in \mathcal{R}^U(S)$.

4. We say informally that an *F–resolution* is one in which one of the clauses is a goal clause (i.e., it contains only negative literals). Give a complete formal definition of F–resolution (that is without referring to the basic definition of resolution) and of $S \vdash_F \square$ (there is an F–resolution proof of $\square$ from S). Prove that $S \in$ UNSAT iff $S \vdash_F \square$.

10. Linear Resolution, Horn Clauses and PROLOG

We wish to consider another refinement of resolution: linear resolution. We will defer the full analysis of this method to the chapter on predicate logic. Here we will simply describe it and analyze its specialization to Horn clauses. In this form it becomes the basic theorem prover underlying PROLOG. The plan here is to try to proceed via a linear sequence of resolutions rather than a branching tree of them. We carry out a sequence of resolutions each of which (after the first) must have as one of its parents the child of the one previously carried out.

Definition 10.1:

(i) A *linear (resolution) deduction or proof* of C from S is a sequence of pairs $\langle C_0, B_0 \rangle, \ldots, \langle C_n, B_n \rangle$ such that $C = C_{n+1}$ and

   (1) C_0 and each B_i are elements of S or some C_j with $j < i$,

   (2) each C_{i+1}, $i \leq n$, is a resolvent of C_i and B_i.

(ii) As usual we say that C is *linearly deducible* (or *provable*) *from* S, $S \vdash_{\mathcal{L}} C$, if there is a linear deduction of C from S. There is a *linear refutation* of S if $S \vdash_{\mathcal{L}} \square$. $\mathcal{L}(S)$ is the set of all clauses linearly deducible from S.

The usual convention is to write linear resolutions with the starting point at the top and the conclusion at the bottom (as opposed to the picture of tree resolutions which put the node, labeled by the conclusion, at the top). Thus we picture a linear resolution as follows:

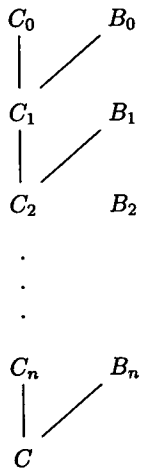


FIGURE 19

Example 10.2: Let $S = \{A_1, A_2, A_3, A_4\}$, $A_1 = \{p, q\}$, $A_2 = \{p, \neg q\}$, $A_3 = \{\neg p, q\}$, $A_4 = \{\neg p, \neg q\}$. Figure 20 gives a linear refutation of S:

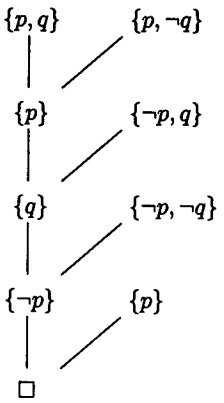


FIGURE 20

Definition 10.3: In the context of linear resolution, the elements of the set S from which we are making our deductions are frequently called *input clauses*. The C_i are called *center clauses* and the B_i *side clauses*. C_0 is called the *starting clause* of the deduction.

If we extend the parent–child terminology by defining the *ancestors* of a clause C in a resolution proof of C from S to be the clauses above it in the tree proof, we can rephrase the definition of linear deduction by saying that each C_i is resolved against an input clause or one of its own ancestors to produce C_{i+1}.

Linear resolution is clearly a refinement of resolution; that is, every linear resolution proof is an ordinary resolution proof. As resolution is sound (Theorem 8.11), so then is linear resolution. In Chapter II, Section 14 we will prove that linear resolution is complete. For now, we wish to consider only the case of Horn clauses and PROLOG programs.

Definition 10.4:

(i) A *Horn clause* is a clause which contains at most one positive literal.

(ii) A *program clause* is one which contains exactly one positive literal. (In PROLOG notation it looks like A :- $B_1, B_2, \ldots, B_n$.)

(iii) If a program clause contains some negative literals it is called a *rule* ($n > 0$ in the notation of (ii)).

(iv) A *fact* (*or unit clause*) is one which consists of exactly one positive literal (Notation: A. or A :- .).

(v) A *goal clause* is one which contains no positive literals. (Thus in PROLOG it is entered as a question with the symbol ?- .)

(vi) A PROLOG *program* is a set of clauses containing only program clauses (rules or facts).

Notice that Horn clauses are either program or goal clauses while program clauses are either rules or facts. An important point is that an inconsistency can arise only from the combination of a goal clause and a fact. The contradiction may be mediated by rules but rules (and facts) alone cannot produce a contradiction.

Lemma 10.5: *If a set of Horn clauses S is unsatisfiable, then S must contain at least one fact and one goal clause.*

Proof: The assignment which makes every propositional letter true satisfies every program clause. The assignment which makes every propositional letter false satisfies every goal clause and every rule. Thus, any unsatisfiable set of Horn clauses must contain both a fact and a goal clause. □

The general view of a PROLOG program is that we are given a collection of facts and rules and wish to deduce consequences from them. Typically, we may want to know if the conjunction of some facts $q_1, q_2, \ldots, q_n$ follows from our program P. We enter this as a question ?- $q_1, q_2, \ldots, q_n$ at the PROLOG prompt and receive an answer telling us if the q_i are consequences of the program. The general idea implemented by PROLOG is to add on a goal clause $G = \{\neg q_1, \neg q_2, \ldots, \neg q_n\}$ to the given program and ask if the resulting set $P \cup \{G\}$ of Horn clauses is unsatisfiable. The simple but crucial point here is that the conjunction of facts $q_1, q_2, \ldots, q_n$ is a consequence of our assumptions P just in case $P \cup \{G\}$ is unsatisfiable. We isolate this basic semantic transformation as a lemma. It is implicitly employed every time we ask a question in PROLOG.

Lemma 10.6: *If P is a* PROLOG *program and $G = \{\neg q_1, \neg q_2, \ldots, \neg q_n\}$ a goal clause, then all of the q_i are consequences of P if and only if $P \cup \{G\}$ is unsatisfiable.*

Proof: The proof simply consists of tracing through the definitions. First note that $P \cup \{G\}$ is unsatisfiable if and only if any assignment satisfying P makes G false. Next note that the goal clause G is false iff none of the $\neg q_i$ are true, i.e., G is false iff all the q_i are true. Thus, our desired conjunction of facts is a consequence of our assumptions P just in case $P \cup \{G\}$ is unsatisfiable. □

Our goal now is to translate this semantic condition into a proof theoretic one that we can verify by resolution methods. In fact, we show that linear resolution suffices to decide unsatisfiability for sets of Horn clauses.

Theorem 10.7 (Completeness of linear resolution for Horn clauses)**:** *If S is an unsatisfiable set of Horn clauses, then there is a linear resolution deduction of* □ *from S, i.e.,* $\square \in \mathcal{L}(S)$.

Proof: By the compactness theorem (Theorem 6.13 or 8.23) we may assume that S is finite. We proceed by induction on the number of literals

in S. By Lemma 10.5 we know that there is at least one positive literal p occurring as a fact $\{p\}$ in S. Consider the formula S^p as described in Definition 8.16. Each clause in S^p is a subset of one in S and so is Horn by definition. We claim that S^p is unsatisfiable. The point here is that, if $\mathcal{A} \models S^p$, then $\mathcal{A} \cup \{p\} \models S$ contradicting the unsatisfiability of S. As S^p contains fewer literals than S (we omit any clause containing p and remove $\bar{p}$ from every other clause), we may apply the induction hypothesis to S^p to get a linear resolution deduction of $\square$ from S^p. As in the inductive step of the proof of the completeness theorem for the general resolution method given for Theorem 8.22, either this is already a linear proof of $\square$ from S or we can convert it into one of $\{\bar{p}\}$ from S by adding $\bar{p}$ to every clause below one not in S. We can now extend this proof one step by adding on $\{p\} \in S$ as a new side clause and resolving against the last center clause $\{\bar{p}\}$ to get $\square$ as required. $\square$

The advantage of linear resolution is obvious. We are now looking for a linear sequence to demonstrate unsatisfiability rather than a whole tree. The tree structure of the searching in PROLOG is generated by the different possibilities for side clauses. Each path in the tree of possible deductions by PROLOG represents a linear resolution. In the actual setting of a PROLOG program and a given goal clause (question to the interpreter) we can be more precise in specifying the order of clauses in the linear resolutions for which we are searching. By Lemma 10.4, we know that the goal clause must be used in the deduction. In fact, we can require our deduction of $\square$ to start with the goal clause and thereafter to use only clauses from the PROLOG program as side clauses. As these clauses are called input clauses, this restriction of resolution is called *linear input resolution.*

Definition 10.8: Let P be a set of program clauses and G a goal clause. A *linear input* (LI) *resolution refutation of* $S = P \cup \{G\}$ is a linear resolution refutation of S which starts with G and in which all the side clauses are from P (input clauses).

The method of LI–resolution is not complete in general as may be seen from the following example.

Example 10.9: Recall the clauses of Example 10.2: $S = \{A_1, A_2, A_3, A_4\}$, $A_1 = \{p, q\}$, $A_2 = \{p, \neg q\}$, $A_3 = \{\neg p, q\}$, $A_4 = \{\neg p, \neg q\}$. The only goal clause here is A_4 which we set equal to G. The remaining clauses are, however, not all program clauses. If we set $P = \{A_1, A_2, A_3\}$ and try to produce a linear input resolution refutation of $S = P \cup \{G\}$ beginning with G we are always thwarted. Figure 21 gives one attempt.

The problem here is that, no matter how we start the resolution, when we get to a center clause which contains exactly one literal, any resolution with a clause from P produces another such clause as resolvent. Thus we can never deduce $\square$.

Linear input resolution does, however, suffice for the cases of interest in PROLOG programming.

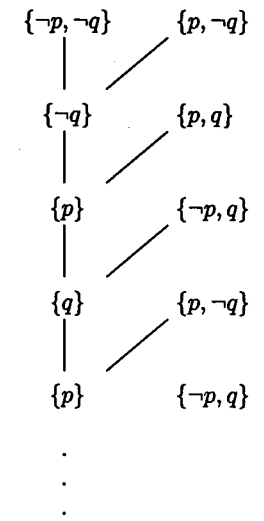


FIGURE 21

Theorem 10.10: *Let P be a set of program clauses and G be a goal clause. If $S = P \cup \{G\} \in$ UNSAT, there is a linear input resolution refutation of S.*

Proof: Note first that we can resolve a goal clause only against a program clause (as opposed to another goal clause) as we must have some literal p in one of the clauses being resolved and $\bar{p}$ in the other while goal clauses contain only negative literals. Moreover, the child of any such resolution must again be a goal clause as the single positive literal in the program clause must be the one resolved on and it is removed from the child leaving only the negative literals of the program clause and the remaining ones of the goal clause. Thus, if we have any linear proof of $\square$ from S starting with G then all the children of the resolutions in the proof must be goal clauses and all the side clauses must be program clauses as desired. It therefore suffices to prove that there is a linear proof of $\square$ from S starting with G. We again proceed by induction on the number of literals in our unsatisfiable set of clauses, but we prove a stronger assertion than that of Theorem 10.7.

Lemma 10.11: *If T is a set of Horn clauses, G a goal clause such that $T \cup \{G\} \in$ UNSAT but $T \in$ SAT, then there is a linear resolution deduction of $\square$ from $T \cup \{G\}$ starting with G.*

Proof: As before, we may assume that T is finite by the compactness theorem. We proceed by induction on the number of literals in T. As in the proof of Theorem 10.7, we know that T contains a fact $\{p\}$ for some positive literal p and that $T' = (T \cup \{G\})^p = T^p \cup \{G\}^p$ is an unsatisfiable set of Horn clauses. (As G is a goal clause, it contains no positive literals

and so $\{G\}^p$ is just $\{G - \{\bar{p}\}\}$.) As T was satisfiable and contained $\{p\}$, T^p is satisfiable by the same assignment that satisfied T (by the proof of the "only if" direction of Lemma 8.19). Thus we may apply the induction hypothesis to T' to get a linear proof of $\Box$ from T' starting with $G - \{\bar{p}\}$. If this proof is not already the desired one of $\Box$ from T starting with G, we may, as in the proofs of Theorem 8.22 or 10.7, convert it into a proof of $\{\bar{p}\}$ from T starting with G. We can again extend this proof one step by adding on $\{p\} \in T$ as a new side clause at the end to do one more resolution to get $\Box$ as desired. $\Box$

As any set of program clauses is satisfiable by Lemma 10.5, this lemma suffices to prove Theorem 10.10. $\Box$

We now know the general format of the resolution proofs for PROLOG: linear input resolution. There are two points left to consider before we have the precise mechanism used by the PROLOG implementation. The most important one is that PROLOG is not restricted to propositional logic; it uses predicates and variables as well. This is the topic of the next chapter. The other point is more technical; it concerns ordering considerations which come in two varieties. The first deals with the actual representation of clauses in the implementation of resolution and the choice of literal on which to resolve. The second deals with the ordering of the search for linear proofs: searching and backtracking.

We begin with the representation of clauses. Our abstract presentation of resolution deals with clauses viewed as sets of literals. As sets, the clauses are intrinsically unordered. A machine, however, typically stores clauses as sequences of literals. Moreover, it manipulates them as sequences and not as sets. Thus, set theoretic operations such as union must be replaced by some sort of merging procedure on sequences. In particular, when $G = \{\neg A_0, \neg A_1, \ldots, \neg A_n\}$ and $H = \{B, \neg B_0, \ldots, \neg B_m\}$ (viewed as ordered clauses) are resolved, say on $A_i = \neg B$, the interpreter simply replaces A_i by $\neg B_0, \ldots, \neg B_m$. The resolvent is then (as an ordered clause) $\{\neg A_0, \neg A_1, \ldots, \neg A_{i-1}, \neg B_0, \ldots, \neg B_m, \neg A_{i+1}, \ldots, \neg A_n\}$. In addition to the ordering itself, one should note that, as a result of this view of clauses, duplications may arise if, for example, one of the B_j is the same as some A_k $(k \neq i)$. The implementation of PROLOG does not check for such duplication, it merely carries along all copies of literals in the appropriate location in the ordered clause. (Ordered clauses are sometimes referred to as *definite clauses*, hence the notation in the next definition using LD for linear–definite.) This ordering of clauses does not cause any serious changes. We embody it in the following definition and lemma.

We continue to use T to denote a set of Horn clauses, P a set of program clauses and G a goal clause.

Definition 10.12: If $P \cup \{G\}$ is given as a set of ordered clauses, then an LD–*resolution refutation* of $P \cup \{G\}$ is a sequence $\langle G_0, C_0 \rangle, \ldots, \langle G_n, C_n \rangle$ of ordered clauses G_i, C_i such that $G_0 = G$, $G_{n+1} = \Box$, and

(i) Each G_i, $i \leq n$, is an ordered goal clause $\{\neg A_{i,0}, \ldots, \neg A_{i,n(i)}\}$ of length $n(i) + 1$.

(ii) Each $C_i = \{B_i, \neg B_{i,0}, \ldots, \neg B_{i,m(i)}\}$ is an ordered program clause of length $m(i) + 2$ from P. (We include the possibility that $C_i = \{B_i\}$, i.e., $m(i) = -1$.)

(iii) For each $i < n$, there is a resolution of G_i and C_i as ordered clauses with resolvent the ordered clause G_{i+1} (of length $n(i) + m(i) + 1$) given by $\{\neg A_{i,0}, \ldots, \neg A_{i,k-1}, \neg B_{i,0}, \ldots, \neg B_{i,m(i)}, \neg A_{i,k+1}, \ldots, \neg A_{i,n(i)}\}$. (In this resolution we resolve on $B_i = A_{i,k}$.)

Lemma 10.13: *If $P \cup \{G\} \in$ UNSAT, then there is an* LD*–resolution refutation of $P \cup \{G\}$ starting with G.*

Proof: This is left as Exercise 1. Proceed by induction on the length of the LI–resolution refutation of $P \cup \{G\}$. (Note that we can only resolve a program clause and a goal clause at each step of the resolution. Each center clause must be a goal clause and each side one a program clause.) $\Box$

Our next task is to describe how we choose the literal of G_i to be resolved on in an LD–resolution proof. The selection rule used in virtually all implementations of PROLOG is to always resolve on the first literal in the ordered goal clause G (in our notation this is just the leftmost literal in G_i). The literals in the resolvent of C_i and G_i are then ordered as indicated in Definition 10.12. We call such linear input resolutions with ordered clauses SLD*–resolutions*. (The S stands for selection.) More generally, we can consider any selection rule R, i.e., any function choosing a literal from each ordered goal clause.

Definition 10.14: An SLD*–resolution refutation* of $P \cup \{G\}$ via (the selection rule) R is an LD–resolution proof $\langle G_0, C_0 \rangle, \ldots, \langle G_n, C_n \rangle$ with $G_0 = G$ and $G_{n+1} = \Box$ in which $R(G_i)$ is the literal resolved on at the $(i+1)$ step of the proof. (If no R is mentioned we assume that the standard one of choosing the leftmost literal is intended.)

Theorem 10.15 (Completeness of SLD–refutation for PROLOG)**:** *If $P \cup \{G\} \in$ UNSAT and R is any selection rule, then there is an* SLD*–resolution refutation of $P \cup \{G\}$ via R.*

Proof: By Lemma 10.13, there is an LD–resolution refutation of $P \cup \{G\}$ starting with G. We prove by induction on the length n of such proofs (for any P and G) that there is an SLD one via R. For $n = 1$ there is nothing to prove as $G = G_0$ is a unit clause and so every R makes the same choice from G_0. Let $\langle G_0, C_0 \rangle, \ldots, \langle G_n, C_n \rangle$, with the notation for these clauses as in Definition 10.12, be an LD–resolution refutation of length n of $\{G_0\} \cup P$. Suppose that the selection rule R chooses the clause $\neg A_{0,k}$ from G_0. As $G_{n+1} = \Box$ there must be a $j < n$ at which we resolve on $\neg A_{0,k}$. If $j = 0$

we are done by induction. Suppose then that $j \geq 1$. Consider the result C of resolving G_0 and $C_j = \{B_j, \neg B_{j,0}, \ldots, \neg B_{j,m(j)}\}$ on $B_j = A_{0,k}$:

$$C = \{\neg A_{0,0}, \ldots, \neg A_{0,k-1}, \neg B_{j,0}, \ldots, \neg B_{j,m(j)}, \neg A_{0,k+1}, \ldots, \neg A_{0,n(0)}\}.$$

We claim that there is an LD–resolution refutation of length $n-1$ from $P \cup \{C\}$ which begins with C. One simply resolves in turn with $C_0, \ldots, C_{j-1}$ on the same literals as in the original proof that started with G. The only change is that we carry along the sequence of clauses $\neg B_{j,0}, \ldots, \neg B_{j,m(j)}$ in place of $\neg A_{0,k}$ in the center clauses of the resolution. After resolving with each side clause $C_0, \ldots, C_{j-1}$, we have precisely the same result G_{j+1} as we had in the original resolution after resolving with C_j. We can then continue the resolution deduction exactly as in the original resolution with $C_{j+1}, \ldots, C_n$. This procedure produces an LD–resolution refutation of length $n-1$ beginning with C. By induction, it can be replaced by an SLD–resolution refutation via R. Adding this SLD–resolution via R onto the single step resolution of G_0 with C_j described above produces the desired SLD–resolution refutation from $P \cup \{G\}$ via R starting with $G = G_0$. □

We now know what the PROLOG interpreter does when a question is entered as in "?- $A_1, \ldots, A_n$.". It searches for an SLD–resolution proof of □ from the current program P and the goal clause $G = \{\neg A_1, \ldots, \neg A_n\}$ starting with G. The remaining uncertainty in our description of its action is just how it organizes this search. At each step i of the SLD–resolution, the only choice to be made is which clause in P to use to resolve on the leftmost term in our current goal clause G_i. We can thus display the space of all possible SLD–derivations as a labeled tree T. The root of T is labeled G. If any node of T is labeled G', then its immediate successors are labeled with the results of resolving on the leftmost literal of G' with the various possible choices of clauses in P. We call such trees SLD–*trees* for P and G.

Example 10.16 (SLD–Trees): As a simple example, consider the program P_0:

```
p:- q,r.        (1)
p:- s           (2)
q.              (3)
q:- s.          (4)
r.              (5)
s:- t.          (6)
s.              (7)
```

PROGRAM P_0.

Suppose we have $G = \{\neg p\}$ as our goal clause. The corresponding SLD–tree is given below in Figure 22. Along each branching we indicate the clause of P_0 resolved against. The convention is that the successors are listed in a left to right order that agrees with the order in which the clauses used appear in P_0. *Success paths* are those ending in □. A path is a *failure path* if it ends with a clause G' such that there is no clause in P with which we can resolve on the leftmost term of G'. In this example there are five possible paths. Two end in failure and three end with success.

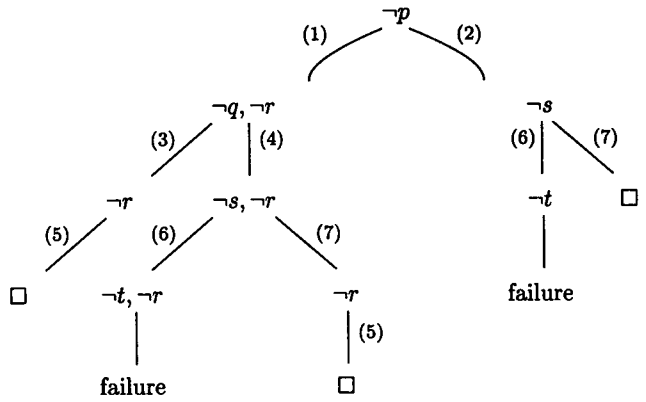


FIGURE 22

The PROLOG theorem prover searches the SLD–tree for a success path by always trying the leftmost path first. That is, it tries to resolve the current G with the first clause in P that is possible. In Figure 22 it would simply follow the path (1), (3) (5) to get the correct answer "yes". If the theorem prover hits a failure point (i.e., not □ and no resolution is possible) it backtracks, that is, it goes back up the tree until it returns to a node N which has a path leading out of it to the right of the one the theorem prover has followed upward. The prover then exits from N along the path immediately to the right of the one it just returned on (i.e., it tries the leftmost successor of N not yet attempted). This process is repeated until a success path is found.

Example 10.17 (Backtracking): If we omit clause (3) from the above program P_0 to produce P_1, we get a new SLD–tree as pictured in Figure 23.

In this case, the theorem prover first tries the path (1), (4), (6), failure. It then backtracks to $\neg s$, $\neg r$ and tries (7), (5), success, to give the answer yes.

Suppose the PROLOG interpreter has searched the tree until it has found an answer and we then enter a semicolon ";" at the prompt. The interpreter will resume backtracking to look for another resolution refutation in

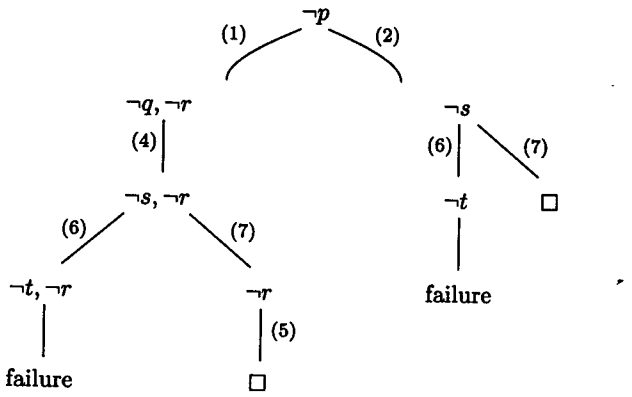


FIGURE 23

the part of the tree that it has not yet searched. A "no" answer now means that there are no more proofs. A "yes" answer indicates another one has been found. In this case, we may once more enter ";" to start the search for yet another proof. In the last example after finding the proof along the path (1), (4), (7), (5) the theorem prover answered "yes". If we asked for another proof by entering a semicolon, it would backtrack all the way up to the top node and try the path (2). It then would proceed down (6) to a failure, backtrack to $\neg s$ and follow (7) to another success and "yes" answer. One more request for a proof via entering a semicolon would finally produce a "no" answer.

If PROLOG searches through the entire SLD–tree without finding a path leading to $\square$, it gives the answer "no" the first time we ask our question. By our general completeness theorem, we then know that in this case $P \cup \{G\}$ is satisfiable and so (by Theorem 10.6) the question asked is not a logical consequence of P.

This type of search procedure is called a *depth–first* search procedure as it tries to go as deeply as possible in the tree by running down to the end of a path before searching sideways along any other branches. In contrast, one that searches the tree in figure (1) in the order $\neg p$; $\neg q$, $\neg r$; $\neg s$; $\neg r$; $\neg s$, $\neg r$; $\neg t$; $\square$; $\square$; $\neg t$; $\neg r$; $\neg r$; failure; failure; $\square$ is called a *breadth–first* search. Clearly many mixed strategies are also possible. In our case, the depth–first search was much faster than breadth–first (3 versus 6 steps). Indeed, this is a quite general phenomena. Depth–first is usually much faster than breadth–first. That, of course, is why the implementations use depth–first searches. The cost of this strategy can, however, be quite high. In a breadth–first search, it is clear that, if there is a path ending in $\square$, we must eventually find it. In contrast, the procedure of depth–first search is not complete: There may be a path leading to $\square$ but we may search the tree forever without finding it.

Example 10.18 (Failure of depth-first searching): Consider the following simple program:

```
q:- r.        (1)
r:- q.        (2)
q.            (3)
```

The usual search procedure applied to the starting clause $\neg q$ will loop back and forth between $\neg q$ and $\neg r$. It will never find the contradiction supplied by (3).

This example seems easy to fix and to depend purely on the order of the clauses in the program. Unfortunately, rearranging clauses will not always produce a terminating program even when a correct proof does exist. (See program P_5 of III.2 for an example.) The full impact of these problems cannot, however, be felt until we deal with full PROLOG rather than restricting our attention to the propositional case. Indeed, it is only with the introduction of predicates and variables that one sees the true power of PROLOG. We now turn to these matters, first in the general setting of full predicate logic and then in just PROLOG.

Exercises

1. Prove Lemma 10.13.
2. Consider the following sentence in "English": If the congress refuses to enact new laws, then the strike will not be over unless it lasts more than one year and the president of the firm resigns.
   a) Represent this sentence in three ways:
      (i) by a statement in the propositional calculus
      (ii) by one in conjunctive normal form
      (iii) as part of a PROLOG program.
   b) Suppose congress has refused to act and the strike has not yet lasted more than a month. Add these to your list (in (a), (b) and (c)) and use a tableau deduction to see if the strike is over yet. (You can try to do a deduction from the appropriate premises or an ordinary one from the appropriate conjunction but be careful how you formulate the problem.)
   c) Suppose instead that the congress refused to act, the strike is now over but the president of the firm did not resign. Use tableau deduction to see if the strike lasted more than a year.
   d) In (d) and (e) if you had a list of the relevant clauses in your PROLOG data base, what would you enter to get the answer.
3. One of the successful applications of expert systems has been analyzing the problem of which chemical syntheses are possible. We consider here an extremely simple example of such a problem.

We know we can perform the following chemical reactions:

(1) $MgO + H_2 \rightarrow Mg + H_2O$

(2) $C + O_2 \rightarrow CO_2$

(3) $CO_2 + H_2O \rightarrow H_2CO_3$.

a) Represent these rules and the assumptions that we have some MgO, H_2, O_2 and C by propositional logic formulas in which assertions say that we have a particular chemical and implications are understood to mean that, if we have the hypotheses, we can get the conclusion. (Thus (1) is $MgO \wedge H_2 \rightarrow Mg \wedge H_2O$.)

b) Describe the state of affairs in clausal form and as a PROLOG Program.

c) Give a resolution proof (in tree or linear form) that we can get some H_2CO_3.

4. Represent the following information as a PROLOG program so that, if you are told that Jones has fallen deathly ill and Smith has gone off to his bedside, you could determine if the directors will declare a dividend.

   If Jones is ill or Smith is away, then the directors will meet and declare a dividend if Robinson comes to his senses and takes matters into his own hands. If Patterson comes, he will force Robinson back to his senses but, of course, he will come only if Jones is ill. On the other hand, if Townsend, who is inseparable from Smith, stays away, Robinson will have to take matters into his own hands.

   Give a resolution proof from your program and the added hypotheses from above that shows that the directors will declare a dividend.

5. Represent the following information as a PROLOG program:

   If congress enacts a line item veto and the president acts responsibly, there will be a decrease in both the budget and trade deficits unless there is a major lobbying campaign by both the protectionists and the advocates of high interest rates. A strong public outcry will get the congress to enact the line item veto and force the president to act responsibly. The protectionists can be kept quiet if productivity increases and the dollar is further devalued.

   (It may help to start with a formulation in propositional logic and convert it appropriately.)

   How would you add on PROLOG clauses to reflect the fact that the public is vocally outraged about the deficits, the dollar is continuing to fall on world markets and productivity is on the increase.

   How do you now ask the PROLOG program if the trade deficit will decrease?

   Give the resolution refutation that shows that it will go down.

6. Draw the SLD–tree illustrating all possible attempts at SLD–refutations using the standard selection rule (always resolve on the leftmost literal)

for the question "?- p." and following program:

```
(1)   p:- s,t.
(2)   p:- q.
(3)   q.
(4)   q:- r.
(5)   r:- w.
(6)   r.
(7)   s.
(8)   t:- w.
```

In what order will the PROLOG interpreter search this tree and what output will it give if we enter a semicolon at each yes answer?

Suggestions for Further Reading

For orderings, partial orderings and trees, consult Birkhoff [1973, 3.8], or almost any of the logic and computer science texts listed in the bibliography [5.2].

For early propositional logic, read Boole [1952, 2.3] and Post [1921, 2.3].

For various alternate formalisms for logic, read the propositional part of the following:

Tableaux: Beth [1959, 3.2], Smullyan [1976, 3.2], and Fitting [1985, 4.2].

Axioms and Rules of Inference: Hilbert and Ackermann [1950, 3.2], Mendelson [1964, 3.2] and Enderton [1976, 3.2]. For an approach at a more advanced level, see Kleene [1971, 3.2], Monk [1976, 3.2] or Shoenfield [1967, 3.2].

Resolution: Chang and Lee [1973, 5.7], J. A. Robinson [1979, 5.2], Lewis and Papadimitriou [1981, 5.2], Maier and Warren [1988, 5.4].

Natural Deduction: Prawitz [1965, 3.5], or at a more advanced level, Girard [1987, 3.5] and Girard et al. [1989, 3.5].

Sequents: Gallier [1986, 5.2], Manaster [1975, 3.2] or Girard [1987, 3.5] and Girard et al. [1989, 3.5].

For a problem–oriented text based on resolution and Horn logic, see Kowalski [1979, 5.4].

For Boolean algebra and its relations to propositional logic, see Halmos [1974, 3.8], Sikorski [1969, 3.8] or Rasiowa and Sikorski [1963, 3.8].

II Predicate Logic

1. Predicates and Quantifiers

The logic of predicates encompasses much of the reasoning in the mathematical sciences. We are already familiar with the informal idea of a property holding of an object or a relation holding between objects. Any such property or relation is an example of a *predicate*. The difference between a property and a relation is just in the arity of the predicate. Unary predicates are simply properties of objects, binary ones are relations between pairs of objects and in general n–ary predicates express relations among n–tuples of objects. A reasonable answer to the question of what are 0–ary predicates might well be that they are propositions. The point here is that they are simply statements of facts independent of any variables. Thus, for example, if we are discussing the natural numbers we may let $\varphi(x, y)$ denote the binary relation (predicate) "x is less than y". In this case $\varphi(3, y)$ denotes the property (unary predicate) of y "3 is less than y" and $\varphi(3, 4)$ denotes the (true) proposition (0–ary predicate) that 3 is less than 4.

In this discussion x and y were used as *variables* while 3 and 4 were used as *constants*. Variables act as placeholders in much the same way as pronouns act as placeholders in ordinary language. Their role is best understood with the following convention: At the beginning of any particular discourse we specify a nonempty domain of objects. All variables that show up in the ensuing discussion are thought of as ranging over this domain. Constants play the role of names for objects (individuals) in this domain. In such a setting, n–ary predicates can be viewed as simply sets of n–tuples from the domain of discourse — the ones for which the predicate holds. Thus, for example, unary predicates can be seen as the subset of the domain consisting of the elements of which the property is true. 0–ary predicates assert facts about the domain of discourse and, in the context of predicate logic, are usually called *sentences* rather than propositions.

Another important class of objects in mathematical discourse is that of *functions*. For example $f(1, 2)$ might stand for the sum of 1 and 2. Functions also have an arity which corresponds to the number of arguments the function takes as input. Ordinary addition on the natural numbers is a binary function as is multiplication. Subtraction, however, is not even a function on the natural numbers. The point here is that the difference of two natural numbers is not necessarily a natural number. We require

that the outputs of any function considered always be elements of our domain of discourse. (We may change the domain of discourse, however, as the need arises or our mood dictates.) On the other hand, not every element of the domain need be a value for a function. As another example consider a ternary function $g(x,y,z)$ defined as $x \cdot y + z$. Here we see that variables can play the role of placeholders in functions as well as predicates. In analogy with our manipulations of the binary relation φ above, we can define binary and unary functions from g by replacing some of the variables by constants: $g(1,y,1)$ is the unary function $y+1$ and $g(x,y,0)$ is multiplication. How then should we view $g(1,1,0)$? It is, of course, (the constant) 1. Thus, just as we can think of propositions as predicates of arity 0, we can think of constants as functions of arity 0. They are objects which have no dependence on any inputs; they simply denote elements of the domain of discourse. More generally, we call all the symbols generated by the function symbols, constants and variables such as $f(x,g(y,y))$, *terms*. We think of them also as ranging over our domain of discourse (or possibly just some subset of the domain — what is usually called the range of the function).

As with propositions, the truth–functional connectives can be used to build compound predicates from simpler ones. For example, if $\varphi(x,y)$ still denotes the relation "x is less than y" and $\psi(x,y)$ denotes the relation "x divides y" then $(\varphi(x,y) \wedge \psi(x,y))$ is a new binary predicate with the obvious meaning. In addition to the truth functional connectives, predicate logic uses two other predicate constructors:

(i) the *universal quantifier* (with the intended meaning "for all") denoted by "$\forall$",

and

(ii) the *existential quantifier* (with the intended meaning "there exists") denoted by "$\exists$".

Example 1.1:

(i) Let the domain of discourse consist of the natural numbers $\mathbb{N}$; let "$\varphi(x,y)$" denote "$x < y$"; $f(x,y)$ the binary function $x+y$ and a, b, c be constants naming the numbers 0, 1, and 2 respectively:

(a) $((\exists x)\varphi(x,y))$ is a unary predicate which says of y that there is a natural number less than it. It is equivalent to "y is not zero". $((\forall x)((\exists y)\varphi(x,y)))$ is the true sentence (predicate of arity 0) saying that for any natural number x, there is a natural number y which is greater than x.

(b) $((\forall x)\varphi(x,f(x,b)))$ is a sentence saying that $x < x+1$ for every x, i.e., every natural number is less than its successor. $\varphi(y,f(y,y))$ is again a unary predicate saying of y that $y < y+y$. This predicate is also equivalent to y being nonzero.

(ii) Let the domain of discourse consist of all rational numbers $\mathbb{Q}$. Again $\varphi(x,y)$ denotes $x < y$, $f(x,y)$ represents addition $(x+y)$, $g(x,y)$ division $(x \div y)$ and a, b, c are constants representing 0, 1 and 2.

(a) The ternary predicate $(\varphi(x,y) \wedge \varphi(y,z))$ says that $x < y$ and $y < z$.

(b) The binary predicate $((\exists y)(\varphi(x,y) \wedge \varphi(y,z)))$ says that there is a rational number between x and z. The unary predicate $((\forall x)(\varphi(x,z) \rightarrow ((\exists y)(\varphi(x,y) \wedge \varphi(y,z)))))$ expresses a property of z which says that, for any x, if x is less than z then there is a rational number between them.

(c) $((\forall x)((\forall y)(\varphi(x,y) \rightarrow (\varphi(x,g(f(x,y),c)) \wedge \varphi(g(f(x,y),c),y)))))$ is a sentence saying that for every x and y, if $x < y$ then $x < \frac{x+y}{2} < y$.

(d) $\varphi(y,f(y,y))$ is again a unary predicate saying that $y < y+y$. Note, however, that in this domain this predicate is equivalent to y being positive.

2. The Language: Terms and Formulas

We can now give a formal definition of what constitutes an appropriate language for predicate logic and then specify the formulas of predicate logic by an inductive definition which selects certain "well formed" strings of symbols which we think of as the meaningful ones.

Definition 2.1: A *language* $\mathcal{L}$ consists of the following primitive symbols:

(i) Variables: x, y, z, v, x_0, x_1, ..., y_0, y_1, ..., ... (an infinite set)

(ii) Constants: c, d, c_0, d_0, ... (any set of them)

(iii) Connectives: $\wedge$, $\neg$, $\vee$, $\rightarrow$, $\leftrightarrow$

(iv) Quantifiers: $\forall$, $\exists$

(v) Predicate symbols: P, Q, R, P_1, P_2, ... (some set of them for each arity $n = 1, 2, \ldots$. There must be at least one predicate symbol in the language but otherwise there are no restrictions on the number of them for each arity).

(vi) Function symbols: f, g, h, f_0, f_1, ..., g_0, ... (any set of them for each arity $n = 1, 2, \ldots$. The 0–ary function symbols are simply the constants listed by convention separately in (ii). The set of constant symbols may also be empty, finite or infinite).

(vii) Parentheses:) , (.

Note that we no longer have propositional letters (which would be 0–ary predicates). They are simply unnecessary in the context of predicate logic.

A true (false) proposition can be replaced by any sentence which is always true (false) such as one of the form $\alpha \vee \neg\alpha$ $(\alpha \wedge \neg\alpha)$. (See Theorem 4.8 for an embedding of propositional logic in predicate logic.)

As a prelude to defining the formulas of a language $\mathcal{L}$, we define the terms of $\mathcal{L}$ — the symbols which, when interpreted, will represent elements of our domain of discourse. We define them inductively. (Readers who prefer to use the formation tree approach exclusively may skip the more traditional syntactic one given here in favor of the presentation of the next section. They should then take the formulations given there as definitions and omit the proofs of their equivalence to the ones given here.)

Definition 2.2: *Terms.*

(i) Every variable is a term.

(ii) Every constant symbol is a term.

(iii) If f is an n–ary function symbol $(n = 1, 2, \ldots.)$ and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is also a term.

Definition 2.3: Terms with no variables are called *variable–free terms* or *ground terms.*

The ground terms are the ones you should think of as naming particular elements of the domain of discourse. They are the constants and the terms built up from the constants by applications of function symbols as in (iii) above.

The base case for the definition of formulas is given by:

Definition 2.4: An *atomic formula* is an expression of the form $R(t_0, \ldots, t_{n-1})$ where R is an n–ary predicate symbol and $t_0, \ldots, t_{n-1}$ are terms.

We now give the full inductive definition of formulas.

Definition 2.5: *Formulas.*

(i) Every atomic formula is a formula.

(ii) If α, β are formulas, then so are $(\alpha \wedge \beta)$, $(\alpha \to \beta)$, $(\alpha \leftrightarrow \beta)$, $(\neg\alpha)$ and $(\alpha \vee \beta)$.

(iii) If v is a variable and α is a formula, then $((\exists v)\alpha)$ and $((\forall v)\alpha)$ are also formulas.

Definition 2.6:

(i) A *subformula* of a formula φ is a consecutive sequence of symbols from φ which is itself a formula.

(ii) An *occurrence* of a variable v in a formula φ is *bound* if there is a subformula ψ of φ containing that occurrence of v such that ψ begins with $((\forall v)$ or $((\exists v)$. An occurrence of v in φ is *free* if it is not bound. (This includes the v in $\forall v$ or $\exists v$.)

(iii) A variable v is said to *occur free* in φ if it has at least one free occurrence there.

(iv) A *sentence* of predicate logic is a formula with no free occurrences of any variable, i.e., one in which all occurrences of all variables are bound.

(v) An *open formula* is a formula without quantifiers.

To see that the definition of sentence corresponds to the idea of a formula with a single fixed meaning and truth value, notice that all references to variables (which are the only way of moving up to predicates of arity greater than 0) occur in the context of a quantifier. That is, they occur only in the form "there exists an x such that ..." or "for all x it is true that ... " . The idea of replacing a variable by some other term to produce predicates of perhaps smaller arity (as we did in Section 1) is captured by the following definition:

Definition 2.7 *Substitution* (or *Instantiation*): If φ is a formula and v a variable we write $\varphi(v)$ to denote that fact that v occurs free in φ. If t is a term then $\varphi(t)$, or if we wish to be more explicit, $\varphi(v/t)$, is the result of substituting (or instantiating) t for all free occurrences of v in φ. We call $\varphi(t)$ an *instance* of φ. If $\varphi(t)$ contains no free variables, we call it a *ground instance* of φ.

There is one important caveat that must be heeded when doing substitutions.

Definition 2.8: If the term t contains an occurrence of some variable x (which is necessarily free in t) we say that t is *substitutable* for v in $\varphi(v)$ if all occurrences of x in t remain free in $\varphi(t)$.

Note that ground terms are always substitutable for any free variable. The problems with substituting a term t (with variables) which is not substitutable in φ will become clearer when we define the semantics of formulas. For now, we consider two examples.

Example 2.9:

(i) Consider first a unary predicate $\psi(y) = ((\exists x)\varphi(x, y))$ where our notation is as in Example 1.1(i). There is no problem substituting z or 2 or even $f(w, w)$ for y to get $((\exists x)\varphi(x, z))$, $((\exists x)\varphi(x, 2))$ and $((\exists x)\varphi(x, f(w, w)))$ respectively. These formulas simply say that z, 2 and $w + w$ are not zero as we would want and expect. However if we try to substitute $f(x, x)$ for y we get $((\exists x)\varphi(x, f(x, x)))$. This formula says nothing about x or $x + x$; it is simply the true sentence asserting that there is some x such that $x < x + x$.

(ii) Next consider a language for the integers $\mathbb{Z}$ with constants 0 and 1, a unary function symbol s for successor and a predicate $A(x, y, z)$ which is interpreted as $x+y = z$. Let φ be the sentence $\forall x \exists y A(x, y, 0)$

which is true in $\mathbb{Z}$. As a true universal sentence, φ should be true of any object. Indeed, any permissible substitution results in a formula valid in $\mathbb{Z}$. On the other hand, if we violate substitutability and substitute $s(y)$ for x we get $\forall x \exists y A(s(y), y, 0)$ which is false in $\mathbb{Z}$.

Example 2.10:

(i) $((\forall x)R(x,y))$ is a formula in which y occurs free but x does not. $((\exists y)((\forall x)R(x,y)))$ has no free variables; it is a sentence.

(ii) A variable may have both a free and a bound occurrence in a single formula as do both x and y in $(((\forall x)R(x,y)) \vee ((\exists y)R(x,y)))$.

(iii) If $\varphi(x)$ is $(((\exists y)R(x,y)) \wedge ((\forall z)\neg Q(x,z)))$ and t is $f(w,u)$ then $\varphi(t) = \varphi(x/t)$ is $(((\exists y)R(f(w,u),y)) \wedge ((\forall z)\neg Q(f(w,u),z)))$. The term $g(y, s(y))$ would, however, not be substitutable for x in $\varphi(x)$.

After the exercises for this section we will usually omit parentheses from formulas when doing so improves readability.

Exercises

For exercises 1–5 let the language be specified by the symbols listed in Definition 2.1.

1. Which of the following are terms?
   a) x e) $f(x,d)$
   b) xy f) $(\forall x)(R(c))$
   c) c g) $g(c, f(y,z))$
   d) $P(c)$ h) $g(R,d)$
2. Which of the following are formulas fully written out in accordance with Definition 2.5?
   a) $f(x,c)$ d) $\forall x(P(x))$
   b) $R(c, f(d,z))$ e) $(\neg R(z, f(w))$
   c) $(\exists y)(P(c))$ f) $((\exists x)(((\forall y)P(z)) \rightarrow R(x,y)))$
3. List all the subformulas of the formulas listed in exercise 2.
4. Which occurrences of variables are free in the formulas listed in answer to exercise 3? Which are bound?
5. Which of the following proposed substitutions are allowable by our definition of substitutable?
   a) $x/f(z,y)$ in $((\exists y)(P(y) \wedge R(x,z)))$.
   b) $x/g(f(z,y),a)$ in $(((\exists x)(P(x) \wedge R(x,y))) \rightarrow P(x)))$.
   c) $x/g(f(z,y),a)$ in $((\exists x)(P(x) \wedge R(x,y)))$.
   d) $x/g(a,b)$ in $((\exists y)(R(a,x) \wedge P(y)))$.

3. Formation Trees, Structures and Lists

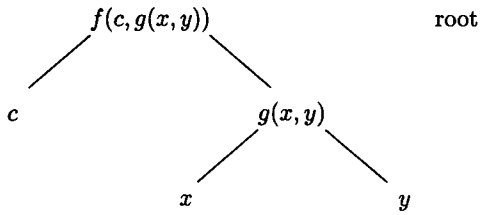
As with the definition of propositions, we can make the formation rules for formulas more explicit and the definition of such terms as "occurrence" more precise by reformulating everything in terms of formation trees. This is also the preferred presentation in most texts on PROLOG programming. Our starting point is again the terms.

Definition 3.1:

(i) *Term formation trees* are ordered, finitely branching trees T labeled with terms satisfying the following condition:
   (1) The leaves of T are labeled with variables or constant symbols.
   (2) Every nonleaf node of T is labeled with a term t of the form $f(t_1, \ldots, t_n)$.
   (3) A node of T which is labeled with a term of the form $f(t_1 \ldots, t_n)$ has exactly n immediate successors in the tree. They are labeled in (lexicographic) order with $t_1, \ldots, t_n$.

(ii) A term formation tree is *associated with* the term with which its root node is labeled.

Example 3.2: (i) The term formation trees associated with $f(c, g(x,y))$ and $h(f(d,z), g(c,a), w)$ are



and

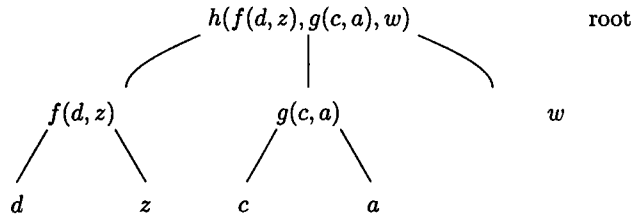


FIGURE 24

Proposition 3.3: *Every term t has a unique formation tree associated with it.*

The proof of this proposition, like those of the other results of this section, is a simple exercise in induction like that of Theorem I.2.4. We will leave them all as exercises. This one is Exercise 4.

Proposition 3.4: *The ground terms are those terms whose formation trees have no variables on their leaves.*

Proof: Exercise 5. □

The atomic formulas are handled as follows:

Definition 3.5:

(i) The *atomic formula auxiliary formation trees* are the labeled, ordered, finitely branching trees of depth one whose root node is labeled with an atomic formula. If the root node of such a tree is labeled with an n–ary relation $R(t_1, \ldots, t_n)$, then it has n immediate successors which are labeled in order with the terms $t_1, \ldots, t_n$.

(ii) The *atomic formula formation trees* are the labeled, ordered, finitely branching trees gotten from the auxiliary trees by attaching at each leaf labeled with a term t the rest of the formation tree associated with t. Such a tree is *associated with* the atomic formula with which its root is labeled.

Example 3.6: The atomic formation trees associated with the formula $R(c, f(x,y), g(a,z,w))$ is

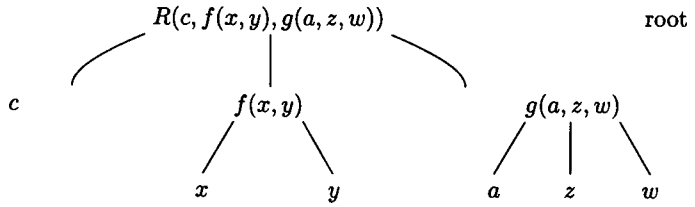


FIGURE 25

Proposition 3.7: *Every atomic formula is associated with a unique formation tree.*

Proof: Exercise 6. □

Definition 3.8:

(i) The *formula auxiliary formation trees* are the labeled, ordered, binary branching trees T such that

(1) The leaves of T are labeled with atomic formulas.

(2) If σ is a nonleaf node of T with one immediate successor $\sigma \wedge 0$ which is labeled with a formula φ, then σ is labeled with $\neg\varphi$, $\exists v\varphi$ or $\forall v\varphi$ for some variable v.

(3) If σ is a nonleaf node with two immediate successors, $\sigma \wedge 0$ and $\sigma \wedge 1$, which are labeled with formulas φ and ψ, then σ is labeled with $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$ or $\varphi \leftrightarrow \psi$.

(ii) The *formula formation trees* are the ordered, labeled trees gotten from the auxiliary ones by attaching to each leaf labeled with an atomic formula the rest of its associated formation tree. Each such tree is again *associated* with the formula with which its root is labeled.

(iii) The *depth of a formula* is the depth of the associated auxiliary formation tree.

Example 3.9: The formula formation tree associated with the formula $\exists x R(c, f(x,y), g(a,z,w)) \wedge \forall y R(c, f(x,y), g(a,z,w))$ is

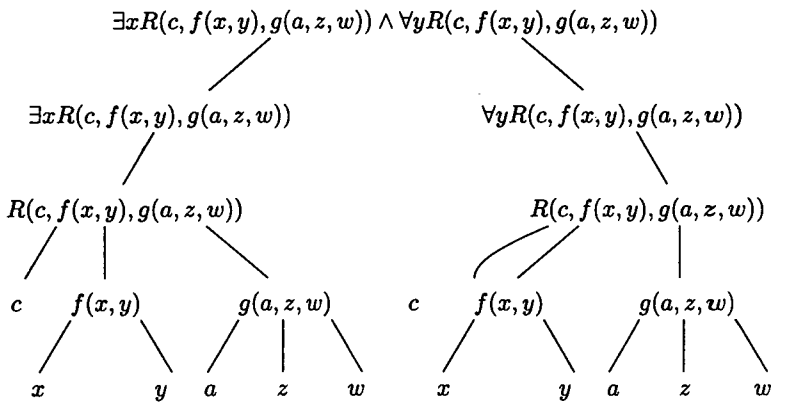


FIGURE 26

Proposition 3.10: *Every formula is associated with a unique (auxiliary) formation tree.*

Proof: Exercise 3.7. □

Formally, we continue to treat the remaining notions about formulas, subformulas and occurrences of variables as the proven equivalents of the notions defined in the last section. Those definitions could, however, actually be replaced with the ones we present here.

Proposition 3.11: *The subformulas of a formula φ are the labels of the nodes of the auxiliary formation tree associated with φ.*

Proposition 3.12:

(i) *The occurrences of a variable v in a formula φ are in one–one correspondence with the leaves of the associated formation tree which are labeled with v. (The correspondence is given by matching the typographical ordering of the occurrences of v in φ with the left–right ordering given by the tree to the leaves labeled with v.) We may also refer to the appropriate leaf labeled with v as the occurrence of v in φ.*

(ii) *An occurrence of the variable v in φ is bound if there is a formula ψ beginning with $((\forall v)$ or $((\exists v)$ which is the label of a node above the corresponding leaf of the formation tree for φ labeled with v.*

Proposition 3.13: *If φ is a formula and v a variable then $\varphi(v/t)$ is the formula associated with the formation tree gotten by replacing each leaf in the tree for $\varphi(v)$ which is labeled with a free occurrence of v with the formation tree associated with t and propagating this change through the tree.*

Proposition 3.14: *The term t is* substitutable *for v in $\varphi(v)$ if all occurrences of x in t remain free in $\varphi(t)$, i.e., any leaf in the formation tree for t which is a free occurrence of a variable x remains free in every location in which it appears in the formation tree described in Proposition 3.9.*

We leave the proofs of these propositions as Exercises 8–11.

Notice that, except for the distinction we have made in our alphabet between function symbols and predicate symbols, the formation trees for terms and atomic formulas are indistinguishable. Each has leaves labeled with constants or variables and every other node is labeled by applying one of the appropriate n–ary symbols to the labels of its immediate successors. The standard implementations of PROLOG, and so the various programming texts, in fact do not make this alphabetic distinction. Terms and atomic formulas are all lumped together and called *structures*. One can therefore have a syntactically acceptable PROLOG clause like "reading(john, reading(jack,list1))". This PROLOG clause might be rendered into English as follows: John is reading Jack's first reading list. Here "reading" is thought of both as a predicate describing who is reading what and a function giving people's items for reading. In general, however, it seems very difficult to make consistent sense out of such combined usages. The semantics we present in the next section, which is the standard one for predicate logic, makes no sense unless we maintain the distinction between function and predicate symbols. As it is the basis for the theoretical analysis of PROLOG (in terms of soundness and completeness, for example) and we know of no reason that it might ever be necessary to exploit such a confusion, we will simply assume that separate alphabets are maintained for function and predicate symbols (at least within any particular program or application).

Example 3.15: As an example of a typical PROLOG structure, we briefly consider one of its most important function symbols (or operators), the *pairing function* denoted by ".". Thus $.(a, b)$ denotes the ordered pair with first element a and second element b. This function is used to form arbitrary lists by repeated application. Practically speaking, the operator "." in PROLOG should be applied only to pairs the second element of which is already a list (the first element can be anything). To get such a procedure off the ground, PROLOG starts with a constant symbol [ ] denoting the empty list (one with no elements). Thus a list consisting of just the element b would be represented by $.(b, [\,])$ and the ordered pair $\langle a, b\rangle$ would actually be realized as $.(a, .(b, [\,]))$. As this notation is cumbersome, lists in PROLOG are also denoted by putting their elements in order within square brackets and separating them by commas. Thus $[a, b, c, d]$ denotes the list with elements a, b, c and d in that order. This notation is really an abbreviation for an iterated use of pairing (with the convention that we always end with the empty list). $[a, b, c, d]$ is treated as if it were $.(a, .(b, .(c, .(d, [\,]))))$. Its formation tree is given in Figure 27 below.

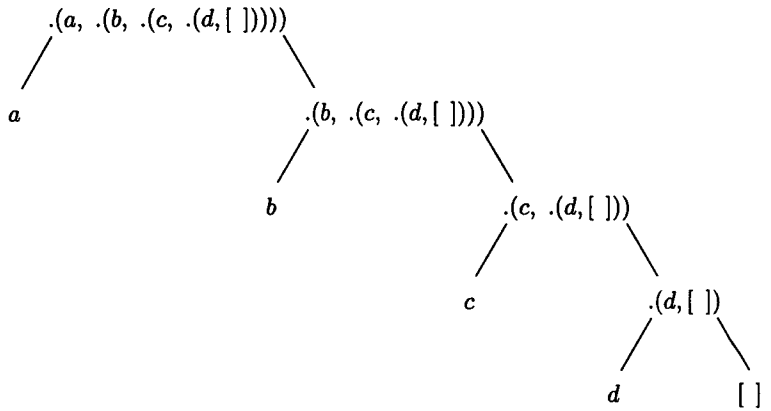


FIGURE 27

The list $[a, b, c, d]$ is also written $[a \mid [b, c, d]]$. The notation with the vertical bar, $\mid$, is another version of the function symbol . for combining lists. $[X \mid Y]$ denotes the list whose first element is X and whose succeeding elements are those of the list Y in order. The terminology that accompanies this notation is that X, the first element of the new list, is called the head of the list $[X \mid Y]$ and the list Y consisting of the remaining elements is called its tail.

The reason for avoiding terms like $[a \mid b]$ or equivalently $.(a, b)$ when b is not a list is that we usually define list handling functions by recursion. The starting point for such recursions is generally the empty list [ ]. Thus a function so defined would never be computed on an input like $.(a, b)$ when b

is not a list. We will return to this point with examples and explanations of definition by recursion in section 5 after we have defined the semantics for predicate logic and PROLOG.

Exercises

1. Draw the formation trees associated with the following terms:
   a) c
   b) $f(x,d)$
   c) $g(f(x,d),c)$
   d) $h(y,g(z,f(f(c,d),g(x,z))))$
2. Draw the formation trees associated with the following formulas:
   a) $R(c,d)$
   b) $R(f(x,y),d)$
   c) $R(c,d) \wedge R(f(x,y),d)$
   d) $\exists y \forall z (R(x,f(c,d)) \vee \neg P(h(y)))$
   e) $\forall z (R(g(x,z,z)) \rightarrow P(y)) \wedge P(z)$
3. Indicate which leaves are free occurrences of variables in the trees of exercise 2.
4. Prove Proposition 3.3.
5. Prove Proposition 3.4.
6. Prove Proposition 3.7.
7. Prove Proposition 3.10.
8. Prove Proposition 3.11.
9. Prove Proposition 3.12.
10. Prove Proposition 3.13.
11. Prove Proposition 3.14.
12. Prove that the length of every term t in a language $\mathcal{L}$ for predicate logic is greater than or equal to the depth of the associated formation tree.
13. Prove that the length of every formula φ of predicate logic is strictly greater than the depth of the associated formation tree.

4. Semantics: Meaning and Truth

A language $\mathcal{L}$ of predicate logic is specified by its predicate (or relation) symbols and function symbols. A single language will have many possible interpretations each suited to a different context or domain of discourse. Thus the language with just one binary predicate $P(x,y)$ can be viewed as talking about any of the following situations:

1) The natural numbers, $\mathbb{N}$, with $<$.
2) The rationals, $\mathbb{Q}$, with $<$.
3) The integers, $\mathbb{Z}$, with $>$.

or any of a host of other possibilities. If we add a binary function symbol $f(x,y)$, we could view f as representing, for example, $x \cdot y$, $x-y$ or $\max\{x,y\}$ in these respective domains. To begin to interpret the language, we must specify a domain of discourse and the intended meanings for the predicate and function symbols.

Definition 4.1: A *structure* $\mathcal{A}$ for a language $\mathcal{L}$ consists of a nonempty domain A, an assignment, to each n–ary predicate symbol R of $\mathcal{L}$, of an actual predicate (i.e., a relation) $R^{\mathcal{A}}$ on the n–tuples $(a_1,\ldots,a_n)$ from A, an assignment, to each constant symbol c of $\mathcal{L}$, of an element $c^{\mathcal{A}}$ of A and, to each n–ary function symbol f of $\mathcal{L}$, an n–ary function $f^{\mathcal{A}}$ from A^n to A.

In terms of the examples considered above, we can specify structures for the language with one binary predicate by letting the domain be $\mathbb{N}$, $\mathbb{Q}$, or $\mathbb{Z}$ respectively. The interpretations $P^{\mathcal{A}}$ of the binary predicate are then $<$, $<$, and $>$ respectively. When we add the binary function symbol f we must specify in each case a binary function $f^{\mathcal{A}}$ on the domain to interpret it. In each of our examples, the function would be the one specified above: multiplication, subtraction or max.

We begin the task of interpreting the formulas of $\mathcal{L}$ in the structure $\mathcal{A}$ by saying, for each ground term of the language $\mathcal{L}$, which element of the domain of discourse A it names.

Definition 4.2 (The *interpretation of ground terms*):

(i) Each constant term c *names* the element $c^{\mathcal{A}}$.

(ii) If the terms $t_1,\ldots,t_n$ of $\mathcal{L}$ *name* the elements $t_1^{\mathcal{A}},\ldots,t_n^{\mathcal{A}}$ of A and f is an n–ary function symbol of $\mathcal{L}$ then the term $f(t_1,\ldots,t_n)$ *names* the element $f(t_1,\ldots,t_n)^{\mathcal{A}} = f^{\mathcal{A}}(t_1^{\mathcal{A}},\ldots,t_n^{\mathcal{A}})$ of A. (Remember that $f^{\mathcal{A}}$ is an n–ary function on A and that $t_1^{\mathcal{A}},\ldots,t_n^{\mathcal{A}}$ are elements of A so that $f^{\mathcal{A}}(t_1^{\mathcal{A}},\ldots,t_n^{\mathcal{A}})$ is in fact an element of A.)

Continuing with our above examples, we might add constants c and d to our language and assign them to elements $c^{\mathcal{A}}$ and $d^{\mathcal{A}}$ of the structures as follows:

1) $c^{\mathcal{A}} = 0$; $d^{\mathcal{A}} = 1$.
2) $c^{\mathcal{A}} = 1/2$; $d^{\mathcal{A}} = 2/3$.
3) $c^{\mathcal{A}} = 0$; $d^{\mathcal{A}} = -2$.

Suppose f is interpreted as multiplication in each of the three structures. Then the ground terms $f(c,d)$ and $f(d,f(d,d))$ name elements of the structures as follows:

1) $(f(c,d))^{\mathcal{A}} = 0$; $(f(d,f(d,d)))^{\mathcal{A}} = 1$.
2) $(f(c,d))^{\mathcal{A}} = 1/3$; $(f(d,f(d,d)))^{\mathcal{A}} = 8/27$.
3) $(f(c,d))^{\mathcal{A}} = 0$; $(f(d,f(d,d))^{\mathcal{A}} = -8$.

It is convenient to deal with structures $\mathcal{A}$ for languages $\mathcal{L}$ which have a ground term naming every element a of A. If we are given a structure $\mathcal{A}$ for a language $\mathcal{L}$ in which not every element of the domain is named by a ground term, we *expand* $\mathcal{L}$ by adding a new constant c_a to $\mathcal{L}$ for each $a \in A$ to get a language $\mathcal{L}^{\mathcal{A}}$ and *extend* $\mathcal{A}$ to a structure for $\mathcal{L}^{\mathcal{A}}$ by interpreting these constants in the obvious way: $c_a^{\mathcal{A}} = a$. Thus in $\mathcal{L}^{\mathcal{A}}$ every element of the domain A is named by a constant. Notice that every structure $\mathcal{A}$ for $\mathcal{L}$ becomes one for $\mathcal{L}^{\mathcal{A}}$ in this way and every structure for $\mathcal{L}^{\mathcal{A}}$ becomes one for $\mathcal{L}$ by simply ignoring the constants c_a.

We can now define when a sentence φ of a language $\mathcal{L}$ is true in a given structure for $\mathcal{L}$. We will write this as $\mathcal{A} \vDash \varphi$. The formal definition is by induction on sentences in the expected manner. The interesting case is that for the quantifiers. Here it is necessary to have ground terms which name each element of A. If there are not enough ground terms in $\mathcal{L}$, we simply use the definition in $\mathcal{L}^{\mathcal{A}}$. Thus, we assume in the following definition that every $a \in A$ is named by a ground term of $\mathcal{L}$.

Definition 4.3: The *truth* of a sentence φ of $\mathcal{L}$ in a structure $\mathcal{A}$ in which every $a \in A$ is named by a ground term of $\mathcal{L}$ is defined by induction. (If not every element of A is so named, we use the definition of $\mathcal{A} \vDash \varphi$ for $\mathcal{L}^{\mathcal{A}}$ to define $\mathcal{A} \vDash \varphi$ for sentences φ of $\mathcal{L}$.)

(i) For an *atomic* sentence $R(t_1, \ldots, t_n)$, $\mathcal{A} \vDash R(t_1, \ldots, t_n)$ iff $R^{\mathcal{A}}(t_1^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}})$, i.e., the relation $R^{\mathcal{A}}$ on A^n assigned to R holds of the elements named by the terms $t_1, \ldots, t_n$. Note that, as $R(t_1, \ldots, t_n)$ is a sentence, the t_i are all ground terms and so name particular elements of A.

(ii) $\mathcal{A} \vDash \neg\varphi \Leftrightarrow$ it is not the case that $\mathcal{A} \vDash \varphi$. [We also write this as $\mathcal{A} \nvDash \varphi$.]

(iii) $\mathcal{A} \vDash (\varphi \vee \psi) \Leftrightarrow \mathcal{A} \vDash \varphi$ or $\mathcal{A} \vDash \psi$.

(iv) $\mathcal{A} \vDash (\varphi \wedge \psi) \Leftrightarrow \mathcal{A} \vDash \varphi$ and $\mathcal{A} \vDash \psi$.

(v) $\mathcal{A} \vDash (\varphi \rightarrow \psi) \Leftrightarrow \mathcal{A} \nvDash \varphi$ or $\mathcal{A} \vDash \psi$.

(vi) $\mathcal{A} \vDash (\varphi \leftrightarrow \psi) \Leftrightarrow (\mathcal{A} \vDash \varphi$ and $\mathcal{A} \vDash \psi)$ or $(\mathcal{A} \nvDash \varphi$ and $\mathcal{A} \nvDash \psi)$.

(vii) $\mathcal{A} \vDash \exists v\varphi(v) \Leftrightarrow$ for some ground term t, $\mathcal{A} \vDash \varphi(t)$.

(viii) $\mathcal{A} \vDash \forall v\varphi(v) \Leftrightarrow$ for all ground terms t, $\mathcal{A} \vDash \varphi(t)$.

Note that truth (or *satisfaction*, as $\vDash$ is often called) for longer sentences is always defined in (ii)–(viii) in terms of truth for shorter sentences. It is for clauses (vii) and (viii) that the assumption that all elements of our structure are named by ground terms is crucial.

Definition 4.4: Fix some language $\mathcal{L}$.

(i) A sentence φ of $\mathcal{L}$ is *valid*, $\vDash \varphi$, if it is true in all structures for $\mathcal{L}$.

(ii) Given a set of sentences $\Sigma = \{\alpha_1, \ldots\}$, we say that α is *logical consequence* of Σ, $\Sigma \vDash \alpha$, if α is true in every structure in which all of the members of Σ are true.

(iii) A set of sentences $\Sigma = \{\alpha_1, \ldots\}$ is *satisfiable* if there is a structure $\mathcal{A}$ in which all the members of Σ are true. Such a structure is called a *model* of Σ. If Σ has no model it is *unsatisfiable*.

Note that we have defined truth only for sentences, that is, formulas with no free variables. The point here is that if $\varphi(v)$ has v free then the formula has no single fixed meaning in a structure $\mathcal{A}$. It rather represents an n–ary predicate on A for $n > 0$ and so we do not say that it is true or false. The notion for formulas with free variables that is analogous to truth for sentences is that of validity.

Definition 4.5: A formula φ of a language $\mathcal{L}$ with free variables $v_1, \ldots, v_n$ is *valid in a structure* $\mathcal{A}$ for $\mathcal{L}$ (also written $\mathcal{A} \vDash \varphi$) if the *universal closure* of φ, i.e., the sentence $\forall v_1 \forall v_2, \ldots \forall v_n \varphi$ gotten by putting $\forall v_i$ in front of φ for every free variable v_i in φ, is true in $\mathcal{A}$. The formula φ of $\mathcal{L}$ is *valid* if it is valid in every structure for $\mathcal{L}$.

As long as we are in a situation in which every element of the structure $\mathcal{A}$ is named by a ground term, this definition of validity in $\mathcal{A}$ is equivalent to saying that every *ground instance* of φ is true in $\mathcal{A}$, i.e., $\mathcal{A} \vDash \varphi(t_1, \ldots, t_n)$ for all ground terms $t_1, \ldots, t_n$ of $\mathcal{L}$. Also note that as sentences have no free variables, a sentence is true in a structure iff it is valid in the structure.

Warning: For a sentence φ and structure $\mathcal{A}$ either φ or $\neg\varphi$ is true in $\mathcal{A}$ (and the other false). It is not true, however, for an arbitrary formula ψ that ψ or $\neg\psi$ must be valid in $\mathcal{A}$. It may well be that some ground instances of ψ are true while others are false. Similarly, one can have a sentence such that neither it nor its negation is valid. It is true in some structures but not in others.

Definition 4.6: A set Σ of formulas with free variables is *satisfiable* if there is a structure in which all of the formulas in Σ are valid (i.e., their universal closures are true). Again such a structure is called a *model* of Σ. If Σ has no models it is *unsatisfiable*.

Example 4.7: Consider a language $\mathcal{L}$ specified by a binary relation symbol R and constants $c_0, c_1, c_2, \ldots$. Here are two possible structures for $\mathcal{L}$ corresponding to two different interpretations of the language.

(i) Let the domain A consist of the natural numbers, let $R^{\mathcal{A}}$ be the usual relation $<$, and $c_0^{\mathcal{A}} = 0$, $c_1^{\mathcal{A}} = 1, \ldots$. The sentence $(\forall x)(\exists y)R(x,y)$ says that for every natural number there is a larger one, so it is true in this structure.

(ii) Let the domain of $\mathcal{A}$ consist of the rational numbers $\mathbb{Q} = \{q_0, q_1, \dots\}$; let $R^{\mathcal{A}}$ again be $<$, and let $c_0^{\mathcal{A}} = q_0$, $c_1^{\mathcal{A}} = q_1, \dots$. The sentence $(\forall x)(\forall y)(R(x,y) \to (\exists z)(R(x,z) \wedge R(z,y)))$ is true in this structure. (It says that the rationals are dense.) It is not, however, valid as it is false in the structure of (i) for the natural numbers.

Warning: We have not included any special or reserved predicate symbol for equality in either our syntax or semantics for predicate logic. In other words, we have made no provisions in our definitions that could be used to force us to interpret some particular predicate, such as "$=$", as true equality. We have avoided this extension of our definition of a language in 2.1 and the corresponding restriction in the definition of truth in 4.3 because it does not mesh well with resolution theorem proving, logic programming and PROLOG. Some of the perhaps unexpected consequences of this choice can be seen in Exercises 2–3 of §7. A view of predicate logic with such a distinguished equality predicate (as well as an approach to equality without it) is presented in III.5. The syntax and semantics presented there can be read now. The proofs of soundness and completeness discussed there for logic with equality are simple modifications of the ones we present in §7.

Now that we have defined the semantics for predicate logic we can make precise the claim that we do not need propositions. Indeed there is a faithful embedding of propositional logic in predicate logic.

Theorem 4.8: *Let φ be an open (i.e., quantifier-free) formula of predicate logic. We may view φ as a formula φ' of propositional logic by regarding every atomic subformula of φ as a propositional letter. With this correspondence, φ is a valid formula of predicate logic if and only if φ' is valid in propositional logic.*

Proof: Exercises 9–11. □

Now that we have both the syntax and semantics for predicate logic it should be clear by analogy with our development of propositional logic what we should do next. We have to give methods of proof in predicate logic and then prove soundness and completeness theorems analogous to those we have seen for the propositional calculus. First, however, we will consider the application (or actually, the specialization) of our semantics to PROLOG.

Notation: We will often use vector notation as in $\vec{x}$, $\vec{t}$ and $\vec{c}$ to denote sequences of variables, terms and constants respectively.

Exercises

1. Let $\mathcal{L}$ contain a constant c, a binary function f and a unary predicate P. Give two structures for $\mathcal{L}$: one in which $\forall x\ P(f(x,c))$ is true and one in which it is false.

2. Show that $\forall x(p(x) \to q(f(x))) \wedge \forall x\ p(x) \wedge \exists x \neg q(x)$ is satisfiable.
3. Give an example of an unsatisfiable sentence.
4. Define a structure for the language containing constant symbols 0 and 1, a binary predicate $<$ and one binary function symbol $+$ in which $x+1 < x$ is valid but $x+x < x$ is not. Indicate why the structure has the required properties.
5. Prove that $\mathcal{A} \models \neg\exists x\varphi(x) \Leftrightarrow \mathcal{A} \models \forall x \neg\varphi(x)$. Does it matter if φ has free variables other than x?
6. Prove that, for any sentence ψ, $\mathcal{A} \models (\psi \to \exists x\varphi(x)) \Leftrightarrow \mathcal{A} \models \exists x(\psi \to \varphi(x))$. What happens if ψ is a formula in which x is free?
7. Prove that for any sentence ψ, $\mathcal{A} \models (\exists x\varphi(x) \to \psi) \Leftrightarrow \mathcal{A} \models \forall x(\varphi(x) \to \psi)$. What happens if ψ is a formula in which x is free?
8. **Theorem on constants:** Let $\varphi(\vec{x})$ be a formula of a language $\mathcal{L}$ with a sequence $\vec{x}$ of free variables. Let $\vec{c}$ be a sequence of new constants (not in $\mathcal{L}$). Prove that $\varphi(\vec{x})$ is valid iff $\varphi(\vec{c})$ is.
9. Prove Theorem 4.8 for formulas with no free variables. (Hint: Convert between models for φ or $\neg\varphi$ to assignments making φ' or $\neg\varphi'$ true.)
10. Combine exercises 8 and 9 to prove Theorem 4.8.

5. Interpretation of PROLOG Programs

In this section we want to specialize the ideas and definitions of the last section to explain the semantics of clausal form and Horn formulas with free variables and so begin the study of the semantics of full PROLOG programs.

The syntax for clausal form and PROLOG format is the same as in the propositional case (Definition I.10.4) except that *literals* can now be any atomic formulas or their negations. Note, however, that implementations of PROLOG uniformly use (initial) capital letters for (names of) variables and lower case ones for (names of) predicates, constants and functions.

Definition 5.1 (*Clausal Notation*)**:**

(i) *Literals* are atomic formulas or their negations. The atomic formulas are called *positive literals* and their negations, *negative literals*.

(ii) A *clause* is a finite set of literals.

(iii) A clause is a *Horn clause* if it contains at most one positive literal.

(iv) A *program clause* is a clause with exactly one positive literal. If a program clause contains some negative literals it is a *rule*, otherwise it is a *fact*.

(v) *A goal clause* is a clause with no positive literals.

(vi) A *formula* is a not necessarily finite set of clauses.

The PROLOG notation for rules and facts is as in the propositional case as well.

Definition 5.2 (PROLOG *Notation*):

(i) In PROLOG, the fact $\{p(\vec{X})\}$ consisting of the single positive literal $p(\vec{X})$ appears in PROLOG programs as follows:

$$p(\vec{X}).$$

(ii) The rule $C = \{p(\vec{X}), \neg q_1(\vec{X},\vec{Y}), \ldots, \neg q_n(\vec{X},\vec{Y})\}$ appears in PROLOG programs as follows:

$$p(\vec{X}) :- \; q_1(\vec{X},\vec{Y}), \ldots, q_n(\vec{X},\vec{Y}).$$

(iii) For a rule C as in (ii), we call $p(\vec{X})$ the *goal* or *head* of C. We call the $q_1(\vec{X},\vec{Y}), \ldots, q_n(\vec{X},\vec{Y})$ the *subgoals* or *body* of C. When the head–body terminology is used, the symbol :- which connects the head and body of C is called the *neck*.

(iv) A (PROLOG) *program* is a formula (set of clauses) containing only program clauses (i.e., rules and facts).

The intended meaning of clauses and formulas is as in the propositional case except that we must explain how we treat the free variables. Each clause is interpreted as the universal closure of the disjunction of its elements. Thus the intended meaning of $C_1 = \{q(X,Y),\ r(Y)\}$ is $\forall X \forall Y [q(X,Y) \vee r(Y)]$. In this vein the intended meaning of the rule C given by $p(X) :- \; q_1(X,Y), \ldots, q_n(X,Y)$ (in clausal notation $C = \{p(X), \neg q_1(X,Y), \ldots, \neg q_n(X,Y)\}$) is $\forall X \forall Y [p(X) \vee \neg q_1(X,Y) \vee \ldots \vee \neg q_n(X,Y)]$. Repeated applications of Exercises 4.8 and I.3.2 would show that this is equivalent to $\forall X [\exists Y (q_1(X,Y) \wedge \ldots \wedge q_n(X,Y)) \rightarrow p(X)]$. (We will later analyze some examples to see how this equivalence is worked out.) Thus C truly embodies a rule: If, for any X, there is a Y such that $q_1(X,Y), q_2(X,Y), \ldots, q_n(X,Y)$ are all true (have been verified), then $p(X)$ is also true (has been verified).

A formula S is interpreted as the conjunction of its clauses. Thus if $S = \{C_1, C_2\}$ where C_1 is as above and $C_2 = \{q(X,Y), m(Y)\}$, then S has the same meaning as $\forall X \forall Y [q(X,Y) \vee r(Y)] \wedge \forall X \forall Y [q(X,Y) \vee m(Y)]$. In particular, if the formula S is a PROLOG program then it is equivalent to a list of universal facts of the form $\forall X p(X)$ and rules like C in the previous paragraph. Implementing PROLOG consists in making deductions from such a list of facts and rules.

Note that in describing the intended meaning of a formula, each clause is universally closed before we take the conjunction. The importance of this convention will become apparent when we consider resolution for predicate calculus. Confusion can be avoided by using distinct variables in each clause. (This corresponds to what is called *standardizing the variables apart* (see §13)). We will later (§9) show that, at the expense of adding new function symbols to our language, every sentence of predicate calculus

is equivalent to a formula in the sense of Definition 5.1. (This result will be the analog of CNF for the predicate calculus.) For now, after one example involving such transformations, we will simply deal with the syntax and semantics of formulas in clausal form directly. The notions of structures for, and interpretations of, formulas in clausal form are immediately specified by the above translation into predicate calculus.

*Knight's moves: an example:

Let us briefly examine the added expressive power given to us by using variables in PROLOG by considering the problem of representing information about a knight's moves on a chessboard. We can label the squares of the board by pairs of numbers from 1 to 8 in the usual way.

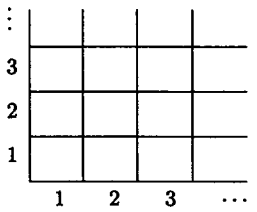


FIGURE 28

We thus might well want our language to include the constant symbols $1, 2, \ldots, 8$. One of the basic predicates of our language should be the 4–ary one, ktmove(X_1, X_2, X_3, X_4). (For the moment we want to avoid worrying about the actual representation of pairs in PROLOG via the list notation introduced in §3 and so will use a 4–ary predicate on $\{1, \ldots 8\}$ rather than a binary one on pairs of numbers.) The **intended** interpretation of "ktmove(X_1, X_2, X_3, X_4)" is that a knight is allowed to move from position $\langle X_1, X_2 \rangle$ to $\langle X_3, X_4 \rangle$. One way to represent the data involved is to simply list all the facts:

ktmove$(1,1,2,3)$.

ktmove$(1,1,3,2)$.

...

The list is, however, quite long (336 facts). Moreover, the situation quickly becomes intolerable when we ask for only a little bit more. Suppose we also wish to have another predicate 2ktmove(X_1, X_2, X_3, X_4) which says that a knight can get from $\langle X_1, X_2 \rangle$ to $\langle X_3, X_4 \rangle$ in two moves. Here too we could enter a very long list of facts:

2ktmove$(1,1,1,1)$.

2ktmove$(1,1,3,5)$.

...

This is really all we could hope to do were we restricted to propositional logic. That situation corresponds to simply having a pure database. Once we have variables, however, we can greatly condense the representation of the data by the introduction of rules. The one for 2ktmove is obvious: 2ktmove(X_1, X_2, X_3, X_4) if there are Y_1, Y_2 such that ktmove(X_1, X_2, Y_1, Y_2) and ktmove (Y_1, Y_2, X_3, X_4). In predicate logic we might express this rule or definition as follows:

$$\forall X_1 \forall X_2 \forall X_3 \forall X_4 [\exists Y_1 \exists Y_2 (\text{ktmove}\,(X_1, X_2, Y_1, Y_2) \wedge \text{ktmove}\,(Y_1, Y_2, X_3, X_4)) \rightarrow \text{2ktmove}\,(X_1, X_2, X_3, X_4)] \quad (\#)$$

We will introduce a general method for converting all such sentences of predicate calculus to clausal equivalents (or PROLOG programs) in §9. For now we analyze this one in an ad hoc way. We begin with eliminating the implication in favor of $\neg$ and $\vee$ as in constructing a CNF in propositional logic to get the following sentence equivalent to (#):

$$\forall X_1 \forall X_2 \forall X_3 \forall X_4 [\neg(\exists Y_1 \exists Y_2)(\text{ktmove}\,(X_1, X_2, Y_1, Y_2) \wedge \text{ktmove}\,(Y_1, Y_2, X_3, X_4)) \vee \text{2ktmove}\,(X_1, X_2, X_3, X_4)].$$

The next steps are to apply the equivalence of $\neg\exists z\varphi$ to $\forall z\neg\varphi$ established in Exercise 4.5 and then De Morgan's laws from Exercise I.3.2 to get

$$\forall X_1 \forall X_2 \forall X_3 \forall X_4 [\forall Y_1 \forall Y_2 [\neg\text{ktmove}\,(X_1, X_2, Y_1, Y_2)] \vee \neg\text{ktmove}\,(Y_1, Y_2, X_3, X_4)] \vee \text{2ktmove}\,(X_1, X_2, X_3, X_4)].$$

Finally we have an equivalent of (#) which is essentially in clausal form

$$\forall X_1 \forall X_2 \forall X_3 \forall X_4 \forall Y_1 \forall Y_2 [\neg\text{ktmove}\,(X_1, X_2, Y_1, Y_2) \vee \neg\text{ktmove}\,(Y_1, Y_2, X_3, X_4) \vee \text{2ktmove}\,(X_1, X_2, X_3, X_4)].$$

(The semantic equivalence of these last two sentences should be clear.)

The clausal form of our rule originally stated in predicate calculus as (#) is thus simply:

$$\{\neg\text{ktmove}\,(X_1, X_2, Y_1, Y_2), \neg\text{ktmove}\,(Y_1, Y_2, X_3, X_4), \text{2ktmove}\,(X_1, X_2, X_3, X_4)\}.$$

This is a Horn clause which we write in PROLOG notation as

$$\text{2ktmove}\,(X_1, X_2, X_3, X_4) \text{ :- } \text{ktmove}\,(X_1, X_2, Y_1, Y_2), \text{ktmove}\,(Y_1, Y_2, X_3, X_4).$$

Thus, we have an example of the general interpretation of a program clause of the form "$p(\vec{X})$:- $q_1(\vec{X}, \vec{Y}), \ldots, q_n(\vec{X}, \vec{Y})$" in PROLOG. It is a rule which says that, for every choice of the variables $\vec{X}$ in the goal (*head*) $p(\vec{X})$ of the clause, p holds of $\vec{X}$ (*succeeds*) if there are $\vec{Y}$ such that all of the subgoal (*body*) clauses $q_1(\vec{X}, \vec{Y}), \ldots, q_n(\vec{X}, \vec{Y})$ hold (*succeed*). In our case,

the clause says as expected that you can get from $\langle X_1, X_2\rangle$ to $\langle X_3, X_4\rangle$ in two knight's moves if there is a $\langle Y_1, Y_2\rangle$ such that you can get from $\langle X_1, X_2\rangle$ to $\langle Y_1, Y_2\rangle$ in one move and from $\langle Y_1, Y_2\rangle$ to $\langle X_3, X_4\rangle$ in another.

Let us see how we might reduce the size of the program representing ktmove from 336 clauses to one of more manageable size by the introduction of other rules. One approach is to introduce symmetry type rules that would enable us to derive every knight's move from a small list of basic moves. One obvious such rule is symmetry itself:

(S1) ktmove (X_1, X_2, X_3, X_4) :- ktmove (X_3, X_4, X_1, X_2).

Remember that this rule says that (for any X_1, X_2, X_3, X_4) if a knight can move from $\langle X_3, X_4\rangle$ to $\langle X_1, X_2\rangle$, it can move from $\langle X_1, X_2\rangle$ to $\langle X_3, X_4\rangle$. Introducing this rule would allow us to cut our database in half. Other possible such rules include the following:

(S2) ktmove (X_1, X_2, X_3, X_4) :- ktmove (X_1, X_4, X_3, X_2).

(S3) ktmove (X_1, X_2, X_3, X_4) :- ktmove (X_2, X_1, X_4, X_3).

(S4) ktmove (X_1, X_2, X_3, X_4) :- ktmove (X_3, X_2, X_1, X_4).

(Check that these are in fact correct rules about a knight's behavior in chess.) We could then list just a few basic moves which, together with these program clauses, would correctly define the predicate ktmove. (It is correct in the sense that any structure satisfying all these facts and rules would give exactly the legal knight's moves as the quadruples of constants $\{1, \ldots, 8\}$ of which the predicate "ktmove" holds. The correctness of the program in terms of execution, which employs resolution–type theorem proving, will be dealt with later.)

Another tack might be to try to define "ktmove" in terms of arithmetic operations on the positions, i.e., to capture, in some sense, the rule as it is usually taught: The knight may move from $\langle X_1, X_2\rangle$ to $\langle X_3, X_4\rangle$ if the change in one coordinate is 1 and 2 in the other, i.e.,

(A0) ktmove (X_1, X_2, X_3, X_4) if $|X_1 - X_3| + |X_2 - X_4| = 3$.

(We must also make sure that the two positions are different. This will be taken care of by the way we define the appropriate arithmetic operations. In particular 0 will not be an allowed value for $|X_1 - X_3|$.) Now PROLOG has many arithmetic operations and predicates built in but a precise understanding of how they are used requires knowing more about how programs are implemented. So for now, we wish to avoid using the built–in predicates. We can, however, put into our program definitions of our own for as much arithmetic as we need. (Be careful not to use the names reserved for built–in predicates for the ones you define.)

To begin with, we might define the "succeeded by" predicate on the set of numbers $\{1, \ldots, 8\}$ by a database:

suc (1, 2).
suc (2, 3).
⋮
suc (7, 8).

We could then define a truncated version of addition by the following rules:

(A1) add $(X, 1, Z)$:- suc (X, Z).

(A2) add (X, Y, Z) :- suc(Y_1, Y), suc(Z_1, Z), add (X, Y_1, Z_1).

We could then directly define $|X_1 - X_2| = Y$ by:

(A3) absolute_difference (X_1, X_2, Y) :- add (X_1, Y, X_2).

(A4) absolute_difference (X_1, X_2, Y) :- add (X_2, Y, X_1).

(These rules do what we want because we are only interested in truncated operations, i.e., only on what happens on $\{1, \ldots, 8\}$. They do not define the operations correctly on all the integers. We will say more about intended structures for a program later.)

So far we have been considering the meaning of clauses in a PROLOG program entered as such, e.g., by "consulting" a file containing the program as listed. We must now explain the semantics of goal clauses entered at the "?" prompt. The intended meaning of, for example, "?- $p(X_1, X_2)$, $q(X_2, X_3)$." is "are there objects a_1, a_2, a_3 such that $p(a_1, a_2)$ and $q(a_2, a_3)$". PROLOG responds not only by answering yes or no to this question but, if the answer is yes, by giving instances that verify it, i.e., actual terms (and so names for objects) a_1, a_2 and a_3 such that $p(a_1, a_2) \wedge q(a_2, a_3)$. (As discussed in the case of propositional logic in I.10.4, entering ";" after one answer has been found asks for another. This may be repeated until there are no more, at which point PROLOG answers "no". The search for additional answers may also be terminated after any reply by simply entering a return.)

As in the propositional case, PROLOG implements the search for such witnesses a_1, a_2 and a_3 by adding the goal clause $G = \{\neg p(X_1, X_2), \neg q(X_2, X_3)\}$ to the current program P and then deciding if the result is an unsatisfiable formula. Let us list various semantic equivalents of the resulting formula to help see how this search produces an answer to our question. First, the meaning of the clause G is $\forall X_1 \forall X_2 \forall X_3 [\neg p(X_1, X_2) \vee \neg q(X_2, X_3)]$. If adding it to the program P produces an unsatisfiable formula $P \cup \{G\}$, then its negation is a logical consequence of P (check through the definitions as we did in Lemma I.10.6). Thus

$$P \models \neg \forall X_1 \forall X_2 \forall X_3 [\neg p(X_1, X_2) \vee \neg q(X_2, X_3)].$$

As we have seen above (and in Exercise 4.5), this is equivalent to $P \models \exists X_1 \exists X_2 \exists X_3 [p(X_1, X_2) \wedge q(X_2, X_3)]$. The implementation of PROLOG tries to establish this consequence relation by producing a resolution refutation of $P \cup \{G\}$. (We will define resolution refutations for predicate calculus in §13 and Chapter III.) A by–product of the proof procedure is that it actually produces witnesses, a_1, a_2, a_3 in this case, that show that $P \cup \{G\}$ is unsatisfiable by providing a proof from P that $p(a_1, a_2) \wedge q(a_2, a_3)$. From the viewpoint of resolution theorem proving, these witnesses are a mere by–product of the proof. From the programming point of view, they are the essential result. They are the output of our program; the answers to our questions.

Because of the way PROLOG represents data via logic, there is an unusual symmetry between input and output. We can put the variables anywhere in our predicate when we ask questions. Thus the simple predicate add (X, Y, Z) not only supplies $a + b$ when we enter "?- add (a, b, Z)." it also supplies $b - a$ when we enter " ?- add (a, Z, b)." (at least if $b > a$). A single PROLOG program can thus be used to answer quite fancy questions that might be difficult to extract from a simple database. Compare asking if one can get from $\langle a, b \rangle$ to $\langle c, d \rangle$ in three knight's moves given one of the above PROLOG programs to explicitly writing such a program in some other language given only the database listing the knight's moves. The arrangement of, and orders for, searching are all done automatically. Again we will return to these points of reversibility and searching later.

Exercises

1. Verify that the symmetry rules (S1)–(S4) are legitimate. (You can do this by applying the arithmetic definition of ktmove (A0).)
2. Explain (in English) the meaning of the rules (A1)–(A2) and why they correctly represent addition on the structure $\{1, \ldots, 8\}$.
3. Explain (in English) the meaning of the rules (A3)–(A4) and why they correctly represent absolute difference on the structure $\{1, \ldots, 8\}$.
4. Suppose that suc(X, Y) were correctly defined in some way on all the natural numbers, i.e., suc(n, m) is true iff $n + 1 = m$.
   a) Do the clauses (A1)–(A2) still correctly define addition?
   b) Do the clauses (A3)–(A4) still correctly define absolute difference?

   Suppose now that suc(X, Y) defines "succeeded by" on the integers. What relations do the clauses (A1)–(A2) and (A3)–(A4) now define on the integers?
5. Suppose we switch to a language containing the constant c, a unary function symbol $s(X)$ and a ternary predicate symbol $a(X, Y, Z)$. Write a set of PROLOG clauses that will make "a" define addition in the

sense that $a(s^n(c), s^m(c), s^t(c))$ will be a consequence of the program iff $n+m=t$. ($s^n(c)$ is shorthand for $s(\ldots(s(c))\ldots)$ where there are n occurrences of s in the string of s's.)

6. Prove that every PROLOG program is satisfiable.

The following problems (and others later on) were designed to be used with a database which we supplied on line. This database consists of the genealogy given in the first few chapters of Chronicles (the last book of the Hebrew Bible). The information there is in terms of male descent only. (Actually there are bits and pieces of information on women and their children but so fragmentary as to make inclusion fairly useless. The information was recorded in the database in terms of the predicate "fatherof(a, b)". Thus the file consisted purely of (many) facts entered as follows:

fatherof(adam, seth).

fatherof(abraham, isaac).

fatherof(isaac, jacob).

fatherof(isaac, esau).

In problems 7 and 8 assume that this is the only type of information available (e.g., in defining grandfather, there is no need to consider ancestry on the mother's side as this sort of information is not available).

We provide a printout of the database as Appendix B. If the reader does not have on–line access to this database or a similar one, the following problems should be answered by just writing down a PROLOG program which is semantically correct according to the interpretations of facts and rules described in this section. Similarly descriptions of how to get the requested information from the programs will suffice.

7. Ancestors:

   a) Write a program defining "grandfatherof".

   b) Find the grandfathers of nimrod, lud and joktan.

   c) Use this program to find a grandson of noah; to find all his grandsons (use the facility which generates alternate answers by entering a semicolon after each answer is given until there are no more).

   d) Write a program defining "greatgrandfatherof".

   e) Find the great–grandfathers of shem and canaan.

   f) Use the program to find a great–grandson of abraham; to find ten of his great–grandsons.

   g) Write a program to define "ancestorof".

   h) Find three ancestors of shem.

8. Uncles:

   a) Write a program defining "uncleof".

   b) Find the uncles of nimrod, lud and joktan.

   c) Use this program to find a nephew of shem; to find all his nephews (use the facility which generates alternate answers by entering a semicolon after each answer is given until there are no more).

   d) Write a program defining "granduncleof" (recall that my grandfather's brothers are my granduncles.)

   e) Find the granduncles of shelah and canaan.

   f) Use the program to find a grandnephew of ham; to find eight of his grandnephews.

6. Proofs: Complete Systematic Tableaux

We will now describe a system for building proofs of sentences in predicate logic. As for propositional logic the proofs will be labeled binary trees called *tableaux*. The labels on the trees will be *signed sentences* (i.e., sentences preceded by T or F to indicate that, for the sake of the analysis, we are assuming them true or false respectively). We will again call these labels the *entries of the tableau*. Formally we will define tableaux for predicate logic inductively by first specifying certain (labeled binary) trees as tableaux (the so–called atomic tableaux) and then giving a development rule defining more complex tableaux from simpler ones. The intent of the proof procedure is to start with some signed sentence such as $F\alpha$ as the root of our tree and to analyze it into its components in such a way as to show that any analysis leads to a contradiction. We will then conclude that we have refuted the original assumption that α is false and so have a proof of α.

The analysis of the connectives will be the same as in propositional logic and the plan of the analysis will again be that if some sentence is correctly signed (T or F) then at least one of its immediate successors in the tree analysis is also correctly signed. The new problem is how to deal with quantifiers. If we consider, for example, $T\exists x\varphi(x)$, the obvious analysis of the assertion that there is an x such that $\varphi(x)$ is simply to supply such an x. Supplying such a witness means specifying a ground term t and asserting that $\varphi(t)$ is true. Thus, our first concern should be that there are as many ground terms available as we might ever need. If we therefore begin with any language $\mathcal{L}$, we immediately expand it to one $\mathcal{L}_C$ by adding on a set of constant symbols $c_0, c_1, c_2, \ldots$ not used in $\mathcal{L}$. Let A be any atomic sentence of $\mathcal{L}$ and α, β be any sentences of $\mathcal{L}_C$. The base case of our inductive definition of tableaux for the analysis of sentences of the language $\mathcal{L}$ starts with the following (labeled binary) trees as the *atomic tableaux*.

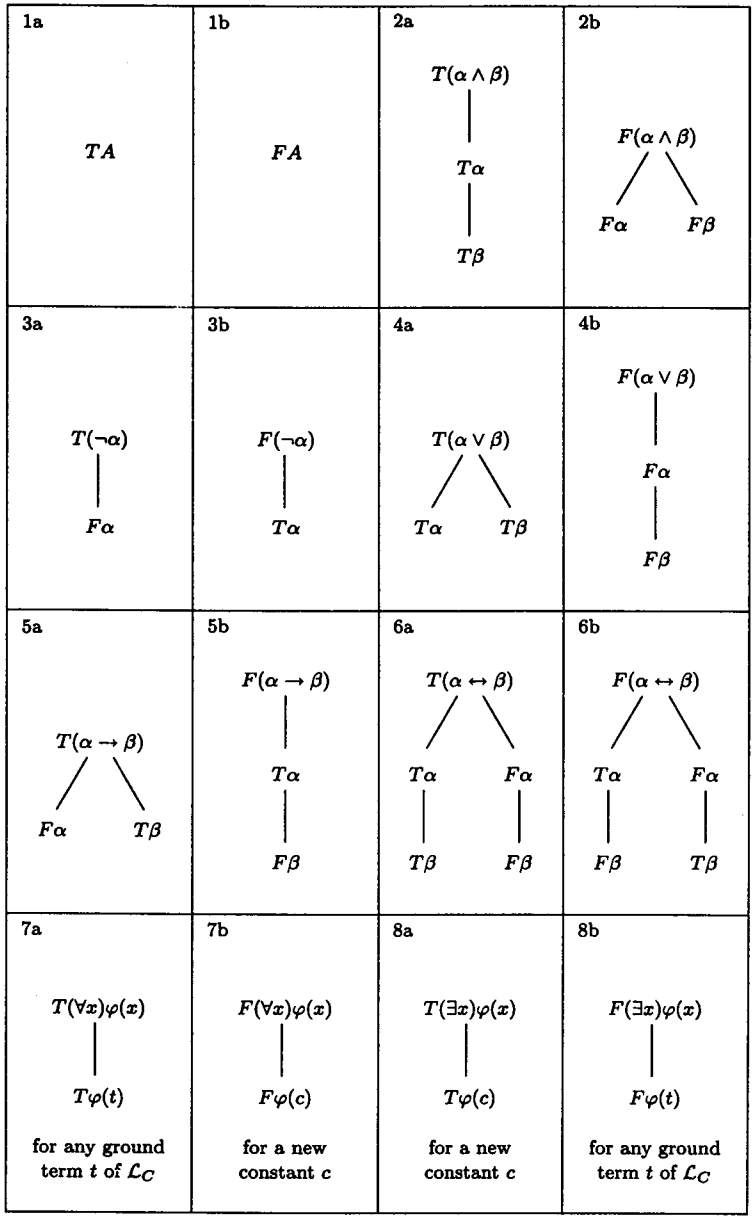


FIGURE 29

Intuitively the requirement that the constant introduced in cases 7b and 8a be "new" is easy to understand. The starting point of the tableau here is the assertion that an x with some property exists. There can be no danger in then asserting that c is such an x as long as we have no prior demands on c. On the other hand, if some other assertions have already been made about c, we have no right to assume that an element with these other properties can also be a witness for this new assertion. The precise syntactic meaning of "new" will be defined simultaneously with the inductive definition of *tableaux* as binary trees labeled with signed statements.

Definition 6.1: We define *tableaux* as binary trees labeled with signed sentences (of $\mathcal{L}_C$) called entries by induction:

(i) All atomic tableaux are tableaux. The requirement that c be new in cases 7b and 8a here simply means that c is one of the constants c_i added on to $\mathcal{L}$ to get $\mathcal{L}_C$ (which therefore does not appear in φ).

(ii) If τ is a finite tableau, P a path on τ, E an entry of τ occurring on P and τ' is obtained from τ by adjoining an atomic tableau with root entry E to τ at the end of the path P then τ' is also a tableau. Here the requirement that c be new in cases 7b and 8a means that it is one of the c_i which do not appear in any entries on P. [In actual practice it is simpler in terms of bookkeeping to choose one not appearing at any node of τ.]

(iii) If τ_0 is a finite tableau and $\tau_0, \tau_1, \ldots, \tau_n, \ldots$ is a sequence of tableaux such that for every $n \geq 0$, τ_{n+1} is constructed from τ_n by an application of (ii) then $\tau = \cup\tau_n$ is also a tableau.

Warning: It is crucial in the setting of predicate logic that the entry E in clause (ii) be repeated when the corresponding atomic tableau is added on to P (at least in cases 7a and 8b). The reason for this will become apparent once we analyze the action needed in these cases and the resulting definition of a finished tableau (Definition 6.7).

We would next like to define tableau proofs of sentences in predicate logic. It is important to realize, however, that in most situations one does not simply prove a sentence outright. One normally proves something based on various assumptions or axioms. The semantic aspect of this procedure was embodied in the notion of logical consequence in Definition 4.4. To capture the corresponding proof theoretic notion we need to define tableaux and proofs from premises for predicate logic analogous to the ones presented in I.6 for propositional logic. The modifications needed are like those incorporated in the definitions of I.6 for propositional logic. The key change is in the definition of a tableau from a set of sentences S. The underlying idea is that we are assuming that every sentence in S is true. Thus, in addition to the formation rules for ordinary tableaux, we may assert at any time that any sentence in S is true. We accomplish this by adding on one new formation rule for tableaux from S.

For the remainder of this section we let S be a set of sentences in the language $\mathcal{L}$. We often refer to the elements of S as *premises*.

Definition 6.1 (Continued): *Tableaux from S*. The definition for tableaux from S is the same as for ordinary tableaux except that we include an additional formation rule

(ii′) If τ is a finite tableau from S, φ a sentence from S, P a path on τ and τ' is obtained from τ by adjoining $T\varphi$ to the end of the path P then τ' is also a tableau from S.

From now on we will define our notions for tableaux from S simultaneously with the ones for ordinary tableaux. The additional clauses pertaining to tableaux from S are parenthesized, as in the following important observation.

Note: It is clear from the definition that every tableau τ (from S) is the union of a finite or infinite sequence $\tau_0, \tau_1, \ldots, \tau_n, \ldots$ of tableaux (from S) in which τ_0 is an atomic tableau and each τ_{n+1} is gotten from τ_n by an application of (ii) (or (ii′)). From now on, we will always assume that every tableau (from S) is presented as such a union.

Definition 6.2: *Tableau proofs (from S)*: Let τ be a tableau and P a path in τ.

(i) P is *contradictory* if, for some sentence α, $T\alpha$ and $F\alpha$ both appear as labels of nodes of P.

(ii) τ is *contradictory* if every path on τ is contradictory.

(iii) τ is a *proof of* α *(from S)* if τ is a finite contradictory tableau (from S) with its root node labeled $F\alpha$. If there is proof τ of α (from S), we say α is provable (from S) and write $\vdash \alpha$ ($S \vdash \alpha$).

(iv) S is *inconsistent* if there is a proof of $\alpha \wedge \neg\alpha$ from S for some sentence α.

Note that, if there is any contradictory tableau (from S) with root node $F\alpha$, then there is one which is finite, i.e., a proof of α (from S). Just terminate each path when it becomes contradictory. As each path is now finite, the whole tree is finite by König's lemma. Thus, the added requirement that proofs be finite tableaux has no affect on the existence of proofs for any sentence. Another way of looking at this is that we could have required the path P in clause (ii) of the definition of tableaux (Definition 6.1) to be noncontradictory without affecting the existence of proofs.

Before describing the appropriate version of finished tableaux and the construction of complete systematic tableaux, it is instructive to look at some examples of proofs by tableaux in predicate logic. Note that we again abbreviate the tableaux by not repeating the entry being analyzed (or developed) unless we are dealing with either case 7a or 8b of the atomic tableaux.

Example 6.3: Suppose we want to check the validity of the formula $((\forall x)\varphi(x) \rightarrow (\exists x)\varphi(x))$. We form the following tableau:

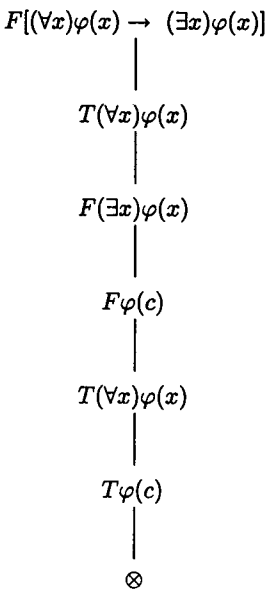


FIGURE 30

For the last entry we chose to use the same constant c as in the previous line so as to get the desired contradiction. We were able to do so because the atomic tableau for $\forall x\varphi(x)$ allows us to use *any* constant.

The next example also yields a contradictory tableau.

Example 6.4: See Figure 31.

In practice, it will generally prove more efficient to extend a tableau by first expanding the atomic tableaux which require the introduction of new terms and to then turn to those for which any ground term can be used.

Example 6.5: See Figure 32.

The atomic tableaux for $T(\forall x)\varphi(x)$ and $F(\exists x)\varphi(x)$ tell us that we can declare $\varphi(t)$ true or false, respectively, for any ground term t. On the other hand, the atomic tableau for $T(\exists x)\varphi(x)$ allows us to declare $\varphi(t)$ true only for one of the constants c_i which have not appeared so far in the tableau. The following example shows how we can get into trouble if we do not obey this proviso.

Example 6.6: Reverse the implication in Example 6.3 to get the sentence $((\exists x)\varphi(x) \rightarrow (\forall x)\varphi(x))$ which is not valid. If, however, we violate the provisions for using new constants, we can produce a "proof" of this sentence, as in Figure 33.

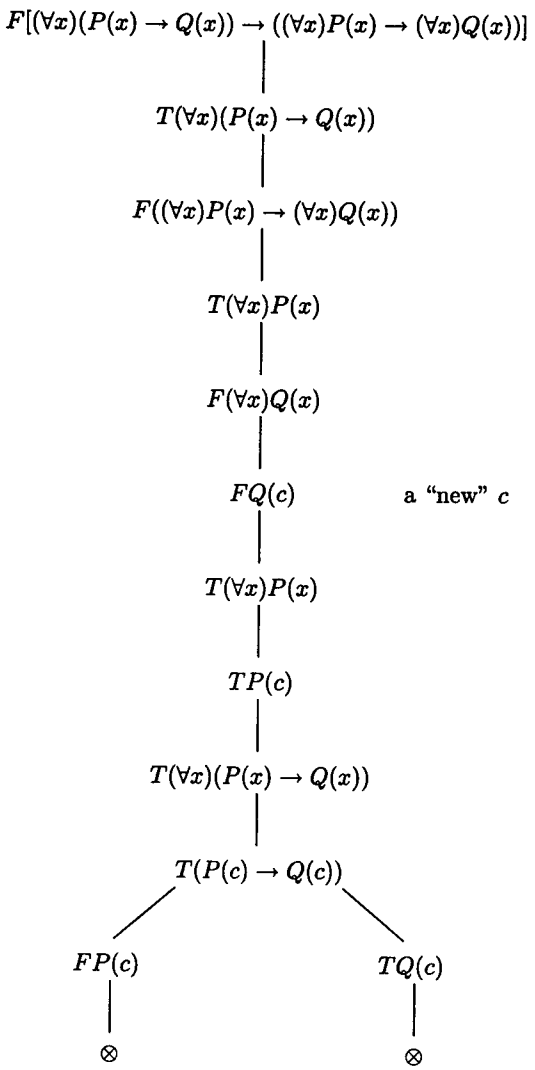


FIGURE 31

It is easy to see that tableaux in predicate logic need never terminate if no contradiction comes up. Thus, there is some question as to when we should say that an entry has been reduced and when a tableau is finished. To motivate these definitions, we first consider the role of the atomic tableaux for the quantifiers and how we use them in building tableaux. When we deal with $T(\exists x)\varphi(x)$ (or $F(\forall x)\varphi(x)$), we analyze it simply by listing $T\varphi(c)$ (or $F\varphi(c)$) for some constant c not yet appearing along the path being extended. The original sentence $(\exists x)\varphi(x)$ contains no more information

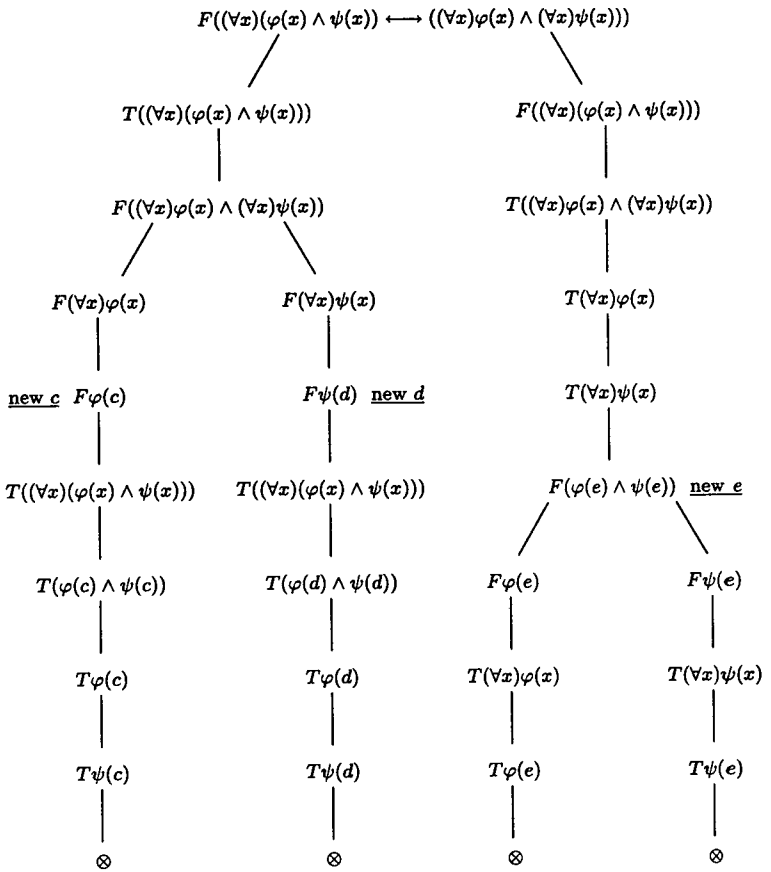


FIGURE 32

than the new one $\varphi(c)$ and so we may reasonably claim to have finished with it. On the other hand, if we are dealing with $T(\forall x)\varphi(x)$ (or $F(\exists x)\varphi(x)$) the situation is quite different. Here we may add $T\varphi(t)$ (or $F\varphi(t)$) to our tableau for any ground term t. This, however, far from exhausts the information in the original sentence. It merely gives us one instance of the universal fact asserted by $T(\forall x)\varphi(x)$. Thus, we cannot say that we have as yet finished with $T(\forall x)\varphi(x)$. With this distinction in mind we can define the notion of when an entry of a tableau has been reduced and when a tableau is finished. As in the propositional case, our goal is to describe a systematic procedure to produce a tableau proof (from S) of a given sentence φ. That this systematic procedure will always succeed if φ is valid (a logical consequence of S) will be the content of the Completeness Theorem (Theorem 7.7).

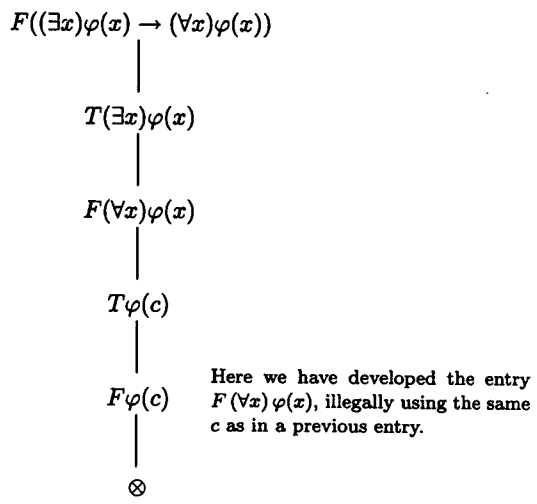


FIGURE 33

Let $t_1, \ldots, t_n, \ldots$ be a list of all the ground terms of our language $\mathcal{L}_C$ which, we recall, includes the new constants c_i.

Definition 6.7: Let $\tau = \cup\tau_n$ be a tableau (from S), P a path in τ, E an entry on P and w the i^{th} occurrence of E on P (i.e., the i^{th} node on P labeled with E).

(i) w is *reduced on* P if

(1) E is neither of the form $T(\forall x)\varphi(x)$ nor $F(\exists x)\varphi(x)$ and, for some j, τ_{j+1} is gotten from τ_j by an application of rule (ii) of Definition 6.1 to E and a path on τ_j which is an initial segment of P. [In this case we say that E occurs on P as the root entry of an atomic tableau.]

or

(2) E is of the form $T(\forall x)\varphi(x)$ or $F(\exists x)\varphi(x)$, $T\varphi(t_i)$ is an entry on P and there is an $j+1^{\text{st}}$ occurrence of E on P.

(ii) τ is *finished* if every occurrence of every entry on τ is reduced on every noncontradictory path containing it (and $T\varphi$ appears on every noncontradictory path of τ for every φ in S). It is *unfinished* otherwise.

The idea here is that signed sentences such as $T(\forall x)\varphi(x)$ must be instantiated for each term t_i in our language before we can say that we have finished with them. We can now show that there is a finished tableau (from S) with any given entry on its root node by constructing the appropriate complete systematic tableau (from S). The plan is to devise an

ordering procedure so that we can reduce each entry in turn to produce the finished tableau. We employ a variant on the lexicographic ordering on the nodes of the tableau.

Definition 6.8: Suppose T is a tree with a left–right ordering on the nodes at each of its levels. Recall (from I.1) that if T is, for example, a tree of binary sequences, the left–right ordering is given by the usual lexicographic ordering. We define the *level–lexicographic ordering* $\leq_{LL}$ on the nodes ν, μ of T as follows:

$\nu \leq_{LL} \mu \Leftrightarrow$ the level of ν in T is less than that of μ or ν and μ are on the same level of T and ν is to the left of μ.

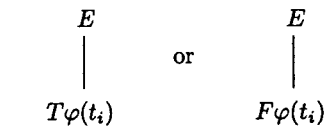
Definition 6.9: We construct the CST, the *complete systematic tableau*, with any given signed sentence as the label of its root, by induction.

(i) We begin with τ_0 an atomic tableau with root the given signed sentence. This atomic tableau is uniquely specified by requiring that in cases 7a and 8b we use the term t_1 and that in cases 7b and 8a we use c_i for the least allowable i.

At stage n, we have, by induction, a tableau τ_n which we extend to one τ_{n+1}. As τ_n is a (finite, labeled) binary tree the level–lexicographic ordering is defined as above on its nodes. If every occurrence of every entry on T is reduced, we terminate the construction. Otherwise, let w be the level–lexicographically least node of τ_n which contains an occurrence of an entry E which is unreduced on some noncontradictory path P of τ_m. We now proceed according to one of the following two cases:

(ii) If E is not of the form $T(\forall x)\varphi(x)$ or $F(\exists x)\varphi(x)$, we adjoin the atomic tableau with apex E to the end of every noncontradictory path in τ that contains w. For E of the form $T(\exists x)\varphi(x)$ or $F(\forall x)\varphi(x)$, we use the least constant c_j not yet appearing in the tableau.

(iii) If E is of the form $T(\forall x)\varphi(x)$ or $F(\exists x)\varphi(x)$ and w is the i^{th} occurrence of E on P we adjoin



respectively, to the end of every noncontradictory path in τ containing w.

The CST from a set of premises S with a given root is defined like the ordinary CST above with one change to introduce the elements of S. At even stages ($n = 2k$) we proceed as in (i), (ii) and (iii) above. At odd stages ($n = 2k+1$) we adjoin $T\alpha_k$ for α_k the k^{th} element of S to every noncontradictory path in τ_n to get τ_{n+1}. We do not terminate the construction of the CST from S unless all elements of S have been put on every noncontradictory path in this way and every occurrence of every entry is reduced on every path containing it.

Note that, in general, a CST will be an infinite tableau (even if S is finite). The crucial point is that it is always a finished tableau.

Proposition 6.10: *Every* CST *is finished.*

Proof: Consider any unreduced occurrence w of an entry E in $\tau_k \subseteq \tau$ which is on a noncontradictory path P of the given CST τ. (If there is none, τ is finished by definition.) Suppose there are n nodes of T which are level–lexicographically less than w. It is clear from the definition of the CST that we must reduce w on P by the time we form τ_{k+n+1}. Thus, every occurrence of each entry on a noncontradictory path in τ is reduced as required.

If we consider the CST from S, the same considerations apply to show that every entry is reduced on every path. (It just takes twice as many steps to get there.) The procedure of adding on the k^{th} member of S at stage $2k+1$ guarantees that every element of S is put on every path of the CST from S. It is therefore a finished tableau from S. □

Example 6.11: Figure 34 gives an example of a finished tableau.

Exercises

In exercises 1–11, let φ and ψ be any formulas either with no free variables or with only x free as appropriate. Give tableau proofs of each of the following.

1. $(\exists x)(\varphi(x) \vee \psi(x)) \leftrightarrow (\exists x)\varphi(x) \vee (\exists x)\psi(x)$.
2. $(\forall x)(\varphi(x) \wedge \psi(x)) \leftrightarrow (\forall x)\varphi(x) \wedge (\forall x)\psi(x)$.
3. $(\varphi \vee (\forall x)\psi(x)) \rightarrow (\forall x)(\varphi \vee \psi(x))$, x not free in φ.
4. $(\varphi \wedge (\exists x)\psi(x)) \rightarrow (\exists x)(\varphi \wedge \psi(x))$, x not free in φ.
5. $(\exists x)(\varphi \rightarrow \psi(x)) \rightarrow (\varphi \rightarrow (\exists x)\psi(x))$, x not free in φ.
6. $(\exists x)(\varphi \wedge \psi(x)) \rightarrow (\varphi \wedge (\exists x)\psi(x))$, x not free in φ.
7. $\neg(\exists x)\varphi(x) \rightarrow (\forall x)\neg\varphi(x)$.
8. $(\forall x)\neg\varphi(x) \rightarrow \neg(\exists x)\varphi(x)$.
9. $(\exists x)\neg\varphi(x) \rightarrow \neg(\forall x)\varphi(x)$.
10. $(\exists x)(\varphi(x) \rightarrow \psi) \rightarrow ((\forall x)\varphi(x) \rightarrow \psi)$, x not free in ψ.
11. $((\exists x)\varphi(x) \rightarrow \psi) \rightarrow (\forall x)(\varphi(x) \rightarrow \psi)$, x not free in ψ.
12. Let φ and ψ be any formulas with free variables x, y and z; let w be any variable not appearing in φ or ψ. Give tableau proofs of the following.
    a) $\forall x\exists y\neg\forall z\ \varphi(x,y,z) \leftrightarrow \forall x\exists y\exists z\neg\varphi(x,y,z)$.
    b) $\exists x\forall y(\forall z\varphi \vee \psi) \leftrightarrow \exists x\forall y\forall w(\varphi(z/w) \vee \psi)$.
    c) $\forall x\exists y(\varphi \vee \exists z\psi) \leftrightarrow \forall x\exists y\exists w(\varphi \vee \psi(z/w))$.
    d) $\forall x\exists y(\varphi \rightarrow \forall z\psi(z)) \rightarrow \forall x\exists y\forall w(\varphi \rightarrow \psi(z/w))$.

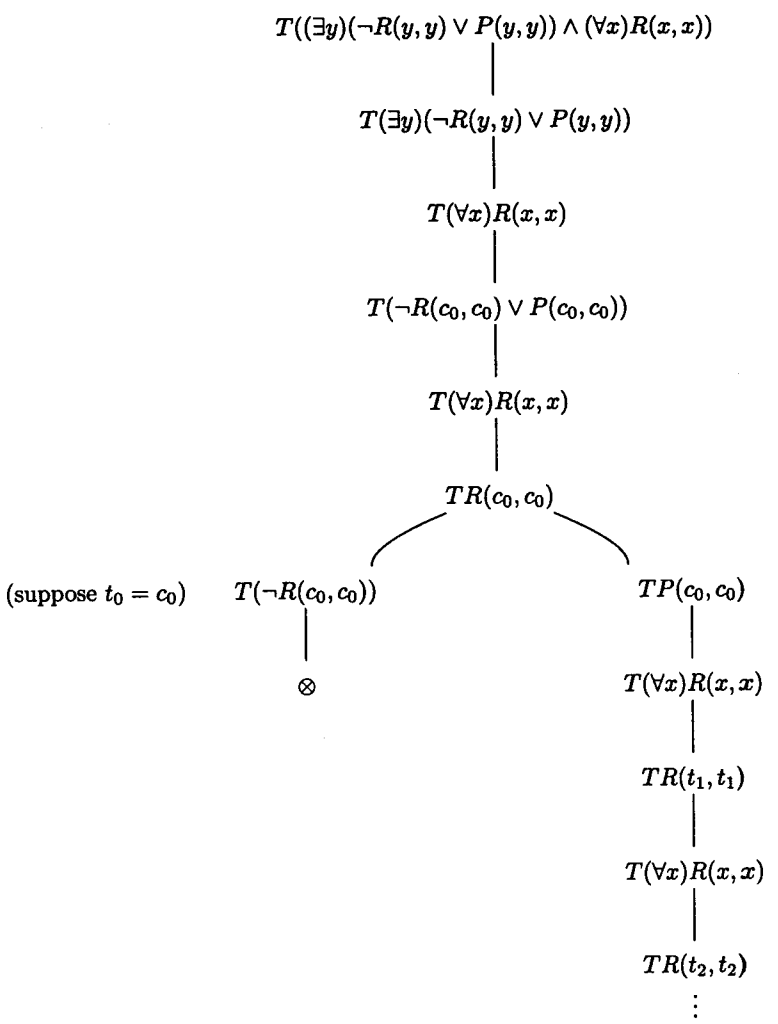


FIGURE 34

13. **Theorem on Constants:** Let $\varphi(x_1,\ldots,x_n)$ be a formula in a language $\mathcal{L}$ with all free variables displayed and let $c_1,\ldots,c_n$ be constant symbols not in $\mathcal{L}$. Show that $\forall x_1 \ldots \forall x_n\varphi(x_1,\ldots,x_n)$ is tableau provable iff $\varphi(c_1,\ldots,c_n)$ is. Argue syntactically to show that, given a proof of one of the formulas, one can construct a proof of the other. (You may assume the proof is given by the CST procedure.)
14. If the left–right orderings on each level of T is a well–ordering (i.e., every subset has a least element), then the ordering $\leq_{LL}$ is a well–ordering of the nodes of T.

7. Soundness and Completeness of Tableau Proofs

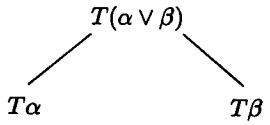
We can now exploit the complete systematic tableaux to prove the basic theorems about predicate logic and provability: Soundness, Completeness and Compactness. We begin with the soundness of proofs by tableaux in predicate logic. Throughout this section, $\mathcal{L}$ is a fixed language of predicate calculus and S is a set of sentences in $\mathcal{L}$. The case of pure tableaux, i.e., with no set S of premises, is simply the case $S = \emptyset$. This remark applies to all the results of this section and so we will deal only with the case of proofs and tableaux from S. The root nodes of our tableaux will also be taken from $\mathcal{L}$.

Lemma 7.1: *If $\tau = \cup\tau_n$ is a tableau from a set of sentences S with root $F\alpha$, then any $\mathcal{L}$-structure $\mathcal{A}$ which is a model of $S \cup \{\neg\alpha\}$ can be expanded to one agreeing with every entry on some path P through τ. (Recall that $\mathcal{A}$ agrees with $T\alpha$ ($F\alpha$) if α is true (false) in $\mathcal{A}$.)*

Proof: The only expansion of $\mathcal{A}$ that is necessary to make it a structure for all the sentences appearing in τ is to define $c_i^{\mathcal{A}}$ for the constants c_i in $\mathcal{L}_C - \mathcal{L}$ appearing on P. (Remember, these are the constants used in τ as the "new" constants in instantiations.)

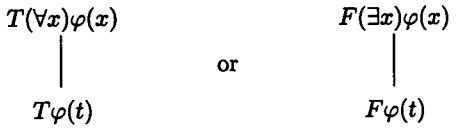
We define P and $c_i^{\mathcal{A}}$ by an induction on the sequence τ_n giving the construction of τ. At each step n we will have a path P_n through τ_n and an extension $\mathcal{A}_n$ of $\mathcal{A}$ (with the same domain) which interprets all the c_i on P_n and agrees with P_n. This clearly suffices to prove the lemma. When τ_{n+1} is gotten from τ_n by extending some path other than P_n we need make no changes in P_n or $\mathcal{A}_n$. Suppose then that τ_{n+1} is gotten by adding on to the end of P_n either an atomic tableau with root E an entry on P_n or an element α_k of S. In the latter case we extend P_n in the only way possible by attaching α_k to its end. No extension of $\mathcal{A}_n$ is necessary and it agrees with α_k (and hence P_{n+1}) by hypothesis. We consider then the case of extending τ_n by adding on an atomic tableau τ' with root E. By induction we may assume that $\mathcal{A}_n$ agrees with E. We wish to extend $\mathcal{A}_n$ to $\mathcal{A}_{n+1}$and find a path P_{n+1} extending P_n through τ_{n+1} agreeing with $\mathcal{A}_{n+1}$. (The base case of our induction is the atomic tableau τ_0 whose root $F\alpha$ agrees with $\mathcal{A}$ by hypothesis. The analysis of the base case is then exactly as in the inductive step: We wish to extend $\mathcal{A}$ to $\mathcal{A}_0$ and find a path P_0 through τ_0 agreeing with $\mathcal{A}_0$.) We consider each type of atomic tableau τ'.

(i) The situation for the propositional connectives is the same as in the proof of soundness for propositional logic (Lemma I.5.4). In particular, no extension of $\mathcal{A}_n$ is necessary. If, for example, we added on



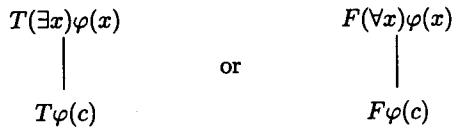
then we know by induction that $\mathcal{A}_n \vDash \alpha \vee \beta$ and so $\mathcal{A}_n \vDash \alpha$ or $\mathcal{A}_n \vDash \beta$. We choose to extend P_n accordingly. The analysis for the other propositional connectives is left as Exercise 7.

(ii) If we added on



we again have no problem. $\mathcal{A}_n \vDash \forall x\varphi(x)$ (or $\mathcal{A}_n \vDash \neg\exists x\varphi(x)$) and so $\mathcal{A}_n \vDash \varphi(t)$ ($\mathcal{A}_n \vDash \neg\varphi(t)$). (Note that if $t^{\mathcal{A}}$ is not yet defined by our inductive procedure we can now define it arbitrarily and still maintain our inductive hypothesis as we know that $\mathcal{A}_n \vDash \forall x\varphi(x)$ (or $\mathcal{A}_n \vDash \neg\exists x\varphi(x)$).

(iii) Finally, if we added on



for some *new constant* symbol c (i.e., one not appearing either in S or in an entry on P_n), we must define $c^{\mathcal{A}}$. By induction, we know that $\mathcal{A}_n \vDash \exists x\varphi(x)(\mathcal{A}_n \vDash \neg\forall x\varphi(x))$ and so we may choose an element $a \in A$ $(= A_n$ by construction) such that, if we expand $\mathcal{A}_n$ to $\mathcal{A}_{n+1}$ by letting $c^{\mathcal{A}} = a$, we have $\mathcal{A}_{n+1} \vDash \varphi(c)(\mathcal{A}_{n+1} \vDash \neg\varphi(c))$ as required. □

Theorem 7.2 (Soundness)**:** *If there is a tableau proof τ of α from S, then $S \vDash \alpha$.*

Proof: If not, then there is a structure $\mathcal{A} \vDash \neg\alpha$ in which every α_k in S is true. Lemma 7.1 then tells us that there is a path P through τ and an expansion $\mathcal{A}'$ of $\mathcal{A}$ which agrees with every node on P. As P is contradictory by assumption, we have our desired contradiction. □

We now turn to the completeness of the tableau method of proof for predicate logic. As in the propositional case (Theorem I.5.3 and especially Lemma I.5.4) the plan is to use a noncontradictory path in a CST to build a structure for $\mathcal{L}_C$ which agrees with every entry on P. The underlying idea here is to build the desired structure out of the only available materials — the syntactic objects, in particular, the ground terms appearing on the path. This idea and its application in the proof of the completeness theorem will be crucial ingredients in the proofs of many other important results including Herbrand's theorem (Theorem 10.4) and the Skolem–Löwenheim theorem (Theorem 7.7).

Theorem 7.3: *Suppose P is a noncontradictory path through a complete systematic tableau τ from S with root $F\alpha$. There is then a structure $\mathcal{A}$ in which α is false and every sentence in S is true.*

Proof: Let the domain of this structure be the set A of ground terms t_i on the master list of ground terms of our expanded language $\mathcal{L}_C$. We define the functions $f^{\mathcal{A}}$ associated with the n–ary function symbols f of our language in the natural way corresponding to the syntax of $\mathcal{L}_C$:

$$f^{\mathcal{A}}(t_{i_1}, t_{i_2}, \dots, t_{i_n}) = f(t_{i_1}, \dots, t_{i_n}).$$

Remember that the elements of our structure are the ground terms and so the t_i appearing on the left–hand side of this equation are being viewed as elements of our structure to which we apply the function $f^{\mathcal{A}}$. On the right–hand side we have another term, and so an element of our structure, which we declare to be the value of this function. If R is an n–ary predicate letter, we define $R^{\mathcal{A}}$ as dictated by the path P:

$$R^{\mathcal{A}}(t_{i_1}, \dots, t_{i_n}) \Leftrightarrow TR(t_{i_1}, \dots, t_{i_n}) \text{ is an entry on } P.$$

We now prove the theorem by establishing a slightly stronger assertion by induction.

Lemma 7.4: *Let the notation be as above.*

(i) *If $F\beta$ occurs on P, then β is false in $\mathcal{A}$.*

(ii) *If $T\beta$ occurs on P, then β is true in $\mathcal{A}$.*

Proof: First recall that, by Proposition 6.10, every occurrence of every entry on P is reduced on P. We now proceed by induction on the depth of β (more precisely, on the depth of the associated auxiliary formation tree as given in Definition 3.8).

(i) If β is an atomic sentence, then β is of the form $R(t_{i_1}, \dots, t_{i_n})$. If $T\beta$ occurs on P then $R^{\mathcal{A}}$ has been declared true of $t_{i_1}, \dots, t_{i_n}$. If $F\beta$ occurs on P then, as P is noncontradictory, $T\beta$ does not occur on P and $R^{\mathcal{A}}$ has been declared false of $t_{i_1}, \dots, t_{i_n}$.

(ii) Suppose β is built using a connective, e.g., β is $(\beta_1 \vee \beta_2)$. As τ is finished, we know that if $T\beta$ occurs on P, then either $T\beta_1$ or $T\beta_2$ occurs on P. By induction hypothesis if $T\beta_1$ occurs on P then β_1 is true in $\mathcal{A}$ (and similarly for β_2). Thus, one of β_1, β_2 is true so $(\beta_1 \vee \beta_2)$ is true in $\mathcal{A}$ (by the inductive definition of truth). On the other hand, if $F(\beta_1 \vee \beta_2)$ appears on P, then we know that both $F\beta_1$ and $F\beta_2$ appear on P. Our inductive hypothesis then tells us that both β_1 and β_2 are false in $\mathcal{A}$. We then have that $(\beta_1 \vee \beta_2)$ is false in $\mathcal{A}$ as required. The cases for the other connectives are similar and are left as Exercise 8.

(iii) Suppose β is of the form $(\forall v)\varphi(v)$. If w is the i^{th} occurrence of $T((\forall v)\varphi(v))$ on P, then $T\varphi(t_i)$ occurs on P and there is an $i{+}1^{\text{st}}$ occurrence of $T((\forall x)\varphi(x))$ on P. Thus, if $T((\forall v)\varphi(v))$ appears on P, then $\varphi(t)$ appears on P for every ground term t. As the depth of $\varphi(t)$ is less than that of $(\forall v)\varphi(v)$, the inductive hypothesis tells us that $\varphi(t)$ is true in $\mathcal{A}$ for every ground term t. As these terms constitute the universe of our structure $\mathcal{A}$, $(\forall v)\varphi(v)$ is true in $\mathcal{A}$ as required.

If $F(\forall v)\varphi(v)$ occurs on P then, again as τ is finished, $F\varphi(t)$ occurs on P for some t. By induction hypothesis $\varphi(t)$ is false in $\mathcal{A}$. So $(\forall v)\varphi(v)$ is false in $\mathcal{A}$.

(iv) The case for the existential quantifier $\exists v\varphi(v)$ is similar and is left as Exercise 9. □

This also completes the proof of Theorem 7.3. □

We now specialize our general remarks on the finiteness of proofs to complete systematic tableaux.

Proposition 7.5: *If every path of a complete systematic tableau is contradictory, then it is a finite tableau.*

Proof: By construction, we never extend a path on a CST once it is contradictory. Thus, every contradictory path on a CST is finite. The theorem then follows from König's lemma (Theorem I.1.4). □

We have thus proven an effective version of the completeness theorem. For any given sentence α and any set of sentences S, we can produce either a proof that α is logical consequence of S or a model of S in which α fails.

Corollary 7.6: *For every sentence α and set of sentences S of $\mathcal{L}$, either*

(i) *the CST from S with root $F\alpha$ is a tableau proof of α from S*

or

(ii) *there is a noncontradictory branch through the complete systematic tableau which yields a structure in which α is false and every element of S is true.*

As the path in (ii) of Corollary 7.6 is countable (i.e., there is a one–one correspondence between its symbols (and hence its terms and formulas) and a subset of the natural numbers), so is the structure associated with it. We have thus also proven the Skolem–Löwenheim theorem. □

Theorem 7.7 (Skolem–Löwenheim): *If a countable set of sentences S is satisfiable (that is, it has some model) then it has a countable model.*

Proof: Consider the CST from S that starts with a contradiction $\alpha \wedge \neg\alpha$ at its root. By the soundness theorem (Theorem 7.2) it cannot be a tableau proof of $\alpha \wedge \neg\alpha$ from S. Thus, it must have a noncontradictory path P. As there are only countably many ground terms in $\mathcal{L}_C$, the structure defined in the proof of Theorem 7.4 is the desired countable model of S. □

The analogous theorem can also be proved for arbitrary cardinalities. Also note that we use countable in the sense of at most countable, that is the model may be finite. In our setting, however, one can always guarantee that the model is infinite (Exercise 3). One can guarantee that a set of sentences has only finite models only by the special treatment of equality which we consider in III.5. Ignoring the remarks on PROLOG, this treatment of equality can be read at this point.

We can reformulate Corollary 7.6, in analogy with the completeness and soundness theorems for propositional calculus, in terms of the equivalences between provability and logical consequence. The point to keep in mind is that, if α is false in some model for S, then it cannot be a logical consequence of S.

Theorem 7.8: (Completeness and Soundness)*:*

(i) *α is tableau provable from $S \Leftrightarrow \alpha$ is a logical consequence of S.*

(ii) *If we take α to be any contradiction such as $\beta \wedge \neg\beta$ in (i) we see that S is inconsistent if and only if S is unsatisfiable.* □

The compactness theorem for predicate logic is also a consequence of these results.

Theorem 7.9 (Compactness)**:** *Let $S = \{\alpha_1, \alpha_2, \dots\}$ be a set of sentences of predicate logic. S is satisfiable if and only if every finite subset of S is satisfiable.*

Proof: The only if direction is immediate. For the if direction consider the CST from S with root entry $F(\alpha \wedge \neg\alpha)$. If the CST is contradictory it is finite by Proposition 7.5. If it is infinite, it has a noncontradictory path and so by Corollary 7.6 there is a structure in which every α_i is true. If it is contradictory and finite then $\alpha \wedge \neg\alpha$ is a logical consequence of the finite subset of S whose elements are those appearing on this tableau. This finite subset can have no model as $\alpha \wedge \neg\alpha$ has no model. □

We should point out one important difference between the completeness proofs for predicate and propositional logic. The finished tableaux for propositional logic were always finite and so for every proposition α we can effectively decide if it is valid or actually produce a counterexample. For predicate logic, if a given sentence φ is valid we eventually find a proof. On the other hand, if it is not valid, the finished tableau and the path providing a counterexample may well be infinite. Thus we may never in the course of our construction actually know that φ is not valid. This phenomena is unavoidable. Church's theorem states that there is no effective method for deciding if a given sentence in the predicate calculus is valid. We prove this result in Corollary III.8.10 as a corollary to a result on termination of PROLOG programs. A proof suitable for insertion at this point can, however, be based on the semantic approach indicated in Exercise III.8.3.

Exercises

1. Give a semantic proof of Exercise 6.13: Let $\varphi(x_1, \dots, x_n)$ be a formula in a language $\mathcal{L}$ with all free variables displayed and let $c_1, \dots, c_n$ be constant symbols not in $\mathcal{L}$. Show that $\forall x_1 \dots \forall x_n \varphi(x_1, \dots, x_n)$ is tableau provable iff $\varphi(c_1, \dots, c_n)$ is by showing that $\varphi(x_1, \dots, x_n)$ is valid iff $\varphi(c_1, \dots, c_n)$ is valid. Now apply the completeness theorem.
2. Let $\mathcal{L}$ be any language which includes a binary relation symbol $\leq$ and S any set of sentences of $\mathcal{L}$ which has an infinite model and includes the axioms for linear orderings:
   (i) $(x \leq y) \vee (y \leq x)$ and
   (ii) $(x \leq y) \wedge (y \leq z) \rightarrow (x \leq z)$.
   (a) Show that there is a model $\mathcal{M}$ for S with an infinite descending chain, that is, one in which there are elements $c_0, c_1, \dots$ such that $\dots \leq c_{n+1} \leq c_n \leq \dots \leq c_0$.
   (b) How can you also guarantee that $c_i \neq c_j$ for $i \neq j$?
   (This problem shows that the notion of well ordering is not definable in predicate logic.)
3. Let $\mathcal{L}$ be any language for predicate logic and S be any set of sentences in $\mathcal{L}$. Prove that S is satisfiable iff it has an infinite model.
4. Let $\mathcal{L}$ be a language for arithmetic on the natural numbers $\mathbb{N}$ (= {0, 1, 2, ...}) including 0, 1, +, · and >. Let $Th(\mathbb{N})$ be the set of all sentences of $\mathcal{L}$ true in $\mathbb{N}$. Show that there is a nonstandard model of $Th(\mathbb{N})$, i.e., a structure $\mathcal{M}$ for $\mathcal{L}$ in which every sentence of $Th(\mathbb{N})$ is true but in which there is an element c greater than every $n \in \mathbb{N}$.
5. Reconsider the applications of compactness in the exercises for propositional logic. Use predicate logic to give considerably simpler proofs of Exercises I.6.7 and I.6.8. (Note that the planarity of a graph G is expressible in predicate logic since by a theorem of Kuratowski it is equivalent to two specific finite graphs not being subgraphs of G.)
6. **Deduction Theorem**: Let Σ be a finite set of sentences in a language $\mathcal{L}$ and $\wedge\Sigma$ the conjunction of its members. Prove that, for any sentence φ of $\mathcal{L}$, the following are equivalent:
   (i) $\Sigma \models \varphi$.
   (ii) $\models \wedge\Sigma \rightarrow \varphi$.
   (iii) $\Sigma \vdash \varphi$.
   (iv) $\vdash \wedge\Sigma \rightarrow \varphi$.
7. Complete the proof case (i) of Lemma 7.1 by describing the required extensions of P_n for the other propositional connectives.
8. Complete the proof case (ii) of Lemma 7.4 by handling the other propositional connectives.
9. Complete the proof of case (iv) of Lemma 7.4 by considering the case that β is $\exists v \varphi(v)$.

10. Let $\mathcal{L}$ be a language with no function symbols. Describe a procedure which, given any sentence ψ of the form $\forall x_1 \dots \forall x_n \exists y_1 \dots \exists y_m \varphi$, with φ quantifier free, decides if ψ is valid. (Hint: First use Exercise 6.13 to reduce the validity of ψ to that of $\exists y_1 \dots \exists y_m \varphi(c_1, \dots, c_n, y_1, \dots, y_m)$ for new constants $c_1, \dots, c_n$. Consider all formulas of the form $\varphi(c_1, \dots, c_n, d_1, \dots, d_m)$ where $d_i \in \{c_1, \dots, c_n\}$. Apply the ideas of Theorem 4.8 and the methods of Chapter I to decide the validity of each of these sentences. If one is provable then so is φ. If none are provable, argue that $\neg\varphi$ has a model.)

11. Let R be a binary relation symbol, and let $R^{\mathcal{A}}$ be its interpretation in a structure $\mathcal{A}$. The *transitive closure* of $R^{\mathcal{A}}$ is the set of all pairs (a, b) for which there exists a finite $R^{\mathcal{A}}$–path from a to b, i.e., a sequence $a_0, a_1, \dots, a_n$, $n \geq 1$, of elements of $\mathcal{A}$ with $a_0 = a$, $a_n = b$, and $R^{\mathcal{A}}(a_i, a_{i+1})$, $0 \leq i < n$. Show that transitive closure is not first–order definable; i.e., show that there does not exist a formula $TC(x, y)$ of predicate logic such that for all structures $\mathcal{A}$ and $a, b \in \mathcal{A}$, $\mathcal{A} \models TC(a, b)$ if and only if (a, b) is in the transitive closure of $R^{\mathcal{A}}$. (Hint: Define the formulas $\rho_n(x, y)$ inductively by:

$$\rho_1(x, y) \equiv R(x, y)$$
$$\rho_{n+1}(x, y) \equiv \exists z(R(x, z) \wedge \rho_n(z, y)).$$

Show that in any structure $\mathcal{A}$, the pair (a, b) is in the transitive closure of $R^{\mathcal{A}}$ iff $\mathcal{A} \models \rho_n(a, b)$ for some n. Suppose there were such a formula $TC(x, y)$ expressing the transitive closure of R. Consider the infinite set of sentences

$$\{TC(a, b)\} \cup \{\neg\rho_n(a, b) \mid n \geq 1\}.$$

Obtain a contradiction using the compactness of predicate logic.)

8*. An Axiomatic Approach

As for the propositional logic we give a brief sketch of a classical approach to predicate logic via axioms and rules. For the sake of brevity, we use as propositional connectives only $\neg$ and $\rightarrow$ as we did in I.7. In the same vein we view the existential quantifier $\exists$ as a defined symbol as well: We replace $\exists x\varphi(x)$ by $\neg\forall x\neg\varphi(x)$. (They are equivalent by Exercise 4.5.) We also fix some list of constants, function symbols and predicate symbols to complete our language $\mathcal{L}$. The axioms include the schemes (I.7.1) for propositional logic but now the variables α, β and γ range over all formulas of $\mathcal{L}$. In addition we include two schemes that express the meaning of the universal quantifier. Note that we are considering all formulas, not just sentences and remember that validity for a formula with free variables is the same as for its universal closure.

8.1 Axioms: Let α, β and γ be any formulas of $\mathcal{L}$. The axioms of our system are all formulas of $\mathcal{L}$ of the following forms:

(i) $(\alpha \rightarrow (\beta \rightarrow \alpha))$

(ii) $((\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)))$

(iii) $((\neg\alpha) \rightarrow (\alpha \rightarrow \beta))$

(iv) $(\forall x)\alpha(x) \rightarrow \alpha(t)$ for any term t which is substitutable for x in α.

(v) $(\forall x)(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow (\forall x)\beta)$ if α contains no free occurrences of x.

It is easy to check that all instances of these axiom schemes are valid. The restriction in (iv) is necessary as we explained when we defined substitutability (Definition 2.8). Recall that we considered in Example 2.9 (ii) the structure $\mathbb{Z}$ of the integers with constants for 0 and 1, a function s for successor and a predicate $A(x, y, z)$ which is interpreted as $x + y = z$. In particular, we considered the true sentence $\varphi = \forall x\exists y A(x, y, 0)$. As a true universal sentence, φ should be true of any object. Indeed (iv) asserts that any permissible substitution results in a formula valid in $\mathbb{Z}$. On the other hand, if we substitute $s(y)$ for x we get $\forall x\exists y A(s(y), y, 0)$ which is false in $\mathbb{Z}$. As for the restriction in (v), consider the true (in $\mathbb{Z}$) sentence $\varphi = \forall x(\forall y A(x, y, y) \rightarrow A(x, 1, 1))$. If we could ignore the restriction in (v) we could conclude from φ (via the rule of modus ponens given below) that $\forall y A(x, y, y) \rightarrow \forall x A(x, 1, 1))$. This formula is not valid in $\mathbb{Z}$ as can be seen by setting the free occurrence of x to 0. (This substitution only affects the left side of the implication by making it the true sentence $\forall y A(0, y, y)$. The right side of the implication is, however, false.)

Our system has two rules of inference. The first rule is modus ponens applied to the formulas of $\mathcal{L}$. The second captures one direction of the equivalence between the validity of a formula with free variables and that of its universal closure. (The other direction is included in axiom scheme (iv). Just take t to be x.)

8.2 The rules of inference:

(i) *Modus Ponens*: From α and $\alpha \rightarrow \beta$, we can infer β for any formulas α and β.

(ii) *Generalization*: From $\forall x\alpha$ infer α.

As in propositional logic, such axiom and rule based systems are generally called Hilbert–style proof systems. The definition of a proof from a set of formulas Σ is the same as for propositional logic except that we have more axioms and rules.

Definition 8.3: Let Σ be a set of formulas of $\mathcal{L}$.

(i) A *proof from* Σ is a finite sequence $\alpha_1, \alpha_2, \dots \alpha_n$ of formulas of $\mathcal{L}$ such that, for each $i \leq n$, one of the following is true:

  (1) α_i is a member of Σ;

  (2) α_i is an axiom;

  (3) α_i can be inferred from some of the previous α_j by an application of a rule of inference.

(ii) α is *provable* (a *theorem*) *from* Σ if there is a proof $\alpha_1, \dots, \alpha_n$ from Σ with $\alpha_n = \alpha$.

(iii) A *proof* of α is simply a proof from $\emptyset$. α is provable if it is *provable* from $\emptyset$.

The standard soundness, completeness and compactness theorems can be proven for the system presented here. It is taken from Elliot Mendelson's *Introduction to Mathematical Logic* [1964, 3.2] and a development of predicate logic using it can be found there. In §13, we will extend the rule based system of resolution to a fragment of predicate logic and prove the corresponding results for it.

9. Prenex Normal Form and Skolemization

We would like to show that, in a certain sense, predicate logic can almost be reduced to propositional logic. Roughly speaking, we want to eliminate the quantifiers by introducing new function symbols and terms. The basic idea is that a formula such as:

$$\varphi = \forall x_1 \dots \forall x_n \exists y_1 \dots \exists y_m R(x_1, \dots, x_n, y_1, \dots, y_m)$$

will be replaced by one

$$\psi = \forall x_1 \dots \forall x_n R(x_1, \dots, x_n, f_1(x_1, \dots, x_n), f_2(x_1, \dots, x_n), \dots, f_n(x_1, \dots, x_n)).$$

Here each f_i is a new function symbol. The intended interpretation of f_i is as a function choosing, for any given $x_1, \dots, x_n$, a y_i which makes the formula true if one exists. Such functions are called Skolem functions. It is clear that φ and ψ are equisatisfiable (i.e., φ is satisfiable iff ψ is) and so we could try to find a tableau (or other) refutation of ψ just as well as one of φ. In order to reap the full benefits from such a procedure, it is convenient to first replace φ by an equivalent formula φ' called a *prenex normal form* of φ in which all the quantifiers are at the beginning. We can then hope to eliminate successive blocks of quantifiers $\forall \vec{x} \exists \vec{y}$ by introducing appropriate Skolem functions. The ultimate goal is to get a *universal formula* ψ (i.e., one with only universal quantifiers which all occur as the initial symbols of ψ) which is *equisatisfiable* with the original φ. (We say that φ and ψ are *equisatisfiable* if both are satisfiable or if neither are.) We would then only need to consider universal formulas in any refutation proof scheme. (Of course, we have resolution in mind.)

As we know how to replace all uses of connectives by expressions involving only $\neg$ and $\vee$ (they form an adequate set of connectives by Corollary I.2.11), we assume for convenience that our given formula φ has no other connectives. We will show how to find a prenex equivalent for such a φ. We first need the basic moves to handle $\neg$ and $\vee$.

Lemma 9.1: *For any string of quantifiers $\overrightarrow{Qx} = Q_1 x_1 Q_2 x_2 \dots Q_n x_n$ (each Q_i is $\forall$ or $\exists$) and any formulas φ, ψ we have the following provable equivalences:*

(1a) $\vdash \overrightarrow{Qx} \neg \forall y \varphi \leftrightarrow \overrightarrow{Qx} \exists y \neg \varphi$.
(1b) $\vdash \overrightarrow{Qx} \neg \exists y \varphi \leftrightarrow \overrightarrow{Qx} \forall y \neg \varphi$.
(2a) $\vdash \overrightarrow{Qx}(\forall y \varphi \vee \psi) \leftrightarrow \overrightarrow{Qx} \forall z(\varphi(y/z) \vee \psi)$.
(2a′) $\vdash \overrightarrow{Qx}(\varphi \vee \forall y \psi) \leftrightarrow \overrightarrow{Qx} \forall z(\varphi \vee \psi(y/z))$.
(2b) $\vdash \overrightarrow{Qx}(\exists y \varphi \vee \psi) \leftrightarrow \overrightarrow{Qx} \exists z(\varphi(y/z) \vee \psi)$.
(2b′) $\vdash \overrightarrow{Qx}(\varphi \vee \exists y \psi) \leftrightarrow \overrightarrow{Qx} \exists z(\varphi \vee \psi(y/z))$.

where z is a variable not occurring in φ or ψ or among the x_i.

Proof: Tableaux proofs of such equivalences are fairly simple and are left as exercises. (Samples of (1), (2a) and (2b′) were given in Exercise 6.12 (a), (b) and (c) respectively.) Alternatively one can argue semantically for the equivalences and then apply the completeness theorem. (Exercise 4.5 essentially gives (1a) and (1b).) A general approach to these equivalences is outlined in Exercises 1–3.

Note: In the context of resolution proofs, the practice of renaming variables as in 2a and 2b to avoid possible conflicts is often called *standardizing the variables apart*.

We can now prove that every formula φ has a prenex equivalent.

Theorem 9.2 (Prenex Normal Form): *For every formula φ there is an equivalent formula φ' with the same free variables in which all quantifiers appear at the beginning. Such an equivalent of φ is called a prenex normal form (PNF) of φ.*

Proof: By induction on the depth of φ. Remember that, by Corollary I.2.11, we may assume that the only propositional connectives occurring in φ are $\neg$ and $\vee$. If φ is atomic there is nothing to prove. If φ is $\forall y \psi$ or $\exists y \psi$ and ψ' is a PNF of ψ then $\forall y \psi'$ or $\exists y \psi'$ is one for φ. (This fact is the base case for the induction in Exercise 1.) If $\varphi = \neg \psi$ and ψ' is a PNF of ψ then repeated applications of the clauses (1a) and (1b) of the lemma will produce the desired PNF for φ. If $\varphi = \psi \vee \theta$ then repeated applications of the clauses (2a), (2a′), (2b) and (2b′) will give the result for φ. □

Note: One can easily introduce prenexing rules that deal directly with the other connectives. The following equivalences may be used to put formulas in PNF without first eliminating any of the connectives except $\leftrightarrow$:

(3a) $\vdash \overrightarrow{Qx}(\forall y \varphi \wedge \psi) \leftrightarrow \overrightarrow{Qx} \forall z(\varphi(y/z) \wedge \psi)$.
(3a′) $\vdash \overrightarrow{Qx}(\varphi \wedge \forall y \psi) \leftrightarrow \overrightarrow{Qx} \forall z(\varphi \wedge \psi(y/z))$.
(3b) $\vdash \overrightarrow{Qx}(\exists y \varphi \wedge \psi) \leftrightarrow \overrightarrow{Qx} \exists z(\varphi(y/z) \wedge \psi)$.
(3b′) $\vdash \overrightarrow{Qx}(\varphi \wedge \exists y \psi) \leftrightarrow \overrightarrow{Qx} \exists z(\varphi \wedge \psi(y/z))$.
(4a) $\vdash \overrightarrow{Qx}(\forall y \varphi \rightarrow \psi) \leftrightarrow \overrightarrow{Qx} \exists z(\varphi(y/z) \rightarrow \psi)$.
(4a′) $\vdash \overrightarrow{Qx}(\varphi \rightarrow \forall y \psi) \leftrightarrow \overrightarrow{Qx} \forall z(\varphi \rightarrow \psi(y/z))$.
(4b) $\vdash \overrightarrow{Qx}(\exists y \varphi \rightarrow \psi) \leftrightarrow \overrightarrow{Qx} \forall z(\varphi(y/z) \rightarrow \psi)$.
(4b′) $\vdash \overrightarrow{Qx}(\varphi \rightarrow \exists y \psi) \leftrightarrow \overrightarrow{Qx} \exists z(\varphi \rightarrow \psi(y/z))$.

Again z is a variable not occurring on the left–hand side of the equivalences.

Example 9.3: We find PNF's for two formulas:

(i) $\forall x \exists y P(x,y) \vee \neg \exists x \forall y Q(x,y)$:
$\forall u[\exists y P(u,y) \vee \neg \exists x \forall y Q(x,y)]$
$\forall u \exists v[P(u,v) \vee \neg \exists x \forall y Q(x,y)]$
$\forall u \exists v[P(u,v) \vee \forall x \neg \forall y\ Q(x,y)]$
$\forall u \exists v[P(u,v) \vee \forall x \exists y \neg Q(x,y)]$
$\forall u \exists v \forall w[P(u,v) \vee \exists y \neg Q(w,y)]$
$\forall u \exists v \forall w \exists z[P(u,v) \vee \neg Q(w,z)]$.

(ii) $\forall x \forall y[(\exists z)(P(x,z) \wedge P(y,z)) \rightarrow \exists u Q(x,y,u)]$:
$\forall x \forall y \forall w[P(x,w) \wedge P(y,w) \rightarrow \exists u Q(x,y,u)]$
$\forall x \forall y \forall w \exists z[P(x,w) \wedge P(y,w) \rightarrow Q(x,y,z)]$.

(iii) Alternatively we could get a different PNF for (i) as follows:
$\forall u[\exists y P(u,y) \vee \neg \exists x \forall y Q(x,y)]$
$\forall u[\exists y P(u,y) \vee \forall x \neg \forall y\ Q(x,y)]$
$\forall u \forall w[\exists y P(u,y) \vee \neg \forall y Q(w,y)]$
$\forall u \forall w \exists v[P(u,v) \vee \neg \forall y Q(w,y)]$
$\forall u \forall w \exists v[P(u,v) \vee \exists y \neg Q(w,y)]$
$\forall u \forall w \exists v \exists z[P(u,v) \vee \neg Q(w,z)]$.

We can now reduce the problem of giving refutation proofs of arbitrary sentences of the predicate calculus to that for universal ones.

Theorem 9.4 (Skolemization): *For every sentence φ in a given language $\mathcal{L}$ there is a universal formula φ' in an expanded language $\mathcal{L}'$ gotten by the addition of new function symbols such that φ and φ' are equisatisfiable.*

(Note that we do not claim that the formulas are equivalent. The procedure will always produce a φ' such that $\varphi' \rightarrow \varphi$ is valid but $\varphi \rightarrow \varphi'$ need not always hold. See Exercise 9.4 for an example.)

Proof: By Theorem 9.2 we may assume that φ is in prenex normal form. Let $y_1, \ldots, y_n$ be the existentially quantified variables of φ in the order in which they appear in φ from left to right and, for each $i \leq n$, let $x_1, \ldots, x_{n_i}$ be all the universally quantified variables preceding y_i. We expand $\mathcal{L}$ to $\mathcal{L}'$ by adding new n_i–ary function symbols f_i for each $i \leq n$. We now form φ' by first deleting each $\exists y_i$ and then replacing each remaining occurrence of y_i by $f_i(x_1, \ldots, x_{n_i})$. We claim that φ' is the desired sentence equisatisfiable with φ. To verify this claim it suffices to apply the following lemma n times:

Lemma 9.5: *For any sentence $\varphi = \forall x_1 \ldots \forall x_n \exists y \psi$ of a language $\mathcal{L}$, φ and $\varphi' = \forall x_1 \ldots \forall x_n \psi(y/f(x_1, \ldots, x_n))$ are equisatisfiable when f is a function symbol not in $\mathcal{L}$.*

Proof: Let $\mathcal{L}'$ be the language obtained from $\mathcal{L}$ by adding the function symbol f. It is clear that if $\mathcal{A}'$ is a structure for $\mathcal{L}'$, $\mathcal{A}$ is the structure obtained from $\mathcal{A}'$ by omitting the function interpeting f and $\mathcal{A}' \vDash \varphi'$, then $\mathcal{A} \vDash \varphi$. On the other hand, if $\mathcal{A}$ is a structure for $\mathcal{L}$ and $\mathcal{A} \vDash \varphi$, we can expand $\mathcal{A}$ to a structure $\mathcal{A}'$ by defining $f^{\mathcal{A}'}$ so thatfor every $a_1, \ldots, a_n \in A = A'$, $\mathcal{A} \vDash \psi(y/f(a_1, \ldots, a_n))$. Of course, $\mathcal{A}' \vDash \varphi'$. Note that n may be 0, that is, f may be a constant symbol. □

Corollary 9.6: *For any set S of sentences of a language $\mathcal{L}$ we can construct a set S' of universal sentences of a language $\mathcal{L}'$ which is an expansion of $\mathcal{L}$ gotten by adding on new function symbols such that S and S' are equisatisfiable.*

Proof: Apply the construction supplied by Theorem 9.4 to each sentence φ of S separately to introduce new function symbols f_φ for each sentence φ of S and form the corresponding universal sentence φ'. Let S' be the collection of all of these sentences φ' and $\mathcal{L}'$ the corresponding expansion of $\mathcal{L}$. As in the proof of the theorem it is clear that, if a structure $\mathcal{A}'$ for $\mathcal{L}'$ is a model of S', then it is one of S. The proof also shows how to expand any model of S to one of S' by defining each new function symbol f_φ independently of what is done for the others. □

Example 9.7: Possible Skolemizations corresponding to the prenex normal forms of Example 9.3 above are as follows:

(i) $\forall u \forall w[P(u, f_1(u)) \vee \neg Q(w, f_2(u,w))]$

(ii) $\forall x \forall y \forall w[P(x,w) \wedge P(y,w) \rightarrow Q(x,y,f(x,y,w))]$

and

(iii) $\forall u \forall w[P(u, f_1(u,w)) \vee \neg Q(w, f_2(u,w))]$.

Example 9.8: There are many familiar examples of Skolemization in the construction of axiom systems for standard mathematical structures such as groups or rings. In these situations, axioms of the form $\forall x \exists y \varphi(x,y)$ can be replaced by open formulas of the form $\varphi(x, f(x))$ by introducing the appropriate Skolem functions.

As a particular example let us reconsider the structure of Example 2.9 for the integers $\mathbb{Z}$ and the sentence $\forall x \exists y A(x,y,0)$ which says that every integer has an additive inverse. The Skolemization of this sentence is $\forall x A(x, f(x), 0)$. The interpretation of f should be the unary function taking every integer x to its additive inverse $-x$. The Skolemized sentence then simply says that, for all x, $x + (-x) = 0$.

Harking back to the clausal forms for predicate calculus introduced in §5, we now see that every set of sentences has an equisatisfiable clausal form.

Corollary 9.9: *For any set S of sentences of $\mathcal{L}$, there is (in the terminology of §5) a formula, that is, a set T of clauses in a language $\mathcal{L}'$ gotten by adding new function symbols to $\mathcal{L}$ such that S and T are equisatisfiable.*

Proof: Consider the set S' of universal sentences $\forall\vec{x}\varphi'(\vec{x})$ equisatisfiable with S given by Corollary 9.6. Let T' consist of the equivalent open formulas $\varphi'(\vec{x})$ gotten by dropping the initial universal quantifiers from the elements of S'. (φ and φ' are equivalent by Exercise 4.8 or 6.13.) If we view each atomic formula of $\mathcal{L}'$ as a propositional letter and form the CNF equivalent $\psi_\varphi = \wedge\psi_{\varphi,i}$ of each formula $\varphi' \in T'$, we get a set of formulas T'' each in CNF and each equivalent to the one of $T' : \wedge\psi_{\varphi,i} = \psi_\varphi \equiv \varphi' \equiv \varphi$ for each $\varphi \in S$. (For each φ, ψ_φ is equivalent to φ' by Theorem 4.8.) The desired set T of clauses then consists precisely of the set of all conjuncts from all of the formulas φ in $T'' : T = \{\psi_{\varphi,i} \mid \varphi \in S\}$. □

Exercises

1. Let φ and ψ be any formulas (with free variables) and let $\overrightarrow{Qx} = Q_1x_1Q_2x_2\ldots Q_nx_n$ be any string of quantifiers. Prove that if φ and ψ are equivalent then so are $\overrightarrow{Qx}\varphi$ and $\overrightarrow{Qx}\psi$. (Hint: proceed by induction on the length n of $\overrightarrow{Qx}$.) Thus in proving the equivalences (1a)–(4b′) we may assume that the formulas have free variables but the strings $\overrightarrow{Qx}$ of initial quantifiers are empty.
2. Use the theorem on constants (Exercise 4.8) to show that we may also assume that there are no free variables in formulas in the equivalences (1a)–(4b′).
3. Now argue for the validity of each equivalence (1a)–(4b′) either semantically or by giving a tableau proof. (Use exercises 1 and 2 to assume that the $\overrightarrow{Qx}$ are empty and that there are no free variables present.)
4. Let $\varphi(x,y)$ be an atomic formula and f a function symbol not appearing in φ. Show that the sentence $\forall x\varphi(x,f(x)) \rightarrow \forall x\exists y\varphi(x,y)$ is valid but its converse, $\forall x\exists y\varphi(x,y) \rightarrow \forall x\varphi(x,f(x))$, is not.
5. Find prenex equivalents and Skolemizations for the following sentences:
   (a) $\forall y(\exists xP(x,y) \rightarrow Q(y,z)) \wedge \exists y(\forall xR(x,y) \vee Q(x,y))$.
   (b) $\exists xR(x,y) \leftrightarrow \forall yP(x,y)$.
   (c) $\forall x\exists yQ(x,y) \vee \exists x\forall yP(x,y) \wedge \neg\exists x\exists yP(x,y)$.
   (d) $\neg(\forall x\exists yP(x,y) \rightarrow \exists x\exists yR(x,y)) \wedge \forall x\neg\exists yQ(x,y)$.

10. Herbrand's Theorem

The introduction of Skolem functions and the reduction of any set of sentences to universal ones gives us a more concrete approach to the dichotomy of unsatisfiability and model building implicit in the completeness theorem for tableau proofs. Consider any set S of universal sentences in a language $\mathcal{L}$ with various Skolem functions already included. We also assume that $\mathcal{L}$ contains at least one constant c. We claim that either S is

inconsistent (i.e., unsatisfiable) or there is a model $\mathcal{A}$ of S whose elements are simply the ground terms of the language $\mathcal{L}$. As all such terms must be interpreted in any structure for $\mathcal{L}$, this is in some sense a minimal structure for $\mathcal{L}$.

Definition 10.1: The set of ground (i.e., variable–free) terms of a language $\mathcal{L}$ is called the *Herbrand universe* of $\mathcal{L}$. A structure $\mathcal{A}$ for $\mathcal{L}$ is an *Herbrand structure* if its universe A is the Herbrand universe of $\mathcal{L}$ and, for every function symbol f of $\mathcal{L}$ and elements $t_1,\ldots,t_n$ of A,

$$f^{\mathcal{A}}(t_1,\ldots,t_n) = f(t_1,\ldots,t_n).$$

(We include here the requirement that $c^{\mathcal{A}} = c$ for each constant symbol c of $\mathcal{L}$.)

The Herbrand universe is reminiscent of the structure produced in the proof of the completeness theorem (Theorem 7.3). As we shall see, they are intimately related. Note also that no restrictions are placed on the interpretations of the predicates of $\mathcal{L}$ so there can be many Herbrand structures for a given language $\mathcal{L}$.

Definition 10.2: If S is a set of sentences of $\mathcal{L}$ then an *Herbrand model* $\mathcal{M}$ of S is an Herbrand structure for $\mathcal{L}$ which is a model of S, i.e., every sentence of S is true in $\mathcal{M}$.

Example 10.3: If our language $\mathcal{L}$ contains the constants a and c, a unary function symbol f and a binary one g and predicates P, Q, R then the Herbrand universe H for $\mathcal{L}$ is $\{a, c, f(a), f(c), g(a,c), ff(a), ff(c), f(g(a,c)), g(a,f(a)), g(a,f(c)), g(a,g(a,c)),\ldots, g(f(a),f(c)),\ldots, fff(a),\ldots\}$.

We claim not only that there is an Herbrand model for any consistent set of universal sentences (or open formulas) S but also that, if S is inconsistent, then its unsatisfiability is demonstrable at the truth–functional level via ground instances of the formulas (that is, instances of substitutions of terms from the Herbrand structure for the universally quantified (free) variables in S).

Theorem 10.4 (Herbrand's Theorem): *Let $S = \{\varphi_i(x_1,\ldots,x_{n_i})\}$ be a set of open formulas of a language $\mathcal{L}$. Either*

(i) *S has an Herbrand model or*

(ii) *S is unsatisfiable and, in particular, there are finitely many ground instances of elements of S whose conjunction is unsatisfiable.*

The latter case, (ii), *is equivalent to*

(ii′) *There are finitely many ground instances of the negations of formulas of S whose disjunction is valid. (As we may view these ground instances as built from propositional letters, the disjunction being valid is equivalent to its being a truth–functional tautology.)*

Proof: Let S' consist of all ground instances from $\mathcal{L}$ of formulas from S. Consider the CST from S' (in the language $\mathcal{L}$ alone, i.e., with no additional constant symbols added on) starting with $F(\alpha \wedge \neg\alpha)$ for any sentence α. There are two possible outcomes. First, there might be a (possibly infinite) noncontradictory path in the tableau. In this case, the proof of Theorem 7.3 supplies us with a model $\mathcal{A}$ of S' whose elements are the ground terms of $\mathcal{L}$, i.e., an Herbrand model for S'. By definition of S' and of tableau proofs from S', $\varphi(t_1, \ldots, t_n)$ is true in $\mathcal{A}$ for every $\varphi \in S$ and every $t_1, \ldots, t_n$ in the Herbrand universe. Thus the structure $\mathcal{A}$ defined on the Herbrand universe by the path is a model for S.

The other possibility is that the tableau is finite and contradictory. In this case, the tableau is, by definition, a proof of the unsatisfiability of the set of elements of S' appearing in the tableau and so we have the unsatisfiable conjunction required in (ii). Moreover, S cannot be satisfiable: A model for S is one in which $\varphi_i(x_1, \ldots, x_{n_i})$ is valid, i.e., true for every instance of the free variables $x_1, \ldots, x_{n_i}$, for every $\varphi_i \in S$. Any example of (ii), however, directly exhibits a set of such instances which cannot be simultaneously satisfied in any model.

Finally, by Theorem 4.8 we may manipulate the variable–free formulas as propositional letters. The unsatisfiability of the conjunction as required in (ii) is then equivalent by propositional rules to the disjunction of their negations being valid or a tautology. Thus, (ii) and (ii′) are equivalent. □

Note that if S is unsatisfiable (and so (i) fails), then (ii) directly exhibits the unsatisfiability of S. Thus we have a method for producing either an Herbrand model for S or a particular finite counterexample to the existence of any model of S.

We can now give some variations on Herbrand's theorem which will be particularly useful in our study of resolution theorem proving and PROLOG. We can also phrase our results positively to give a direct reduction of provability or validity in predicate logic to provability or validity in propositional logic. We begin with the special case of an existential formula.

Corollary 10.5: *If $\varphi(\vec{x})$ is a quantifier–free formula in a language $\mathcal{L}$ with at least one constant symbol, then $\exists\vec{x}\varphi(\vec{x})$ is valid if and only if there are ground terms $\vec{t_i}$ of $\mathcal{L}$ such that $\varphi(\vec{t_1}) \vee \ldots \vee \varphi(\vec{t_n})$ is a tautology.*

Proof: First, note that $\exists\vec{x}\varphi(\vec{x})$ is valid $\Leftrightarrow$ $\forall\vec{x}\neg\varphi(\vec{x})$ is unsatisfiable $\Leftrightarrow$ $\neg\varphi(\vec{x})$ is unsatisfiable. By Theorem 10.4 (ii), $\neg\varphi(\vec{x})$ is unsatisfiable iff there are finitely many ground terms $\vec{t_i}$ of $\mathcal{L}$ such that $\varphi(\vec{t_1}) \vee \ldots \vee \varphi(\vec{t_n})$ is a tautology. □

Translating these results into the terminology of clauses of §5, we have what will be the key to resolution theorem proving in the predicate calculus.

Theorem 10.6: *A set S of clauses is unsatisfiable if and only if the set S' of all ground instances from the Herbrand universe of the clauses in S is unsatisfiable.*

Proof: If some set of instances of elements of S (instantiated with terms from the Herbrand universe) is unsatisfiable then S, which asserts the validity of its member clauses, is surely unsatisfiable. In the other direction, if S is unsatisfiable then, by Herbrand's theorem (ii), there is, in fact, a finite set of instances of clauses of S which is unsatisfiable. □

The restriction in our version of Herbrand's theorem that S contain only universal formulas (or equivalently that we consider only sets of clauses) is necessary as can be seen from the example in Exercise 1. On the other hand, further restricting S to consists of only program clauses allows us to establish the existence of minimal and indeed least Herbrand models. (See Exercises 3.) Moreover, in the case of a deduction from a set of program clauses, which is the case of interest for PROLOG, we can eliminate the disjunction in the analog of Corollary 10.5 in favor of a single valid instance. That is, if P is a set of program clauses and $\theta(\vec{x})$ is an atomic formula, then $P \vDash \exists\vec{x}\theta(\vec{x}) \Leftrightarrow$ there are Herbrand terms $\vec{t}$ such that $P \vDash \theta(\vec{t})$ (Exercise 5).

Finally, although it is not directly relevant to resolution theorem proving, we can use Skolemization to get a generalization of Corollary 10.5 to arbitrary sentences. This result provides a propositional equivalent for validity in predicate logic.

Theorem 10.7: *Let φ be a sentence in prenex normal form in a language $\mathcal{L}$, ψ a prenex equivalent of $\neg\varphi$ and $\theta(\vec{x})$ an open Skolemization of ψ in the language $\mathcal{L}'$ as in Theorem 9.4. (Note that the free variables in ψ are precisely the existentially quantified ones of φ.) Then φ is valid if and only if there are terms $\vec{t}_1, \ldots, \vec{t}_n$, of $\mathcal{L}'$ such that $\neg\theta(\vec{t_1}) \vee \ldots \vee \neg\theta(\vec{t_n})$ is a tautology.*

Proof: By Corollary 10.5, it suffices to prove that φ is valid if and only if $\exists\vec{x}\neg\theta(\vec{x})$ is valid. Now φ is valid iff $\neg\varphi$ is not satisfiable. On the other hand, Theorem 9.4 says that $\neg\varphi$ is satisfiable if and only if $\theta(\vec{x})$ is satisfiable. Thus, φ is valid iff $\theta(\vec{x})$ is not satisfiable. Finally, note that $\theta(\vec{x})$ (or, equivalently, $\forall\vec{x}\theta$) is not satisfiable iff $\exists\vec{x}\neg\theta(\vec{x})$ is valid. □

Exercises

1. Let $\mathcal{L}$ consist of the constant c and the unary predicate R.
   (a) What is the Herbrand universe for $\mathcal{L}$?
   (b) What are the possible Herbrand structures for $\mathcal{L}$?
   (c) Let $S = \{R(c), \exists x\neg R(x)\}$. Note that S does not consist solely of universal formulas and so is not in clausal form. Show that while S is satisfiable, it has no Herbrand model.

2. Let $\mathcal{L}$ consist of the constants c and the function symbol f.
   (a) What is the Herbrand universe for $\mathcal{L}$?
   (b) Describe infinitely many possible Herbrand structures for $\mathcal{L}$.
3. Prove that every set P of program clauses has a minimal (indeed least) Herbrand model. (Hint: Prove that the intersection of all Herbrand models for P is itself an Herbrand model for P.)
4. Let M_P be the minimal Herbrand model for a set P of program clauses in a language $\mathcal{L}$. Prove that for each atomic formula φ of $\mathcal{L}$, $M_P \models \varphi$ iff φ is a logical consequence of P.
5. Let P be a set of program clauses and $G = \neg\theta(\vec{x})$ be a goal clause. Prove that, if $P \models \exists\vec{x}\theta(\vec{x})$ (or equivalently, $P \cup \{G\}$ is unsatisfiable), then there are Herbrand terms $\vec{t}$ such that $P \models \theta(\vec{t})$. (Hint: If $P \models \exists\vec{x}\theta(\vec{x})$, look at the minimal model $\mathcal{M}_P$ and apply Exercise 4.)

11. Unification

We saw in Theorem 9.4 that, for every formula φ of predicate logic, there is another one ψ which is open, in conjunctive normal form and equisatisfiable with φ. Thus if we are interested in the satisfiability of (sets of) formulas in predicate logic, it suffices to consider open formulas in clausal form. The only difference from the propositional case is that literals are now atomic formulas (possibly with free variables and the added Skolem function symbols) rather than simply propositional letters. Of course, a clause with free variables is understood to be equivalent to its universal closure. From the viewpoint of resolution theorem proving, the only difference between predicate and propositional logic in deducing $\square$ from S is the problem of how to instantiate the free variables (i.e., make substitutions) in the available clauses so that we may then apply the reduction rule.

Of course, we could, as in the tableau proof of Herbrand's theorem, simply list all ground term substitutions in the Herbrand structure and start running our resolution machine with all of them as inputs. Needless to say, this is not an efficient procedure. We need a better guide.

For example, if we have $C_1 = \{P(f(x), y),\ \neg Q(a, b, x)\}$ and $C_2 = \{\neg P(f(g(c)), g(d))\}$ we should be able to resolve C_1 and C_2 by directly substituting $g(c)$ for x and $g(d)$ for y to get $\{\neg Q(a, b, g(c))\}$. (Remember that C_1 is equivalent to its universal closure $\forall x \forall y (P(f(x), y) \vee \neg Q(a, b, x))$ from which we can deduce any substitution instance.) The general approach to the problem of which substitutions to make when doing resolution proofs is called *unification* (or *matching*). We describe it before giving the resolution algorithm for predicate calculus. First, we need some notation for substitutions.

Definition 11.1: A *substitution* θ is a finite set of the form $\{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$ where the x_i are distinct variables and each t_i is a term other than x_i. If the t_i are all ground terms, we call θ a *ground substitution*. If the t_i are distinct variables, we call θ a *renaming substitution*.

As we are concerned with substitutions in clauses, we must define the action of θ on a clause C. In order to define the composition (successive application) of different substitutions, it is convenient to define the action of a substitution θ on terms as well.

Definition 11.2: An *expression* is any term or literal. Given a substitution θ and an expression E (or a set of expressions S) we write $E\theta$ ($S\theta$) for the result of replacing each occurrence of x_i in E (in every element of S), by t_i for every $i \leq n$. If the resulting expression $E\theta$ (set of expressions $S\theta$) is ground, i.e., variable–free, then the substitution is called a *ground instance* of E (S).

Note: The substitution θ is written as a set of elements of the form x_i/t_i rather than as a sequence of such terms because the intended substitutions of t_i for each x_i are performed simultaneously rather than successively. Thus, in $E\{x_1/t_1, x_2/t_2\}$, any occurrences of x_2 in t_1 will be unaffected by the substitution of t_2 for x_2.

Example 11.3:

(i) Let $S = \{f(x, g(y)), P(a, x), Q(y, z, b), \neg P(y, x)\}$ and $\theta = \{x/h(a), y/g(b), z/c\}$. Then $S\theta = \{f(h(a), g(g(b)), P(a, h(a)),\ Q(g(b), c, b), \neg P(g(b), h(a))\}$. Here θ is a ground substitution and $S\theta$ is a ground instance of S.

(ii) Let S be as in (i) and let $\sigma = \{x/h(y), y/g(z), z/c\}$. Then $S\sigma = \{f(h(y), g(g(z))), P(a, h(y)), Q(g(z), c, b), \neg P(g(z), h(y))\}$.

Composition is a natural operation on substitutions, i.e., we want to define $\theta\sigma$ to be the substitution which when applied to any expression E to get $E(\theta\sigma)$ gives the same result as applying σ to $E\theta$, i.e., $(E\theta)\sigma$.

Example 11.4: Let $E = P(x, y, w, u)$, $\theta = \{x/f(y), y/g(z), w/v\}$ and $\sigma = \{x/a, y/b, z/f(y), v/w, u/c\}$. Then $E\theta = P(f(y), g(z), v, u)$ and $(E\theta)\sigma = P(f(b), g(f(y)), w, c)$. What then should $\theta\sigma$ be? Well, x is replaced first by $f(y)$. We then replace y by b. The result is $x/f(b)$. y is replaced by $g(z)$ and then z by $f(y)$ and so we get $y/g(f(y))$. w gets replaced by v which is in turn replaced by w. The result might be written w/w but this is omitted from the description as it causes no changes. The substitution x/a in σ also has no bearing on the final outcome since there are no $x's$ left after applying θ. The final substitution in σ, u/c, however, acts unimpeded as there is no substitution for u made by θ. Thus $\theta\sigma = \{x/f(b), y/g(f(y)), u/c\}$.

Guided by this example we can write out the formal definition of composition of substitutions.

Definition 11.5:

(i) If $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ and $\sigma = \{y_1/s_1, \ldots, y_m/s_m\}$ then $\theta\sigma$ is $\{x_1/t_1\sigma, \ldots, x_n/t_n\sigma, y_1/s_1, \ldots, y_m/s_m\}$ less any $x_i/t_i\sigma$ for which $x_i = t_i\sigma$ and any y_j/s_j for which $y_j \in \{x_1, \ldots, x_n\}$.

(ii) The *empty substitution* ϵ (which does nothing to any expression) is an identity for this operation, i.e., $\theta\epsilon = \epsilon\theta = \theta$ for every substitution θ.

We now check that we have defined composition correctly and that it is associative.

Proposition 11.6: *For any expression E and substitutions θ, ψ, and σ:*

(i) $(E\theta)\sigma = E(\theta\sigma)$ *and*

(ii) $(\psi\theta)\sigma = \psi(\theta\sigma)$.

Proof: Let θ and σ be as in the definition of composition and let $\psi = \{z_1/r_1, \ldots, z_k/r_k\}$. As the result of a substitution consists simply of replacing each variable in an expression by some term, it suffices to consider the case in which E is a variable, say v, in (i) and the result of applying $(\psi\theta)\sigma$ and $\psi(\theta\sigma)$ to v in (ii).

(i) We divide the argument into two cases.

Case 1: $v \notin \{x_1, \ldots, x_n\}$. In this case $v\theta = v$ and $(v\theta)\sigma = v\sigma$. If $v \notin \{y_1, \ldots, y_m\}$ then $v\sigma = v = v(\theta\sigma)$ as $v \notin \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ and so no substitution is made. If, on the other hand, $v = y_j$ for some $j \leq n$ then $y_j \notin \{x_1, \ldots, x_n\}$, $(v\theta)\sigma = v\sigma = s_j = v(\theta\sigma)$.

Case 2: $v = x_i$ for some $i \leq n$. In this case $v\theta = t_i$ and $(v\theta)\sigma = t_i\sigma$ but this is exactly $v(\theta\sigma)$ by definition.

(ii) The result follows from several applications of (i):

$$\begin{aligned} v((\psi\theta)\sigma) &= (v(\psi\theta))\sigma \\ &= ((v\psi)\theta)\sigma \\ &= (v\psi)(\theta\sigma) \\ &= v(\psi(\theta\sigma) \end{aligned}$$

□

Thus we may omit parentheses when composing sequences of compositions. The composition operation on substitutions is, however, not commutative. (Exercise 3 asks for a counterexample.)

Our interest in substitutions, we recall, is to make certain elements of different clauses identical so that we may apply the resolution rule. The process of making substitutions that identify expressions is called *unification*.

Definition 11.7: If $S = \{E_1, \ldots, E_n\}$ is a set of expressions we say a substitution θ is a *unifier* for S if $E_1\theta = E_2\theta = \ldots = E_n\theta$, i.e., $S\theta$ is a singleton. S is said to be *unifiable* if it has a unifier.

Example 11.8: (i) Neither $\{P(x,a), P(b,c)\}$ nor $\{P(f(x),z), P(a,w)\}$ are unifiable. (Exercise 2).

(ii) $S_1 = \{P(x,c), P(b,c)\}$ and $S_2 = \{P(f(x),y), P(f(a),w)\}$ are, however, both unifiable. The first can be unified by $\{x/b\}$ and only by this substitution. The situation for S_2 is a bit different. $\theta = \{x/a, y/w\}$ unifies S_2 but so do $\sigma = \{x/a, y/a, w/a\}$ and $\psi = \{x/a, y/b, w/b\}$ as well as many others. Here θ has a certain advantage over the other substitutions in that it allows more scope for future substitutions. If we first applied θ to unify S_2 we could then unify the resulting set with the expression $P(f(a),c)$ by applying $\{w/c\}$. Had we used either of σ or ψ, however, we would be stuck. On the other hand, we can always go from θ to σ or ψ by applying the substitution $\{w/a\}$ or $\{w/b\}$ respectively. We capture this property of θ in the following definition.

Definition 11.9: A unifier θ for S is a *most general unifier (mgu) for S* if for every unifier σ for S there is a substitution λ such that $\theta\lambda = \sigma$.

Up to renaming variables there is only one result of applying an mgu:

Theorem 11.10: *If θ and ψ are both mgu's for S then there are renaming substitutions σ and λ (i.e., ones which consist solely of replacements of distinct variables by other distinct variables) such that $S\theta\sigma = S\psi$ and $S\theta = S\psi\lambda$.*

Proof: By the definition of an mgu there are σ and λ such that $S\theta\sigma = S\psi$ and $S\psi\lambda = S\theta$. Clearly, we may assume that σ and λ make substitutions only for variables occurring in $S\theta$ and $S\psi$ respectively. (They consist of the single terms $E\theta$ and $E\psi$, respectively, as θ and ψ both unify S.) Suppose σ makes some substitution x_i/t_i where t_i is not a variable or a constant. In this case the complexity (e.g., length) of the expression $E\theta\sigma$ in $S\theta\sigma = \{E\theta\sigma\}$ must be strictly larger than that of $E\theta$ in $S\theta$. As no substitutions of terms for variables (e.g., λ) can decrease the length of an expression we could not then have $S\psi\lambda = S\theta\sigma\lambda = S\theta$ as required. If there were in σ a substitution x_i/c, for some constant c, then no further substitution (e.g., λ) could return the resulting instances of c in an expression $E\theta\sigma$ in $S\theta\sigma$ back to instances of the variable x_i in $E\theta \in S\theta$. Thus, once again, we could not have $S\theta\sigma\lambda = S\theta$ for any λ. We now know that σ can contain only substitutions of one variable by another. If σ identified distinct variables by such a substitution, then λ could not distinguish them again. Thus σ (and similarly λ) is simply a renaming substitution. □

Exercises

1. Find substitutions which will unify the following sets of expressions.
   (a) $\{P(x, f(y), z), P(g(a), f(w), u), P(v, f(b), c)\}$.
   (b) $\{Q(h(x,y), w), Q(h(g(v), a), f(v)), Q(h(g(v), a), f(b))\}$.
2. Explain why neither expression in Example 8 (i) is unifiable.

3. Show that composition of substitutions is not commutative, that is, find two substitutions σ and λ such that $\sigma\lambda \neq \lambda\sigma$.
4. We say that expressions E and F are *variants* (or E is a variant of F) if there are substitutions θ and ψ such that $E\theta = F$ and $F\psi = E$. Recall that a renaming substitution is one of the form $\{x_1/y_1, \ldots, x_n/y_n\}$ where $x_1, \ldots, x_n$ are distinct variables in E and the $y_1, \ldots, y_n$ are distinct variables not including any variables in E other than perhaps some of $x_1, \ldots, x_n$. Prove that if E and F are variants, then there is a renaming substitution σ for E such that $E\sigma = F$.

12. The Unification Algorithm

In this section we will give an effective procedure for finding an mgu for a finite set of expressions S. We start with two illustrations, $S_1 = \{f(x, g(x)), f(h(y), g(h(z)))\}$ and $S_2 = \{f(h(x), g(x)), f(g(x), h(x))\}$. We begin our search for mgu's for S_1 and S_2 by noting that in each case the terms to be unified begin with f. Of course, if they each began with different function or predicate symbols there would be no hope as unification only replaces variables. The next step must be to check the first and second place arguments of f in the terms of S_1 (S_2). If we can unify them both by a single substitution, then we can unify S_1 (S_2). For S_1, we get $T_1 = \{x,\ h(y)\}$ and $T_2 = \{g(x),\ g(h(z))\}$ respectively. In order to unify T_1 in the most general way, we should substitute $h(y)$ for x. As we must do the substitution throughout the expressions being unified, the second place arguments in T_2 become $g(h(y))$ and $g(h(z))$. These first differ at y. We can now unify them by applying $\{y/z\}$. Again this must be applied to the entire expression and we get $f(h(z),\ g(h(z)))$ and $f(h(z),\ g(h(z)))$ as required for unification. Thus the composition $\{x/h(y)\}\{y/z\} = \{x/h(z)\}$ is our desired unifier. For S_2 the process halts when we try to unify the arguments of f. Here the first difference occurs in the set of first arguments where we get $\{h(x), g(x)\}$. As these terms differ at the function symbol rather than at a variable there is no hope of unifying them and so S_2 is not unifiable.

The general procedure for unification is to move along each of the expressions in the given set to the first position of disagreement. If, in any one of the expressions, it is not a variable the set is not unifiable. If it does not contain a variable in one of the expressions, we can replace it by one of the terms in the corresponding position at another expression. As long as the variable being replaced does not occur in the term replacing it, this substitution makes some progress towards unification. We can now try to repeat the process in the hope of eventually unifying the set of expressions. We now formalize this process.

Definition 12.1: Let S be a finite nonempty set of expressions. To define the *disagreement set* of S find the first (i.e., leftmost) position at which not all elements E of S have the same symbol. The set of subexpressions of each $E \in S$ that begin at this position is the disagreement set $D(S)$ of S. (In terms of formation trees, we find the lexicographically least node of the formation trees associated with each expression such that not all the labels of these nodes begin with the same symbol. $D(S)$ is then the set of labels of these nodes.)

Note that any unifier ψ of S must necessarily unify $D(S)$.

Example 12.2: For $S_1 = \{f(x, g(x)), f(h(y), g(h(z)))\}$ and $S_2 = \{f(h(x), g(x)), f(g(x), h(x))\}$ as above, the disagreement sets are $D(S_1) = \{x, h(y)\}$ and $D(S_2) = \{h(x), g(x)\}$. For $T_1 = S_1\{x/h(y)\} = \{f(h(y), g(h(y))), f(h(y), g(h(z)))\}$ the disagreement set is $\{y, z\}$.

12.3 The Unification Algorithm for S: Let S be a set of expressions. We attempt to unify it as follows:

Step 0. Set $S_0 = S$, $\sigma_0 = \epsilon$.

Step $k+1$. If S_k is a singleton, terminate the algorithm with the announcement that $\sigma_0\sigma_1\sigma_2 \ldots \sigma_k$ is an mgu for S. Otherwise, see if there is a variable v and a term t not containing v both of which are in $D(S_k)$. If not, terminate the algorithm with the announcement that S has no mgu. (Note that, in this case, it is at least clear that S_k is not unifiable.) If so, let v and t be the least such pair (in any fixed ordering of terms). (Indeed, we could nondeterministically choose any such t and v as will become clear from the proof that the algorithm succeeds.) Set $\sigma_{k+1} = \{v/t\}$ and $S_{k+1} = S_k\sigma_{k+1}$ and go on to step $k+2$.

Example 12.4: Consider the set of expressions

$$S = \{P(f(y, g(z)), h(b)),\ P(f(h(w), g(a)), t),\ P(f(h(b), g(z)), y)\}.$$

Step 1. $S = S\epsilon = S_0\sigma_0$ is not a singleton. $D(S_0) = \{y, h(w), h(b)\}$. Depending on the ordering of terms, there are two possibilities for σ_1: $\{y/h(w)\}$ and $\{y/h(b)\}$. It is better to choose the second (see step 2) but suppose we are not so clever and blindly set $\sigma_1 = \{y/h(w)\}$. We then get $S_1 = S_0\sigma_1$ which is
$\{P(f(h(w), g(z)), h(b)),\ P(f(h(w), g(a)), t),\ P(f(h(b), g(z)), h(w))\}$.

Step 2. $D(S_1) = \{w, b\}, \sigma_2 = \{w/b\}$ (and so we get to $\{y/h(b)\}$ after all). Then S_2 is
$\{P(f(h(b), g(z)), h(b)),\ P(f(h(b), g(a)), t),\ P(f(h(b), g(z)), h(b))\}$.

Step 3. $D(S_2) = \{z, a\}$, $\sigma_3 = \{z/a\}$. Then S_3 is
$\{P(f(h(b), g(a), h(b)),\ P(f(h(b), g(a)), t),\ P(f(h(b), g(a)), h(b))\}$.

Step 4. $D(S_3) = \{h(b), t\}, \sigma_4 = \{t/h(b)\}$. Then S_4 is
$\{P(f(h(b), g(a)), h(b)),\ P(f(h(b), g(a)), h(b)),\ P(f(h(b), g(a)), h(b))\}$.

Step 5. S_4 is a singleton and the mgu for S is
$\{y/h(w)\}\{w/b\}\{z/a\}\{t/h(b)\} = \{y/h(b), w/b, z/a, t/h(b)\}$.

Theorem 12.5: *For any S, the unification algorithm terminates at some step $k+1$ with a correct solution, i.e., either S is not unifiable as announced or $\psi = \sigma_0\sigma_1 \dots \sigma_k$ is in fact an mgu for S. Moreover, ψ has the special property that for any unifier θ of S, $\theta = \psi\theta$.*

Proof: First of all, the algorithm always terminates as each nonterminal step eliminates all occurrences of one of the finitely many variables in S. It is obvious that if the algorithm terminates with an announcement that there is no unifier, then S is not unifiable. On the other hand, if the algorithm terminates with the announcement that $\psi = \sigma_0 \dots \sigma_n$ is an mgu for S, then it is at least clear that ψ is a unifier for S. Suppose then that θ is any unifier for S. We must show that $\theta = \psi\theta$. We prove by induction that, for every i, $\theta = \sigma_0 \dots \sigma_i\theta$.

For $i = 0$, the claim clearly holds. Suppose we have $\theta = \sigma_0\sigma_1 \dots \sigma_i\theta$ and $\sigma_{i+1} = \{v/t\}$. It suffices to show that the substitutions $\sigma_{i+1}\theta$ and θ are equal. We show that their actions on each variable are the same. For $x \neq v$, $x\sigma_{i+1}\theta$ is clearly the same as $x\theta$. For v itself $v\sigma_{i+1}\theta = t\theta$. As θ unifies $S\sigma_0 \dots \sigma_i$ and v and t belong to $D(S\sigma_0 \dots \sigma_i)$, θ must unify v and t as well, i.e., $t\theta = v\theta$ as required. □

The unification algorithm given here is simple but inefficient. As presented, it is the search for a v and t with v not occurring in t that can take excessive amounts of time. The problem is that we may have to check each pair of items in the disagreement set rather than simply taking the first variable and term that we come across. As an example, consider the problem of unifying $S = \{P(x_1, \dots, x_n), P(f(x_0, x_0), \dots, f(x_{n-1}, x_{n-1}))\}$:

$$D(S_0) = \{x_1, f(x_0, x_0)\}; \quad \sigma_1 = \{x_1/f(x_0, x_0)\};$$
$$S_1 = \{P(f(x_0, x_0), x_2, \dots, x_n), P(f(x_0, x_0), f(f(x_0, x_0), f(x_0, x_0)), f(x_2, x_2), \dots, f(x_{n-1}, x_{n-1}))\}.$$
$$D(S_1) = \{x_2, f(f(x_0, x_0), f(x_0, x_0))\}; \quad \sigma_2 = \{x_2/f(f(x_0, x_0), f(x_0, x_0))\};$$

etc.

Note that before announcing σ_1 we had to check that x_1 was not either of the two occurrences of variables in $f(x_0, x_0)$. For σ_2 there were four occurrences to check. In general $D(S_{i+1})$ will have twice as many occurrences of variables as $D(S_i)$ and so the "occurs check" takes exponential time.

More efficient (even linear time) procedures for unification are now available (Martelli and Montanari [1982, 5.4]). Unfortunately, all current PROLOG implementations simply omit the "occurs check". They simply take the first variable x in $D(S_k)$ and substitute for it the first term t other than x in $D(S_k)$ in the expressions contributing x to $D(S_k)$. Thus, the implementations believe that $S = \{x, f(x)\}$ is unifiable. (They cannot actually carry out the substitution. They would try to replace x by $f(x)$ and then return to x which would again be replaced by $f(x)$ and so on forever.) Needless to say, this type of unification destroys the soundness

of the resolution method. Some protections against such incorrect deductions can be put into programs. We will discuss one in III.2 after we have more fully described the actual deduction procedure of PROLOG. Unfortunately, almost nothing the programmer can do can fully compensate for omitting the occurs check. We will, however, prove (Corollary II.8.7) that certain programs sufficient to calculate all effective functions do in fact run correctly even without the occurs check.

Exercises

1. Apply the unification algorithm to each of the following sets to find an mgu or show that none exists.
   (a) $\{P(x, y), P(y, f(z))\}$
   (b) $\{P(a, y, f(y)), P(z, z, u)\}$
   (c) $\{P(x, g(x)), P(y, y)\}$
   (d) $\{P(x, g(x), y), P(z, u, g(a)), P(a, g(a), v)\}$
   (e) $\{P(g(x), y), P(y, y), P(y, f(u))\}$
2. Apply the unification algorithm to each of the following sets to find mgu's or show that they are not unifiable.
   (a) $\{P(h(y), a, z), P(hf(w), a, w), P(hf(a), a, u)\}$
   (b) $\{P(h(y), a, z), P(hf(w), a, w), P(hf(a), a, b)\}$

13. Resolution

We now describe how to combine unification with the resolution method for propositional logic to give a proof scheme for full predicate logic. As before, we consider formulas in clausal form. Remember, however, that literals are now atomic formulas or their negations with free variables allowed. The results of §9 and §10 show that, as long as we are willing to add function symbols to our language, every sentence has an equisatisfiable version in clausal form. Note that all the variables in a sentence S are local, that is, each clause is understood as its universal closure. S is then the conjunction of the universally quantified clauses. Thus, there are no connections between the variables of distinct clauses. To reflect this syntactically, we will generally rename variables when using two clauses together so that they have no variables in common. (This procedure is called *standardizing the variables apart.*)

As in the propositional case, clauses with at most one positive literal are called *Horn clauses*. The rest of the terminology from Definition I.10.4 (or II.5.1) describing *program clauses, rules, facts* and *goals*, is also carried over intact from the propositional case. Thus, for example, a (PROLOG) *program* is a formula which contains only program clauses, that is ones with exactly one positive literal. We continue to use PROLOG notation.

Example 13.1: Consider the following list of clauses:

mother (X,Y) :- daughter(Y,X), female (X). (1)
mother (X,Y) :- son (Y,X), female (X). (2)
daughter (X,Y) :- mother (Y,X), female (X). (3)
son (X,Y) :- mother (Y,X), male (X). (4)
father (X,Y) :- son (Y,X), male (X). (5)
father (X,Y) :- daughter (Y,X), male (X). (6)
daughter (X,Y) :- father (Y,X), female(X). (7)
son (X,Y) :- father (Y,X), male (X). (8)
male (jim). (9)
male (tim). (10)
female(jane). (11)
female (pam). (12)
father (jim, tim). (13)
father (jim, pam). (14)
mother (jane, tim). (15)
mother (jane, pam). (16)

These clauses are the PROLOG versions of

$$\{\{\text{mother}(x,y), \neg\text{daughter}(y,x), \neg\text{female}(x)\},$$
$$\{\text{mother}(x,y), \neg\text{son}(y,x), \neg\text{female}(x)\},$$
$$\vdots$$
$$\{\text{mother(jane, pam)}\}\}.$$

Which are in turn the clausal forms of

$$\forall x\forall y[\text{daughter}(y,x) \wedge \text{female}(x) \rightarrow \text{mother}(x,y)]\wedge$$
$$\forall x\forall y[\text{son}(y,x) \wedge \text{female}(x) \rightarrow \text{mother}(x,y)]\wedge$$
$$\ldots\wedge$$
$$\ldots\wedge$$
$$\ldots\wedge$$
$$\text{mother(jane, pam)}.$$

Definition 13.2: Suppose that we can rename the variables of C_1 and C_2 so that they have no variables in common and are of the form $C_1' \sqcup \{P\vec{t}_1, \ldots, P\vec{t}_n\}$ and $C_2' \sqcup \{\neg P\vec{s}_1, \ldots, \neg P\vec{s}_m\}$ respectively. If σ is an mgu for $\{P\vec{t}_1, \ldots, P\vec{t}_n, P\vec{s}_1, \ldots, P\vec{s}_m\}$ then $C_1'\sigma \cup C_2'\sigma$ is a *resolvent* of C_1 and C_2. ($C_1'\sigma \cup C_2'\sigma$ is also called the *child* of the *parents* C_1 and C_2.)

Resolution proofs of C from S and resolution refutations of S in both linear and tree form are now defined as in the propositional case (Definitions I.8.4 and I.8.6) except that we use the version of the resolution rule given above and allow the premises inserted from S, or equivalently the leaves of the tree proof, to be $C\sigma$ for any renaming substitution σ and any $C \in S$. Similarly, we define $\mathcal{R}(S)$ as the closure under resolution of the set of all renamings of elements of S.

Two points should be noted in this definition of resolvent. The first is that the renaming of variables is necessary. For example, the sentence $\{\{P(x)\}, \{\neg P(f(x))\}\}$ is (unsatisfiable and) resolution refutable but the clauses cannot be unified without renaming the variables. The second point is that we cannot assume in the above definition that n or m are equal to 1 as we did in propositional logic. We must be able to eliminate several literals at once. (This aspect of the procedure is often called *factoring*.) As an example, consider $S = \{\{P(x), P(y)\}, \{\neg P(x), \neg P(y)\}\}$. It is (unsatisfiable and) resolution refutable but no resolution proof from S which eliminates only one literal at a time can produce $\square$.

Example 13.3:

(i) We can resolve

$$C_1 = \{Q(x), \neg R(y), P(x,y), P(f(z), f(z))\}$$

and

$$C_2 = \{\neg N(u), \neg R(w), \neg P(f(a), f(a)), \neg P(f(w), f(w)\}$$

to get

$$C_3 = \{Q(f(a)), \neg R(f(a)), \neg N(u), \neg R(a)\}.$$

To do this we unify $\{P(x, y), P(f(z), f(z)), P(f(a), f(a)), P(f(w), f(w))\}$ via the mgu $\{x/f(a), y/f(a), z/a, w/a\}$ and perform the appropriate substitutions and union on C_1 and C_2.

(ii) From the clauses corresponding to (3) and (16) in Example 13.1 above, we can form the resolvent {daughter(pam, jane), ¬female(pam)} by the substitution $\{X/\text{pam}, Y/\text{jane}\}$.

Example 13.4:

(i) From (a) and (b) below we wish to conclude (c):
(a) $\forall x\forall y\forall z[P(x,y) \wedge P(y,z) \rightarrow P(x,z)]$ (transitivity) and
(b) $\forall x\forall y[P(x,y) \rightarrow P(y,x)]$ (symmetry)
(c) $\forall x\forall y\forall z[P(x,y) \wedge P(z,y) \rightarrow P(x,z)]$.
In clausal form, we wish to derive C_3 from $S = \{C_1, C_2\}$ where

$$C_1 = \{\neg P(x,y), \neg P(y,z), P(x,z)\},$$
$$C_2 = \{\neg P(u,v), P(v,u)\},$$

and

$$C_3 = \{\neg P(x,y), \neg P(z,y), P(x,z)\}.$$

(Note that we have standardized the clauses of S apart.)

We exhibit a resolution tree proof with the substitutions used in the resolution displayed on the branches. For clarity, we also underline the literal on which we resolve.

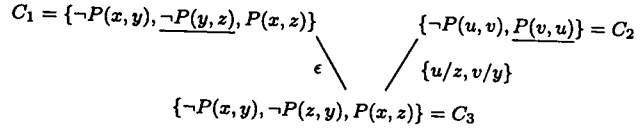


FIGURE 35

(ii) We can show that son(tim, jim) follows from the clauses in example 13.1 by the following resolution proof:

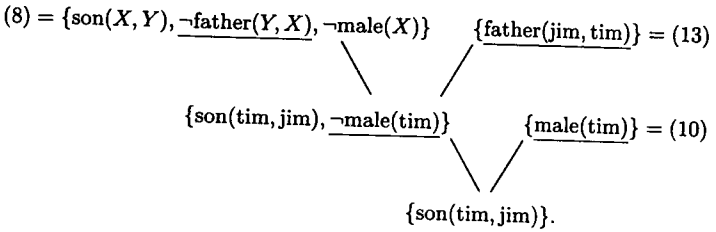


FIGURE 36

One can ask for the results gotten by resolution in the above examples from PROLOG. If one has the clauses of Example 13.1 in the database, one enters the desired result at the " ?- " prompt as "?- son(tim, jim).". One then gets the answer yes. PROLOG interprets the question as a request to prove son(tim, jim) from the database. More precisely, if positive literals $C_1, C_2, \ldots, C_n$ are entered at the "?" prompt, PROLOG tries to deduce $\square$ from the database S and the goal clause $G = \{\neg C_1, \ldots, \neg C_n\}$ which is also written :- $C_1, C_2, \ldots, C_n$. (Recall that a goal clause is one without positive literals, i.e., one of the form :- $A_1, \ldots, A_n$ where the A_i are positive. The reason for this terminology is apparent from such examples.)

If the C_i are ground, then we would expect a successful deduction of $\square$ from $S \cup \{\neg C_1, \ldots, \neg C_n\}$ to imply that all of the C_i are consequences of S. (This implication will follow from the soundness of resolution for predicate logic, Theorem 13.6 below. With some syntactic effort it can also be viewed, in the case that the C_i are ground, as a consequence of the soundness of resolution for propositional logic.) Consider, however, the meaning of entering an atomic formula with free variables such as "male(X), female(Y)" at the "?" prompt. Again PROLOG takes this as a request to prove $\square$ from S and the added goal clause $\{\neg\text{male}(X), \neg\text{female}(Y)\}$. Success here only means

that from S we can conclude $\neg\forall X \forall Y[\neg\text{male}(X) \vee \neg\text{female}(Y)]$ as the clause $\{\neg\text{male}(X), \neg\text{female}(Y)\}$ means $\forall X \forall Y[\neg\text{male}(X) \vee \neg\text{female}(Y)]$. That is, we conclude $\exists X \exists Y[\text{male}(X) \wedge \text{female}(Y)]$. What PROLOG actually does is return a substitution, say $X = \text{jim}$, $Y = \text{jane}$ which demonstrates the truth of the conclusion $\exists X \exists Y[\text{male}(X) \wedge \text{female}(Y)]$ based on the information in S – i.e., $\{\neg\text{male(jim)}, \neg\text{female(jane)}\}$ is inconsistent with S. Of course, in applications it is almost always this correct answer substitution that we really want, not the mere assertion that $\exists X \exists Y[\text{male}(X) \wedge \text{female}(Y)]$.

Definition 13.5: If P is a program and $G = \{\neg A_1, \ldots, \neg A_n\}$ a goal clause, we say that the substitution θ (for the variables of G) is a *correct answer substitution* if $(A_1 \wedge A_2 \wedge \ldots \wedge A_n)\theta$ is a logical consequence of P (that is, of its universal closure).

Note that, by an application of Herbrand's theorem given in Exercise 10.5, if $P \cup \{G\}$ is unsatisfiable then there is a correct answer substitution which is a ground substitution. That one can always find such a substitution via resolution is essentially a statement of the completeness theorem. We will return to the issue of completeness of resolution after proving its soundness. We discuss general resolution methods in this and the next sections and leave the specific search procedures used in PROLOG to Chapter III.

Theorem 13.6 (Soundness of Resolution): *If $\square \in \mathcal{R}(S)$ then S is unsatisfiable.*

Proof: Suppose, for the sake of a contradiction, that $\mathcal{A} \models S$. Let the notation for a resolution be as in Definition 13.2. It suffices to show that if $\mathcal{A} \models C_1$, C_2 and C is a resolvent of C_1, C_2 then $\mathcal{A} \models C$, i.e., $\mathcal{A} \models C\tau$ for every ground substitution τ. (If so, we could show by induction that $\mathcal{A} \models C$ for every $C \in \mathcal{R}(S)$. As $\mathcal{R}(S)$ contains $\square$, we would have the desired contradiction.) The only point to notice here is that if $\mathcal{A} \models C_i$ then $\mathcal{A} \models C_i\sigma_i$ for any σ_i as the C_i are open. For every ground instantiation τ of the variables of $C = C_1'\sigma \cup C_2'\sigma$ we can argue as in the propositional case. (See Lemma I.8.12 and Theorem I.8.11.) As, for each ground instantiation τ, either $C_1'\sigma\tau$ or $C_2'\sigma\tau$ is true in $\mathcal{A}$ (depending on whether the literal resolved on is true in $\mathcal{A}$ or not and in which of the $C_i'\tau$ it appears positively), then so is their union $C\tau$. $\square$

We now want to prove the completeness of the resolution method for predicate logic by reducing it to the case of propositional logic. We begin with two lemmas. The first (Lemma 13.7) relates single resolutions in propositional and predicate logic. The second (Lemma 13.8) extends the correspondence to resolution proofs. This lemma (which is often called the lifting lemma as it "lifts" proofs in propositional logic to ones in predicate logic) will be quite useful in the analysis of restricted versions of resolution in §14 and III.1. The special case of proofs of $\square$ will be especially useful and is singled out as Corollary 13.9.

Lemma 13.7: *If C_1' and C_2' are ground instances (via the substitutions θ_1 and θ_2) of C_1 and C_2 respectively and C' is a 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 (via $\theta_1\theta_2$ if C_1 and C_2 have no variables in common).*

Proof: As the resolution rule allows us to rename the variables in C_1 and C_2 as part of the resolution, we may as well assume that they (and so also θ_1 and θ_2) have no variables in common. As $C_1' = C_1\theta_1$ and $C_2' = C_2\theta_2$ are resolvable, say on the ground literal $P(t_1, \ldots, t_n)$, there are sets of literals

$$A_1 = \{P(\vec{s}_{1,1}), \ldots, P(\vec{s}_{1,n_1})\} \subseteq C_1$$

and

$$A_2 = \{\neg P(\vec{s}_{2,1}), \ldots, \neg P(\vec{s}_{2,n_2})\} \subseteq C_2$$

which become unified to $\{P(t_1, \ldots, t_n)\}$ and $\{\neg P(t_1, \ldots, t_n)\}$ by θ_1 and θ_2 respectively. As the sets of variables in θ_1 and θ_2 are disjoint, $\theta_1\theta_2$ unifies both sets of literals A_1 and A_2 simultaneously. Thus, by the definition of resolution for the predicate calculus (Definition 13.2), $C = ((C_1 - A_1) \cup (C_2 - A_2))\sigma$ is a resolvent of C_1 and C_2 where σ is the mgu for

$$\{\neg P(\vec{s}_{1,1}), \ldots, \neg P(\vec{s}_{1,n_1})\} \cup \{\neg P(\vec{s}_{2,1}), \ldots, \neg P(\vec{s}_{2,n_2})\}$$

given by the unification algorithm. The only point left to verify is that C' is an instance of C. We claim that $C' = C\theta_1\theta_2$. Note that as $\theta_1\theta_2$ unifies $\neg A_1 \cup A_2$, the special property of the mgu given by our algorithm (Theorem 12.5) guarantees that $\sigma\theta_1\theta_2 = \theta_1\theta_2$. Thus

$$\begin{aligned}
C\theta_1\theta_2 &= ((C_1 - A_1) \cup (C_2 - A_2))\sigma\theta_1\theta_2 \\
&= ((C_1 - A_1) \cup (C_2 - A_2))\theta_1\theta_2 \\
&= (C_1\theta_1 - A_1\theta_1) \cup (C_2\theta_2 - A_2\theta_2) \quad \text{(by disjointness of variables)} \\
&= (C_1' - \{P(t_1, \ldots, t_n)\}) \cup (C_2' - \{\neg P(t_1, \ldots, t_n)\}) \\
&= C' \qquad \text{(by definition).} \qquad \square
\end{aligned}$$

Lemma 13.8 (Lifting Lemma): *Let S be a formula in a language $\mathcal{L}$ and let S' be the set of all ground instances of clauses in S in the Herbrand universe for $\mathcal{L}$. If T' is a resolution tree proof of C' from S', then there is a clause C of $\mathcal{L}$, a resolution tree proof T of C from S and a substitution θ such that $T\theta = T'$ (i.e., T and T' are labelings of the same tree and $C_i\theta = C_i'$ for C_i, C_i' the respective labels of each node of the common tree underlying T and T'. Thus, in particular, $C' = C\theta$.) Moreover, if the leaves of T' are labeled R_i and each R_i is an instance of an S_i in S, then we may arrange it so that the corresponding leaves of T are labeled with renamings of the appropriate S_i.*

Proof: We proceed by induction on the depth of resolution tree proofs from S'. For the base case of elements R_i of S', the lemma is immediate as each such R_i is a substitution instance of an element of S. Consider now a proof of C' from S' of depth $n+1$. It consists of two proofs, T_1' and T_2' (of depth $\leq n$) of ground clauses C_1', C_2' from S' and a final resolution of C_1' and C_2' to get C'. Suppose that $P(t_1, \ldots, t_n) \in C_1'$, $\neg P(t_1, \ldots, t_n) \in C_2'$ and that we resolved on this literal to get

$$C' = C_1' \cup C_2' - \{P(t_1, \ldots, t_n), \neg P(t_1, \ldots, t_n)\}.$$

By induction, we have predicate clauses C_1 and C_2, proof trees T_1 and T_2 of C_1 and C_2 and substitutions θ_1 and θ_2 such that $T_i\theta_i = T_i'$. (The leaves of T_i are also labeled appropriately by induction.) At the cost perhaps of renaming variables in T_1 and T_2, we may assume that θ_1 and θ_2 have no variables in common. (As the resolution rule allows for arbitrary renamings of the parents, the T_i remain resolution proofs. As our lemma only calls for the leaves to be labeled with some renamings of the given clauses from S, this renaming does not alter the fact that we have the leaves appropriately labeled.) We now apply Lemma 13.7 to get a resolvent C of C_1 and C_2 with $C' = C\theta_1\theta_2$. We can now form a resolution tree proof T from S of C by combining T_1 and T_2. As θ_1 and θ_2 are disjoint, $T\theta_1\theta_2$ restricted to T_1 and T_2 simply gives us back $T_1\theta_1$ and $T_2\theta_2$. Of course, on the remaining node C of T we have $C\theta_1\theta_2 = C'$. Thus T is the required predicate logic resolution proof from S' of C and $\theta_1\theta_2$ is the substitution required in our lemma. $\square$

Corollary 13.9: *If T' is a resolution tree proof of $\square$ each of whose leaves L_i is labeled with a ground instance R_i of the clause S_i, then there is a relabeling T of the underlying tree of T' which gives a resolution proof of $\square$ each of whose leaves L_i is labeled with (a renaming) of S_i.*

Proof: This is simply the special case of the theorem with $C' = \square$. The only point to notice is that the only clause C which can have $\square$ as a substitution instance is $\square$ itself. $\square$

Theorem 13.10 (Completeness of Resolution): *If S is unsatisfiable then $\square \in \mathcal{R}(S)$.*

Proof: Let S' be the set of all ground instances of clauses in S in the Herbrand universe for the language $\mathcal{L}$ of S. By one of the consequences (Theorem 10.6) of Herbrand's theorem, S and S' are equisatisfiable. Thus if we assume that S is unsatisfiable then so is S'. By the completeness of resolution for propositional logic (Theorem I.8.15 or I.8.22) we then know that $\square \in \mathcal{R}_p(S')$ where we use $\mathcal{R}_p$ to represent the resolution procedure in propositional logic. (As usual we consider the atomic formulas as propositional letters in this situation.) The completeness of resolution for predicate logic (i.e., $\square \in \mathcal{R}(S)$ if S is unsatisfiable) is now immediate from Corollary 13.9. $\square$

Exercises

1. Find resolvents for the following:
   a) $\{P(x,y), P(y,z)\}, \{\neg P(u, f(u))\}$
   b) $\{P(x,x), \neg R(x, f(x)\}, \{R(x,y), Q(y,z)\}$
   c) $\{P(x,y), \neg P(x,x), Q(x, f(x), z)\}, \{\neg Q(f(x), x, z), P(x,z)\}$.
2. Translate the following sentences into predicate logic, put in clausal form and prove by resolution:
   a) Suppose all barbers shave everyone who does not shave himself. Moreover, no barber shaves anyone who shaves himself. Conclude that there are no barbers.
   b) Suppose John likes anyone who doesn't like himself. Conclude that it is not the case that John likes no one who likes himself.
3. Suppose I believe the following:
   (i) There exists a dragon.
   (ii) The dragon either sleeps in its cave or hunts in the forest.
   (iii) If the dragon is hungry, it cannot sleep.
   (iv) If the dragon is tired, it cannot hunt.

   Translate (i)–(iv) into predicate logic. Use resolution to answer the following questions:
   (a) What does the dragon do when it is hungry?
   (b) What does the dragon do when it is tired?

   (Assume that if X cannot do Y then X does not do Y.)
4. (a) Express the following in clausal form:
   Everyone admires a hero.
   A failure admires everyone.
   Anyone who is not a hero is a failure.
   (b) Use resolution to find X and Y who admire each other.
5. Give a resolution refutation of the following set of clauses. Indicate the literals being resolved on and the substitutions being made to do the resolutions:
   1) $\{P(a, x, f(y)), P(a, z, f(h(b))), \neg Q(y,z)\}$
   2) $\{\neg Q(h(b), w), H(w,a)\}$
   3) $\{\neg P(a, w, f(h(b))), H(x,a)\}$
   4) $\{P(a, u, f(h(u))), H(u,a), Q(h(b), b)\}$
   5) $\{\neg H(v,a)\}$.

6. Consider the following sentences.
   1) All the stockholders who will have real estate partners will vote against the proposal but no others.
   2) john and jim (and similarly mary and jane) will form real estate partnerships if some bank will give them a loan unless none of the lawyers can get them the needed zoning variance.
   3) No banker will give a loan to form a real estate partnership without a lawyer's getting the needed zoning variances. With such an assurance they require only a good appraisal to agree to the loan.
   4) john and jane are stockholders.
   5) joyce is a lawyer who can get zoning approval for anyone with enough money.
   6) john is immensely wealthy and his and jim's land has been given a good appraisal.

   Translate these sentences into predicate logic, put them in clausal form and use resolution to deduce that someone will vote against the proposal. Who is it?

14. Refining Resolution: Linear Resolution

Systematic attempts at generating resolution proofs are often redundant and inefficient. As in the propositional case, we can impose various restrictions to make the procedure more efficient. The analogous procedures (to those considered in I.9) are covered in the exercises. We now wish to analyze, in the setting of full predicate logic, the refinement dealt with in I.10 for propositional Horn clauses: linear resolution. The plan here is to try to proceed via a linear sequence of resolutions rather than a branching tree. We carry out a sequence of resolutions each of which (after the first) must have one of its parents the child of the one previously carried out.

Definition 14.1: Let C be a clause and S a formula.

(i) A *linear deduction of C from S* is a sequence of pairs of clauses $\langle C_0, B_0 \rangle, \ldots, \langle C_n, B_n \rangle$ such that C_0 and each B_i are either renaming substitutions of elements of S or some C_j for $j < i$; each C_{i+1}, $i \leq n$, is a resolvent of C_i and B_i; and $C_{n+1} = C$.

(ii) *C is linear deducible from S*, $S \vdash_{\mathcal{L}} C$, if there is a linear deduction of C from S. There is a *linear resolution refutation* of S if $\square$ is linearly deducible from S. $\mathcal{L}(S)$ is the set of all clauses linearly deducible from S.

We picture a linear resolution as follows:

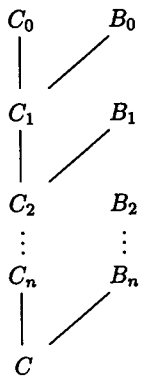


FIGURE 37

as in the following familiar example (I.10.2):

$$S = \{A_1, A_2, A_3, A_4\}, \quad A_1 = \{p(x), q(x)\}, \quad A_2 = \{p(x), \neg q(x)\},$$
$$A_3 = \{\neg p(x), q(x)\}, \quad A_4 = \{\neg p(x), \neg q(x)\}.$$

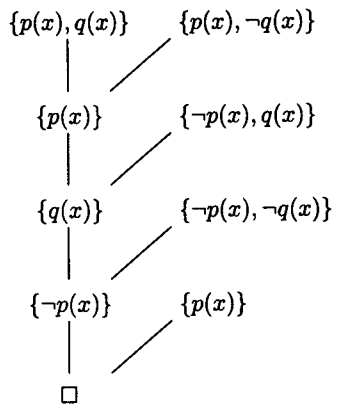


FIGURE 38

Definition 14.2: In this context, the elements of S are frequently called *input clauses*. The C_i are called *center clauses* and the B_i *side clauses*.

If we extend the parent–child terminology by defining the *ancestors* of a clause C in a resolution proof of C from S to be the clauses above it in the tree proof, we can rephrase the definition of linear deduction by saying that each C_i is resolved against an input clause or one of its own ancestors to produce C_{i+1}.

We want to prove that linear resolution is complete. (As it is a restriction of the sound method of full resolution its soundness is, as usual, automatic.) For the sake of the eventual induction argument as well as applications to the special case of proofs in PROLOG, we will actually prove a stronger result which gives us some control over the starting point of the linear proof (C_0 in the above notation).

Definition 14.3: $U \subseteq S$ is a *set of support* for S if $S - U$ is satisfiable. We say that a linear resolution proof $\langle C_i, B_i \rangle$, $i \leq n$, of C from S *has support* U if $C_0 \in U$.

The intuition here is that we will consider a formula $S \in$ UNSAT. In this case the "cause" of the unsatisfiability has been isolated in U (which "supports" the fact that $S \in$ UNSAT).

We can now state a strengthened version of the completeness theorem.

Theorem 14.4: *If $S \in$ UNSAT and U is a set of support for S, then there is a linear refutation of S with support U.*

Our first step is to reduce the proof of Theorem 14.4 to the case in which every nonempty subset of S is a set of support for S.

Definition 14.5: S is *minimally unsatisfiable* if it is unsatisfiable but every proper subset is satisfiable, i.e., $\{C\}$ is a set of support for S for every $C \in S$.

Lemma 14.6: *If $S \in$ UNSAT then there is a minimally unsatisfiable $S' \subseteq S$. Moreover, if U is a set of support for S, $U \cap S'$ is one for S'.*

Proof: By compactness, some finite subset of S is unsatisfiable. If S' is an unsatisfiable subset of S with the least possible number of clauses, S' is certainly a minimally unsatisfiable subset of S. Let U be any set of support for S. If $U \cap S' = \emptyset$, S' would be contained in the satisfiable set $S - U$ for a contradiction. Thus $S' - (S' \cap U)$ is a proper subset of S' and so, by the minimality of S', is satisfiable. $\square$

Proof (of Theorem 14.4)**:** Our plan now is once again to reduce the proof to the case of propositional logic. (We supply a proof for this case below.) As in the case of general resolution, we apply Herbrand's theorem. If S is unsatisfiable and has support U, so is S', the set of all ground instances of elements of S, and it has support U', the set of all ground instances of elements of U. We wish to show that any linear resolution proof T' of $\square$ from S' with support U' lifts to one from S with support U. This is immediate from Corollary 13.9 to the lifting lemma. The lifting lemma preserves the shape of the resolution tree and so lifts linear proofs to linear proofs. It also lifts instances R_i of clauses S_i on the leaves of the tree to

(renamings of) S_i. Thus if the clause C'_0 of the proof T' is in U' and so is an instance of a clause C in U, then it lifts to a (renaming of) the same clause C. □

We now turn to the proof of the strengthened completeness theorem for linear resolution in the propositional calculus. The proof here is more difficult than for semantic or ordered resolution but is connected with the set of support and lock resolution methods considered in the exercises for I.9.

Proof (of the propositional version of Theorem 14.4): By Lemma 14.6, it suffices to consider only those S which are minimally unsatisfiable. (Any linear resolution refutation of $S' \subseteq S$ with support $U \cap S'$ is one of S with support U by definition.) We proceed by induction on $E(S)$ = the *excess literal number of* S, that is, the number of occurrences of literals in all clauses of S minus the number of clauses in S. (Note that we need S to be finite to even define the excess literal number.) We in fact prove by induction that, for any $C \in S$, there is a linear refutation of S which begins with C, i.e. $C = C_0$ in the proof tree. At the bottom, if $E(S) = -1$, $\square \in S$ and there is nothing to prove. Suppose now that $E(S) \geq 0$.

Case 1. C is a unit clause, i.e., it contains exactly one literal ℓ. There must be a clause $C' \in S$ with $\bar{\ell} \in C'$ as otherwise any assignment satisfying $S - C$ (which is satisfiable by the minimality of S) could be extended to one satisfying S by adding on ℓ. Note that $\ell \notin C'$ for if it did, C' would be a tautology and S would not be minimally unsatisfiable contrary to our assumption. Thus $C' - \{\bar{\ell}\}$ is in S^ℓ by the definition of S^ℓ (Definition I.8.16). If $C' = \{\bar{\ell}\}$, we are done as we can simply resolve C and C' to get $\square$. Suppose then that $C' = \{\bar{\ell}, \dots\}$ has more than one literal. As $S \in$ UNSAT, $S^\ell \in$ UNSAT by Lemma I.8.19. Each clause removed from S in forming S^ℓ has at least one literal (ℓ) (again by definition). Thus their removal cannot increase the excess literal number. On the other hand, at least C' loses one literal ($\bar{\ell}$) in its transition to S^ℓ. Thus $E(S^\ell) < E(S)$.

We next claim that S^ℓ is also minimally unsatisfiable: Suppose $D \in S^\ell$ but $S^\ell - \{D\}$ is unsatisfiable. Now, by the definition of S^ℓ, $D \in S$ or $D \cup \{\bar{\ell}\} \in S$ and in either case $\ell \notin D$. Let D' represent whichever clause belongs to S. We know, by the minimal unsatisfiability of S, that $S - \{D'\}$ is satisfiable. Let $\mathcal{A}$ be an assignment satisfying it. As $C = \{\ell\} \in S - \{D'\}$, $\mathcal{A} \models \ell$. Consider now any $F \in S^\ell - \{D\}$ and the associated $F' \in S - \{D'\}$. As $\mathcal{A} \models \ell$ and $\mathcal{A} \models F'$, $\mathcal{A} \models F$ in either case of the definition of F'. (F' is defined from F as D' was defined from D.) Thus $\mathcal{A} \models S^\ell - \{D\}$ contrary to our assumption.

Our induction hypothesis now gives us a linear resolution deduction of $\square$ from S^ℓ starting with $C' - \{\bar{\ell}\} : \langle C_0, B_1 \rangle, \dots, \langle C_n, B_n \rangle$ with $C_0 = C' - \{\bar{\ell}\}$. Each B_i is a member of S^ℓ or is C_j for some $j < i$ and C_n, B_n resolve to $\square$. We construct a new proof $\langle D_j, A_j \rangle$ in segments with the i^{th} one ending with $D_k = C_i$. We begin by setting $D_0 = \{\ell\} = C$ and $A_0 = C'$. Of course, they can be resolved to get $D_1 = C_0$. Now

we proceed by induction. Suppose we have A_j, $j < k$ and $D_k = C_i$. If $B_i = C_j$ for some $j < i$, we let $A_k = C_j$ (which by induction is a previous D_m) and resolve to get $D_{k+1} = C_{k+1}$. Otherwise, $B_i \in S^\ell$ and we have two cases to consider. If $B_i \in S$ we set $A_k = B_i$ and resolve to get $D_{k+1} = C_{i+1}$. If $B_i \notin S$ then $B_i \cup \{\bar{\ell}\} \in S$. We set $A_k = B_i \cup \{\bar{\ell}\}$ and resolve to get $D_{k+1} = C_{i+1} \cup \{\bar{\ell}\}$. In this case, we set $A_{k+1} = \{\ell\}$ and resolve to get $D_{k+2} = C_{i+1}$ and so continue the induction. As $\{\ell\} = D_0$, we now have a linear resolution refutation of S as required.

Case 2. $C = \{\ell, \dots\}$ has more than one literal. Now consider $S^{\bar{\ell}}$. As above, it is minimally unsatisfiable and has lower excess literal number than S. We thus have, by induction, a linear resolution deduction of $\square$ from $S^{\bar{\ell}}$ starting with $C - \{\ell\}$. If we add on ℓ to every center clause and to any side clause which is in $S^{\bar{\ell}}$ but not S, we get a linear proof of $\{\ell\}$ from S starting with C. Consider now $S' = S - \{C\} \cup \{\{\ell\}\}$. It too is unsatisfiable. (Any assignment satisfying it satisfies S.) As C has more than one literal, $E(S') < E(S)$. Now as $\square \notin S'$, for any $S'' \subseteq S'$, $E(S'') \leq E(S')$. If we take $S'' \subseteq S'$ to be minimally unsatisfiable we have, by induction, a linear resolution proof of $\square$ from $S'' \subseteq S \cup \{\{\ell\}\}$ beginning with $\{\ell\}$. (Note that $S' - \{\ell\} = S - \{C\}$ is satisfiable by the minimal unsatisfiability of S. Thus, any unsatisfiable subset S'' of S' must contain $\{\ell\}$.) Attaching this proof to the end of the one of $\{\ell\}$ from S gives the desired linear refutation of S starting with C. □

Further refinements of general resolution are possible. Some of these (such as ordered linear resolution) are considered in the exercises. Instead of pursuing the general problem further, we turn our attention to the special case of resolution from Horn clauses — that is, the deduction mechanisms for PROLOG.

Exercises

1. Following the model in I.9 for ordered resolution in propositional logic, give a complete definition of an ordered resolution and an ordered resolution refutation for the predicate calculus in which you index the predicate symbols of the language.
2. State and prove the soundness theorem for ordered resolution for predicate logic.
3. State and prove the completeness theorem for ordered resolution for predicate logic.
4. Give the definitions and prove the soundness and completeness theorems for the predicate version of F–resolution (Exercise I.9.4).
5. Give the definitions and prove the soundness and completeness theorems for the predicate version of lock resolution (Exercise I.9.1).

Suggestions for Further Reading

To see how predicate logic got its start, read Frege [1879] in van Heijenoort [1967, 2.1].

For more on the predicate logic versions of tableaux, axioms and rules of inference, resolution, natural deduction and sequents, see the references to these topics at the end of Chapter I.

The basic development of model theory (beyond the few theorems given here) can be found in Chang and Keisler [1973, 3.4] or any of the other texts in list 3.4 of the bibliography.

To see where Herbrand universes and unification come from, read the first few pages of Herbrand [1930] in van Heijenoort [1967, 2.1] or Herbrand [1971, 2.3].

To see Herbrand's theorem used as a basis for the exposition of logic, see Chang and Lee [1972, 5.7]. Many varieties of resolution can be found there; see also Loveland [1978, 5.7] and Bibel [1982, 5.7].

The standard text on the theory of PROLOG is Lloyd [1987, 5.4] which also has an extensive bibliography. Other suggestions for reading about PROLOG can be found at the end of Chapter III.

III PROLOG

1. SLD–Resolution

In this chapter we will consider the full PROLOG language for logic programming in predicate logic. Much of the basic terminology is simply the predicate logic version of that introduced in I.10. We will, nonetheless, restate the basic definitions in a form suitable for resolution theorem proving in the predicate calculus. PROLOG employs a refinement of linear resolution but we have made the presentation independent of the (rather difficult) completeness theorem for linear resolution (Theorem II.14.4). We do, however, assume familiarity with the definitions for linear resolution in Definitions II.14.1–3. Thus our proofs will be based on the analysis of the propositional version of PROLOG discussed in I.10, together with Herbrand's theorem (II.10.4) and the reduction of predicate logic to propositional logic that it entails. At times when a knowledge of II.14 would illuminate certain ideas or simplify proofs, we mark such alternate results or proofs with an *.

Recall from II.5.1 that a PROLOG program P is a set of *program* clauses, i.e., ones with precisely one positive literal. We ask questions by entering a sequence of positive literals $A_1, \ldots, A_n$ at the " ?- " prompt. The PROLOG interpreter answers the question by converting our entry into a goal clause $G = \{\neg A_1, \ldots, \neg A_n\}$ and asking if $P \cup \{G\}$ is unsatisfiable. We will now describe the way in which PROLOG discovers if $P \cup \{G\}$ is unsatisfiable. Our starting point is the method of linear resolution introduced in I.10 for propositional logic and proved complete for predicate logic in II.14. We next restrict ourselves to an input version of linear resolution. Although this is not in general a complete version of resolution (as can be seen from the example following Definition II.14.1), it turns out to be complete in the setting of PROLOG. For the remainder of this section P is a PROLOG program and G a goal clause.

Definition 1.1: A linear resolution proof $\langle G_0, C_0\rangle, \ldots, \langle G_n, C_n\rangle$, G_{n+1} is a *linear input*, or LI, *(resolution) refutation* of $P \cup \{G\}$ if $G_0 = G$, $G_{n+1} = \square$, each G_i is a goal clause and each C_i is a (renaming of a) clause in P.

***Theorem 1.2:** *If* $P \cup \{G\} \in$ UNSAT, *there is a linear resolution refutation of* $P \cup \{G\}$.

***Proof:** As every PROLOG program is satisfiable (Exercise II.5.6), $\{G\}$ is a set of support for $P \cup \{G\}$. By the completeness theorem for linear resolution (Theorem II.14.4) there is a linear refutation $\langle G_0, C_0\rangle, \ldots, \langle G_n, C_n\rangle$

with $G_0 = G$. We claim that every G_i is a goal clause and, modulo renaming, every $C_i \in P$. Proceeding by induction, the only point to notice is that we cannot resolve two goal clauses and so the next C_i must be from P while the result of the resolution of a goal clause and a program clause is again a goal clause or $\square$. $\square$

We now know the general format of resolution proofs for PROLOG: linear input resolution. Before continuing to examine the additional refinements implemented in PROLOG, we should note (and this is usually true of implementations of all resolution methods) that when $G = \{\neg A_0, \ldots, \neg A_n\}$ and $C = \{B, \neg B_0, \ldots, \neg B_m\}$ are resolved, say on $A_i = B$ via mgu θ, the interpreter does not check to see if perhaps some resulting duplications should be eliminated (e.g., $A_j\theta = A_\ell\theta$ or $A_j\theta = B_\ell\theta$ or $B_k\theta = B_\ell\theta$) . It simply replaces $\neg A_i$ by $\neg B_0, \ldots, \neg B_m$ and then applies θ to each term. It does no further simplification. To understand the actual implementations, we should therefore think of clauses (as the machine does) not as sets of literals but as *ordered clauses*, i.e., sequences of literals. A resolution as above then inserts the literals $\neg B_0, \ldots, \neg B_m$ in place of $\neg A_i$ and applies θ to each literal in the sequence to get the next (ordered) goal clause. This ordering of clauses does not cause any serious changes. We embody it in the following definition and lemma.

Definition 1.3:

(i) If $G = \{\neg A_0, \ldots, \neg A_n\}$ and $C = \{B, \neg B_0, \ldots, \neg B_m\}$ are ordered clauses and θ is an mgu for A_i and B, then we can perform an *ordered resolution* of G and C on the literal A_i. The *(ordered) resolvent* of this resolution is the ordered clause $\{\neg A_0, \ldots, \neg A_{i-1}, \neg B_0, \ldots, \neg B_m, \neg A_{i+1}, \ldots, \neg A_n\}\theta$.

(ii) If $P \cup \{G\}$ is given as a set of ordered clauses, then a *linear definite* or LD*-refutation* of $P \cup \{G\}$ is a sequence $\langle G_0, C_0\rangle, \ldots, \langle G_n, C_n\rangle$ of ordered clauses G_i, C_i in which $G = G_0$, $G_{n+1} = \square$, each G_i is an ordered goal clause, each C_i is a renaming of an element of P containing only variables which do not appear in G_j for $j \leq i$ or C_k for $k < i$ and each G_{i+1} $(0 \leq i \leq n)$ is an ordered resolvent of G_i and C_i. If C_n is not $\square$, we call the sequence, as usual, an LD*-resolution proof*.

Note that this method does not employ the strategy of collapsing literals. We resolve on one literal from each clause and remove only these two literals from the resolvent.

Lemma 1.4 (Completeness of LD–resolution): *If $P \cup \{G\}$ is an unsatisfiable set of ordered clauses, then there is an* LD*–refutation of $P \cup \{G\}$ beginning with G.*

Proof: Consider all (ordered) ground instances $P' \cup G'$ of the (ordered) clauses in $P \cup \{G\}$ in the appropriate Herbrand universe. By Herbrand's theorem (Corollary II.10.6) $P' \cup G'$ is unsatisfiable. By the compactness

theorem (II.7.9), some finite subset of $P' \cup G'$ is unsatisfiable. As all sets of program clauses are satisfiable (Exercise II.5.6), any such subset must include elements of G', i.e., instances of G. Let $P'' \cup G''$ be an unsatisfiable subset of $P' \cup G'$ of minimal size. By minimality there is a $G_0'' \in G''$ such that $P'' \cup G'' - \{G_0''\}$ is satisfiable. By Lemma I.10.11, there is then an LD–resolution refutation of $P'' \cup G''$ starting with G_0''. By Lemma 1.5 below this can be lifted to the desired LD–refutation of $P \cup G$. $\square$

***Proof:** Let P' and $\{G'\}$ be the sets of unordered clauses corresponding to P and $\{G\}$ respectively. The proof proceeds by a simple induction on the length of the LI–refutation of $P' \cup \{G'\}$. Note, however, that one LI–resolution may be replaced by a sequence of LD–resolutions to compensate for the collapsing of literals allowed in LI–resolution. We leave the details as Exercises 1–2. $\square$

Lemma 1.5: *The lifting lemma holds for* LD*–resolution proofs. More precisely* II.13.7 *holds for ordered resolutions;* II.13.8 *holds for* LD*–resolution proofs; and* II.13.9 *holds for* LD*–resolution refutations.*

Proof: The proofs are essentially the same as in II.13. The lifting of a single resolution (II.13.7) is, however, somewhat simpler in that no collapsing of literals occurs here (and so the parameters n_1 and n_2 are both equal to 1 in the proof). In the proof of the lifting lemma itself, we note that, for linear resolutions, an induction on the depth of the tree is the same as one on the length of the proof. The leaves of the tree are the starting clause and the side clauses. We leave the details of rewriting the proofs of II.13 in this setting as Exercises 3–5. $\square$

Our next task is to describe how, in an LD–resolution proof, we should choose the literal of G_i on which to resolve. The *selection rule* used in essentially all implementations of PROLOG is to always resolve on the first, i.e., the leftmost, literal in G_i. The literals in the resolvent deriving from C_i are then always inherited with their original order and put to the left of all of the clauses coming from G_i. We call this an SLD*–resolution*. (The S stands for *selection*.) More generally, we can consider any selection rule R, i.e., any function choosing a literal from each ordered goal clause.

Definition 1.6: A *selection rule* R is simply a function that chooses a literal $R(C)$ from every nonempty ordered clause C. An SLD*–refutation of $P \cup \{G\}$ via R* is an LD–resolution proof $\langle G_0, C_0\rangle, \ldots, \langle G_n, C_n\rangle$ with $G_0 = G$, $G_{n+1} = \square$ in which $R(G_i)$ is the literal resolved on at step i of the proof. (If no R is mentioned we assume that the standard one of choosing the leftmost literal is intended.)

Our next goal is to prove a completeness theorem for SLD–refutations. We first give a simple proof along the lines of our previous arguments that uses a version of the lifting lemma for SLD–proofs. On the propositional level, the heart of this proof is essentially that of Theorem I.10.13,

the completeness of SLD–resolution for propositional logic. The difficulty with lifting that result directly to predicate logic is that the lifting lemma does not apply directly to SLD–proofs with arbitrary selection rules. The problem is that the rule may choose a literal out of the predicate lifting of a given clause which is not the lifting of the literal chosen by the rule from the ground instance of the given clause. However, this problem does not arise for a wide class of selection rules, including the standard one of always choosing the leftmost literal. So in the case in which we are interested for implementations of PROLOG, we can directly lift the completeness theorem.

Definition 1.7: A selection rule R is *invariant* if, for every (ordered) clause C and every substitution θ, $R(C\theta) = (R(C))\theta$.

Note that the standard selection rule is obviously invariant.

Theorem 1.8 (Completeness of SLD–refutations): *If $P \cup \{G\} \in$ UNSAT, there is an SLD–resolution refutation of $P \cup \{G\}$ via R for any selection rule R.*

Proof (for invariant selection rules): We argue exactly as in the proof of Lemma 1.4 except that we apply Theorem I.10.13 in place of Lemma I.10.9. We then apply the lifting lemma for SLD–resolutions with an invariant selection rule (Exercise 6). □

We could now supply a direct but fairly complicated proof of Theorem 1.8 for arbitrary selection rules. It is somewhat simpler to instead prove a lemma asserting an independence result: given an LD–refutation starting from G, we can find an SLD one via any selection rule R. The general form of Theorem 1.8 would then follow directly from Lemma 1.4. The proof of this independence result is itself somewhat technical and we postpone it to Lemma 1.12.

We now know what the PROLOG interpreter does when a question is entered as "?- $A_1, \ldots, A_n$.". It searches for an SLD–resolution proof of $\square$ from the current program P and the goal clause $G = \{\neg A_1, \ldots, \neg A_n\}$. Before analyzing the search method for finding such a proof, let us consider what happens at the end. If all attempts at finding a proof fail PROLOG answers "no". If a proof is found, PROLOG gives us an *answer substitution*, that is a substitution for the variables in G. In fact, if the proof found by the interpreter is $\langle G_0, C_0\rangle, \ldots, \langle G_n, C_n\rangle$ with mgu's $\theta_0, \ldots, \theta_n$ then it gives us the answer substitution $\theta = \theta_0 \ldots \theta_n$ restricted to the variables of G. Most importantly, these are always correct answer substitutions, i.e., $(A_1 \wedge \ldots \wedge A_n)\theta$ is a logical consequence of P.

Theorem 1.9 (Soundness of implementation): *If θ is an answer substitution given by an SLD–refutation of $P \cup \{G\}$ (via R for any R) then θ is correct, i.e., $(A_1 \wedge \ldots \wedge A_n)\theta$ is a logical consequence of P.*

Proof: We proceed by induction on the length of the SLD–refutation. For the base case of a refutation of length one, $G = G_0 = \{\neg A\}$ and $C_0 = \{B\}$ are singletons with θ an mgu for $\{A\}$ and $\{B\}$. As $B \in P$, it is a logical consequence of P as is its substitution instance $B\theta$. As θ is a unifier, $B\theta = A\theta$ which is then also a logical consequence of P as required. Suppose now that $G = \{\neg A_0, \ldots, \neg A_n\}$ and $P \cup \{G\}$ has an SLD–refutation of length $n+1$ starting with $G_0 = G$ and $C_0 = \{B, \neg B_0, \ldots, \neg B_m\}$ and a resolution on $\neg A_i$ with mgu θ_0. The resolvent $G_1 = \{\neg A_0, \ldots, \neg A_{i-1}, \neg B_0, \ldots, \neg B_m, \neg A_{i+1}, \ldots, \neg A_n\}\theta_0$ has an SLD–refutation from P of length n with mgu $\theta' = \theta_1 \ldots \theta_n$. Thus, by the induction hypothesis, $G_1\theta_0\theta'$ is a logical consequence of P. Let $\theta = \theta_0\theta'$. As $C_0 \in P$, $C_0\theta$ is also a logical consequence of P. Now $C_0\theta$ is equivalent to $(B_0\theta \wedge \ldots \wedge B_m\theta) \rightarrow B\theta$. So by propositional logic, $\{\neg A_0\theta, \ldots, \neg A_{i-1}\theta, \neg B\theta, \neg A_{i+1}\theta, \ldots, \neg A_n\theta\}$ is then also a consequence of P. As $\theta = \theta_0\theta'$, θ also unifies A_i and B, i.e., $A_i\theta = B\theta$. Thus $G\theta$ is a logical consequence of P as required. □

The answers supplied by SLD–resolutions are, in a sense, all the correct ones there are.

Theorem 1.10 (Completeness of implementation): *Let R be a selection rule. If $P \models (A_1 \wedge \ldots \wedge A_n)\sigma$ then there is an SLD–refutation $T = \langle \langle G_i, C_i\rangle \mid i \leq n\rangle$ of $P \cup \{G\}$ via R with answer substitution θ and also a substitution ψ such that $G\sigma = G\theta\psi$.*

Proof (for invariant selection rules): We prove by induction on the length n of an SLD–refutation of $P \cup \{G\sigma\}$ via R that there is a refutation with answer substitution θ and a ψ such that $\sigma = \theta\psi$ on the variables of G. Choose a substitution γ instantiating all the variables of $G\sigma$ to new constants (i.e., ones not appearing in $P \cup \{G\}$). As $P \models (A_1 \wedge \ldots \wedge A_n)\sigma$ it is clear that $P \cup \{G\sigma\gamma\} \in$ UNSAT. Let $T = \langle \langle G_i, C_i\rangle \mid i < n\rangle$ be a ground SLD–refutation of $P \cup \{G\sigma\gamma\}$ via R. By the invariance of R, reversing the substitution γ (i.e., replacing the constants by the original variables) gives an SLD–refutation $T' = \langle \langle G_i', C_i'\rangle \mid i < n\rangle$ of $P \cup \{G\sigma\}$ via R in which the unifiers ψ_i restricted to the variables in $G\sigma$ are the identity. The ground refutation can also be lifted, by Exercise 6, to one

$$T'' = \langle \langle G_i'', C_i''\rangle \mid i < n\rangle$$

of $P \cup \{G\}$ with mgu's $\theta_0, \ldots, \theta_n$. Suppose that R selects the literal A_i from G. We can now apply the induction hypothesis to

$$G_1' = \{\neg A_0\sigma, \ldots, \neg A_{i-1}\sigma, \neg B_{0,1}\psi_0, \ldots, \neg B_{0,m_0}\psi_0, \neg A_{i+1}\sigma, \ldots, \neg A_n\sigma\}$$

and

$$G_1'' = \{\neg A_0\theta_0, \ldots, \neg A_{i-1}\theta_0, \neg B_{0,1}\theta_0, \ldots, \neg B_{0,m_0}\theta_0, \neg A_{i+1}\theta_0, \ldots, \neg A_n\theta_0\}$$

as $G_1' = G_1''\sigma\psi_0 = G_1''\psi_0\sigma$ (remember that $G\sigma\psi_0 = G\sigma$ and, as $\sigma\psi_0$ unifies A_0 and B_0, $\theta_0\sigma\psi_0 = \sigma\psi_0$). Thus, we have an SLD–refutation of $P \cup \{G_1''\}$ via R with mgu's $\theta_1' \ldots \theta_n'$ and a λ' such that $\theta_1' \ldots \theta_n'\lambda' = \sigma$ on the variables in G_1''. If x occurs in $G = G_0$, but $x\theta_0$ does not appear in G_1'', then $x\theta_0$ does not appear in $\theta_1', \ldots, \theta_n'$. Since $\theta_0\sigma = \sigma$ on the variables in A_0, we can extend λ' to λ by setting $\lambda(x) = \sigma(x)$ for each variable x in A_0 such that $x\theta_0$ is not in G_1''. Then $\theta_0\theta_1' \ldots \theta_n'\lambda = \sigma$ on all the variables of G, as required. (Keep in mind that $y\theta_0\sigma = y\sigma$ for y occurring in G.) □

To provide proofs for Theorem 1.8 and 1.10 for arbitrary selection rules, we now prove the independence lemma, as promised. We begin with a basic procedure to change a single choice of literal in an LD–refutation.

Lemma 1.11 (Switching Lemma): *Let $G = G_0 = \{\neg A_0, \ldots, \neg A_n\}$ and let $\langle G_0, C_0 \rangle, \ldots, \langle G_k, C_k \rangle$ be an LD–refutation of $P \cup \{G_0\}$ with answer substitution $\psi = \psi_0 \ldots \psi_k$. Suppose that $A_j\psi_0 \ldots \psi_{s-1}$ is the literal resolved on at step $s > 0$. There is then an LD–refutation $\langle G_0, C_0' \rangle, \ldots, \langle G_k', C_k' \rangle$ of $P \cup \{G_0\}$ with answer substitution $\theta = \theta_0 \ldots \theta_k$ in which we resolve on $A_j\theta_0 \ldots \theta_{s-2} = A_j\psi_0 \ldots \psi_{s-2}$ at step $s-1$ such that $G\theta$ is a renaming of $G\psi$.*

Proof: Let $C_i = \{B_i, \neg B_{i,0}, \ldots, \neg B_{i,m_i}\}$ for $i \leq k$. Let $G_{s-1} = \{\neg A_0', \neg A_1', \ldots, \neg A_t'\}$ where $A_j\psi_0 \ldots \psi_{s-2} = A_{j'}'$ and suppose we resolved on A_r' at stage $s-1$. Thus $G_s = \{\neg A_0', \neg A_1', \ldots, \neg A_{r-1}', \neg B_{s-1,0}, \ldots, \neg B_{s-1,m_{s-1}}, \neg A_{r+1}', \ldots, \neg A_t'\}\psi_{s-1}$ and G_{s+1} is ψ_s applied to the result of replacing $\neg A_j = \neg A_{j'}'$ by $\neg B_{s,0}, \ldots, \neg B_{s,m_s}$ in G_s. (Recall that by the definition of LD refutations, $\neg B_{s,0}, \ldots, \neg B_{s,m_s}$ have no variables acted on by ψ_{s-1}.) We know that $A_{j'}'\psi_{s-1}\psi_s = B_s\psi_s = B_s\psi_{s-1}\psi_s$ and so we can unify $A_{j'}'$ and B_s. Let ψ_{s-1}' be the corresponding mgu and let λ be such that $\psi_{s-1}\psi_s = \psi_{s-1}'\lambda$. We now replace step $s-1$ of the original refutation by this resolution. We want to show that we can resolve with C_{s-1} on the literal $A_r'\psi_{s-1}'$ at step s with mgu ψ_s'. If we can also show that the result of this resolution is a renaming of G_{s+1}, we can continue the refutation, modulo renamings, to get the desired result.

We first note that, from our original refutation, $A_r'\psi_{s-1} = B_{s-1}\psi_{s-1}$. Combining this fact with the relation for λ above we can get the following sequence of equalities: $A_r'\psi_{s-1}'\lambda = A_r'\psi_{s-1}\psi_s = B_{s-1}\psi_{s-1}\psi_s = B_{s-1}\psi_{s-1}'\lambda$. Thus λ unifies $A_r'\psi_{s-1}'$ and $B_{s-1} = B_{s-1}\psi_{s-1}'$ (by convention, B_{s-1} has no variables acted on by ψ_{s-1}'). We may therefore resolve on $A_r'\psi_{s-1}'$ with mgu ψ_s' as required. We also get a λ' such that $\lambda = \psi_s'\lambda'$. Combining the equations for λ, we have $\psi_{s-1}\psi_s = \psi_{s-1}'\psi_s'\lambda'$. If we can now prove the existence of a substitution φ' such that $\psi_{s-1}'\psi_s' = \psi_{s-1}\psi_s\varphi'$, then, by Exercise II.11.4, we will have established that $\psi_{s-1}'\psi_s'$ is a renaming of $\psi_{s-1}\psi_s$ as required to complete the proof. The argument for the existence of φ' is similar to that of λ'. We know that $A_r'\psi_{s-1}'\psi_s' = B_{s-1}\psi_{s-1}'\psi_s'$ while in the original proof ψ_{s-1} is an mgu for A_r' and B_{s-1}. Thus there is a φ such that $\psi_{s-1}'\psi_s' = \psi_{s-1}\varphi$. We next note that $A_{j'}'\psi_{s-1}\varphi = A_{j'}'\psi_{s-1}'\psi_s' = B_s\psi_s'\psi_{s-1}' = B_s\psi_{s-1}\varphi$. Thus φ unifies $A_{j'}'\psi_{s-1}$ and $B_s\psi_{s-1}$. As ψ_s in the original proof is an mgu for this unification, there is a φ' such that $\varphi = \psi_s\varphi'$. Combining this with the first equation for φ, we see that $\psi_{s-1}'\psi_s' = \psi_{s-1}\psi_s\varphi'$ as required. □

Lemma 1.12 (Independence Lemma): *For any LD–refutation of $P \cup \{G\}$ of length n with answer substitution θ and any selection rule R, there is an SLD–refutation of $P \cup \{G\}$ via R of length n with an answer substitution ψ such that $G\psi$ is a renaming of $G\theta$.*

Proof: We proceed by induction on the length of the given LD–refutation. As usual, there is nothing to prove if it has length 1. Suppose we have an LD–refutation of G of length $k+1$ with answer substitution φ. Let A_j be the literal selected from G by R and suppose it is resolved on at step s of the given LD–refutation. Apply Lemma 1.11 $s-1$ times to get an LD–refutation $\langle G_0, C_0 \rangle, \ldots, \langle G_k, C_k \rangle$ of $P \cup \{G\} = P \cup \{G_0\}$ with answer substitution $\varphi = \varphi_0\varphi_1 \ldots \varphi_k$ in which A_j is the literal resolved on at step 1 and such that φ is a renaming of θ via λ. Now apply the induction hypothesis to the LD–refutation $\langle G_1, C_1 \rangle, \ldots, \langle G_k, C_k \rangle$ to get one of G_1 via R with answer substitution θ' such that $\varphi_1 \ldots \varphi_k\lambda' = \theta'$ with λ' a renaming substitution. We can now prefix the resolution of $G = G_0$ with C_0 on A_j (with mgu φ_0) to this refutation to get the desired SLD–refutation of $P \cup \{G\}$ via R with answer substitution $\varphi_0\theta'$. We complete the induction argument by noting that $G_0\varphi_0\theta' = G_0\varphi_0\varphi_1 \ldots \varphi_k\lambda' = G_0\varphi\lambda' = G_0\psi\lambda\lambda'$. We have now found the required SLD–refutation and renaming substitution $\lambda\lambda'$. □

Proof (of Theorems 1.8 and 1.10): It is now clear that, by applying Lemma 1.12, Theorem 1.8 follows immediately from Lemma 1.4. Similarly, Theorem 1.10 follows from Lemma 1.4 together with the special case proved above for invariant selection rules. □

Exercises

1. Prove the following lemma: If G is an ordered goal clause, C an ordered program clause, G' and C' the unordered clauses corresponding to G and C respectively (i.e., the union of the elements of the sequence) and the goal clause D' is an LI–resolvent of G' and C', then there is a sequence of LD–resolutions starting with G and C and ending with an ordered goal clause D which corresponds to D'.
2. Use the results of Exercise 1 to carry out the inductive *proof of Lemma 1.4.
3. Prove Lemma II.13.7 for ordered clauses and LD–resolution.
4. Prove Lemma II.13.8 for ordered clauses and LD–resolution.
5. Prove Corollary II.13.9 for ordered clauses and LD–refutations.

6. Notice that the proofs of Exercises 3-5 work for SLD–resolutions if the selection rule is invariant.
7. Let S be a set of clauses in a language $\mathcal{L}$. The *success set* of S is the set of all ground instances $A(t_1, \ldots, t_n)$ in $\mathcal{L}$ of the predicates of $\mathcal{L}$ such that there is a resolution refutation of $S \cup \{\neg A(t_1, \ldots, t_n)\}$. Prove that if P is a PROLOG program then a ground atomic formula $A(t_1, \ldots, t_n)$ is in the success set of P iff it is true in every Herbrand model of P.
8. Suppose a graph G is represented as a database of edges via the list of facts "edge(n, m)." for each pair of nodes, n, m for which there is an edge connecting them in the graph. (Assume that the graph is undirected and so "edge(n, m)." appears in the database iff "edge(m, n)." also appears.) Define, via a PROLOG program, the predicate "connected(X, Y)" so that connected(n, m) is a logical consequence of the program iff there is a sequence of nodes $n = n_1, n_2, \ldots ; n_k = m$ such that each successive pair of nodes n_i, n_{i+1} is joined by an edge.

2. Implementations: Searching and Backtracking

Although SLD–resolution is both sound and complete, the available implementations of PROLOG are neither. There are two sources of problems. The first, and at least theoretically relatively minor one, is in the implementation of the unification algorithm. As we have mentioned, the available PROLOG implementations omit the "occurs check" in the unification algorithm. Thus, for example, the PROLOG unifier believes that X and $f(X)$ are unifiable. In addition, the PROLOG theorem prover does not make the substitutions needed by the unifier until they are required in theSLD–resolution. These two facts combine to destroy the soundness of the system. Thus, for example, given the program:

$$\text{test} :- \ p(X, X).$$
$$p(X, f(X)).$$

and the question "?– test." PROLOG will answer "yes.". What has happened here is that the theorem prover says that to verify "test", we must verify $p(X, X)$. As $p(X, f(X))$ is given, it suffices to see if $p(X, X)$ and $p(X, f(X))$ can be unified. The unifier answers that they can. As no further information is needed (to instantiate "test", for example) the theorem prover gives the answer "yes" to our question. Thus, it gives an answer which is not a logical consequence of the program — a violation of soundness.

A key point in the above implementation is that the theorem prover did not have to carry out the substitution $\{X/f(X)\}$ implicit in the unifier's answer. If it had tried, it would have fallen into the endless loop of trying

to output the results of the substitution. This situation is illustrated by the program:

$$\text{test1}(X) :- \ p(X, X).$$
$$p(X, f(X)).$$

If we now ask "?– test1(X).", the result of the looping is displayed as PROLOG tries to give the "correct" answer substitution of $X = f(f(f(\ldots$. (Hit control–break to stop the display.) This type of failure can also occur undisplayed in the search for an SLD–resolution. Consider the program:

$$\text{test2} :- \ p(X, X).$$
$$p(X, f(X)) :- \ p(X, X).$$

In this case, the only indication that something has gone wrong is that no answer is forthcoming.

Unfortunately, these problems may well occur in natural programs as well. Of course, both of these problems could be eliminated by implementing the unification algorithm correctly. As there are now reasonably efficient (i.e., linear) unification algorithms, this is not an unreasonable expectation. For now, however, one can fairly easily eliminate the first type of problem in favor of the second. One simply writes programs in which any variable appearing in the head of a clause also appears in the body. This can be done by simply adding $X = X$ to the body for any variable X in the head not already in the body. The first program would then become

$$\text{test} :- \ p(X, X).$$
$$p(X, f(X)) :- \ X = X.$$

Now, when the theorem prover tries to resolve on $p(X, X)$ and $P(X, f(X))$ and the unifier says they are unifiable, the prover asks for the substitution so that it can put the expression resulting from $X = X$ into the goal clause in place of $p(X, X)$. As the substitution is circular, it never gets to complete the proof. Thus, this simple programming trick will restore the soundness of the PROLOG theorem prover (at the expense, of course, of taking longer to run). Completeness, however, is another matter.

The source of the incompleteness of the PROLOG implementation is the method employed to search for an SLD–refutation. Suppose we are given a program P and a goal clause G. We wish to find an SLD–resolution refutation of $P \cup \{G\}$ beginning with G. At each step i of the SLD–resolution, the only choice to make is which clause in P to use to resolve on the leftmost term in our current goal clause G_i. We can thus display the space of all possible SLD–derivations as a tree: the root node is labeled G and if a node is labeled G' then the labels of it successors are the results of all possible choices of clauses of P for the next resolution on the leftmost term of G'.

We call such trees SLD–*trees for* P *and* G. As a simple example, consider the program P_1:

$$
\begin{aligned}
&p(X,X) \text{ :- } q(X,Y), r(X,Z). && (1)\\
&p(X,X) \text{ :- } s(X). && (2)\\
&q(b,a). && (3)\\
&q(a,a). && (4)\\
&q(X,Y) \text{ :- } r(a,Y) && (5)\\
&r(b,Z). && (6)\\
&s(X) \text{ :- } q(X,a). && (7)
\end{aligned}
$$

The SLD–tree for P_1 starting with the goal clause $G = \{\neg p(X,X)\}$ is displayed in Figure 39. Along each branching we indicate the clause of P_1 resolved against. The convention is that the successors are listed in a left to right order that agrees with the order in which the clauses used appear in P_1. *Success paths*, corresponding to yes answers, are those ending in $\square$. At the end of each such success path we put the answer substitution given by the proof (of $\square$) represented by the path. A path is a *failure path* if it ends with a clause G' such that there is no clause in P with which we can resolve on the leftmost term of G'.

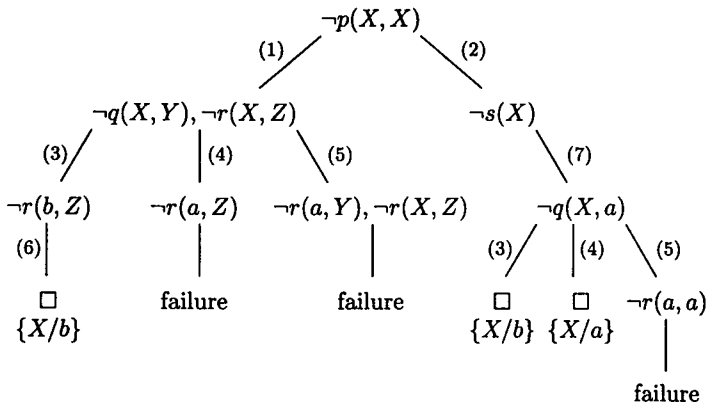


FIGURE 39

We see here that, of the six possible paths, three end in failure, two end with the correct substitution $\{X/b\}$ and one with $\{X/a\}$.

The PROLOG theorem prover searches the SLD–tree for a success path by always trying the leftmost path first. That is, it tries to resolve the current G with the first clause in P that is possible. In Figure 39 it would simply follow the path (1), (3), (6) to get the correct answer substitution $\{X/b\}$. If the theorem prover hits a *failure point* (i.e., not $\square$ and no resolution is possible) it *backtracks*. Backtracking means retracing the path one has just followed until one finds a node with a branch to the right of the path being retraced. If there is more than one such path, take the leftmost one. The theorem prover repeats this backtracking procedure until a success path is found.

Copies of printouts for runs of the programs P_1 (and other programs listed in this section) are included at the end of the section. We also include printouts of the runs with "tracing". (Tracing is a facility supplied with most implementations of PROLOG that displays the actual steps of the search of the SLD–tree. It is an important tool for understanding the flow of control in a program and for debugging. Note that items of the form "_0nnn" are just names for variables.)

If, for example, we omit clause (3) from the above program P_1 to produce P_2 , we get a new SLD–tree as pictured in Figure 40.

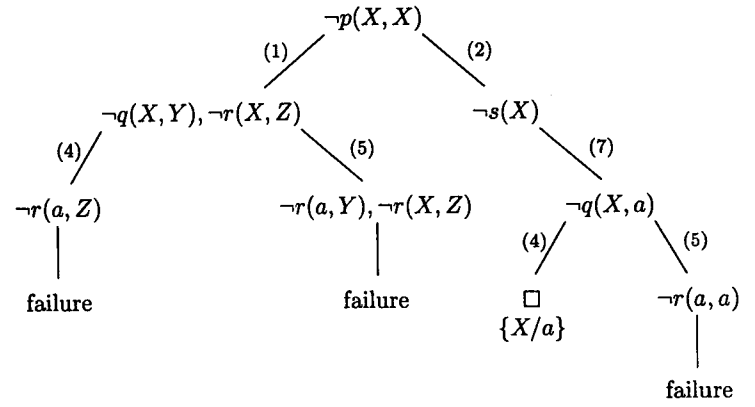


FIGURE 40

In this case, the theorem prover first tries the path (1), (4), failure. It then backtracks to $\neg q(X,Y), \neg r(X,Z)$ and tries (5), failure. It then backtracks all the way to $\neg p(X,X)$ and tries (2), (7), (4), success to give the answer substitution $X = a$.

The same backtracking procedure is implemented when we ask PROLOG for a second answer. Thus, with the original program P_1 and goal $\{\neg p(X,X)\}$, i.e., "?- $p(X,X)$." we got the reply "$X = b \rightarrow$". If we now ask for another answer by entering ";" the theorem prover backtracks from the success node until it hits a node with alternate paths — here $\neg q(X,Y)$, $\neg r(X,Z)$. It then tries (4), failure and (5), failure and then backtracks to $\neg p(X,X)$. It now tries (2), (7), (3) giving once again the answer $X = b$. If we enter ";" , once again, it backtracks from the success node to $\neg q(X,a)$ to try (4) and give the answer $X = a$. Should we once again ask for another answer, PROLOG will backtrack to $\neg q(X,a)$, try the remaining path (5), fail and report no — no more success nodes have been found and there are no more paths to try.

The situation is similar if we get a "no" answer on first entering our question. Here it means that all paths in the SLD–tree have been traversed and they are all failures. By our general completeness theorem, we then know that $P \cup \{G\}$ is satisfiable and so there is no substitution for which the question asked is a logical consequence of P_1. By the completeness theorem for the implementation we also know that when we finally get a no after a series of ";" requests, every correct answer substitution is an instance of one of the answer substitutions already displayed.

This type of search procedure is called a *depth–first search* procedure as it tries to go as deeply as possible in the tree by running down to the end of a path before searching along any other branches. In contrast, a procedure that searches the tree in Figure 40 in the order $\neg p(X,X)$; $\neg q(X,Y)$; $\neg r(X,Z)$; $\neg s(X)$; $\neg r(b,Z)$; $\neg r(a,Z)$; $\neg r(a,Y)$; $\neg r(a,Z)$; $\neg q(X,a)$; $\square$; failure; failure; $\square$; $\square$; $\neg r(a,a)$; failure, is called a *breadth–first search*. Clearly many mixed strategies are also possible. In our case, the depth–first search was much faster than breadth–first (3 versus 9 steps). Indeed, this is a quite general phenomena. Depth–first is usually much faster than breadth–first. That, of course, is why the implementations use depth–first searches. The cost, however, is quite high — we lose the completeness of the SLD–resolution method (quite independently of the procedure used to implement unification).

The general completeness theorem guarantees that, if $P \cup \{G\} \in$ UNSAT, there is a (necessarily finite) SLD–refutation proof beginning with G. If there is one of length n it is clear that in a breadth–first search we must find such a proof by the time we have searched the tree to depth n, i.e., we have traversed every possible path of length n. Unfortunately, there are no such guarantees for depth-first searching. The problem is that some paths of the tree may be infinite. Depth–first searching may then keep us on such a path forever when a short proof lies along another path. As an example, consider the program P_3 gotten by replacing (6) in P_1 by

$$r(W,Z) \text{ :- } r(b,Z). \qquad (6')$$

The SLD–tree for P_3 beginning with $\neg p(X,X)$ is displayed in Figure 41.

Here we see that, even though there are SLD–refutations along the paths (2), (7), (3) and (2), (7), (4), the depth–first search employed by PROLOG will not find them. Instead, it will endlessly pursue the leftmost path (1), (3), (6′) and then continue trying to use clause (6′) again and again forever.

We can now see why the ordering of the clauses plays a crucial role in the actual running of a PROLOG program. If we rearranged P_3 to get P_4 by interchanging clauses (1) and (2), the theorem prover would first find the proof (2), (7), (3) and then (2), (7), (4). Only then, if asked for another answer, would it fall into the endless search part of the tree. Unfortunately,

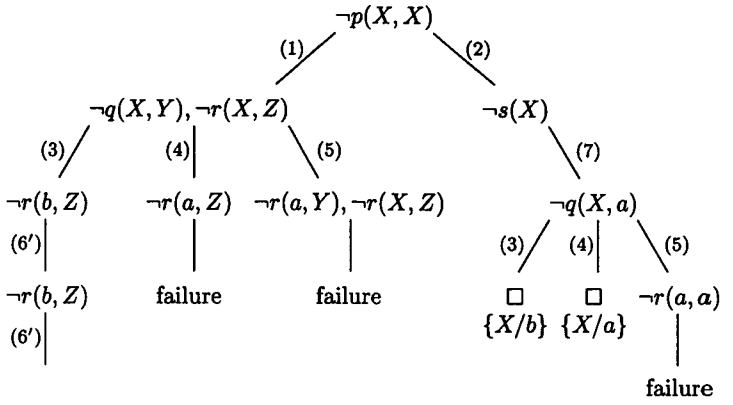


FIGURE 41

the arrangement of clauses, though an important programming consideration, is not enough to guarantee completeness. To see this, consider the program P_5 :

$$\text{equivalent}(X,Y) \text{ :- } \text{equivalent}(Y,X). \qquad (1)$$
$$\text{equivalent}(X,Z) \text{ :- } \text{equivalent}(X,Y), \text{equivalent}(Y,Z). \qquad (2)$$
$$\text{equivalent}(a,b). \qquad (3)$$
$$\text{equivalent}(b,c). \qquad (4)$$

and the goal $G = \neg\text{equivalent}(c,a)$. It is clear that equivalent(c,a) is a logical consequence of P_5 (Exercise 1). The problem is that no matter in which order clauses (1) and (2) are listed in the program, a depth–first search will always keep trying to apply the first of them since both heads will unify with any expression of the form equivalent(t_2,t_3) for any terms t_2 and t_3. Thus the theorem prover will be able to use one of (1) and (2) but not both. It is, however, easy to see that if either clause is omitted from P_5 then the result together with G is satisfiable. Thus, no depth–first search procedure can find an SLD–refutation of $P \cup \{G\}$ regardless of the ordering of the clauses in P. (Note that the depth–first implementation remains unable to find the proof here even if we use another selection rule.) Thus, depth–first search methods, although efficient, are inherently incomplete. In the next section we present a programming tool to minimize or get around these problems. In §4 we briefly consider the general problem of guaranteeing termination (for certain types) of programs in pure PROLOG with the standard selection rule and depth–first searching of SLD–trees. In §8 we show that the general termination problem for PROLOG programs is undecidable.

Exercises

1. Find an SLD–refutation of $P_5 \cup \{\neg\text{equivalent}(c, a)\}$.
2. Prove that every set P of program clauses has a *minimal* (actually least) *Herbrand model* M_P (i.e., prove that the intersection of all Herbrand models of P is also a model of P) but not every set S of universal sentences has a minimal Herbrand model.
3. Consider the program P

$$p(X,Z) \text{ :- } q(X,Y), p(Y,Z). \quad (1)$$
$$p(X,X). \quad (2)$$
$$q(a,b). \quad (3)$$

   Draw SLD–trees for P and goal "?- $p(X,Y)$." for the standard selection rule and also for the rule which always chooses the rightmost literal from a clause.

4. a) Draw the SLD–refutation tree illustrating all possible attempts at SLD–proofs using the standard selection rule and the associated correct answer substitutions for the following program and goal:

   (1) $p(X,Y)$:- $s(X), t(X)$.
   (2) $p(X,Y)$:- $q(X, f(X)), r(X, f(Y))$.
   (3) $q(b,Y)$.
   (4) $q(a,Y)$:- $r(a, f(a))$.
   (5) $r(a, f(a))$.
   (6) $r(b, f(b))$.
   (7) $s(a)$.
   (8) $s(b)$.
   (9) $t(a)$.
   Goal: ?- $p(X,Y)$.

   b) Explain what happens in terms of PROLOG searching and backtracking when we enter "?- $p(X,Y)$." and then repeatedly ask for more answers via ";".

5. Define PROLOG programs for the following operations on lists. (The basic notations for lists were introduced in II.3.)
   a) second_element (X,Y) which is to define the predicate "X is the second element of the list Y".
   b) substitute_for_second_element $(Y, L1, L2)$ which is to define the predicate "$L2$ is $L1$ with Y substituted for its second element".
   c) switch (L,M) which is to define the predicate "M is L with its first and last elements exchanged".

   d) Check that your programs work correctly, at least when all variables are instantiated, by running some examples: Is b the second element of $[a,[b,c],d,e]$ or of $[a,b,c]$? Is $[b,c]$ the second element of either of them? Is $[a,[b,c],d]$ the result of replacing the second element of $[a,b,c]$ with $[a,[b,c],d]$? Is $[a,[a,[b,c],d],c]$? Is $[a,c,b]$ the result of interchanging the first and last elements of $[a,b,c]$? Is $[c,b,a]$?
   e) Try a couple of examples with uninstantiated variables as well: find the second element of $[a,[b,c],d,e]$ and $[a,b,c]$; replace the second element of $[a,b,c]$ with $[a,[b,c],d]$ and then replace the second element of the result with b; interchange the first and last elements of $[a,b,c]$ and then of the resulting list. Now some examples with the roles of the variables interchanged: Find a list with second element b; one which, when its second element is replaced by b, would be $[a,b,c]$; one which, when its first and last elements are interchanged, would be $[a,b,c]$.
   f) What simple universal statements suggested by the above examples should be true about your programs? Try asking as a PROLOG question if substituting the second element of a list X for its own second element returns the original list. Similarly for the result of switching the first and last elements twice. Explain the output when these questions are entered.

6. Trace the execution of your programs on the examples in Exercise 5 and explain any anomalous behavior (depending on your program there may not be any).

In Exercises 7–8 we use a unary function symbol s (with the intended meaning "successor of") to define a counter beginning with the constant 0. Thus 0 corresponds to 0, $s(0)$ to 1, $s(s(0))$ to 2, and in general $s^n(0)$ $(= s \ldots s(0)$ with n repetitions of the function $s)$ corresponds to n. Do not try to use the built–in arithmetic operations supplied with PROLOG.

7. Write a program to calculate the length of a list using this counter. Try it on a couple of inputs: $[a,b,c]$, $[a,b,c,[b,c],[d,c,e]]$. What happens when you try to find a list of length three?
8. We add on to our language two predicate symbols "add(X,Y,Z)" and "multiply(X,Y,Z)".
   a) Prove that, if we interpret "0" as 0 in the natural numbers and "s" as successor, then the following PROLOG program defines addition on the natural numbers as represented by $\{0, s(0), s(s(0)), \ldots\}$ in the sense that "add$(s^n(0), s^m(0), s^r(0))$" is a consequence of the program iff $n + m = r$:

      add$(X, 0, X)$.
      add$(X, s(Y), s(Z))$:- add(X,Y,Z).

b) Write a similar PROLOG program to define multiplication in terms of addition and successor so that "multiply($s^n(0), s^m(0), s^r(0)$) will be a consequence of the program iff $mn = r$. (Hint: $x(y+1) = xy + x$.)

c) Prove that "multiply($s^n(0), s^m(0), s^r(0)$) is in fact a consequence of the program iff $mn = r$.

9. Recall the procedure in II.5 for defining a knight's move on the chess board.

a) Write a program defining a queen's move. (It can move horizontally, vertically or diagonally.)

b) Write one defining when a queen cannot move from one position to another.

c) Write a program to find a way to put a queen in each column so that none of them could move to any square occupied by any of the others.

Do not use any built–in predicates (i.e., no arithmetic, no cut, no "not").

d) Find two solutions to the problem in (c).

10. Write a program for the function FLATTEN which strips off all brackets from a list except the outermost, i.e., it returns a list of atoms in the order in which they would appear on the page when the input list is written out.

11. Consider the following program:

$$tc(X,Y) :- r(X,Y).$$
$$tc(X,Y) :- r(X,Z), tc(Z,Y).$$

The goal

$$?- tc(a,b).$$

will succeed exactly when the pair (a,b) is in the transitive closure of the relation $r(,)$ as defined in Exercise 11 of II.7. How do you reconcile this with the result of that exercise? (This problem is also relevant to Exercise 1.8.)

The following problems are continuations of Exercises II.5.7–8 and follow the same conventions about the assumed genealogical database. A printout of the database and some words of warning are included in Appendix B. They also make use of the counter defined in Exercise 7. Once again, if not using the database simply write out the programs defining the required predicates and explain how one would obtain the desired information. Now, in addition to explaining why your answersare semantically correct, discuss how they would run as implemented by PROLOG.

12. a) Define nthgrandfather($X, Y, s^n(c)$) to mean that X is an ancestor of Y n generations up. (Start with $n = 1$ for X is the father of Y.)

b) Use this program to find libni's grandfather's great–grandfather.

c) Can you find more than one such ancestor? Should you be able to do so if all were well with the database?

d) Use this program to find four of levi's great-great-great–grandchildren.

e) Find three of esau's grandchildren.

13. a) Define cousin(X, Y) to give the usual meaning of first cousin.

b) Find five cousins of tola.

c) Define secondcousin(X, Y).

d) Find six second cousins of libni.

Recall the usage in English: My children and my brother's children are first cousins; my grandchildren and his grandchildren are second cousins; my children and his grandchildren are first cousins once removed; my children and his great–grandchildren are first cousins twice removed; my grandchildren and his great–grandchildren are second cousins once removed.

e) Define cousin($X, Y, s^n(c), s^m(c)$) to mean that X is the n^{th} cousin m times removed of Y.

f) Find seven second cousins once removed of libni. Can you tell or guess and verify what relation they are to the people listed in (*d*)?

g) Find three third cousins twice removed of libni. Can you predict from your program how they are likely to be related to libni (i.e., what routing istaken to find these instances)?

14. Various anomalies may creep into your results when implemented with an actual genealogical database. Consider how the following typical problems with such databases might affect your programs.

a) The same data may be recorded twice. For example the fact fatherof(abraham,isaac) appears twice in the database. Will this cause any wrong answers to be reported? What effect if any will it have on running a program such as the one for ancestor or for n^{th} cousins m times removed?

b) Different people may have the same name and not be distinguished in the database. (Try to see who is the father of enoch.) How will this affect the ancestor and cousin programs? Can you devise a method for identifying people in the database that might be used to eliminate or reduce the impact of this problem? (Hint consider using a counter.) Try the father of enoch again. Also see if you have eliminated all solutions to ancestor(X, X).

c) The same person may appear under more than one name. We know, for example, from other Biblical passages that esau is seir. How will this affect the ancestor and cousin programs? What could you do (short of editing the data file) to correct for such a situation? Can you add rules to the database that would take care of this problem without editing the database? (Examples in the database used for trying your solution out are finding the grandfather of lotan and the cousins of hori.)

Runs of Programs P_1 – P_4

PROGRAM P_1.

```
?- listing.
p(A,A) :-
   q(A,B),
   r(A,C).
p(A,A) :-
   s(A).
q(b,a).
q(a,a).
q(A,B) :-
   r(a,B).
r(b,A).
s(A) :-
   q(A,a).
yes
?- p(X,X),
X = b →;
X = b →;
X = a →;
no
?- trace.
yes
?- leash(full).
yes
?- spy(p/2).
yes
?- p(X,X).
   ** (0) CALL: p(_0085,_0085)?>
   (1) CALL: q(_0085,_0255)?>
   (1) EXIT: q(b,a)?>
   (2) CALL: r(b,_0261)?>
   (2) EXIT: r(b,_0261)?>
   ** (0) EXIT: p(b,b)?>
X = b →;
   ** (0) REDO: p(b,b)?>
   (2) REDO: r(b,_0262)?>
   (2) FAIL: r(b,_0261)?>
   (1) REDO: q(b,a)?>
   (1) EXIT: q(a,a)?>
   (3) CALL: r(a,_0261)?>
   (3) FAIL: r(a,_0261)?>
   (1) REDO: q(a,a)?>
   (4) CALL: r(a,_0255)?>
   (4) FAIL: r(a,_0255)?>
   (1) FAIL: q(_0085,_0255)?>
   (5) CALL: s(_0085)?>
   (6) CALL: q(_0085,a)?>
   (6) EXIT: q(b,a)?>
   (5) EXIT: s(b)?>
   ** (0) EXIT: p(b,b)?>
X = b →;
   ** (0) REDO: p(b,b)?
   (5) REDO: s(b)?>
   (6) REDO: q(b,a)?>
   (6) EXIT: q(a,a)?>
   (5) EXIT: s(a)?>
   ** (0) EXIT: p(a,a)?>
X = a →;
   ** (0) REDO: p(a,a)?>
   (5) REDO: s(a)?>
   (6) REDO: q(a,a)?>
   (7) CALL: r(a,a)?>
   (7) FAIL: r(a,a)?>
   (6) FAIL: q(_0085,a)?>
   (5) FAIL: s(_0085)?>
   ** (0) FAIL: p(_0085,_0085)?>
no
```


PROGRAM P_2.

```
?- listing.
p(A,A) :-
   q(A,B),
   r(A,C).
p(A,A) :-
   s(A).
q(a,a).
q(A,B) :-
   r(a,B).
r(b,A).
s(A) :-
   q(A,a).
yes
?- p(X,X).
X = a →;
no
?- trace.
yes
?- leash (full).
yes
?- spy(p/2).
yes
?- p(X,X).
   ** (0) CALL: p(_005D,_005D)?>
   (1) CALL: q(_005D,_022D)?>
   (1) EXIT: q(a,a)?>
   (2) CALL: r(a,_0239)?>
   (2) FAIL: r(a,_0239)?>
   (1) REDO: q(a,a)?>
   (3) CALL: r(a,_022D)?>
   (3) FAIL: r(a,_022D)?>
   (1) FAIL: q(_005D,_022D)?>
   (4) CALL: s(_005D)?>
   (5) CALL: Q(_005D,a)?>
   (5) EXIT: q(a,a)?>
   (4) EXIT: s(a)?>
   ** (0) EXIT p(a,a)?>
X = a →;
   ** (0) REDO: p(a,a)?>
   (4) REDO: s(a)?>
   (5) REDO: q(a,a)?>
   (6) CALL: r(a,a)?>
   (6) FAIL: r(a,a)?>
   (5) FAIL: q(_005D,a)?>
   (4) FAIL: s(_005D)?>
   ** (0) FAIL: p(_005D,_005D)?>
no
```

PROGRAM P_3.

```
?- listing.
p(A,A) :-
   q(A,B),
   r(A,C).
p(A,A) :-
   s(A).
q(b,a).
q(a,a).
q(A,B) :-
   r(a,B).
r(A,B) :-
   r(b,B).
s(A) :-
   q(A,a).
yes
?- p(X,X).
?- trace.
yes
?- leash(full).
yes
?- spy(p/2).
yes
?- p(X,X).
   ** (0) CALL: p(_005D,_005D)?>
   (1) CALL: q(_005D,_022D)?>
   (1) EXIT: q(b,a)?>
   (2) CALL: r(b,_0239)?>
   (3) CALL: r(b,_0239)?>
   (4) CALL: r(b,_0239)?>
   (5) CALL: r(b,_0239)?>
```

PROGRAM P_4.

```
?- listing.
p(A,A) :-
   s(A).
p(A,A) :-
   q(A,B),
   r(A,C).
q(b,a).
q(a,a).
q(A,B) :-
   r(a,B).
r(A,B) :-
   r(b,B).
s(A) :-
   q(A,a).

yes
?- p(X,X).
X = b →;
X = a →;
?- trace.
yes
?- leash(full).
yes
?- spy(p/2).
yes
```

```
?- p(X,X).
   ** (0) CALL: p(_005D,_005D)?>
   (1) CALL: s(_005D)?>
   (2) CALL: q(_005D,a)?>
   (2) EXIT: q(b,a)?>
   (1) EXIT: s(b)?>
   ** (0) EXIT: p(b,b)?>
X = b →;
   ** (0) REDO: p(b,b)?>
   (1) REDO: s(b)?>
   (2) REDO: q(b,a)?>
   (2) EXIT: q(a,a)?>
   (1) EXIT: s(a)?>
   ** (0) EXIT: p(a,a)?>
X = a →;
   ** (0) REDO: p(a,a)?>
   (1) REDO: s(a)?>
   (2) REDO: q(a,a)?>
   (3) CALL: r(a,a)?>
   (4) CALL: r(b,a)?>
   (5) CALL: r(b,a)?>
```

3. Controlling the Implementation: Cut

We have seen that the success of an execution of even a semantically correct PROLOG program depends in many ways on the specifics of the implementation. So far, the only control we have had over the path of execution has been the ordering of clauses in the program. We know, for example, that the base case of a recursion should always precede the inductive case (why?). Similarly, facts about a predicate should generally precede the asserted rules. Such heuristics can, however, go only so far. At times we might wish to exercise more detailed control over implementing the searching of the SLD–tree. Sometimes this is "merely" for the sake of efficiency. At other times there just seems to be no other way of getting a program that will run at all. In this section we consider one such built–in control facility — cut.

Syntactically *cut*, written "!", appears to be simply another literal. Thus we write:

$$p :- \ q_1, q_2, !, q_3, q_4.$$

It does not, however, have any (declarative) semantics. Instead, it alters the implementation of the program. When the above clause is called in a

search of the SLD–tree, the subgoals q_1, q_2, !, q_3 , q_4 are inserted at the beginning of our current goal clause as usual. We try to satisfy q_1 and q_2 in turn, as before. If we succeed, we skip over the cut and attempt to satisfy q_3 and q_4 . If we succeed in satisfying q_3 and q_4 all continues as if there were no cut. Should we, however, fail and so by backtracking be returned to the cut, we act as if p has failed and we are returned by "*deep backtracking*" to the node of the SLD–tree immediately above that for p, called the *parent* goal, and try the next branch to the right out of that node. (If none exists, the current search fails as usual.)

Example 3.1: Consider the following program:

$$\begin{aligned} t &:- \ p, r. && (1)\\ t &:- \ s. && (2)\\ p &:- \ q_1, q_2, !, q_3, q_4. && (3)\\ p &:- \ u, v. && (4)\\ q_1. & && (5)\\ q_2. & && (6)\\ s. & && (7)\\ u. & && (8) \end{aligned}$$

For the goal $\{\neg t\}$ we get the following SLD–tree:

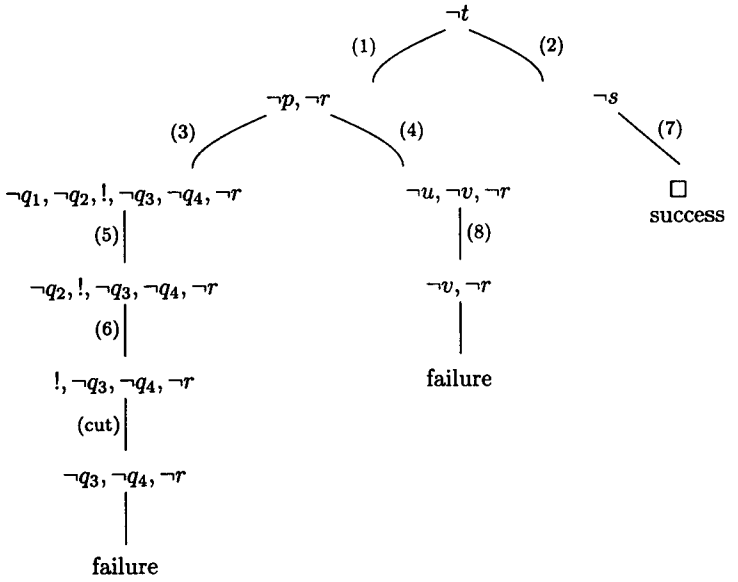


FIGURE 42

When we run the search we travel down the leftmost branch, as usual, going through the cut until we hit the failure point. (Note that ! succeeds by definition the first time through; we simply remove it when reached.) However, when we hit the failure node at the end of the leftmost path, control is passed to the node labeled $\neg s$ from which we immediately succeed. The point is that, once the goal clauses after the cut, $\neg q_3, \neg q_4, \neg r$, have failed, we return not to the parent, $(\neg p, \neg r)$, of the clause with the cut, but instead go back to *its* parent $(\neg t)$. We then try the next branch to the right (through $\neg s$) and proceed with our search.

The use of cut in this example merely saved us time in the execution of the program by pruning the SLD–tree without cutting off any success branches. Such uses of cut are, of course, innocuous (or *safe*). It is, however, difficult to write programs so that the uses of cut are always restricted to such safe situations. In general, the cut can be a source of both incompleteness and unsoundness. Clearly, if the cut prunes a success path, we may lose some correct answer substitutions. Worse yet, it could prune all the success paths. In this case we might wind up running down an infinite path — demonstrating the incompleteness of the search system much as in the analysis of depth–first searching. Finally it might prune both all success paths and all infinite paths. In this case PROLOG will answer "no" when in fact $P \cup \{G\}$ is unsatisfiable. In this way cut can introduce an actual unsoundness into programs, as a "no" answer means that $P \cup \{G\}$ is satisfiable.

Nonetheless, cut can be very useful if extreme care is taken when using it. Some implementations of PROLOG have other facilities for controlling the implementation of searching. One such is called *snip*. It acts like a restricted form of cut. For example when a clause p :– $q_1, q_2, [!q_3, q_4!], q_5$ is encountered, the search progresses normally through the snip (q_4), that is through the clauses between the exclamation points. Should backtracking return you to right end of the snip, however, it instead skips over the snip to return to q_2. Although this is at times a convenient tool, we will see in the exercises that a snip can always be replaced by a use of cut. In general, we simply warn the reader — be very careful, cut can have unexpected consequences. At the very least, uses of cut which subvert the declarative semantics of the program should be avoided. We will, however, briefly consider one important use of cut — defining negation — as a lead in to some other topics.

Although "not" comes as a built–in predicate in PROLOG, we can see what it really means by defining it in terms of cut. Its meaning in PROLOG is different from the usual logical meaning of negation. "not(P)" means that we have a demonstration that P fails (i.e., is not provable). Thus PROLOG answers "yes" to the goal "not(A)" if and only if it would answer "no" to the goal A. We could replace uses of "not" by uses of cut by inserting a definition of not(A) :

```
not(A) :- A, !, fail.
not(A).
```


We see here that if not(A) is called, PROLOG turns to the first clause defining it and calls A. If A succeeds, we pass over the cut and hit "fail". "fail" is a built–in predicate which always fails (and so could be replaced with any other clause which always fails). Thus, if A succeeds, not(A) fails. On the other hand, if A fails, we try the second clause defining not(A) and it succeeds.

In fact, the tables can often be turned. Many uses of cut can and should be replaced by uses of "not". We say "should" because "not" can have some declarative sense in PROLOG programs (even though not the classical one) while cut is much more problematic. We will explore the theoretical semantic underpinnings of PROLOG's use of "not" in §6. In order to do that, however, we will first make a start on dealing with equality in §5.

Exercises

1. In Exercise 2.4 what would be the effect of inserting a cut, !, between the two literals in the body of clause (2) on your answer to (b)?

   Recall the list notation introduced in Example II.3.15. Consider the following program APPEND for appending one list to another:

   (a1) $a([\,], Y, Y).$

   (a2) $a([X|Y], Z, [X|W])$:– $a(Y, Z, W).$

2. What is the advantage of modifying this program to APPEND′:

   (a1) $a([\,], Y, Y)$:– !.

   (a2) $a([X|Y], Z, [X|W])$:– $a(Y, Z, W).$

   Consider the situation when we have two given lists x and y and we wish to find out the result of appending one to the front of the other, that is consider goals of the form ?– $a([x, y, z], [u, v], Z)$. Consider also ones of the form ?– $a([x, y, z], V, W)$.

3. What problems arise in the implementation of APPEND′ (in contrast to that of APPEND) when considering a goal of the form ?–$a(X, Y, [x, y, z])$. Consider what happens when you try to get more than one answer substitution for the variables.

4. This question is somewhat open–ended and refers to the database mentioned in the exercises for II.7–8 and III.11–12. Can you use cut to take advantage of the ordering of clauses in the genealogical database to alleviate any of the problems that you had earlier on with the family relation programs such as grandfather, uncle or cousin? Assume that the clauses in the database reflect births in chronological order. You might consider both rewriting your programs and writing a new program (with cut) to revise the database in some way to prevent such results as someone being his own grandfather.

4. Termination Conditions for PROLOG Programs

An important problem in the analysis of programs is that of determining when programs terminate. Of course, the problem in general is undecidable (Theorem 8.9 and Corollary 8.10), but it may be manageable for specific programs of practical interest. In this section we present a method of analysis that can be used to abstractly characterize termination of programs running with the standard selection rule of always choosing the leftmost literal. The approach presented is that of Apt and Pedreschi [1991, 5.4]. They adapted the ideas of Bezem [1989, 5.4] for characterizing termination of all LD–proofs from a given program and goal to deal with the situation where the selection rule is fixed as the standard one.

For the rest of this section P will be a PROLOG program and G a goal in a language $\mathcal{L}$. All clauses and resolutions are ordered. We let P' and G' be the set of ground instances of P and G respectively. SLD–proofs will mean ones using the standard selection rule. The basic property of the program P that we wish to characterize is given in the following definition:

Definition 4.1: P is *left–terminating for a goal* G if all SLD–proofs from $P \cup \{G\}$ starting with G are finite. P is *left–terminating* if it is left–terminating for every ground goal G.

Note that if P is left–terminating for G, then the standard implementation of P in PROLOG (using leftmost selection and depth–first search) will terminate on the goal G. Indeed, as all SLD–proofs starting with G are finite, it would terminate with any search rule. Thus, if we can prove left–termination for the goal clauses of interest, we will be guaranteed that our programs terminate under the standard implementation.

The basic strategy in nearly all proofs of termination of deductions has two parts. First, one carefully defines a well ordering of clauses or proofs. Then one shows that individual deductions (or in our case resolutions) produce decreases in this ordering. Once we have reached such a situation, it is clear that all proofs terminate as each step represents a decrease in a well ordering. The description of the well ordering we want begins with the basic notion of a level mapping.

Definition 4.2:

(i) A *level mapping* for P is a function f from the atomic sentences (positive ground literals) of $\mathcal{L}$ to $\mathbb{N}$. We denote the value of this function on a literal A by $|A|_f$, called *the level of A with respect to f*. If the function f is clear from the context, we will omit the subscript.

(ii) If $\mathcal{M}$ is a structure for $\mathcal{L}$ and $\vec{A} = A_1, \ldots, A_n$ a sequence of atomic sentences, we let $\mathcal{M}(\vec{A})$ be the least $i \leq n$ such that $\mathcal{M} \not\models A_i$, if there is one, and n itself otherwise. If $G = \{\neg A_1, \ldots, \neg A_n\}$, we also write $\mathcal{M}(G)$ for $\mathcal{M}(A_1, \ldots, A_n)$.

(iii) P is *acceptable with respect to a level mapping f and a model $\mathcal{M}$ of P* if for every $B :- \vec{A}$ in P', i.e., every ground instance of a clause in P, $|B| > |A_i|$ for each $i \leq \mathcal{M}(\vec{A})$. P is *acceptable* if it is acceptable with respect to some level mapping and model.

To grasp the idea behind this definition, consider first a level mapping without any mention of a model. It is then clear that, if P is acceptable, every ground resolution of a goal clause $G = \{\neg A_1, \ldots, \neg A_n\}$ with a clause in P gives a resolvent of lower level than G. This modified notion of acceptability corresponds to the condition that all LD–proofs from P starting with G are finite. (See Exercises 1–2.) The restriction to SLD–proofs will be captured by the restriction to literals A_i for $i \leq \mathcal{M}(G)$ by the right choice of the model $\mathcal{M}$. The basic idea is that, if there is no SLD–refutation of $\{\neg A_i\}$, then no SLD–refutation starting with G can get beyond A_i. Thus there is no need to consider later literals in an analysis of the SLD–tree from G. On the other hand, if there is no SLD–refutation of $\{\neg A_i\}$, there is a model $\mathcal{M}$ in which A_i is false. Thus the right choice of model will cut off the level function precisely at the point at which one need not consider any more literals.

To begin our analysis we define, from a given level mapping f and model $\mathcal{M}$ of P, the required well ordering of clauses. We must consider first the ground goal clauses and then certain nonground ones.

Definition 4.3:

(i) Let P be acceptable with respect to f and $\mathcal{M}$. We extend f to ground goal clauses $G_i = \{\neg A_{i,1}, \ldots, \neg A_{i,n_i}\}$ by setting $|G_i| = \langle |A_{i,j_1}|, \ldots, |A_{i,j_k}| \rangle$ where we have listed the literals $A_{i,1}, \ldots, A_{i,\mathcal{M}(G_i)}$ in decreasing (although not necessarily strictly decreasing) order of their level. We order these tuples by the lexicographic ordering. (Formally, to make this literally an extension of f we identify each $n \in \mathbb{N}$ with the one element sequence $\langle n \rangle$.) Note that the lexicographic ordering on finite sequences is a well ordering by Exercise I.1.8(b).

(ii) A goal clause G is *bounded* (with respect to f and $\mathcal{M}$) if there is a maximal element in $\{|G^*| : G^* \in G'\}$. (Recall that G' is the set of ground instances of G.) If G is bounded, we denote this maximal element by $|G|$. (Note that with the identification of n and $\langle n \rangle$ this agrees with the definition in (i) if G is ground.)

The use of the word bounded is justified by the following lemma.

Lemma 4.4: *$G = \{\neg A_1, \ldots, \neg A_n\}$ is bounded (with respect to f and $\mathcal{M}$) if and only if there is a sequence τ such that $G^* \leq \tau$ for every $G^* \in G'$.*

Proof: The "only if" direction is immediate. Suppose then that there is a τ as described in the Lemma. Let t be a number larger than any element of τ. If $G^* \in G'$, then there are at most n elements in the sequence $|G^*|$. By the choice of τ and t, each of these elements is less than t. Thus there are only finitely many sequences of the form $|G^*|$ for $G^* \in G'$. As every finite set has a maximum in any ordering we are done. □

Our next task is to prove that any acceptable program which starts with a bounded goal must terminate. As the ordering induced on the goal clauses is a well ordering, it suffices to show that resolutions with clauses in P decreases the level of the goal clause. We begin with ground resolutions.

Lemma 4.5: *Let P be acceptable with respect to a level mapping f and a model $\mathcal{M}$. If $G = \{\neg A_1, \ldots, \neg A_n\}$ is a ground goal and $H = \{\neg B_1, \ldots, \neg B_m, \neg A_2, \ldots, \neg A_n\}$ is an* SLD*-resolvent of G with some ground clause $C = \{B, \neg B_1, \ldots, \neg B_m\}$ in P', then $|H| < |G|$.*

Proof: We proceed by cases. First, suppose that $\mathcal{M}(G) = 1$, i.e., $\mathcal{M} \not\models A_1$ and so by definition $|G| = \langle |A_1| \rangle$. As we resolved G and C, $B = A_1$ and it is false in $\mathcal{M}$ by assumption. As $C \in P'$, it must be true in the model $\mathcal{M}$ of P. Thus, B_i must be false in $\mathcal{M}$ for some $i \leq m$ and so by definition $\mathcal{M}(H) \leq m$. As $|B_i| < |A_1|$ for every $i \leq m$ by acceptability, $|H| < |G| = |A_1|$ by the definition of the ordering.

Next suppose that $\mathcal{M}(G) > 1$. In this case H and G succeed in $\mathcal{M}$ for the first time at the same literal, i.e., $\mathcal{M}(H) = \mathcal{M}(G) + m - 1$. Thus the sequence $|H|$ has the same elements as $|G|$ except that $|A_1|$ is replaced by the set of elements $|B_i|$ for $1 \leq i \leq m$. As $|B_i| < |A|$ for each i, it is clear from the definition of the extension of the level mapping to clauses and the ordering on these sequences that $|H| < |G|$ as required. □

We now prove our lemma for bounded goals.

Lemma 4.6: *Let P be acceptable with respect to a level mapping f and a model $\mathcal{M}$. If $G = \{\neg A_1, \ldots, \neg A_n\}$ is a bounded (with respect to f and $\mathcal{M}$) goal and $H = \{\neg B_1, \ldots, \neg B_m, \neg A_2, \ldots, \neg A_n\}\theta$ an* SLD*-resolvent of G with some $C = \{B, \neg B_1, \ldots, \neg B_m\}$ in P, then H is bounded and $|H| < |G|$.*

Proof: Consider any ground instance $H\gamma$ of H. By extending γ if necessary, we may assume that $\theta\gamma$ also grounds B. $H\gamma$ is then a resolvent of $G\theta\gamma$ and $C\theta\gamma \in P'$ and so by Lemma 4.5 $|H\gamma| < |G\theta\gamma|$. As G is bounded, $|G\theta\gamma| \leq |G|$. As $H\gamma$ was an arbitrary ground instance of H, H is bounded by Lemma 4.4. If we now choose γ so that $|H\gamma| = |H|$ we see that $|H| < |G|$ as required. □

Theorem 4.7: *If P is acceptable and G is a goal clause which is bounded (with respect to any level mapping and model showing that P is acceptable) then every* SLD*-proof from $P \cup \{G\}$ beginning with G is finite.*

Proof: Consider any SLD–proof from $P \cup \{G\}$ starting with $G = G_0$. Each successive resolution produces a new goal clause G_n. By Lemma 4.6, the sequence $|G_n|$ is strictly decreasing. As the ordering on goal clauses is a well ordering, the sequence of resolutions must be finite. □

Corollary 4.8: *Every acceptable program is left–terminating.*

Proof: The corollary follows from the theorem as every ground goal is by definition bounded. □

We now wish to characterize the left–terminating programs by proving the converse to Corollary 4.8. We also want to deal with nonground goals by proving some form of converse to Theorem 4.7. We begin with the ingredients of our level mapping.

Definition 4.9: If the SLD–tree from $P \cup \{G\}$ beginning with G is finite, $N(G)$ is the number of its nodes; otherwise, $N(G)$ is undefined.

Theorem 4.10: *If P is left–terminating, there is a level mapping f and a model $\mathcal{M}$ such that*

(i) *P is acceptable with respect to f and $\mathcal{M}$ and*

(ii) *For every goal clause G, G is bounded with respect to f and $\mathcal{M}$ if and only if every* SLD*-proof from $P \cup \{G\}$ beginning with G is finite.*

Proof: We define the required level mapping f and model $\mathcal{M}$ by setting $|A| = N(\{\neg A\})$ for each atomic sentence A of $\mathcal{L}$ and requiring that $\mathcal{M} \models A \Leftrightarrow$ there is an SLD–refutation of $P \cup \{\neg A\}$. Note that, as we are assuming that P is left–terminating, f is well defined. Also, by the completeness of SLD–resolution, each atomic sentence A is true in $\mathcal{M}$ if and only if it is a logical consequence of P. We now prove that f and $\mathcal{M}$ have the desired properties.

(i) Consider any clause $C = A :- B_1, \ldots, B_m$ in P'. Let $n = \mathcal{M}(B_1, \ldots, B_m)$. There is an SLD–proof from $P \cup \{\neg A\}$ beginning with $\{\neg A\}$ and a resolution with C. By the definition of $\mathcal{M}$, there is an SLD–refutation of $P \cup \{\neg B_i\}$ beginning with $\neg B_i$ for each $i < n$. The SLD–search tree for each such refutation is clearly a subtree of the SLD–tree for $\{\neg A\}$. As each such search for $i < n$ succeeds, the SLD–tree for $\neg B_n$ is attached to the end of each successful search for refutations of all of the $\neg B_i$ for $i < n$. Thus the SLD–tree for $\{\neg A\}$ contains copies of the SLD–tree for $\{\neg B_i\}$ for every $i \leq n$. The definition of f then tells us that $|A| > |B_i|$ for each $i \leq n$ as required.

(ii) Let G be a bounded goal clause. Suppose, for the sake of a contradiction, that there is a nonterminating SLD–proof $\langle G_0, C_0 \rangle, \langle G_1, C_1 \rangle, \ldots$ starting with $G = G_0$. For any n, we can (starting with $\langle G_n, C_n \rangle$) find a substitution θ which grounds all the clauses of $\langle G_0, C_0 \rangle, \langle G_1, C_1 \rangle, \ldots, \langle G_n, C_n \rangle$. This gives an SLD–proof beginning with the ground instance $G\theta$ of G of length n. As n was arbitrary, this contradicts the assumption that G is bounded.

Finally, suppose that every SLD–proof beginning with G terminates. Notice that the SLD–tree from any goal clause is finitely branching: Each immediate successor of a fixed node corresponds to a clause in the finite program P. Thus, by König's lemma (Theorem I.1.4) the SLD–tree for G is finite. Suppose it has n nodes. Again, as every SLD–tree has branchings

of at most the number of clauses in P, there can be SLD–trees from ground instances of G with arbitrarily large numbers of nodes only if there are ones of arbitrarily large depth. Thus, if G is not bounded, there is an SLD–proof beginning with a ground instance of G of length $n+1$. The lifting lemma for SLD–proofs (Exercise 1.6) then lifts such a proof to one of length $n+1$ beginning with G. As this proof must be a path on the SLD–tree beginning with G we have the desired contradiction. □

Corollary 4.11: *P is left–terminating if and only if it is acceptable.*

As an example we will show that the program PERM for deciding if one list is a permutation of another is left–terminating. The language of our program consists of a constant [ ] for the empty list and the binary list combining function "." of Example II.3.15. We also use the alternate notations and abbreviations introduced there such as $[a] \mid [b,c,d]$ for $[a,b,c,d]$. The ground terms of the language, i.e., its Herbrand universe H, consists of the closure of the constant [ ] under the list combining operation. The program for PERM includes a program APPEND (consisting of (a1) and (a2) below) for appending one list to another as well as two clauses (p1) and (p2) defining PERM from APPEND:

$(a1)$ $a([\,],Y,Y).$

$(a2)$ $a([X|Y],Z,[X|W]) \text{ :- } a(Y,Z,W).$

$(p1)$ $p([\,],[\,]).$

$(p2)$ $p(X,[Y,Z]) \text{ :- } a(W,[Y|V],X), a(W,V,U), p(U,Z).$

Now not all LD–proofs from PERM starting with a ground goal are finite (Exercise 3) but we will show that PERM is acceptable and so all SLD–proofs starting with ground goals will terminate.

Theorem 4.12: PERM *is acceptable (and so left–terminating).*

Proof: We first let $|x|$ be the length of x for any list x in the Herbrand universe, H. Thus, for example, $|[y|v]| = |v|+1$ for all $y,v \in H$. We define a level mapping by setting $|p(x,y)| = |x|+|y|+1$ and $|a(x,y,z)| = \min\{|x|,|z|\}$. As our model we could take the intended interpretations for p and a on B (Exercise 6). Rather than verifying the semantic correctness of the program, however, it is easier to create an artificial model that embodies just enough to cut off the resolutions when needed. We define $\mathcal{M}$ with universe H by saying that $p(x,y)$ holds for every $x,y \in H$ and that $a(x,y,z)$ holds iff $|x|+|y| = |z|$. It is obvious from the definitions that $\mathcal{M}$ is a model of PERM. We prove that, with this choice of level mapping and model, PERM is acceptable.

We only have to check ground instances of clauses (a2) and (p2). For (a2) just note that, for any $x,y \in H$, $|y| < |[x|y]|$ by definition. Thus for any $x,y,z,w \in H$,

$$|a(y,z,w)| = \min\{|y|,|w|\} < |a([x|y],z,[x|w]) = \min\{|[x|y]|,|[x|w]|\}$$

as required. Now consider any ground instance of (p2):

$$p(x,[y,z]) \text{ :- } a(w,[y|v],x),\ a(w,v,u),\ p(u,z).$$

It is clear that

$$|p(x,[y,z])| = |x|+|[y|z]|+1 > |a(w,[y|v],x)| = \min\{|w|,|x|\}.$$

If $|w|+|[y|v]| \neq |x|$, then we are done by our choice of $\mathcal{M}$ and the definition of acceptability. Suppose then that $|w|+|[y|v]| = |x|$ and so $|w| \leq |x|$. Thus $|p(x,[y,z])| = |x|+|[y|z]|+1 > |w| \geq |a(w,v,u)|$. Once again we are done unless $|w|+|v| = |u|$ as well. In this case, $|u| < |x|$ and so $|p(x,[y,z])| = |x|+|[y|z]|+1 > |u|+|[y|z]|+1 = |p(u,z)|$ as required to complete the proof of acceptability. □

As acceptability implies left–termination, we have shown that PERM running with the standard implementation of PROLOG will terminate on any ground clause. As the logical consequences of PERM are the intended ones by Exercise 5, PERM will terminate with a correct answer on any ground goal. Thus we have a proven method for checking if one list is a permutation of another. More interestingly, we can use the characterization of termination in terms of boundedness to prove that it can do much more. For example, by starting with a goal of the form $G = \{\neg p(x,X)\}$ we would hope to be able to find all the permutations of a given list x. To see this, it suffices to prove that G is bounded. We prove much more.

Theorem 4.13: *For all terms $t,t_1,\ldots,t_n$ of $\mathcal{L}$, every goal G of the form $\{\neg p([t_1,\ldots,t_n],t)\}$ is bounded (with respect to the level mapping and model of the proof of Theorem 4.12).*

Proof: For any ground instance $G\gamma$ of G, $|G\gamma| = n+m+1$ where $m = |t\gamma|$. As the length of $t\gamma$ is constant for any ground substitution γ, G is bounded. □

Many other types of goal clauses can be proven bounded for PERM. See, for example, Exercise 7.

Exercises

Definition:

(i) P is *terminating for a goal G* if all LD–proofs from $P \cup \{G\}$ starting with G are finite. P is terminating if it is terminating for every ground goal G.

(ii) P is *recurrent with respect to a level mapping f* if, for every clause $A \text{ :- } A_1,\ldots,A_n$ in P', $|A| > |A_i|$ for each $1 \leq i \leq n$. P is *recurrent if* it is recurrent with respect to some level mapping.

1. Prove that if P is recurrent it is terminating.
2. Prove that if P is terminating it is recurrent.
3. Prove that PERM is not terminating.
4. Prove that APPEND is recurrent.
5. Prove that the logical consequences of PERM of the form $p(x,y)$ for $x,y \in B$ are the intended ones.
6. Prove that one could use the intended interpretation of p and a on B as the model in the proof of Theorem 4.12.
7. Suppose $G = \{\neg A_1, \ldots, \neg A_n\}$. Prove that, if each A_i is bounded (with respect to some level mapping and model), then so is G.

5. Equality

Until now, we have ignored the whole question of mathematical equality. (Recall that in PROLOG, "$t_1 = t_2$" is used to mean that t_1 and t_2 can be unified.) The time has come to at least face the problem of "true" equality for it is indeed a problem for PROLOG. Syntactically, we introduce a special (reserved) two–place predicate written infix as $x = y$. (The use of $=$ for equality is too widespread to give it up in our exposition simply because PROLOG syntax took it for some other use. In all contexts other than PROLOG programs, we will use "=" for the equality relation.) We must now expand our deduction methods and semantics for predicate calculus to deal with this new special predicate.

The basic properties of equality (in a language $\mathcal{L}$) are captured by the following:

Definition 5.1: *The equality axioms for $\mathcal{L}$:*

(1) $x = x$.

(2) $x_1 = y_1 \wedge \ldots \wedge x_n = y_n \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$ for each n–ary function symbol f of $\mathcal{L}$.

(3) $x_1 = y_1 \wedge \ldots \wedge x_n = y_n \rightarrow (P(x_1, \ldots, x_n) \rightarrow P(y_1, \ldots, y_n))$ for each n–ary predicate symbol P of $\mathcal{L}$ (including the binary one "=").

Reflexivity of "=" is guaranteed by (1). The other usual properties of equality (symmetry and transitivity) follow from (3) by taking P to be $=$ (Exercise 4).

We can now think of these axioms as being added to any set of sentences we are dealing with under any proof system. Thus, by a tableau refutation proof of a sentence S with "=" in its language, we mean one from the set of sentences S^* where S^* is S plus the universal closures of (1)–(3) for all the function and predicate symbols of S. Similarly, a resolution refutation

of S is one from S^*. Unfortunately, simply adding in these clauses makes for a very inefficient procedure. We will return to this point shortly when we consider more specialized methods such as paramodulation.

The next step is to decide on the intended semantics for equality. Here there are two choices. We can treat equality as we did all other predicates and simply require that the interpretation of "=" be a two–place relation which satisfies all the equality axioms. From the proof theoretic, and so the PROLOG point of view, this is the only approach on which we have any real handle, and within the confines of a fixed language, it is all we can say about equality. On the other hand, from an abstract mathematical point of view, we would like to require that "=" always be interpreted as true equality.

We can, in fact, require this of our interpretations and still prove all the basic theorems of II.7 as before: soundness, completeness, compactness, etc.. The only problem arises in the proof of the completeness theorem. In the proof of Theorem II.7.3, our construction via the CST gives us a structure in which the interpretation of "=" satisfies all the above axioms but this does not guarantee that it is true equality. The solution is to divide out by the equivalence relation induced by "=". To be precise, let $\mathcal{A}$ be the structure determined by a noncontradictory path on the CST for a given set of sentence S. The elements of $\mathcal{A}$, we recall, are the ground terms t of a language $\mathcal{L}$. We define a relation $\equiv$ on them by: $t_1 \equiv t_2 \Leftrightarrow \mathcal{A} \vDash t_1 = t_2$. Using the equality axioms, it is easy to see that $\equiv$ is an equivalence relation (i.e., $t \equiv t$ for every t, if $t_1 \equiv t_2$ then $t_2 \equiv t_1$ and if $t_1 \equiv t_2$ and $t_2 \equiv t_3$, then $t_1 \equiv t_3$). We then define a structure $\mathcal{B}$ for $\mathcal{L}$ on the equivalence classes of $\equiv$. That is, the elements of $\mathcal{B}$ are the sets of the form $[t_1] = \{t | t \equiv t_1\}$ for each $t_1 \in \mathcal{A}$. The functions and relations are defined on $\mathcal{B}$ by choosing representatives and referring to $\mathcal{A}$: $\mathcal{B} \vDash P([t]_1, \ldots, [t_n]) \Leftrightarrow \mathcal{A} \vDash P(t_1, \ldots, t_n)$ (for P other than "=") and $f^{\mathcal{B}}([t]_1, \ldots, [t_n]) = [f^{\mathcal{A}}(t_1, \ldots, t_n)]$. Of course "=" is interpreted as true equality in $\mathcal{B}$. At this point, one must check that these definitions are independent of the choice of representatives (that is, the elements t_i chosen from the sets $[t_i]$). The final step is to show by induction, as we did for $\mathcal{A}$, that $\mathcal{B}$ agrees with every signed statement on the path used to construct $\mathcal{A}$. Thus $\mathcal{B}$ is the required model for S in which "=" is interpreted as true equality.

Theorem 5.2 (Completeness): *If S is any set of sentences which includes the equality axioms for the language of S, then either there is a tableau proof of $\square$ from S or there is a model for S in which $=$ is interpreted as true equality.*

As our main concern now is with the proof theoretic, i.e., resolution method, point of view we leave the details of this construction and the proof of the appropriate compactness theorem for such interpretations as exercises. From now on we use "=" simply as a reserved predicate symbol.

Definition 5.3: An *equality structure for a language* $\mathcal{L}$ with "=" is any structure for $\mathcal{L}$ which satisfies the equality axioms. Similarly, an *equality model of a set of sentences* S of $\mathcal{L}$ is an equality structure for $\mathcal{L}$ in which all sentences of S are true. An *equality resolution (or tableau) proof* from S is then one from S plus the equality axioms.

The soundness, completeness and compactness theorems for resolution (or tableaux) are then by definition true for equality interpretations and proofs. In terms of actually carrying out equality resolutions with any sort of efficiency, however, we are considerably worse off than in the original case. There are simply too many new rules. At this point we will give one revised resolution rule to handle equality that goes a long way towards alleviating the problem.

The inference scheme (paramodulation) we want will take the place of the equality axioms (2) and (3). That is, we want a rule (like resolution) which, when combined with resolution, will be complete for equality interpretations: If $\{x = x\} \in S$ and S has no equality model, then $\square$ is derivable from S using resolution and paramodulation. (The point here is that S may mention "=" but contain no equality axioms other than $x = x$.) The basic idea is that if we have a clause C_1 containing a literal $L(t,\dots)$ in which a term t occurs and a clause C_2 (with no variables in common with C_1) containing $t = s$ then we can conclude from C_1 and C_2 not only $C_1 \cup C_2$ but also the modification of $C_1 \cup C_2$ in which we replace t by s in $L(t,\dots)$. Of course we need not replace t by s everywhere in L. Thus, we want to consider replacement of a single occurrence of t by s in L. (Obviously, multiple replacements can then be generated by repeated uses of the rule.) We use $L[t/s]$ to represent the result of replacing some one occurrence of t by s in L.

Example 5.4: From $C_1 = \{\neg P(a),\ Q(b)\}$ and $C_2 = \{a = b, R(b)\}$ conclude $\{\neg P(b), Q(b), R(b)\}$. Note that we also drop $a = b$ from the result. As in resolution, it has been used and absorbed.

Of course as in resolution, we must consider the possibilities introduced by unifications.

Example 5.5: From $C_1 = \{\neg P(x), Q(x)\}$ and $C_2 = \{b = b, R(b)\}$ we can conclude $\{\neg P(b), Q(b), R(b)\}$. This is simply instantiation via unification.

More generally we should consider the following:

Example 5.6: From $C_1 = \{\neg P(f(g(x))), Q(x)\}$ and $C_2 = \{g(h(c)) = a, R(c)\}$ we can conclude $\{\neg P(f(a)), Q(h(c)), R(c)\}$. Here $g(x)$ is the term t being considered. We unified it with $g(h(c))$ by the substitution $\{x/h(c)\}$. After applying this substitution to C_1 to get $\{\neg P(fgh(c)), Q(h(c))\}$ we replaced the occurrence of $gh(c)$ by a as allowed by $gh(c) = a$ of C_2 and combined to get our result.

As with resolution, we may also collapse literals via unification before applying the substitution. Here it is necessary to separate these operations.

Definition 5.7: Suppose we can rename the variables of C_1 and C_2 so that they have no variables in common and C_1 is of the form $\{L(t_1),\dots,L(t_n)\} \sqcup C_1'$ and C_2 is of the form $\{r_1 = s_1,\dots,r_m = s_m\} \sqcup C_2'$. If σ_1 is an mgu for $\{L(t_1),\dots,L(t_n)\}$, σ_2 one for $\{r_1 = s_1,\dots,r_m = s_m\}$ and σ one for $\{t_1\sigma_1, r_1\sigma_2\}$ then any clause of the form

$$\{L\ \sigma_1\sigma[t_1\sigma_1\sigma/s_1\sigma_2\sigma]\} \cup C_1'\sigma_1\sigma \cup C_2'\sigma_2\sigma$$

is a *paramodulant of* C_1 and C_2.

Together with resolution, this rule is complete for equality interpretations. In fact, as with resolution, a linear version is also complete.

Theorem 5.8: *If* $\{x = x\} \in S$, $C \in S$ *and* $S - \{C\}$ *has an equality interpretation but* S *does not, then there is a linear proof of* $\square$ *from* S *starting with* C *via resolution and paramodulation.* (As you would expect, such a linear proof is a sequence $\langle C_0, B_0\rangle,\dots,\langle C_n, B_n\rangle$ in which $C_0 = C$, $C_{n+1} = \square$, each B_i is in S or is a C_j for $j < i$ and each C_{i+1} follows from C_i and B_i via one resolution or paramodulation step.)

The proof is very much like that for resolution alone and we omit it. The general problem of dealing with just equality is quite complicated. It is a well developed subject on its own, that of rewrite rules. The problems of integrating equality with PROLOG or more general theorem provers is an as yet underdeveloped topic of current research that lies beyond the scope of this book.

Exercises

1. Prove that the tableau method with equality axioms is sound for the class of interpretations in which = is interpreted as true equality.
2. Give a complete proof of the completeness theorem for tableau proofs (from the equality axioms) with the true equality interpretation (Theorem 5.2).
3. Prove the compactness theorem (a set of sentences is satisfiable iff every finite subset is) for the true equality interpretation.
4. Prove that the symmetry and transitivity of "=" follow from (1)–(3) of Definition 5.1. (Hint: For the first let $x_1 = x$, $x_2 = x$, $y_1 = y$ and $y_2 = x$ in (3). For the second let $x_1 = x$, $x_2 = y$, $y_1 = x$ and $z_2 = z$.)

6. Negation as Failure

PROLOG, as we have described it so far, has no way to derive negative information. (The Herbrand structure for a PROLOG program P in which every ground atomic formula is true is obviously a model of P.) Nonetheless, it is often the case that we want to conclude that certain facts do not hold.

In PROLOG derivations, negation is implemented in terms of failure, i.e., the goal $\neg A$, often written "not(A)", succeeds if and only if PROLOG would return "no" to the goal A. There are a number of ways of understanding negation which are used to justify this implementation. The two earliest are the *closed world assumption* (CWA) and the *completed database* (CDB) view. The first falls outside the scope of predicate logic so we will explain it fairly briefly; we will give a more detailed treatment of the second. These two approaches also apply to a more general implementation of negation as failure which allows arbitrary clauses in the program as well as in the goal.

In the next section we will describe a more recent approach to negation, that of stable models. The characterization of such models also goes beyond predicate logic. It is closely related to nonmonotonic logic which we will also describe in §7.

The CWA arises naturally in the context of databases. If we are given a database of grades for students in the math department, we may have reason to believe that it is a correct and complete list. Thus, if the fact that Jones got an A in Math 486 does not appear, we may reasonably assume that it is false. The extension of this principle to a setting like PROLOG with rules as well as data leads to the CWA for a program P: If a ground atomic formula (positive literal) A is not a logical consequence of P then we may infer $\neg A$.

The first thing to note here is that the CWA deals with the abstract notion of logical consequence or, equivalently, provability in some complete proof systems for predicate logic. By the undecidability of provability in predicate logic (Corollary 8.10), however, we cannot hope to implement such a rule even in theory. The closest we can expect to come is to conclude $\neg A$ when we have a proof that A is not a logical consequence of P. For a PROLOG–like system such a proof might reasonably consist of a finite SLD–tree starting with the goal $\neg A$ in which every branch ends a failure. In this case, we know that there is no SLD–refutation starting with the given goal. The completeness theorem for SLD–refutations (Theorem 1.8) then tells us that A is not a logical consequence of P. Such a tree is called a *finitely failed* SLD*–tree for* $P \cup \{A\}$.

The usual implementations of PROLOG only check for a finitely failed SLD–tree via the standard selection rule. For theoretical analyses, however, we are better off considering a more general definition which has a clearer semantic content. To this end, we will have to consider refutation search procedures which do not follow the standard rule or even any selection rule that always chooses the same literal from a given goal. (See Exercise 2.)

We begin with a generalized notion of a selection rule that makes its choice of literal at any step based on the entire history of the proof up to that point rather than on just the current goal clause.

Definition 6.1:

(i) A *generalized selection rule* R is a function which, given any LD–proof $\langle G_0, C_0\rangle, \ldots, \langle G_n, C_n\rangle$, chooses a literal from G_n.

(ii) An LD–proof $\langle G_0, C_0\rangle, \ldots, \langle G_n, C_n\rangle$ is a *proof via a generalized selection rule* R if the literal resolved on at each step i, $0 \leq i < n$ is the one chosen by R from the proof $\langle G_0, C_0\rangle, \ldots, \langle G_i, C_i\rangle$ up to that point.

We now give the formal definition of the SLD–tree associated with a given goal, program and generalized selection rule.

Definition 6.2: Let P be PROLOG program, G a goal clause and R a generalized selection rule. The *associated* SLD*–tree (from* P *starting with* G *via* R) is a finitely branching tree T labeled with goal clauses such that each path of T is associated with an SLD proof via R. We define T by induction. The root node of T is labeled with G. If any node σ of T is labeled with a goal clause G' and the generalized selection rule R chooses the literal $\neg A$ from G' given the proof associated with the path through T leading to G', then the immediate successors of σ correspond to the clauses C_i of P which can be resolved with G' on $\neg A$. These nodes are labeled with the results of resolving G' on $\neg A$ with the corresponding clause C_i of P. (The proof associated with the path to a successor of G' is the one associated with the path to G' followed by the appropriate resolution.)

Note that, in general, the SLD–tree associated with some P, G and R may be infinite. Some paths may succeed, i.e., end with $\square$ (success) and so be (necessarily) finite refutations of G from P. Others may be failed, i.e., there is no clause of P with which the final goal can be resolved on the selected literal. Other paths, however, may never terminate.

We can now approximate the set of literals A which are not logical consequences of a program P by considering those for which the search for a refutation of A fails in a finite and hence observable way.

Definition 6.3:

(i) The SLD–tree associated with P, G and R is *finitely failed* if it is finite and every path ends because of a failure to find a clause of P with which the selected literal can be resolved. (In particular no path ends with success, i.e., $\square$.)

(ii) The SLD *finite failure set* of a PROLOG program P is the set of ground atoms A such that there is a generalized selection rule R such that the SLD–tree associated with P, $\{\neg A\}$ and R is finitely failed.

Recall the contrast between breadth–first and depth–first searches of SLD–trees in terms of completeness. In this setting, there are generalized selection rules R which guarantee that if A is in the finite failure set of P then the SLD–tree via R is finitely failed and there are others which do not have this property.

Definition 6.4: An SLD–proof (via R) is *fair* if it is either finite (and so either failed or a successful refutation) or, for every occurrence of a literal Q in the proof (say in G_i), either R selects Q at step i or there is a stage $j > i$ at which $Q\theta_i \dots \theta_{j-1}$ is selected by R, where θ_k is the mgu used at step k of the proof. A generalized selection rule R is *fair* if every SLD-proof via R is fair.

It is not hard to design a fair generalized selection rule R. However, no ordinary selection rule can be fair. (See Exercises 1–2.) The following result says that we can restrict our attention to any fair generalized selection rule.

Theorem 6.5: *For a program P and ground atomic formula A, A is in the SLD–finite failure set of P iff the SLD–tree for A via R is failed for every fair generalized selection rule R. Thus, there is a finitely failed SLD–tree for A using any selection rule if and only if the SLD–tree for A is finitely failed for any fair generalized selection rule.*

Proof: The "if" direction is immediate from the definition of the finite failure set. Suppose, therefore, that A is in the finite failure set of P and R is any fair generalized selection rule. We wish to prove that the SLD–tree via R starting with $\{\neg A\}$ is finitely failed. We prove a stronger lemma by induction.

Lemma 6.6: *Let P be a PROLOG program and R a fair generalized selection rule. If a goal clause $G = \{\neg A_1, \dots, \neg A_m\}$ has a finitely failed SLD–tree of depth k (via any generalized selection rule) and $\langle G_0, C_0 \rangle, \dots, \langle G_n, C_n \rangle$ is an SLD–proof from P via R with $G_n = G$, then every path on the SLD–tree via R that has $\langle G_0, C_0 \rangle, \dots, \langle G_n, C_n \rangle$ as an initial segment is finitely failed.*

Proof: We proceed by induction on k. Suppose $k = 1$ and A_s is the literal selected from G in the given failed SLD–tree S of depth 1. By the definition of a failed SLD–tree, no clause in P has a head that will unify with A_s. Now consider any path Q on the SLD–tree T via R starting with $\langle G_0, C_0 \rangle, \dots, \langle G_n, C_n \rangle$. As R is fair there is a node on Q at which R chooses $A_s\theta$ for some substitution θ. By our assumption about A_s, no clause of P has a head which can be unified with $A_s\theta$. Thus the path Q terminates with a failure at this point. As Q was an arbitrary path on T with the specified initial segment, T is finitely failed below this point.

For the induction step, let A_s be the literal chosen at the first level of the given finitely failed SLD–tree S of depth $k+1$. The first level of S then consists of all the resolvents of all resolutions on A_s with clauses in P. Each node at this level then has the form

$$H = \{\neg A_1, \dots, \neg A_{s-1}, \neg B_1, \dots, \neg B_k, \neg A_{s+1}, \dots, \neg A_n\}\theta$$

where θ is the mgu associated with the appropriate resolution and clause C of P. Note that H has a finitely failed SLD–tree of depth k.

Now let Q be any path on the SLD–tree T via R starting with $\langle G_0, C_0 \rangle$, $\dots, \langle G_n, C_n \rangle$ with $G_n = G$. Again, by the fairness of R, there is some level, say $m-1$, of Q at which we first select a literal of the form $A_s\psi$ coming from A_s in G. Let $\langle G_0, C_0 \rangle, \dots, \langle G_m, C_m \rangle$ be the path up to the point at which we have performed the resolution on $A_s\psi$. The last resolution on this path was with some $C \in P$ whose head unified with $A_s\psi$ and so with A_s. The proof of the switching lemma (Lemma 1.11) shows that G_m is a renaming of the clause H on the first level of S which corresponds to C_{m-1}. As starting a finitely failed SLD–tree is obviously invariant under renamings, G_m starts a finitely failed SLD–tree of depth k and we can apply the induction hypothesis to $\langle G_0, C_0 \rangle, \dots, \langle G_m, C_m \rangle$ to conclude that Q is finitely failed. □

In view of Theorem 6.5, we may assume that we have specified any fair generalized selection rule R to define our SLD–trees. It is now reasonably clear how a PROLOG type system equipped with an implementation of a fair R should attack a question asking for negative as well as positive conclusions. We start with a clause G containing both positive and negative literals. We then carry out an SLD–proof via R except that when we select a positive literal A we try to construct a finitely failed SLD–tree via R starting with A. If we succeed, we eliminate the literal A and continue. If we fail, the attempt at refutation of G fails as well. We formalize this procedure as SLDNF–refutations (SLD–refutations with negation as failure).

Definition 6.7:

(i) A *general goal clause* G is simply an arbitrary clause.

(ii) Let P be a PROLOG program. An SLDNF–*proof via R from P beginning with G* is a sequence $\langle G_i, C_i \rangle$ of general goal clauses G_i and clauses $C_i \in P$ with $G_0 = G$ and $G_{n+1} = \square$ which is generated as follows: If $R(G_i)$, the literal chosen by R, is negative, then G_{i+1} is a resolvent of G_i and C_i via mgu θ_i on literal $R(G_i)$. If $R(G_i)$ is positive, it must be a ground literal A. In this case, there must be a finitely failed SLD–tree (via R) starting with the goal $\neg A$. We then have G_{i+1} equal to G_i with A deleted, C_i plays no role and θ_i is the identity.

As usual, the composition sequence of mgu's $\theta_0 \dots \theta_1 = \theta$ for such a proof is called its *answer substitution*.

The definition of the SLDNF–*tree* from P starting with G via R is the same as that of the corresponding SLD–tree (Definition 6.2), modulo the modification in (ii). A path on the tree is just an attempt to construct such an SLDNF–refutation. The path *succeeds* if the attempt eventually produces □. Suppose at some point on such a path we encounter an SLD–tree T starting with some $\neg R(G_i)$ where $R(G_i)$ is a positive ground literal. If T has □ on one of its paths, T is not finitely failed. In this case, we say that this attempt at finding an SLDNF–refutation *fails*. (Of course, even if the SLDNF–tree is not finitely failed, we may never discover this fact and the proof procedure may simply fall into an infinite search.) If $R(G_i)$ is positive but not ground, we say that the proof *flounders* at this point.

Warning: We allow the SLDNF–refutation to proceed when R chooses a positive literal only if it is ground. Such a choice is called *safe*. This restriction is essential as we are interpreting the success of $\neg q$ as the failure of q. If q has a free variable X, this is clearly unfounded. A question "?- $\neg q(X)$." asks for a c such that $\neg q(c)$ holds while "?- $q(X)$." asks for a d such that $q(d)$ holds. Clearly neither one is the negation or failure of the other. Unfortunately, most PROLOG systems do not bother to check that a positive literal is ground before applying the negation as failure procedure. This can lead to unexpected (and indeed false) results.

Before describing the relationship between negation as failure and the CWA, we introduce a more general approach in terms of "*completed databases*" (CDB). The idea here is that, when one specifies the conditions under which something occurs, one specifies them all. In terms of a particular program P, suppose we consider one n–ary predicate r and all the clauses of P with r in their heads:

$$r(t_{1,1},\dots,t_{1,n}) \text{ :- } q_{1,1},\dots,q_{1,n_1}.$$
$$\vdots$$
$$r(t_{k,1},\dots,t_{k,n}) \text{ :- } q_{k,1},\dots,q_{k,n_k}.$$

If we view this list as a complete description of when $r(X_1,\dots,X_n)$ holds (for new variables X_i), then we can express this view by the sentence

$$r(X_1,\dots,X_n) \leftrightarrow q_1 \vee \dots \vee q_k.$$

where $Y_1,\dots,Y_{p_i}$ are the variables in $q_{i,1},\dots,q_{i,n_i}$, $X_1,\dots,X_n$ are new variables and q_i is $\exists Y_1,\dots,Y_{p_i}(X_1 = t_{i,1} \wedge \dots \wedge X_n = t_{i,n} \wedge q_{i,1} \wedge \dots \wedge q_{i,n_i})$. The "if" ($\leftarrow$) direction in these equivalences is simply the assertions of the given program. The "only if" ($\rightarrow$) direction say that we have completely specified r by the program clauses. Comp(P), *the completion of* P, includes such an axiom for each predicate r appearing in P. If r does not occur in the head of any clause of P, we include the axiom $\forall \vec{X} \neg r(\vec{X})$ in Comp(P). In the absence of equality, Comp(P) consists of these axioms and no others.

To deal with equality, we include in Comp(P) the basic equality axioms (1)–(3) of §5 for the language of P. In addition, the database point of view dictates that distinct terms (names) represent distinct objects. We incorporate this point of view (to the extent possible in first order logic) by including the following axioms in Comp(P) as well:

(4) $f(x_1,\dots,x_n) \neq g(y_1,\dots,y_m)$ for each distinct pair of function symbols f and g of arities n , $m \geq 0$ respectively.

(5) $t(x) \neq x$ for each term $t(x)$ (other than x itself) in which x occurs.

(6) $f(x_1,\dots,x_n) = f(y_1,\dots,y_n) \rightarrow x_1 = y_1 \wedge \dots \wedge x_n = y_n$ for each n–ary function symbol f.

This completes the construction of Comp(P) from P. Every clause in P is clearly a consequence of Comp(P) and so Comp(P) $\models P$. Moreover, if P is a PROLOG program, Comp(P) is consistent. (Again, the Herbrand structure in which "=" is interpreted as true equality and every other ground atomic formula is true is a model.) We can now use Comp(P) to prove soundness and completeness theorems "justifying" the negation as failure rule. We begin with a lemma relating unification and the equality axioms (1)–(6).

Lemma 6.8: *Let* $S = \{s_1 = t_1,\dots,s_n = t_n\}$.

(i) *If* S *is unifiable and* $\theta = \{x_1/r_1,\dots,x_m/r_m\}$ *is the mgu given by the unification algorithm* (II.12.3), *then* (1)–(6) $\models (s_1 = t_1 \wedge \dots \wedge s_n = t_n) \rightarrow (x_1 = r_1 \wedge \dots \wedge x_m = r_m)$.

(ii) *If* S *is not unifiable, then* (1)–(6) $\models (s_1 = t_1 \wedge \dots \wedge s_n = t_n) \rightarrow A \wedge \neg A$ *for any sentence* A.

Proof: Consider the unification algorithm as applied to S. It produces a sequence of substitutions $\theta_0, \theta_1, \dots, \theta_n$. Let $\{x_1/r_{k,1},\dots,x_m/r_{k,m}\}$ be the composition $\theta_0 \dots \theta_k$. One proves by induction on k that (1)–(6) $\models (s_1 = t_1 \wedge \dots \wedge s_n = t_n)\theta_0 \dots \theta_k \rightarrow (x_1 = r_{k,1} \wedge \dots \wedge x_m = r_{k,m})$ for each k up to the point at which the algorithm terminates. If it terminates with a unifier we have proved (i). If it terminates with the announcement that S is not unifiable, it is easy to see that (1)–(6) prove that no instance of $(s_1 = t_1 \wedge \dots \wedge s_n = t_n)\theta_0 \dots \theta_k$ can be true, as is required for (ii). We leave the details of the induction and verifications asExercise 3. □

Theorem 6.9 (Soundness of SLDNF–refutation)**:** *Let* P *be a* PROLOG *program.*

(i) *If the* SLDNF*–tree via* R *from* P *beginning with a general goal* $G = \{L_1,\dots,L_m\}$ *is finitely failed, then Comp*(P) $\models L_1 \vee \dots \vee L_m$.

(ii) *If there is a success path, i.e., an* SLDNF*–refutation of* G *from* P, *on the tree with answer substitution* θ, *then Comp*(P) $\models (\neg L_1 \wedge \dots \wedge \neg L_m)\theta$.

Proof:

(i) We proceed by induction on the depth of the finitely failed SLDNF–tree starting with G. We begin with the case that the tree is finitely failed at its first level.

If $R(G)$ is a negative literal $L = \neg r(s_1,\dots,s_n)$, then there is no clause $C \in P$ with whose head we can unify $r(s_1,\dots,s_n)$. If there is no clause in P with r in its head then Comp(P) includes the axiom $\forall \vec{X} \neg r(\vec{X})$ and so Comp(P) $\models \neg r(s_1,\dots,s_n)$. Otherwise, with the notation as in the definition of Comp(P), we see that Comp(P) $\models r(s_1,\dots,s_n) \leftrightarrow \vee\{\exists Y_1,\dots,Y_{p_i}(s_1 = t_{i,1} \wedge \dots \wedge s_n = t_{i,n} \wedge q_{i,1} \wedge \dots \wedge q_{i,n_i}) | i \leq k\}$. As $r(s_1,\dots,s_n)$ does not unify with any $r(t_{i,1},\dots,t_{i,n})$, by assumption, the falsity of each of the disjuncts follows from the equality axioms by Lemma 6.8(ii). Thus Comp(P) $\models \neg r(s_1,\dots,s_n)$ and so Comp(P) $\models G$ as required.

If $R(G)$ is a positive literal L, it must be ground (or the proof would flounder rather than fail) and the SLD–tree starting with $\neg L$ must have a path ending in $\square$. Thus, by the soundness of SLD–refutations (Theorem 1.9), $P \models L$ as required. (Note that as L is ground, the answer substitution given by the SLD–refutation is irrelevant.)

Now consider the inductive step. Suppose that G has a finitely failed SLDNF–tree of depth $k+1$. If $R(G)$ is a positive ground literal L, then the SLD–tree starting with $\neg L$ is finitely failed and G_1 is $G - \{L\}$. It has a finitely failed SLDNF–tree of depth k and so by induction, $\text{Comp}(P) \models G_1$. As G contains G_1, $\text{Comp}(P) \models G$ as well.

Finally, suppose $R(G)$ is a negative literal $L = \neg r(s_1, \ldots, s_n)$. (Again we adopt the notation of the definition of Comp(P).) Each immediate successor H_i of G on level 1 of the given failed SLDNF–tree is the result of applying the appropriate mgu θ_i to G with L replaced by $\neg q_{i,1}, \ldots, \neg q_{i,n_i}$ (for $i \leq k$). Each has a failed SLDNF–tree of depth $\leq k$ and so, by induction, $\text{Comp}(P) \models H_i$ for each $i \leq k$. It thus suffices to prove that $\text{Comp}(P) \models \wedge H_i \rightarrow \forall \vec{X} G$. To see this, it suffices in turn to prove that

$$\text{Comp}(P) \models \wedge\{(\neg q_{i,1} \vee \ldots \vee \neg q_{i,n_i})\theta_i | i \leq k\} \rightarrow \neg r(s_1, \ldots, s_n).$$

Now by the definition of Comp(P), $\neg r(s_1, \ldots, s_n)$ can fail to hold only if $\exists Y_1, \ldots, Y_{p_i}(s_1 = t_{i,1} \wedge \ldots \wedge s_n = t_{i,n} \wedge q_{i,1} \wedge \ldots \wedge q_{i,n_i})$ for some $i \leq k$. By Lemma 6.8, this can happen only if there is a $\vec{Y}$ which unifies s_j and $t_{i,j}$ for each $j \leq n$ as well as witness $q_{i,1} \wedge \ldots \wedge q_{i,n_i}$. As θ_i is the mgu for this unification, the assumption that $(\neg q_{i,1} \vee \ldots \vee \neg q_{i,n_i})\theta_i$ implies that there are no such $\vec{Y}$ as required. $\square$

(ii) We proceed by induction on the length of the SLDNF–refutation. Suppose first that the refutation has length 1 and so G contains only one literal L. If L is positive, it is ground and there is a finitely failed SLD–tree starting with $\neg L$. By (i), $\text{Comp}(P) \models \neg L$ as required. If L is negative, say $\neg r(s_1, \ldots, s_n)$, then there is a clause of the form $r(t_1, \ldots, t_n)$ in P that can be unified with L by some θ. Thus $\neg L\theta$ is a consequence of P and hence of Comp(P) as required.

Next, consider an SLDNF–refutation of G of length $k+1$. If $R(G)$ is a positive literal L_i, then L_i is ground, θ_0 is the identity and $G_1 = \{L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m\}$ has an SLDNF–refutation of length k with mgu's $\theta_1 \ldots \theta_k$. As in the base case, $\text{Comp}(P) \models \neg L_i$; by the induction hypothesis $\text{Comp}(P) \models (\neg L_1 \wedge \ldots \wedge \neg L_{i-1} \wedge \neg L_{i+1} \wedge \ldots \wedge \neg L_m)\theta_1 \ldots \theta_k$. Thus, $\text{Comp}(P) \models (\neg L_1 \wedge \ldots \wedge \neg L_m)\theta_0\theta_1 \ldots \theta_k$ as required.

Finally, suppose $R(G)$ is a negative literal $L_i = \neg r(s_1, \ldots, s_n)$, and $G_1 = \{L_1, \ldots, L_{i-1}, \neg q_{j,1}, \ldots, \neg q_{j,n_j}, L_{i+1}, \ldots, L_m\}\theta_0$. By induction, $\text{Comp}(P) \models \{\neg L_1 \wedge \ldots \wedge \neg L_{i-1} \wedge q_{j,1} \wedge \ldots \wedge q_{j,n_j} \wedge \neg L_{i+1} \wedge \ldots \wedge \neg L_m\}\theta_0 \ldots \theta_{k+1}$. Now by the definition of Comp(P), the fact that θ_0 unifies $r(s_1, \ldots, s_n)$ and $r(t_{j,1}, \ldots, t_{j,n})$ and Lemma 6.8(i), $\text{Comp}(P) \models (q_{j,1} \wedge \ldots \wedge q_{j,n_j})\theta_0 \rightarrow r(s_1, \ldots, s_n)\theta_0$. Thus we see that $\text{Comp}(P) \models \neg L_i\theta_0 \ldots \theta_k$ as required to complete the induction step. $\square$

Theorem 6.10 (Completeness of SLDNF–refutation): *If P is a* PROLOG *program, $G = \{\neg A_1, \ldots, \neg A_k\}$ an ordinary goal clause, R a fair generalized selection rule and $\text{Comp}(P) \models \neg A_1 \vee \ldots \vee \neg A_k$ then there is a finitely failed* SLD*–tree from P via R beginning with G.*

Proof: The idea of the proof is much like that for the proof of completeness of the tableau method (II.7) as modified to handle equality in §5. If the SLD–tree beginning with G is not finitely failed, then it has a (possibly infinite) path Q. We use the terms and formulas appearing on Q to define a model $\mathcal{M}$ of Comp(P) such that $\mathcal{M} \models \exists \vec{X}(A_1 \wedge \ldots \wedge A_k)$. ($\vec{X}$ lists the free variables of G.) The construction of $\mathcal{M}$ then establishes the contrapositive of the desired theorem.

Let $G_0, G_1, \ldots$ and $\theta_0, \theta_1, \ldots$ be the goals and the mgu's appearing in the SLD–proof associated with Q. As in §5, the elements of M are equivalence classes of terms of the language and the function symbols operate on them in the obvious way defined there. The crucial new idea here is to use the sequence of mgu's to define the equivalence relation on the terms. We say that two terms s and t are equivalent, $s \equiv t$, if they are eventually identified by the sequence of mgu's, i.e., there is an m such that $s\theta_0 \ldots \theta_m = t\theta_0 \ldots \theta_m$. It is easy to see that this relation is, in fact, an equivalence relation (Exercise 4). We denote the equivalence class of a term t by $[t]$ and let the universe M of our intended model be the set of these equivalence classes. It is now easy to see that the equality axioms (1)–(6) of Comp(P) are satisfied in M when $=$ is interpreted as true equality on the equivalence classes (Exercises 5).

As the first step in defining the atomic formulas true in $\mathcal{M}$, we declare $r([t_1], \ldots, [t_n])$ true if there are $s_i \in [t_i]$ such that $\neg r(s_1, \ldots, s_n)$ appears as a literal in one of the goal clauses G_m. Note that this immediately makes $\mathcal{M} \models \exists \vec{X}(A_1 \wedge \ldots \wedge A_k)$ (as the (classes of the) terms in G provide the witnesses).

Our next, and most critical, claim is that the set S of atomic facts declared true so far satisfies the "only if" direction of the axioms for predicate letters in Comp(P). Suppose $\neg r(s_1, \ldots, s_n)$ first appears as a literal in the goal clause G_m. By the fairness of R, there is a $u > m$ at which $\neg r(s_1, \ldots, s_n)\theta_m \ldots \theta_u$ is selected. Note that $\neg r(s_1, \ldots, s_n)\theta_m \ldots \theta_u = \neg r(s_1, \ldots, s_n)\theta_0 \ldots \theta_u$ by the usual convention on the choice of variables and mgu's. At that step it is replaced by the literals $(\neg q_{i,1}, \ldots, \neg q_{i,n_i})\theta_{u+1}$ $(= (\neg q_{i,1}, \ldots, \neg q_{i,n_i})\theta_0 \ldots \theta_{u+1})$ from the body of the appropriate clause of P. As θ_{u+1} is an mgu for this resolution, each $q_{i,j}\theta_{u+1}$ is in S. So by Lemma 6.8(i) we have the desired witnesses for the disjunct q_i of the instance of the axiom of Comp(P) associated with $r([s_1], \ldots, [s_n])$.

We now have to extend S so as to make $\mathcal{M}$ a model of the "if" direction of the axioms, i.e. of P, without losing the "only if" direction. Let P' be the set of ground substitution instances of clauses of P by elements of M and let S' be the set of resolution consequences of $S \cup P'$. Let $\mathcal{M}$ be such that S' is the set of atomic facts true in M. We claim that $\mathcal{M}$ is

the desired model of Comp(P). As it is obviously a model of P, we only have to check that the "only if" direction of the axioms for each predicate r have been preserved as well. This claim is immediate by induction on the length of the resolution deduction putting any $r(t_1, \ldots, t_n)$ into S': It can be deduced only if some appropriate instances of the $q_{i,1}, \ldots, q_{i,n_i}$ in one of the clauses of P with r in its head have already been deduced. □

The definition of Comp(P) formalizes the intuition behind the CDB approach. Analogously, CWA(P) is the set of all sentences which should be associated with P according to the CWA (closed world assumption). The basic intuition of the CWA is that, for any positive ground literal L, if $P \nvdash L$ then we should infer $\neg L$. We can thus view it as adjoining to P the following clauses:

(0) $\{\neg L\}$ for each ground positive literal L such that $P \nvdash L$.

While the CWA shares with CDB the view that the domain of discourse is correctly captured by the ground terms, the CWA takes it even farther. In addition to the equality axioms (1)–(6) described above, it asserts that the universe consists precisely of the ground terms. This assertion cannot, however, be guaranteed by a formula of predicate logic (Exercise 6). If we consider only logical consequence ($\models$) rather than provability, we can express this requirement by an infinitary clause, DCA, the *domain closure axiom*:

(7) $x = t_1 \vee \ldots \vee x = t_n \vee \ldots$

where $\langle t_i \rangle$ a list of all the ground terms.

We now write CWA(P) to denote the extension of P by (0)-(7). Note that any model for CWA(P) is an Herbrand model for P. As the adjunction of (0) guarantees that the truth of every ground literal is determined by CWA(P), there can be at most one such model. Indeed for any PROLOG program P, CWA(P) is always satisfiable and its only model is the minimal Herbrand model for P (Exercise 7). As this model is also one of Comp(P) (Exercise 8), the soundness results (Theorem 6.9) proved for negation as failure and Comp(P) hold automatically for CWA(P) as well. There can, however, be no completeness theorem comparable to Theorem 6.10 for CWA(P). Indeed, no effective procedure (such as searching for an SLDNF–refutation) can list all the logical consequences of CWA(P) for every P (Exercise 8.6).

In addition to sometimes wanting to derive negative information, the PROLOG programmer might like to use such expressions in the program as well. This leads to the notion of general programs and general SLDNF–resolution.

Definition 6.11: A *general program clause* is one which contains at least one positive literal (but perhaps more). A *general program* is a set of general program clauses.

In any given general program clause $\{R, \bar{L}_1, \ldots, \bar{L}_n\}$ we single out one positive literal, R, as the head and consider all others as the body of the clause. We then write the clause in PROLOG notation (with $\neg$) as R :- $L_1, \ldots, L_n$. (Unfortunately, the interpretation and analysis will depend on which positive literal is chosen as the head.) In the same vein, we write general goal clauses in the form $\{\neg L_1, \ldots, \neg L_n\}$; however, as before the hope is to show that $P \models \exists X_1 \ldots X_m (L_1 \wedge \ldots \wedge L_n)\theta$ by deriving □ (via some form of resolution) from $P \cup \{G\}$ with mgu's $\theta_0 \ldots \theta_k = \theta$.

We can now extend the definition of SLDNF–refutations to general programs by introducing a recursion at the point at which we search for a finitely failed tree. We now look for a finitely failed SLDNF–tree. The extensions of a general program P to CWA(P) and Comp(P) are defined as before. Soundness results like those of Theorems 6.8, 6.9 can be proved in this general setting as well. The completeness result of Theorem 6.10 no longer holds. Indeed, the completeness theorem cannot be extended to general goal clauses even for all PROLOG programs (Exercise 9). Weaker forms that deal only with the cases in which every branch of the SLDNF–tree ends in success or failure do hold. Under these conditions, it is possible to show that the SLDNF–tree gives, in some sense, "all" the answers that are consequences of CWA(P) or Comp(P). We refer the reader to Shepherdson [1984, 5.4] for a treatment of CWA(P) and to Chapter 3 of Lloyd [1987, 5.4] for a thorough discussion of the CDB approach and Comp(P).

The crucial caveat in the setting of general programs P is that it may turn out that CWA(P) or Comp(P) or both are unsatisfiable even though P is satisfiable (Exercises 10–12). Conditions like those of recurrence and acceptability considered in §4 can, however, be used to guarantee the consistency of Comp(P) and to produce a completeness theorem for SLDNF–refutations with respect to the semantics given by Comp(P). (Again we refer the reader to Lloyd [1987, 5.4].)

Exercises

1. Show that no selection rule that always chooses the same literal from each goal clause can be fair.
   (Hint: Consider the program P with three clauses:

   (1) r :- p, q. (2) p :- p. (3) q :- q.)

2. Describe a fair generalized selection rule and prove that it is fair.
   (Hint: Always choose the first literal to appear in the proof so far that has not yet been chosen.)
3. Complete the proof of Lemma 6.8.
4. Verify that the relation $\equiv$ defined in the proof of Theorem 6.10 is an equivalence relation.

5. Verify that the equality axioms (1)–(6) are satisfied in the set M defined in the proof of Theorem 6.10 when "=" is interpreted as true equality of equivalence classes.
6. Prove that no set of sentences of predicate logic can imply axiom (7) of CWA. (Hint: Use the compactness theorem.)
7. Prove that the unique model for CWA(P) for a PROLOG program P is the minimal Herbrand model for P.
8. Prove that the minimal Herbrand model for a PROLOG program P is also a model of Comp(P).
9. Give a counterexample to the generalization of Theorem 6.10 to general goal clauses. (Hint: Write a short program and choose a general goal such that every attempted SLDNF–refutation flounders.)
10. Give an example of a general program P such that Comp(P) (and hence P) is satisfiable but CWA(P) is not.
11. Give an example of a general program P such that CWA(P) (and hence P) is satisfiable but Comp(P) is not.
12. Give an example of a satisfiable general program P such that neither Comp(P) nor CWA is satisfiable.

7. Negation and Nonmonotonic Logic

The general procedure of implementing negation as failure described in the last section is both useful and important. Nonetheless, it violates one of the most basic tenets of mathematical reasoning. In mathematical reasoning (and indeed in all the systems we consider elsewhere in this book) a conclusion drawn from a set of premises can be also be drawn from any larger set of premises. More information or axioms cannot invalidate deductions already made. This property of monotonicity of inferences is basic to standard mathematical reasoning, yet it is violated by many real life procedures as well as by the negation as failure rule.

In the absence of evidence to the contrary, we typically take consistency with the rest of our general belief system to provide grounds for a belief. The classic example concerns Tweety the bird. At some stage in the development of our knowledge we observe and learn about various birds. Based on this information we conclude that birds fly. One day we are told about Tweety the bird and naturally assume that he can fly. When we are later introduced to Tweety, we discover that he is a pet ostrich and can no more fly than we can. We reject our previous belief that all birds fly and revise our conclusions about Tweety. We now face the world with a new set of beliefs from which we continue to make deductions until new evidence once again proves our beliefs false. Such a process is typical of the growth of knowledge in almost all subjects except mathematics. Beliefs and conclusions are often based on a lack of evidence to the contrary.

A similar approach is embodied in the notion of negation as failure. If we have no evidence to the contrary (i.e., a deduction of L), we assume that L is false. This procedure clearly embodies a nonmonotonic system of reasoning. Minsky [1975, 5.5] was the first to propose such systems and beginning with McCarthy's study of circumscription [1980, 5.5] various researchers have proposed and studied a large number of nonmonotonic systems which have been suggested by various problems in computer science and AI. To list just a few: Hintikka's theory of multiple believers, Doyle's truth maintenance system, Reiter's default logic and Moore's autoepistemic logic as well as various versions of negation as failure in extensions of PROLOG by Apt, Clark and others.

We will now briefly present a new approach to an abstract view of nonmonotonic systems as given in Marek, Nerode and Remmel [1990, 5.5]. It seems to capture the common content of many of the systems mentioned. The literature has dealt primarily with the propositional case and we restrict ourselves to it as well. For negation in PROLOG, this means that we will always be looking at the set of ground instances of a given program in the appropriate Herbrand universe. After describing the general system we will connect it to one interesting way of picking out a distinguished Herbrand model that captures many aspects of negation in PROLOG (although it is not precisely the same as the negation as failure rules of §6): the stable model semantics of Gelfond and Lifschitz [1988, 5.4].

We present the idea of nonmonotonic systems in the form of rules of inference like resolution or the one given for classical monotonic logic in I.7. In such a setting, a rule of inference is specified by giving a list of hypotheses and a conclusion which may be drawn from them. The standard rule of modus ponens (I.7.2) concludes β from the hypotheses α and $\alpha \to \beta$. An appropriate style for describing this rule is to write the hypotheses in a list above the line and the conclusion below:

$$\frac{\alpha, \alpha \to \beta}{\beta}.$$

In this notation, the axioms are simply rules without hypotheses such as in I.7.1(i):

$$\frac{}{(\alpha \to (\beta \to \alpha))}$$

The crucial extension of such a system to nonmonotonic logic is to add restraints to the deduction. In addition to knowing each proposition in the set of hypotheses, it may be necessary to not know (believe, have a proof of, have already established, etc.) each of some other collection of propositions in order to draw the conclusion permitted by a given rule. The notation for this situation is to list the usual kind of premises first and then, separated by a colon, follow them with the list of restraints. The restraints are the

propositions which the rule requires us not to know (believe, etc.). Thus we read the rule

$$\frac{\alpha_1, \ldots, \alpha_n : \beta_1, \ldots, \beta_m}{\gamma}$$

as saying that if $\alpha_1, \ldots, \alpha_n$ are known (proven, established) and $\beta_1, \ldots, \beta_m$ are not, then we may conclude that we know (can prove or establish) γ.

Definition 7.1 (*Nonmonotonic formal systems*): Let U be a set (of propositional letters).

(i) A *nonmonotonic rule of inference* is a triple $\langle P, G, \varphi \rangle$ where $P = \{\alpha_1, \ldots, \alpha_n\}$ and $G = \{\beta_1, \ldots, \beta_m\}$ are finite lists of elements of U and $\varphi \in U$. Each such rule is written in the form

$$r = \frac{\alpha_1, \ldots, \alpha_n : \beta_1, \ldots, \beta_m}{\varphi}.$$

We call $\alpha_1, \ldots, \alpha_n$ the *premises* of the rule r and $\beta_1, \ldots, \beta_m$ its *restraints*. Note that either P or G or both may be empty.

(ii) If $P = G = \emptyset$, then the rule r is called an *axiom*.

(iii) A *nonmonotonic formal system* is a pair $\langle U, N \rangle$ where U is a nonempty set (of propositional letters) and N is a set of nonmonotonic rules.

(iv) A subset S of U is *deductively closed* in the system $\langle U, N \rangle$ if, for each rule r of N such that all the premises $\alpha_1, \ldots, \alpha_n$ of r are in S and none of its restraints $\beta_1, \ldots, \beta_m$ are in S, the conclusion φ of r is in S.

The essence of the nonmonotonicity of a formal system is that the deductively closed sets are not in general closed under arbitrary intersections. Thus there is, in general, no deductive closure of a set I of propositional letters; i.e., no least set $S \supseteq I$ which is deductively closed. The intersection of a decreasing sequence of deductively closed sets is, however, deductively closed (Exercise 1) and so there is always (at least one) minimal deductively closed subset of U (Exercise 2).

The deductively closed sets containing I can be viewed as the rational points of view possible in a given system when one assumes all the elements of I to be true. Each one expresses a set of beliefs which is closed under all the rules. There may, however, be many such points of view which are mutually contradictory. The intersection of all deductively closed sets containing I represents the information common to all such rational points of view. It is often called the set of *secured consequences* of I or the *skeptical reasoning* associated with the system and I.

Example 7.2: Let $U = \{\alpha, \beta, \gamma\}$ and let

$$r_1 = \frac{}{\alpha} \qquad r_3 = \frac{\alpha : \beta}{\gamma}$$

$$r_2 = \frac{\alpha : \beta}{\beta} \qquad r_4 = \frac{\alpha : \gamma}{\beta}$$

(i) Let $N_1 = \{r_1, r_2\}$. There is only one minimal deductively closed set for $\langle U, N_1 \rangle : S = \{\alpha, \beta\}$. S is then the set of secured consequences of $\langle U, N_1 \rangle$.

(ii) Let $N_2 = \{r_1, r_3, r_4\}$. There are two minimal deductively closed set for $\langle U, N_2 \rangle : S_1 = \{\alpha, \beta\}$ and $S_2 = \{\alpha, \gamma\}$. $S = \{\alpha\}$ is then the set of secured consequences of $\langle U, N_2 \rangle$. In this case the set of secured consequences is not deductively closed.

The analog in nonmonotonic logic of a classical deduction from premises I involves a parameter S for the set of propositions we are assuming we do not know. We use this notion of deduction to characterize the *extensions* of a nonmonotonic system which are analogous to the set of consequences of a monotonic system.

Definition 7.3: Let $\langle U, N \rangle$ be a nonmonotonic formal system and let S, $I \subseteq U$. An *S-deduction of φ from I in $\langle U, N \rangle$* is a finite sequence $\varphi_1, \ldots, \varphi_k$ such that $\varphi = \varphi_k$ and, for all $i \leq k$, φ_i is either in I, an axiom of $\langle U, N \rangle$ or the conclusion of a rule $r \in N$ all of whose premises are included among $\varphi_1, \ldots, \varphi_{i-1}$ and all of whose restraints are contained in $U - S$. In this situation φ is called an *S-consequence of I* and we denote by $C_S(I)$ the set of all S-consequences of I.

Note that the role of S in the above definitions is to prevent applications of rules with any restraint in S; it does not contribute any members of U directly to $C_S(I)$. Indeed, $C_S(I)$ may not contain S and may not be deductively closed.

Example 7.4: With the notation as in Example 7.2, define a system $\langle U, N \rangle$ by setting $N = \{r_1, r_3\}$. If $S = \{\beta\}$, then $C_S(\emptyset) = \{\alpha\}$ is not deductively closed as it does not contain γ in violation of rule r_3.

Proposition 7.5: *If $S \subseteq C_S(I)$, then $C_S(I)$ is deductively closed.*

Proof: Suppose all the premises of a rule r with conclusion φ are in $C_S(I)$ and all of r's restraints are outside it. By the definition of $C_S(I)$, we can produce an S-deduction containing all the premises of r. All of the restraints in r are outside S by hypothesis. We can thus extend the S-deduction to one of φ by applying r to get $\varphi \in C_S(I)$ as desired. □

Definition 7.6: $S \subseteq U$ is an *extension of I* if $C_S(I) = S$. S is an *extension* if it is an extension of the empty set $\emptyset$.

The extensions S of I are the analogs for nonmonotonic systems of the logical consequences of I. Every member of an extension is deducible from I and all the S–consequences of I are in fact in S. We give some basic properties of extensions in Exercises 3–5.

It turns out that extensions capture many procedures in mathematics and computer science. We give some mathematical examples in Exercises 8–9. Now we return to PROLOG programs with negation and their connection to extensions through the notion of stable models.

From our current point of view, it is natural to try to consider the negation as failure as a nonmonotonic system. The underlying idea of negation as failure as presented in the last section is that we may assert $\neg p$ when we do not know (cannot deduce) p. This suggests a natural translation of a general PROLOG program into a nonmonotonic formal system.

Recall from Definition 6.11 that a general program clause has the form $p :\!-\ q_1, \ldots, q_n, \neg s_1, \ldots, \neg s_m$ where p, q_i and s_j are atoms.

Remember also that we are in the propositional case. Thus, if a program of interest has variables, we consider instead all ground instances of the program clauses in the Herbrand universe. We can now easily translate a general program P containing only ground atoms into a nonmonotonic formal system in a natural way. We consider each ground atom as a propositional letter. These atoms constitute our universe U. Each program clause C of P of the form $p :\!-\ q_1, \ldots, q_n, \neg s_1, \ldots, \neg s_m$ is translated into a rule $tr(C)$:

$$\frac{q_1, \ldots, q_n : s_1, \ldots, s_m}{p}$$

The nonmonotonic system is then specified by letting its set of rules N be the collection $\{tr(C) : C \in P\}$ of translations of clauses of P.

Definition 7.7: Let P be a general program with only ground clauses. $tr(P)$, the *translation of* P, is the nonmonotonic system $\langle U, N \rangle$ where U is the set of atoms appearing in P and $N = \{tr(C) : C \in P\}$ is the set of translations of clauses of P.

It turns out that it is precisely the extensions of $tr(P)$ which are the stable models of P introduced by Gelfond and Lifschitz [1988, 5.4] to capture a strong notion of semantics for general programs with negation as failure.

Definition 7.8: If U is the set of atoms appearing in a ground general program P and $M \subseteq U$, then P_M is the program obtained from P by deleting each clause which has in its body a negative literal $\neg s$ with $s \in M$ and also deleting all negative literals in the bodies of the remaining clauses. As P_M is clearly a PROLOG program (it has no negative literals), it has a unique minimal Herbrand model by Exercise II.10.3. A *stable model* of P is an $M \subseteq U$ such that M is the unique minimal Hebrand model of P_M.

This terminology is justified by the following theorem which shows that stable models of P are in fact models of P.

Theorem 7.9: *Every stable model of P is a minimal model of P.*

Proof: Suppose M is a stable model of P. Consider a clause C of P of the form $p :\!-\ q_1, \ldots, q_n, \neg s_1, \ldots, \neg s_m$. If some $s_j \in M$, then M trivially satisfies C. If none of the s_j are in M, then $p :\!-\ q_1, \ldots, q_n$ is in P_M. As M is a model of P_M, $p \in M$ if $q_1, \ldots, q_n \in M$. Thus M satisfies C in this case as well and so M is a model of P.

To see that M is a minimal model of P, consider any $M' \subseteq M$ which is also a model of P. We need to show that $M = M'$. By the definition of a stable model, it suffices to show that M' is a model of P_M. Now any clause C' of P_M comes from some C in P as above with $s_j \notin M$ for $1 \leq j \leq m$. It is then of the form $p :\!-\ q_1, \ldots, q_n$. Suppose then that $q_1, \ldots, q_n \in M'$. We need to show that $p \in M'$. As $M' \subseteq M$, $s_j \notin M'$ for every $1 \leq j \leq m$. Thus, as M' is a model of $C \in P$, $p \in M'$ as required. □

Example 7.10: Let Q be the following general program of Gelfond and Lifschitz [1988, 5.4]:

$$p(1,2).$$
$$q(x) :\!-\ p(x,y), \neg q(y).$$

Q has two minimal Herbrand models: $M_1 = \{p(1,2), q(1)\}$ and $M_2 = \{p(1,2), q(2)\}$ (Exercise 6). The usual negation as failure rule applied to this program answers "no" to the question "?- $q(2)$." but "yes" to the question "?- $q(1)$." Thus we should prefer the first model over the second.

Now consider the possible subsets of the Herbrand universe as candidates for stable models of the ground instances of this program. First, the program itself is transformed into the following ground version P:

$$p(1,2).$$
$$q(1) :\!-\ p(1,1), \neg q(1).$$
$$q(1) :\!-\ p(1,2), \neg q(2).$$
$$q(2) :\!-\ p(2,1), \neg q(1).$$
$$q(2) :\!-\ p(2,2), \neg q(2).$$

We now consider the subset $M = \{q(1)\}$ of the Herbrand universe. P_M is then

$$p(1,2).$$
$$q(1) :\!-\ p(1,2).$$
$$q(2) :\!-\ p(2,2).$$

The minimal Herbrand model of P_M is $\{p(1,2), q(1)\} \neq M$. Thus, M is not a stable model of P. Indeed, any set M not containing $p(1,2)$ is not a model of P and so by Theorem 7.9 not a stable model of P.

Next, we consider the two minimal Herbrand models M_1 and M_2 of the original program Q. We claim that M_1 is stable but not M_2. First, P_{M_1} is

$$p(1,2).$$
$$q(1) :- p(1,2).$$
$$q(2) :- p(2,2).$$

The minimal model of this program is clearly M_1 which is therefore stable. On the other hand, P_{M_2} is

$$p(1,2).$$
$$q(1) :- p(1,1).$$
$$q(2) :- p(2,1).$$

Its minimal model is $\{p(1,2)\} \neq M_2$. Thus M_2 is not stable.

In fact, M_1 is the only stable model of P (Exercise 7) and so the stable model is the "right" one from the viewpoint of negation as failure.

A direct and precise connection of stable models with nonmonotonic formal systems is provided by the Theorem 7.12. We continue with the notation introduced above and begin with a lemma.

Lemma 7.11: *If M' is a model of P_M, then $M' \supseteq C_M(\emptyset)$.*

Proof: Suppose that M' is a model of P_M. We prove by induction on the length of M–deductions that every member of $C_M(\emptyset)$ is contained in M'. Consider any M–deduction $\varphi_1, \ldots, \varphi_k, p$ (from $\emptyset$) and suppose the rule applied at the last step of this deduction to conclude p is $tr(C)$ for some clause C in P. By induction, we may assume that $\varphi_i \in M'$ for every $1 \leq i \leq k$ and so every premise q_i of $tr(C)$ is in M'. As this is an M–deduction, no restraint s_j of $tr(C)$ is in M. By definition then, $p :- q_1, \ldots . q_n$ is one of the clauses of P_M. As M' is a model of P_M, $p \in M'$ as required. □

Theorem 7.12: *A subset M of U is a stable model of P if and only if it is an extension of $tr(P)$.*

Proof: Suppose that M is an extension of $\langle U, tr(P) \rangle$. First, we claim that M is a model of P_M. Consider any clause $p :- q_1, \ldots, q_n$ in P_M such that $q_1, \ldots, q_n \in M$. By the definition of P_M, there is a clause $C = p :- q_1, \ldots, q_n, \neg s_1, \ldots, s_m$ in P with no s_j in M. Thus there is a rule $tr(C)$ in $tr(P)$ with all its premises in M and none of its restraints in M. As extensions are deductively closed by Proposition 7.5, $q \in M$ as required. Next, we must prove that no M' strictly contained in M is a model of P_M. As $M = C_M(\emptyset)$, this is immediate from Lemma 7.11.

For the converse, suppose that M is a minimal Herbrand model of P_M. We first note that, by Lemma 7.11, $M \supseteq C_M(\emptyset)$. By the minimality assumption on M, it suffices to prove that $C_M(\emptyset)$ is a model of P_M to conclude that $M = C_M(\emptyset)$ as required. Consider, therefore, any clause $p :- q_1, \ldots, q_n$ in P_M with all the q_i in $C_M(\emptyset)$. There is then an M–deduction $\varphi_1, \ldots, \varphi_k$ containing all of the q_i. By definition of P_M, there is a clause $C = p :- q_1, \ldots, q_n, \neg s_1, \ldots, \neg s_m$ in P with none of the s_j in M and so a rule $tr(C)$ in $tr(P)$ with all its premises in $C_M(\emptyset)$. We may thus form an M–deduction with p as the consequence. So $p \in C_M(\emptyset)$ as required. □

Gelfond and Lifschitz show that certain classes of programs with properties like those considered in §4 have unique stable models and propose the term stable model semantics for such programs. The special case of a unique stable model is certainly of particular interest. From the viewpoint of nonmonotonic logic, however, all the extensions of $tr(P)$ are equally good candidates for models of the system.

Exercises

1. Let $S_1 \supseteq S_2 \supseteq \ldots$. be a nested sequence of deductively closed sets for a nonmonotonic system $\langle U, N \rangle$. Prove that $\cap S_i$ is deductively closed.
2. Zorn's lemma (an equivalent of the axiom of choice) states that any nonempty family of sets closed under the intersection of downwardly nested sequences has a minimal element. Use it and Exercise 1 to prove that every nonmonotonic formal system has a minimal deductively closed subset.
3. Prove that the operation $C_S(I)$ is monotonic in I and antimonotonic in S, that is if $I \subseteq J$, then $C_S(I) \subseteq C_S(J)$ and if $S \subseteq T$, then $C_S(I) \supseteq C_T(I)$.
4. Prove that, if S is an extension of I, then S is a minimal deductively closed superset of I and for every J such that $I \subseteq J \subseteq S$ we have $C_S(J) = S$.
5. If S and T are extensions of I and $S \subseteq T$, then $S = T$.
6. Prove that the minimal Herbrand models of programs P and Q of Example 7.10 are the sets M_1 and M_2 given there.
7. Prove that M_1 is the only stable model of P in Example 7.10.

   (Hint: To begin the analysis note that any candidate must contain $p(1,2)$ to be a model of P but will not contain any other instance of P by minimality considerations.)

Refer to Exercises I.6.7–8 for the basic terminology about graphs and partial orderings used below.

8. Let n be a natural number and G be a locally finite graph, i.e., a graph in which, for each node x there are at most finitely many nodes y such that $\{x, y\}$ is an edge of G. We define a nonmonotonic formal system $\langle U(G), N(G)\rangle$ by first setting $U(G) = \{Cxi \mid x \text{ is a node of } G \text{ and } i \leq n\}$. We then put into $N(G)$, for each node x of G and $j \leq n$, the rule

$$\frac{: Cx1, \ldots, Cx(j-1), Cx(j+1), \ldots, Cxn}{Cxj}.$$

Finally, we put into $N(G)$, for each pair x, y of distinct nodes of G, each $i \leq n$ and each $\varphi \in U9G)$, the rule

$$\frac{Cxi, Cyi}{\varphi}$$

Prove that $S \subseteq U(G)$ is an extension for $\langle U(G), N(G)\rangle$ if and only if coloring each node x of G with color i iff $Cxi \in S$ produces an n–coloring of G.

9. Let P be a partial ordering of width n. We define a nonmonotonic system $\langle U(P), N(P)\rangle$ by first setting $U(P) = \{Cxi | x \in P \text{ and } i \leq n\}$. For each $x \in P$ we put into $N(P)$ the rule

$$\frac{: Cx1, \ldots, Cx(j-1), Cx(j+1), \ldots, Cxn}{Cxj}.$$

Finally, for each x and y in P which are incomparable in the partial ordering, we put into $N(P)$ the rule

$$\frac{Cxi, Cyi}{\varphi}.$$

Prove that $S \subseteq U$ is an extension of $\langle U(P), N(P)\rangle$ if and only if $\{C_1, \ldots, C_n\}$ is a set of disjoint chains covering P where $C_i = \{x | C_{x_i} \in S\}$.

8. Computability and Undecidability

One of the major tasks of the logicians of the 30's and 40's was the formalization of the basic intuitive notion of an algorithm or an effective procedure. (For convenience we consider procedures on the natural numbers.) Many seemingly different definitions were proposed by a number of researchers including Church, Gödel, Herbrand, Kleene, Markov and Post. They suggested schemes involving recursion, equational deduction systems, idealized models of computing machines and others. Perhaps the philosophically most convincing proposal was that of Turing. He gave what is undoubtedly now the best known definition in terms of a simple machine model of computation: the Turing machine.

Every function calculable by any of these models was clearly effective. As investigations progressed, it became evident that any function that was intuitively computable could be calculated in any of the systems. Indeed, over a number of years all these proposals were proven equivalent, that is, the class of functions computable in any one model is the same as that computable in any other. These functions are now called the *recursive functions*. Early results along these lines led Church to formulate what is now known as *Church's thesis*: the effectively calculable functions are precisely the recursive ones. The weight of the evidence has by now produced an almost universal acceptance of this thesis. Thus, to prove that any computation scheme is universal in the sense that it computes every effective function, it suffices to prove that it computes every function computable by any of the schemes known to define the class of recursive functions.

It is not difficult to model almost any of the standard definitions by deduction in predicate logic: for each recursive function f, we can write down axioms in a language for arithmetic which includes a term $\bar{n}$ for each natural number n and a two–place predicate symbol p_f such that $f(n) = m$ iff $p_f(\bar{n}, \bar{m})$ is a logical consequence of the axioms. (We restrict our attention to unary functions simply to avoid strings of variables. Everything we do will work just as well for functions of any specified arity m by simply replacing the single variable x_1 by a sequence of variables $x_1, x_2, \ldots, x_m$.) For the most part, these representations can be naturally expressed in the form of PROLOG programs. (See Exercises 1–2 for an example.) Thus, any sound and complete implementation of PROLOG (e.g. with breadth–first searching) will correctly compute all recursive functions. By choosing the right model of computation (Shepherdson's register machines as described in Definition 8.1) and exercising some cleverness in the translation into PROLOG (Definition 8.4), we will prove that the standard implementation via the leftmost literal selection rule and depth-first searching of the SLD–tree also suffices to compute all recursive functions. (In fact, the "right" programs will run correctly with essentially any selection rule and search procedure.) Thus PROLOG is a universal computing machine model (Corollaries 8.6–8.7).

Once one has an agreed upon the mathematical definition of an algorithm or the class of effectively computable functions, one can hope to prove that various procedures (or decisions) cannot be carried out (made) by any algorithm or that particular functions cannot be computed effectively. (These notions really coincide. Decision procedures such as deciding if a number n is in some given set A, or if some polynomial has an integral root or the like correspond to calculating the characteristic function C_A of A ($C_A(n) = 1$ if $n \in A$ and $C_A(n) = 0$ if $n \notin A$) or of the set of tuples of numbers which correspond to the coefficients of polynomials with integral roots.) Indeed, beginning with the first definitions of the recursive functions and continuing to the present, many classical problems asking for algorithms have been solved negatively by showing that there is no recursive function which computes the desired result. One of the earliest and

best known of these results is Turing's proof of the undecidability of the halting problem: There is no algorithm (recursive function) for deciding if a given computer program halts on a given input. Thus, once we have provided a translation of a standard model of computation into PROLOG, we will have proven the undecidability of the termination problem for PROLOG programs with the standard implementation. As the arguments apply to semantically complete implementations as well, we will also have proven Church's celebrated result on the undecidability of validity for predicate logic (Theorem 8.10).

We begin our proof of the universality of PROLOG for computation by presenting Shepherdson's register machine model for computability. It is "mechanically" simpler than Turing's model and considerably easier to implement in PROLOG. A *register machine* consists of some number of storage locations called registers. Each register contains a natural number. There are only two types of operations that these machines can perform in implementing a program. First, they can increase the content of any register by one and then proceed to the next instruction. Second, they can check if any given register contains the number 0 or not. If so they go on to the next instruction. If not they decrease the given register by one and can be told to proceed to any instruction in the program. Formally, we define register machine programs and their execution as follows:

Definition 8.1: A *register machine program* I is a finite sequence $I_1, \ldots, I_t, I_{t+1}$ of instructions operating on a sequence of numbers $x_1, \ldots, x_r$, where each instruction I_m, for $m \leq t$, is of one of the following two forms:

(i) $x_k := x_k + 1$ (replace x_k by $x_k + 1$)

(ii) If $x_k \neq 0$, then $x_k := x_k - 1$ and go to j. (If $x_k \neq 0$, replace it by $x_k - 1$ and proceed to instruction I_j.)

It is assumed that after executing some instruction I_m, the execution procedes to I_{m+1}, the next instruction on the list, unless I_m directs otherwise. The execution of such a program proceeds in the obvious way on any input of values for $x_1, \ldots, x_r$ (the initial content of the registers) to change the values of the x_k and progress through the list of instructions.

The final instruction, I_{t+1}, is always a halt instruction. Thus, if I_{t+1} is ever reached, the execution terminates with the current values of the x_k. In general, we denote the assertion that an execution of the program I is at instruction I_m with values $n_1, \ldots, n_r$ of the variables by $I_m(n_1, \ldots, n_r)$.

Definition 8.2: A register machine program I *computes a function* $f : \mathbb{N} \to \mathbb{N}$, if, when started at instruction I_1 with $x_1 = n$ and $x_k = 0$ for all $k > 1$, its execution eventually terminates (at instruction I_{t+1}) with $x_1 = n$ and $x_2 = f(n)$, i.e., we eventually have $I_{t+1}(n, f(n), n_3, \ldots, n_r)$ for some numbers $n_3, \ldots, n_r$. If f is partial function from $\mathbb{N}$ into $\mathbb{N}$, i.e., it is not defined at every $n \in \mathbb{N}$, we also require that the execution of the program beginning at $I_1(n, 0, \ldots, 0)$ terminates if and only if n is in the domain of f.

A (partial) function from $\mathbb{N}$ to $\mathbb{N}$ is (*partial*) *recursive* if it is computed by some register machine program.

The reader familiar with Turing machines can find proofs that the partial functions computable by a register machine program are exactly those computed by a Turing machine program or any of the other more common models in the original papers of Shepherdson and Sturgis [1963, 3.6] and Minsky [1961, 3.6] or in many basic texts on computability such as Cutland [1980, 3.6] or Tourlakis [1984, 3.6]. For our purposes, we simply take this model as the defining one for the class of partial recursive functions.

For the sake of definiteness, we give a specific definition of what it means for a PROLOG program with a particular implementation to compute a partial function f. We assume that a minimum amount of arithmetic is represented in our language. In fact, all we need is a constant for the number zero, say 0, and a unary function s representing the successor function on $\mathbb{N}$. In this setting (which we considered in Exercises 2.7–8), $s(x)$ represents $x + 1$ and the term $s^n(0)$ represents the number n.

Definition 8.3: A PROLOG program P with a (two-place) predicate p *computes the partial function* f (under some specific implementation) if, for any natural number a, asking the question "?- $p(s^a(0), Z)$" produces a nonterminating computation if $f(a)$ is undefined and a terminating one with the answer substitution $Z = s^b(0)$ (and no other answers) if $f(a) = b$. If no implementation is mentioned, we assume that any sound and complete one is intended (they are, of course, all equivalent). By the standard implementation, we mean the sound but incomplete one using the leftmost selection rule and depth-first search.

Perhaps the most natural way to represent a register machine in the language of predicate, or even Horn, logic is to introduce a predicate letter p_m of r variables for each instruction I_m. The intended interpretation of $p_m(n_1, \ldots, n_r)$ is that the machine is at instruction m with values $n_1, \ldots, n_r$ for the variables. We can now easily express the step by step execution of a given program by implications corresponding to its instructions.

A first attempt at such a translation might well proceed as follows:

For each instruction I_m, $1 \leq m \leq t$, include an axiom of the appropriate form:

(i) $p_m(x_1, \ldots, x_r) \to p_{m+1}(x_1, \ldots, x_{k-1}, s(x_k), x_{k+1}, \ldots, x_r)$.

(ii) $p_m(x_1, \ldots, x_{k-1}, 0, x_{k+1}, \ldots, x_r) \to p_{m+1}(x_1, \ldots, x_{k-1}, 0, x_{k+1}, \ldots, x_r)$

$\wedge\, p_m(x_1, \ldots, x_{k-1}, s(y), x_{k+1}, \ldots x_r) \to p_j(x_1, \ldots, x_{k-1}, y, x_{k+1}, \ldots, x_r)$.

(Note that being a successor is equivalent to being nonzero.)

Let $Q(I)$ be the finite set of axioms corresponding under this translation to register program I. It is not hard to prove that, if program I computes the partial function f, then, for any n, m and $r-2$ numbers $a, b, c \ldots$, $p_1(s^n(0), 0 \ldots 0) \rightarrow p_{t+1}(s^n(0), s^m(0), a, b, c \ldots)$ is a logical consequence of $Q(I)$ if and only if $f(n)$ is defined and equal to m. Such a translation then suffices to give the computability of all recursive functions by, and so the undecidability of, any sound and complete version of Horn clause deduction or PROLOG. (See Exercise 3.) Unfortunately, the standard implementation will not always find the terminating computation (Exercise 4).

Shepherdson's strategy (as presented in Bezem [1989, 5.4]) for converting a register machine program into a PROLOG program that will correctly compute the same function (even, as we shall see, with the standard implementation) involves two ideas. The first is to reverse the direction of the implications, thus translating the calculation into a verification. The second is to move the variable in the goal clause out of the way so as to eliminate the possibility of unnecessary and unwanted attempts at unifying it.

Definition 8.4: The PROLOG program $P(I)$ associated with a register machine program I contains the following clauses:

A clause transferring the variable in the goal clause out of the way and passing control to the first instruction:

$$p(X_1, Z) :- p_1(X_1, 0, \ldots, 0, Z).$$

(The string of zeros has length $r-1$.)

For each instruction I_m of type (i), the single clause:

$$p_m(X_1, \ldots, X_r, Z) :- p_{m+1}(X_1, \ldots, X_{k-1}, s(X_k), X_{k+1}, \ldots, X_r, Z).$$

For each instruction of type (ii), the two clauses:

$$p_m(X_1, \ldots, X_{k-1}, 0, X_{k+1}, \ldots X_r, Z) :- p_{m+1}(X_1, \ldots, X_{k-1}, 0, X_{k+1}, \ldots X_r, Z)$$

$$p_m(X_1, \ldots, X_{k-1}, s(Y), X_{k+1}, \ldots X_r, Z) :- p_j(X_1, \ldots, X_{k-1}, Y, X_{k+1}, \ldots X_r, Z).$$

Finally, the clause corresponding to the terminal states of the register machine:

$$p_{t+1}(X_1, X_2, \ldots, X_r, X_2).$$

Theorem 8.5: *For every register program I, the* PROLOG *program $P(I)$ with any implementation computes the same partial function as I.*

Proof: Fix a natural number a and a register machine program I which computes a partial recursive function f. Consider the program $P(I)$ with the goal clause $G_0 = \{\neg p(s^a(0), Z)\}$. Notice first that every clause in the program $P(I)$ has at most one literal in its body. Thus every goal clause appearing in any SLD–proof beginning with G has at most one literal.

Thus the course of any such SLD–proof is independent of the selection rule. We also wish to show that result of the search for an SLD–refutation is both correct and independent of the search procedure. To see this we need a more detailed analysis of the execution.

When we start with the goal clause $G_0 = \{\neg p(s^a(0), Z)\}$ the first resolution must be with the first clause in $P(I)$; the result is

$$G_1 = \{\neg p_1(s^a(0), 0, \ldots, 0, Z)\}.$$

Suppose the execution of the register program I produces at its n^{th} step the state $I_{m(n)}(m(1,n), m(2,n), \ldots, m(r,n))$ with $m(n) \leq t+1$. We claim that, as long as the machine has not halted before step n, the succeeding goal clause G_n of the SLD–proof is precisely

$$\{\neg p_{m(n)}(s^{m(1,n)}(0), s^{m(2,n)}(0), \ldots, s^{m(r,n)}(0), Z)\}$$

and that no substitutions are made for Z. The proof is by induction so suppose the claim is true for n.

If $I_{m(n)}$ is of type (i) the register machine moves to state

$$I_{m(n)+1}(m(1,n), \ldots m(k-1,n), m(k,n)+1, m(k+1,n), \ldots, m(r,n)).$$

There is only one program clause with $p_{m(n)}$ in its head (the one corresponding to $I_{m(n)}$) and so only one with which we can resolve G_n. This resolution obviously produces the desired result:

$$\{\neg p_{m(n)+1}(s^{m(1,n)}(0), s^{m(2,n)}(0), \ldots, s^{m(k,n)+1}(0), \ldots, s^{m(r,n)}(0), Z)\}.$$

If $I_{m(n)}$ is of type (ii) there are two program clauses whose heads contain $p_{m(n)}$. The argument divides into cases according to whether or not $m(k,n) = 0$. In either case there is exactly one clause whose head can be unified with G_n so the resolution is uniquely determined and G_{n+1} has the required form.

Thus, if I is nonterminating when started in state $I_1(a,0)$ (i. e. f is not defined at a), then $P(I)$ is nonterminating when started with the goal $\{\neg p(s^a(0), Z)\}$ as required. Suppose then $f(a) = b$. If I is started in state $I_1(a,0)$ then it must terminate at step $n+1$ of its execution by entering some state $I_{t+1}(a, b, c_2, \ldots, c_r)$. By the above induction argument, clause G_{n+1} of the (unique) SLD–proof is

$$\{\neg p_{t+1}(s^a(0), s^b(0), s^{c_2}(0), \ldots, s^{c_r}(0), Z)\}.$$

Once again, there is exactly one program clause with p_{t+1} in its head: the final one of the program. Resolving G_{n+1} with this fact gives $\square$ with an mgu including the substitution $Z/s^b(0)$, as desired. $\square$

Corollary 8.6: *Every (partial) recursive function f is computable by a* PROLOG *program P_f that executes correctly under any implementation of* PROLOG.

Proof: Every (partial) recursive function f is computed by some register machine program I_f. By the theorem, $P(I_f)$ is a PROLOG program that computes f under any implementation. □

Of course, when referring to implementations of PROLOG, we have (tacitly) assumed we had one which correctly implements resolution (otherwise it would be meaningless). In particular, contrary to all actual implementations, one must typically assume some correct unification algorithm to be assured of the correctness of any result. It is worth pointing out that the PROLOG programs $P(I)$ constructed above from register machine programs run correctly even when the occurs check is omitted from the unification algorithm.

Corollary 8.7: *Every (partial) recursive function f is computable by a* PROLOG *program P_f that executes correctly under any implementation of* PROLOG *(even with the occurs check omitted from its unification algorithm).*

Proof: A careful look at the progression of the SLD–proof from $P(I) \cup \{G\}$ starting with $G = \{p(s^a(0), Z)\}$, as analyzed in the proof of Theorem 8.5, shows that all substitutions employed are ground. The unifications always work on a goal clause in which Z is the only nonground term and, until the last step of the proof, no substitution is made for Z. Thus all substitutions for variables in the program clauses before the last step are ground. At the last step (if there is one), we make a ground substitution for all the variables in the program clause including Z. As no ground substitution can violate the occurs condition on substitutions, the fact that we omitted the check has no significance. □

We can now prove the undecidability of the halting problem for PROLOG and the general validity problem for predicate calculus.

Theorem 8.8: *The halting problem for register machine programs is undecidable, that is, there is no effective procedure for determining if a given program halts on a given input.*

Proof: We assume Church's thesis that every effectively calculable function is recursive, i.e., can be computed by a register machine program. A formal proof that there is no recursive solution for the halting problem can be obtained without Church's thesis by explicitly writing the programs for the few functions that appear in the following argument. Such programs can be found in the basic papers or many standard texts on computability. Just as we can make a list of all finite sequences of numbers, we can make an effective list of all programs for register machines. If the halting problem

were decidable, there would be a recursive function $h(x)$ such that, if the x^{th} program on this lists halts on input x with output y, then $h(x) = y+1$ and otherwise $h(x) = 0$. We immediately deduce a contradiction by the classical diagonalization argument: As h is recursive (by Church's thesis), it is computed by some register machine program. Suppose h is computed by the n^{th} program on our list. Consider $h(n)$. As h is defined on every number, the n^{th} program with input n halts with some value y. By definition then, $h(n) = y+1$ contradicting the assumption that the n^{th} program computes h. □

Corollary 8.9: *The halting problem for* PROLOG *programs with any implementation, with or without the occurs check in the unification algorithm, is undecidable.*

Proof: The proof is immediate from the theorem and results 8.5–8.7. □

Corollary 8.10 (Church's Theorem)**:** *The validity problem for predicate logic is undecidable, i.e., there is no effective procedure for deciding if a given sentence of the predicate calculus is valid.*

Proof: If I is a register machine program then I halts with input a if and only if the search for an SLD–refutation of $P(I) \cup \{\neg p(s^a(0), Z)\}$ terminates successfully by Theorem 8.5. By the completeness and soundness of SLD–resolution this happens if and only if $\exists Z p(s^a(0), Z)$ is a logical consequence of $P(I)$. As $P(I)$ consists of a finite set $\{C_1, \ldots, C_n\}$ of clauses, the termination is equivalent to the conjunction of the universal closure of the clauses C_i implying $\exists Z p(s^a(0), Z)$. Thus, if we could decide the validity problem for predicate calculus, we could decide the halting problem and contradict Theorems 8.8–9. □

Exercises

A very common definition of the *partial recursive functions* proceeds by closing some simple functions under composition, primitive recursion and a least number operator:

The successor function $s(x) = x + 1$ is partial recursive.

The constant function $c(x) = 0$ is partial recursive.

For each i and j the projection function $p_{i,r}(x_1, \ldots, x_r) = x_i$ is partial recursive.

If $g_1, \ldots, g_n$ and h are partial recursive, then so is $f(x_1, \ldots, x_r) = h(g_1(x_1, \ldots, x_r), \ldots, g_n(x_1, \ldots, x_r))$.

If $r > 1$ and g and h are partial recursive, then so is the function f defined "by primitive recursion" as follows:

$$f(0, x_2, \ldots, x_r) = g(x_2, \ldots, x_r)$$
$$f(x_1 + 1, x_2, \ldots, x_r) = h(x_1, f(x_1, x_2, \ldots, x_r), x_2, \ldots, x_r).$$

If $f(x_1, \ldots, x_r, y)$ is partial recursive then so is the function $g(x_1, \ldots, x_r)$ defined by setting $g(x_1, \ldots, x_r)$ equal to the least number y such that $f(x_1 \ldots, x_r, z)$ is defined for every $z \leq y$ and $f(x_1, \ldots, x_r, y) = 0$.

1. Show (by induction on the definition given above) how to write PROLOG programs that compute each partial recursive function.
2. Prove that the programs given in answer to Exercise 1 compute correctly for any sound and complete implementation of PROLOG. (Hint: By completeness of the implementation, it suffices to find one LD–refutation for each desired computation. Now follow the path of the computation. For the other direction, use soundness and the fact $\mathbb{N}$ is a natural model for the program with the intended semantics.)
3. Let $Q(I)$ be the set of clauses corresponding to register machine program I for the partial recursive function f as given before Definition 8.4.
   (i) Argue semantically to show that, for any numbers $n, m, a, b, c, \ldots$, $p_1(s^n(0), 0, \ldots, 0) \rightarrow p_{t+1}(s^n(0), s^m(0), s^a(0), s^b(0), \ldots,)$ is a logical consequence of $Q(I)$ if and only if $f(n)$ is defined and equal to m.
   (ii) Apply Theorem 8.8 to give a proof of Theorem 8.10 that does not depend on the notions of PROLOG implementation or even resolution.
   (iii) Prove that $Q(I)$ as a PROLOG program correctly computes the same partial function as register machine I for any sound and complete implementation. (Hint: Follow the hint for Exercise 2.)
4. Give an example of a register machine program I such that $Q(I)$ as a PROLOG program does not correctly compute the same partial function as does I if the standard implementation of PROLOG is used.
5. A set W of natural numbers is *recursively enumerable* if there is an effective procedure to list its members, i.e., it is empty or the range of a (total) recursive function. Use Church's Thesis to prove that a set S is recursive iff both W and its complement, $\mathbb{N} - W$, are recursively enumerable. (Hint: Argue informally that, given listings for both S and $\mathbb{N} - W$, you can calculate the characteristic function of S.)
6. Show that there is a PROLOG program P such that the logical consequences of CWA(P) as defined in §6 are not recursively enumerable. (Hint: By Church's Thesis and the results of this section, there is a program P which computes the partial recursive function f such that $f(n, x) = 0$ iff the n^{th} register machine program halts on input x. As the only model of CWA(P) is $\mathbb{N}$, the logical consequences of CWA(P) are precisely the true facts about this function. In particular, $\neg p(s^n(0), 0)$ is a logical consequence of CWA(P) iff the n^{th} register program does not halt on input x. Now use Exercise 5 to argue that this would contradict Theorem 8.8.)

Suggestions for Further Reading

A good survey article on logic programming is Apt [1990, 5.4]. The March 1992 issue of the *Communications of the ACM* is devoted to Logic Programming. It includes a brief history (Robinson [1992, 5.4]) as well as surveys of the impact of logic programming on databases (Grant and Minker [1992, 5.4]) and the Fifth Generation Computing Project (Furukawa [1992, 5.4]). The standard text on the theory of PROLOG is Lloyd [1987, 5.4] which also has an extensive bibliography. Another text which stresses resolution and the connection of logic programming to relational databases is Maier and Warren [1988, 5.4].

For practical programming in PROLOG, see Bratko [1986, 5.4], Sterling and Shapiro [1987, 5.4] and Dodd [1989, 5.4]. Clocksin and Mellish [1981, 5.4] used to be the standard text and is still good.

For more on termination problems in logic programming and PROLOG, see Apt and Pedreschi [1991, 5.4] and the references there.

Logic with equality is treated in all the standard texts on mathematical logic mentioned in the suggestions for further reading at the end of Chaper I. Consider in particular, Mendelson [1964, 3.2], Enderton [1976, 3.2] or Shoenfield [1967, 3.2]. A basic model theory book such as Chang and Keisler [1973, 3.4] is also a good place to look

For the treatment of negation as failure in PROLOG, see Chapters 3–4 of Lloyd [1987, 5.4] and the appropriate references including Clark [1978, 5.4] and Shepherdson [1984, 1985 and 1987, 5.4]. A good current survey and extensive bibliography can be found in Shepherdson [1992, 5.4].

For basic articles on nonmonotonic logic, see Ginsberg [1979, 5.5]. For further developments along the lines of §7, we recommend the series of papers by Marek, Remmel and Nerode [1990, 5.5]. For the different approaches and sources of nonmonotonic logics mentioned at the beginning of §7, see Apt, Blair and Walker [1987, 5.4], Clark [1978, 5.4], Doyle [1979, 5.5], Hintikka [1962, 5.5], McCarthy [1980, 5.5], Minsky [1975, 5.5], Moore [1985, 5.5] and Reiter [1980, 5.5]. A good source of current research articles which are often quite accessible are the TARK (*Theoretical Aspects of Reasoning about Knowledge*) volumes in list 5.5.

For an excellent introduction to recursive function theory (the theory of computability) from an informal viewpoint assuming Church's thesis, we recommend Rogers [1967, 3.6]. For more details on the equivalence of various definitions and actual "implementations", we suggest Davis and Weyuker [1983, 5.2] and Odifreddi [1989, 3.6]. For more specifically on register machines, see Fitting [1987, 5.2], Cutland [1980, 3.6], Tourlakis [1984, 3.2] or the original papers by Shepherdson and Sturgis [1961, 3.6] and Minsky [1961, 3.6]. An approach somewhat more oriented to computer science can be found in Machtey and Young [1978, 5.4].

The next result along the lines of the undecidability of predicate logic is Gödel's celebrated incompleteness theorem: There are sentences φ in the predicate logic language for arithmetic such that neither φ nor $\neg\varphi$ is provable from any reasonable axioms for arithmetic. For a brief introduction to incompleteness, see Crossley [1990, 3.2]. Expositions of this result can also be found in many standard textbooks such as Boolos and Jeffrey [1989, 3.2], Enderton [1972, 3.2] and Mendelson [1979, 3.2]. A treatment from a recursion–theoretic point of view is in Odifreddi [1989, 3.6]. A puzzler's approach can be found in Smullyan [1987 and 1978, 3.2]. The original paper Gödel [1931, 2.3] is still worth reading.

IV Modal Logic

1. Possibility and Necessity; Knowledge or Belief

Formal modal logics were developed to make precise the mathematical properties of differing conceptions of such notions as possibility, necessity, belief, knowledge and temporal progression which arise in philosophy and natural languages. In the last twenty–five years modal logics have emerged as useful tools for expressing essential ideas in computer science and artificial intelligence.

Formally, modal logic is an extension of classical propositional or predicate logic. The language of classical logic is enriched by adding of new "modal operators". The standard basic operators are traditionally denoted by $\Box$ and $\Diamond$. Syntactically, they can be viewed as new unary connectives. (We omit a separate treatment of propositional logic and move directly to predicate logic. As we noted for classical logic in II.4.8, propositional logic can be viewed as a subset of predicate logic and so is subsumed by it. The same translation works for modal logic.)

Definition 1.1: If $\mathcal{L}$ is a language for (classical) predicate logic (as defined in II.2) we extend it to a *modal language* $\mathcal{L}_{\Box,\Diamond}$ by adding (to Definition II.2.1) two new primitive symbols $\Box$ and $\Diamond$. We add a new clause to the definition (II.2.5) of formulas:

(iv) If φ is formula then so are $(\Box\varphi)$ and $(\Diamond\varphi)$.

The definitions of all other related notions such as subformula, bound variable and sentence are now carried over verbatim.

When no confusion is likely to result we drop the subscripts and refer to $\mathcal{L}_{\Box,\Diamond}$ as simply the (modal) language $\mathcal{L}$.

Interpretations of modal languages were originally motivated by philosophical considerations. One common reading of $\Box$ and $\Diamond$ are "it is necessary that" and "it is possible that". Another is "it will always be true that" and "it will eventually be true that". One should note that the intended relation between $\Box$ and $\Diamond$ is like that between $\forall$ and $\exists$. They are dual operators in the sense that the intended meaning of $\Diamond\varphi$ is usually $\neg\Box\neg\varphi$. The two interpretations just mentioned have ordinary names for both operators. At times it is natural to use just one. Interpretations involving knowledge or belief, for example, are typically phrased in a language with just the operator $\Box$ (which could be denoted by $\mathcal{L}_{\Box}$) and $\Box\varphi$ is understood