Springer
New York
Berlin
Heidelberg
Barcelona
Budapest
Hong Kong
London
Milan
Paris
Santa Clara
Singapore
Tokyo

Melvin Fitting

FIRST-ORDER LOGIC AND AUTOMATED THEOREM PROVING

Second Edition

Springer

Melvin Fitting
Department of Mathematics and Computer Science
Lehman College, The City of New York University
Bronx, NY 10468-1589 USA

Series Editors

David Gries
Fred B. Schneider

Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853-7501 USA

With 15 line illustrations.

Library of Congress Cataloging-in-Publication Data
Fitting, Melvin. 1942-
First-order logic and automated theorem proving / Melvin Fitting. – 2nd ed.
p. cm. – (Graduate texts in computer science)
Includes bibliographical references and index.
ISBN-13:978-1-4612-7515-2 e-ISBN-13:978-1-4612-2360-3
DOI:10.1007/978-1-4612-2360-3
1. Automatic theorem proving. 2. Logic. Symbolic and mathematical. I. Title. II. Series III. Series: Graduate texts in computer science (Springer-Verlag New York Inc.)
QA76.9.A96F68 1996
511.3–dc20 95-45225

Printed on acid-free paper.

Production managed by Robert Wexler; manufacturing supervised by Jacqui Ashri.
Photocomposed copy prepared from the author's LaTeX files using Springer's `svwidecm` style file.

9 8 7 6 5 4 3 2 1

To Raymond Smullyan
who brought me up
into the trees.

Preface

There are many kinds of books on formal logic. Some have philosophers as their intended audience, some mathematicians, some computer scientists. Although there is a common core to all such books, they will be very different in emphasis, methods, and even appearance. This book is intended for computer scientists. But even this is not precise. Within computer science formal logic turns up in a number of areas, from program verification to logic programming to artificial intelligence. This book is intended for computer scientists interested in automated theorem proving in classical logic. To be more precise yet, it is essentially a theoretical treatment, not a how-to book, although how-to issues are not neglected. This does not mean, of course, that the book will be of no interest to philosophers or mathematicians. It does contain a thorough presentation of formal logic and many proof techniques, and as such it contains all the material one would expect to find in a course in formal logic covering completeness but not incompleteness issues.

The first item to be addressed is, What are we talking about and why are we interested in it? We are primarily talking about truth as used in mathematical discourse, and our interest in it is, or should be, self-evident. Truth is a semantic concept, so we begin with models and their properties. These are used to define our subject.

The second issue is how we, as limited human beings, can know what is true. For this we have a device called a *proof.* Many formal proof procedures have been developed over the years: axiom systems, natural deduction, tableaux, resolution. We present several of these and show how they are used. Of course, the connections between these proof procedures and truth must be established. We need what are called *soundness* and *completeness* results. Ours are demonstrated in a uniform way for all the systems we consider. Thus, we are able to discuss many formal proof procedures without doing much more work than if we had only discussed one.

Finally, how can we get a machine—a computer—to use one of our proof procedures? When we human beings prove things, we bring all our insight and experience to bear, not to mention the wisdom of the past stored up in books. We are still learning how to give machines such knowledge. Generally, we must be satisfied if we can give a computer a simple-minded recipe by which it can find proofs, even though the proofs may not be very clever. So, which of the formal proof procedures that humans have developed will allow themselves to be applied blindly, mechanically? What recipes can we give a computer that are reasonably efficient and still are guaranteed to work? After semantics and formal proof procedures, this constitutes the third major topic of the book.

We discuss automation for tableaux and for resolution. For tableau systems we give usable implementations in Prolog and we prove, of these implementations, that they do the job. Similar implementations of resolution are set up as exercises and projects. We have chosen Prolog as our implementation language because it allows us to get to the heart of the matter almost immediately, and results in code that is rather easy to follow. If you do not already know Prolog, here is a good opportunity; the understanding of Prolog that is necessary is fairly basic. We use few programming tricks. Indeed, if one understands the Prolog code given here, implementing comparable theorem provers in other languages should be straightforward. We do not claim that our theorem provers are particularly efficient, though. We tried to commit our quota of sins on the side of clarity, instead of efficiency. First understand, then speed it up if you can.

Automated theorem proving has two goals: (1) to prove theorems and (2) to do it automatically. Over the years experience has shown these goals are incompatible. Fully automated theorem provers for first-order logic have been developed, starting in the 1960s, but as theorems get more complicated, the time that theorem provers spend tends to grow exponentially. As a result, no really interesting theorems of mathematics can be proved this way—the human life span is not long enough.

The problem is to prove interesting theorems; the solution is to give theorem provers *heuristics*—roughly, rules of thumb for knowledge and wisdom. Some heuristics are fairly general; for example, in a proof that is about to break into several cases, do as much as possible that will be of broad applicability before the division into cases occurs. But many heuristics are area-specific; for instance, heuristics appropriate for plane geometry will probably not be appropriate for group theory. The development of good heuristics is a major area of research and requires much experience and insight.

But still, heuristics must be imposed on some basic, underlying proof mechanism. And it is here that fully automated theorem provers play their role. Generally, heuristics are implemented as restrictions or modifications to such systems. So, as a basis for more sophisticated work, for general understanding, and for historical reasons as well, fully automated theorem provers are the place to start, and they are what we will be concerned with here. We will develop the foundations for further reading and research, but we will not go beyond that.

Almost all fundamental work on automated theorem proving has been based on *Resolution*, a method that is due to J. A. Robinson [42] in the 1960s and that descends from techniques developed by Herbrand [25] in the 1930s. But there is another method, *Semantic Tableaux*, also developed in the 1960s, by R. M Smullyan [48], which descends from work of Gentzen [22] in the 1930s and Beth [5] in the 1950s. (Tableaux actually first appeared in [30], though this paper was largely unknown until recently.) Gentzen's and Herbrand's work are closely related, but still resolution and semantic tableaux have different flavors to them. Tableau-based theorem provers have been comparatively rare in the field. Both methods are, we think, of basic importance. A few books have treated both [2, 21] and are recommended for additional reading.

So, a brief outline of the book is this. We begin with propositional logic, move on to first-order logic, then finish up with first-order logic with equality. For each of these we present both resolution and semantic tableau systems as primary. Implementations of semantic tableaux in Prolog are given, and similar implementations of resolution are outlined as projects. We also present natural deduction, Gentzen sequent calculi, and axiomatic systems, because these require little additional work and are common in the literature, though they are generally less appropriate for automated theorem proving. Also, for each level of logic that we consider, we discuss necessary semantical background: Boolean valuations in the propositional case; models in the first-order case; and normal models in the first-order case with equality. Soundness and completeness of our theorem provers is established. Details of syntax, as well as semantics are presented, including normal form theorems. In general we use the device of uniform notation that is due to R. M. Smullyan, which allows us to have many connectives and quantifiers present in our language without the need for elaborate theorem provers with many special cases. We hope to keep it clean and elegant.

Preface to the Second Edition

This edition differs from the previous one in three ways. First, it contains much new material. Second, revisions of the original material have been made throughout. Third, it contains this preface.

Chapter 8 is almost entirely new (a few sections were moved to it from elsewhere). The new material consists of the following:

1. A discussion of the **AE** calculus. This is a decidable part of first-order logic that is natural from the tableau perspective, powerful, easy to implement, and of historical significance.
2. Herbrand's theorem. This is presented twice: non-constructively and constructively. The non-constructive version is based on the model existence theorem; the constructive one, on the tableau formulation.
3. Gentzen's theorem (Gentzen's Hauptsatz). The proof is essentially the constructive one of Gentzen, but following Smullyan, it is formulated explicitly for the tableau calculus. This reformulation tends to clear away some of the unnecessary detail.
4. A proof that is due to Statman that cut elimination can make proofs blow up exponentially.
5. Craig's interpolation theorem. A non-constructive proof, based on the model existence theorem, was in the first edition and is still present. But now a second, constructive, proof has been added. This proof extracts an interpolant from a tableau proof.

6. Lyndon's interpolation theorem. This refinement of Craig's theorem is obtained from the same proof, with no additional work.
7. Lyndon's homomorphism theorem. This adds to Beth's theorem an additional application of interpolation results and is of considerable interest for its own sake.

It was pointed out by Reiner Hähnle and Peter Schmitt, and independently by Wilfried Sieg, that the so-called Free-Variable δ-Rule given in the first edition was unnecessarily inefficient. (This rule deals with occurrences of existential quantifiers in tableau or resolution proofs.) Following their suggestions, a new version of the rule has replaced the old one. This required some changes in soundness proofs, but primarily it also forced the rewriting of much of the Prolog implementation of the first-order tableau theorem prover (and the version incorporating equality).

At the suggestion of Krzysztof Apt, the original treatment of Multiple Unification in Chapter 7 has been modified. In the original version it used the proof of the Unification Theorem, now it uses the Theorem itself.

A number of exercises were added. In addition, those exercises that are essentially programming projects have been clearly marked (with **P** as a superscript). There are almost 30 such exercises—they range from relatively easy modifying of programs in the text to substantial pieces of new work.

Since the first edition of this work appeared, numerous papers and books on automated theorem-proving have been published, most of which are beyond our scope. There are, however, two items that are most pertinent, both handbooks. The first is the *Handbook of Logic in Artificial Intelligence and Logic Programming* [20]. This is an extensive multi-volume work; fortunately most volumes have already appeared. It includes coverage of topics like unification, and resolution and tableau theorem-proving. The other item is the *Handbook of Tableau Methods* [13], which should appear shortly. As its name implies, it contains thorough presentations of tableau techniques, applied to both classical and non-classical logics.

Finally, lengthy sections of Prolog code are included in the text. These can be obtained in the following ways: First, by anonymous ftp, at the address ftp.springer-ny.com, in the directory /pub/supplements/mfitting (log on as anonymous, and use your e-mail address as password); second, at web site http://www.springer-ny.com/supplements/mfitting.html.

Contents

List of Tables

1

Background

There are many useful logics: temporal, modal, relevance, intuitionistic, etc. They differ in what concepts are being considered and in what the basic features of these concepts are thought to be. For example, should the passage of time play a formal role (as in temporal logic), or not? Must existence assertions have constructive content (as in intuitionistic logic), or not? In the family of formal logics, one is central: classical logic. It is the most widely used logic, the logic underlying mathematics as it is generally practiced, and the logic on top of which many others have been built. Indeed, for most people who have occasion to use formal methods, logic is synonymous with classical logic. Classical logic is the subject of this book. Our formal treatment of classical logic begins in the next chapter; here we present the intuitive background. We want to make sure there is a ground-level informal understanding before erecting a rigorous mathematical structure.

In everyday life there are assertions whose truth value is unclear or problematic, such as "That is an ugly chair," and "Then so are you." Classical logic is incapable of dealing with such things. In fact, classical logic was created to embody the reasoning principles of mathematics, where ambiguity and imprecision are a Bad Thing. When classical logic is applied to non-mathematical examples, the examples are first "mathematicized." In the real world we might argue about whether block B is behind block A or not—maybe it depends on one's point of view. But we can create an ideal world, a mathematical model, in which either B is behind A, or it isn't. Classical logic can be used to reason correctly about such a model. Whether the model accurately reflects the real world is a separate issue. But this is standard operating procedure generally, not just where logic is concerned. Differential equations don't describe a real-world vibrating string; they describe the behavior of a mathematical model of a vibrating string, which in turn captures the real thing more or less well, depending.

So, in classical logic we investigate the principles of reasoning for perfect worlds, where truth is unqualified and there are no shades of grey.

When we reason we use *sentences.* These sentences are built up from primitive assertions about the world (or rather, our model of it). They are built up using words like *and*, *or*, *not*, *implies*, *every*, *some*, and *equals.* Basically, it is the behavior of these words that we will study. Most of these words have meanings that are fairly straightforward to grasp in an informal, intuitive way. But we should be a little careful about *implies*, because it plays a very central role, and is used in several senses in everyday discourse (even among mathematicians).

Commonly, one says "P implies Q" only if one already knows that P is the case. "P implies Q" is thus often used as another way of asserting Q. But in reading mathematics, we are frequently faced with the assertion "P implies Q" where the truth of P is not known, or even where P is false. Consider, for example: "whatever integer x is, x is even implies $x+2$ is even." This seems correct, but then we could take x to be 7, and we find we are committed to the truth of "7 is even implies $7+2$ is even." This is disconcerting, but it would be wrong to call "7 is even implies $7+2$ is even" false, because the more general sentence from which it came would then be false, which is not acceptable. We could say that "7 is even implies $7 + 2$ is even" is neither true nor false, but in mathematics we want every sentence to have an unambiguous truth value. Then we are forced to take it to be true. This is not really counterintuitive, it is simply a case that does not come up in everyday life, that of something false implying something false. Similar difficulties are met in two other cases: false implying true, and true implying true. Both are taken to be true in mathematical usage. What all this amounts to is that mathematicians use *implies* in what is called the *material* sense; "P implies Q" is taken to mean nothing more than "either P is false or (P is true and so) Q is true." "P implies Q" entails no particular relationship between P and Q (as it would in relevance logic say) save the single condition: Q must be true if P is true.

Special symbols are introduced to represent the logical constructs we have been discussing. We write $\wedge$ for *and*, $\vee$ for *or*, $\neg$ for *not*, $\supset$ for *implies*, $\forall$ for *every* and $\exists$ for *some.* The symbols $\wedge$, $\vee$, $\neg$ and $\supset$ are called *logical connectives* (there are others as well that we will introduce in the next chapter). The symbols $\forall$ and $\exists$ are called *quantifiers.* Other books may use different symbols.

Now we begin looking at how these notions are used in a particular mathematical theory. We have chosen *arithmetic*, the theory of the natural numbers, 1, 2, 3,..., because it is familiar to everyone.

Numbers are abstract objects. If we are to talk about them, we need names for them. Let us say we have a *constant symbol* 1, which we intend as a name for the first of the natural numbers. (It is important not to mix up the symbol with the object it is naming.) Also suppose we have a one-place *function symbol* s, intended to denote the successor function. Then $s(1)$ names the second natural number, $s(s(1))$ names the third, and so on. We also want some *operation symbols*, say $+$ and $\times$, intended to denote addition and multiplication. Now we can form more complicated names like $s(s(1)) \times (1 + s(1))$, and even name-like expressions containing variables, such as $x + s(1)$. These expressions are called *terms*. Loosely, a term is a name, or something that would become a name if names were substituted for variables.

Next, there are expressions that make assertions but that do not involve any logical connectives or quantifiers; examples are $s(1) + s(s(1)) = s(s(s(s(1))))$ and $s(1) > 1$. There are also assertion-like expressions, containing variables, such as $x + 1 = s(s(1))$. Such expressions are called *atomic formulas*. Notice that we have assumed we have in our language *relation symbols* $=$ and $>$. Saying what relation symbols we have available is part of the specification of the language of arithmetic. If we think of $=$ as denoting the equality relation and $>$ as denoting the greater-than relation, then atomic formulas without variables are either true or false, and those with variables become true or false when values for the variables are specified.

Starting with atomic formulas, using connectives and quantifiers, we may build up more complex expressions, or *formulas*, for example: $\neg(1 + s(1) = 1)$. If we take the various symbols as denoting numbers, functions and relations as described previously, $1 + s(1) = 1$ is false, and hence $\neg(1 + s(1) = 1)$ is true. Further, $(1 + 1 = s(1)) \wedge \neg(1 + s(1) = 1)$ asserts that both $(1 + 1 = s(1))$ and $\neg(1 + s(1) = 1)$ hold, and so it is true. $(\exists x)(x + 1 = s(1))$ asserts that for some value of the variable x, $x+1 = s(1)$ is true, and this is so, hence $(\exists x)(x+1 = s(1))$ is true. On the other hand, $(\forall x)(x+1 = s(1))$ is false. For a more complicated example, $(\forall x)[(\exists y)(s(1) \times y = x) \supset (\exists z)(s(1) \times z = s(s(x)))]$ is true (informally, it asserts that whatever number x is, if x is even, so is $x + 2$).

Generally, we will use the word *sentence* for a formula that does not have variables we can substitute values for. It is sentences that we think of as making assertions.

Two separate notions were intertwined in the preceding discussion. We have a formal language that we use to make assertions in the form of sentences, and we have a mathematical structure that the assertions are about. The relationship between sentence and subject matter is something we rarely think about in most of mathematics, but it is a central issue in logic.

On the one hand we have created a formal language of arithmetic, with a constant symbol 1, a function symbol s, two operation symbols $+$ and $\times$, and two relation symbols $=$ and $>$, and various other constructs like variables, connectives, and quantifiers. This formal language could be modified in many ways: We could add other constant symbols like 2 and 3, we could add predicate symbols like *is_even*, and so on. We could make many changes and still have a language suitable for saying things about numbers. Likewise, we could have designed a formal language suitable for a different purpose altogether—we will do so shortly. This part of formal logic is *syntax*, the specification of a formal language and its constituent parts.

On the other hand we have a mathematical *structure*. There is a *domain*, the positive integers, that we think of our variables as ranging over and our quantifiers as quantifying over. There is a particular member of this domain for the constant symbol 1 to name; there are particular functions, operations, and relations on the domain for our function symbol, operation symbols, and relation symbols to designate. In short, our formal language has been given a particular *interpretation* in a *model*. Unless we have done this, it makes no sense to say a term is a name, or a sentence is true. Meanings must be assigned to the expressions of a formal language. This part of formal logic is *semantics*.

We consider a few more examples before we turn to general considerations. A *group* is a mathematical structure with a binary operation that is associative, for which there is a (right and left) identity element, with each element having a (right and left) inverse. Suppose we create a language having one constant symbol, e, intended to denote the group identity; one function symbol i, intended to denote the map from objects to inverses; one operation symbol, $\circ$, to denote the group operation; and the equality symbol $=$. The following sentences, the group axioms, embody the properties required of a group:

1. $(\forall x)(\forall y)(\forall z)[x \circ (y \circ z) = (x \circ y) \circ z]$
2. $(\forall x)[(x \circ e = x) \wedge (e \circ x = x)]$
3. $(\forall x)[(x \circ i(x) = e) \wedge (i(x) \circ x = e)]$

Now, what is a group? It is any structure in which the relation symbol $=$ is interpreted by the equality relation and in which these three sentences are true. One of the first results established in a course on group theory is that the inverse operation is an involution, $(\forall x)(i(i(x)) = x)$. In other words, this sentence is true in all groups. In some sense it "follows from" the three group axioms by purely logical considerations. We will provide methods that allow us to show this is so.

For the next example, suppose we have a formal language with relation symbols *parent_of*, *male*, and *female*. A 'standard model' is one whose domain is a collection of people, and in which $parent\_of(x, y)$ is true just when x is a parent of y, $male(x)$ is true just when x is a male, and $female(x)$ is true just when x is a female. Then in each standard model, the formula $parent\_of(x, y) \wedge male(x)$ is true if x is the father of y. You might try writing formulas that, in standard models, characterize the brother, sibling, and aunt relations. Also add an equality symbol to the language, and write a sentence asserting that each person has a unique father and a unique mother.

The final example also assumes we have not only a formal language, but a particular family of models in mind. This time we have predicate symbols *Cube*, *Small*, *Large*, and so on, and two-place relation symbols *Smaller*, *RightOf*, *BackOf*, and so on. A standard model now is one in which the domain consists of three-dimensional objects, $Cube(x)$ is true if x is a cube, and so on. We can write sentences asserting things like "If one object is to the right of another, then the first is a small cube." In fact, this example is taken from a very nice piece of software entitled "Tarski's World," which allows one to become familiar with the basic concepts of the semantics of first-order logic. We recommend it highly [3]. ("Tarski's World" is available as a program for the Macintosh, for Windows, and for the Next, either alone, or as part of a book package, "The Language of First-Order Logic." Information is available from the distributor, Cambridge University Press, or from the web site http://csli-www.stanford.edu/hp/.)

Next we turn to general considerations. It is not up to us to decide which sentences of our formal language of arithmetic are true when interpreted as being about numbers. This is a book on logic, not arithmetic. But it is part of our job to determine which sentences are true because of their logical structure alone, and not because of any special facts about numbers.

For example, consider the following sentence: $(\forall x)[(x + 1 = s(s(1))) \vee \neg(x+1 = s(s(1)))]$. This is a true sentence of arithmetic, but we can show it is true without using any special facts about the natural numbers. To see this, replace $(x + 1 = s(s(1)))$ by $(\cdots x \cdots)$, thus concealing much of the sentence structure. Then the sentence becomes $(\forall x)[(\cdots x \cdots) \vee \neg(\cdots x \cdots)]$, and this will be true, provided, for each object n in our model, $(\cdots n \cdots) \vee \neg(\cdots n \cdots)$ is true. This, in turn, will be true for any particular n provided, either $(\cdots n \cdots)$ is true or $\neg(\cdots n \cdots)$ is true. But this must be the case, since $\neg(\cdots n \cdots)$ is true precisely when $(\cdots n \cdots)$

is not true. Thus, we are entitled to say $(\forall x)[(x+1 = s(s(1))) \vee \neg(x+1 = s(s(1)))]$ is true *by logic alone*, since its truth depends on no particular facts about numbers but only on the meanings of the logical connectives and quantifiers.

If we have a sentence of arithmetic, and we want to show it is true by its logical form alone, what sorts of things may we use, and what sorts of things are we not allowed to use? We may use the fact that 1 denotes an object, but we may not use the fact that it denotes the first natural number. We may use the fact that $+$ denotes an operation; we may not use the fact that it denotes addition. Similar considerations apply to $\times$ and s. We may use the fact that $>$ denotes a relation, but not that it denotes greater-than. What we may assume about $=$ is a special issue, which we will discuss further. Finally, we may not even assume particular facts about the domain, such as that it is countably infinite, or even that it is infinite.

Then what we are left with is this. A sentence is true by logic alone if it is true, no matter what domain we take its variables and quantifiers to range over, no matter what members of that domain its constant symbols name, and no matter what functions, operations, and relations on that domain its function, operation, and relation symbols designate. Such sentences are called *valid*.

In addition to validity, we also want to know what follows from what; which sentences are consequences of which others. This notion has a characterization similar to validity. A sentence X is a *logical consequence* of a set S of sentences if X is true whenever all the members of S are true, no matter what domain the variables and quantifiers of the language are thought of as ranging over, and no matter how the symbols of the language are interpreted in that domain.

It is one of the jobs of formal logic to properly define the notions of formal language, of model, of truth in a model, of validity, and of logical consequence. We have sketched the ideas informally; formal counterparts will be found in subsequent chapters.

It is not enough to have a *definition* of validity; we also want to be able to establish that certain sentences are, in fact, valid. In elementary arithmetic, algorithms have been devised that let us compute that $249 + 128 = 377$. To apply such algorithms, we do not use the *numbers* 249 and 128 or the *operation* $+$. Indeed, most of us have rather vague conceptions of what these things actually are. Rather, we perform certain mechanical manipulations on strings of symbols. Similar algorithms have been developed for classical logic; they are called *proof procedures*. A proof procedure does not make use of the meanings of sentences, it only manipulates them as formal strings of symbols. A proof procedure is said to *prove* a sentence or not, as the case may be. A proof procedure, then, is an algorithm in the usual mathematical sense and can be studied just like any other algorithm. In particular, correctness results can be established. It should be verified, of a proof procedure, that

1. It only proves valid sentences (it is sound).
2. It proves every valid sentence (it is complete).

In fact, even more is wanted; we should have proof procedures capable of dealing with logical consequence, not just with validity. The creation and investigation of proof procedures that capture validity and consequence are also part of formal logic.

We have yet a further goal. We want to investigate proof procedures that are suitable for computer implementation, and we want to implement them. Most proof procedures in the literature are non-deterministic. Many of them are capable of producing 'short' proofs when used with knowledge and intelligence, but when used purely mechanically, they can be extraordinarily inefficient. Certain ones, however, are well adapted to mechanical use. *Resolution* is the best-known such proof procedure; the *semantic tableaux* procedure is another, though it has not had the intensive development that resolution has. We will present both systems in considerable detail and discuss their implementation. We will also present several other proof procedures of more traditional kinds, partly for comparisons' sake, partly because they are common in the literature and should be part of one's general knowledge.

Issues of truth, validity, and consequence can be divided into levels of difficulty. At the simplest level we ask, What sentences are true solely because of the way they use the logical connectives—never mind the role of the quantifiers, variables, constant symbols, function symbols, or relation symbols? Restricting our attention to the role of logical connectives'yields what is called *propositional logic.* This is much simpler to deal with, and a wide variety of techniques have been developed that are suitable for it. We devote the first several chapters to propositional logic, both for its own sake, and because it provides a good background for the more difficult logics that follow.

Next we ask, What sentences are true because of the way they use the logical connectives *and* the quantifiers? This is the study of *first-order logic* and is the heart of the book.

The greater-than relation makes sense only for certain domains, and this also holds true for virtually all other relations. But the equality relation is meaningful for every domain. Suppose we ask which sentences are true because of their use of the logical connectives, the quantifiers, and the equality relation. *Normal* models are models in which the symbol = is interpreted as designating the equality relation. Restricting models to those that are normal gives us *first-order logic with equality.* It is the appropriate setting for the development of most formal mathematical theories. As it happens, the theoretical role of equality is easy to characterize, but it makes things much more difficult when implementing proof procedures. We take this up as our final major topic.

This is not the entire of classical logic. The quantifiers we have been discussing range over *objects.* We can also consider quantifiers ranging over *properties.* This is *second-order logic.* There is also *third-order logic*, *fourth-order logic*, and so on, and also *type theory*, which embodies them all. We do not go beyond first-order logic here; though we recommend Andrews' work [2] for those who are interested. Finally, we recommend several other books on logic, with particular relevance to automated theorem proving [21, 31, 44, 57].

2 Propositional Logic

2.1 Introduction

In classical propositional (or sentential) logic we study the behavior of the *propositional connectives*, such as *and* and *or*. We assume we have a family of sentences (hence the name 'sentential') that can be thought of as expressing propositions (hence the name 'propositional'). For instance, the English sentence "Snow is white," expresses the proposition that snow is white, which happens to be true. We begin with a family of elementary, or atomic, sentences, whose internal structure we do not analyze. Indeed, we represent these by single letters, propositional letters. All that matters for us is that these are either true or false, but never both. (Better said, the propositions expressed by these sentences are either true or false, but we will ignore such niceties from now on.) Then we form more elaborate sentences from these using the propositional connectives. Our major concern is how the truth or falsity of compound sentences depends on the truth or falsity of the atomic sentences making them up. In particular, we would like to know which compound sentences must be true, independently of the status of their atomic constituents. For propositional logic these are called *tautologies.* Much of classical propositional logic amounts to determining which sentences are tautologies.

We begin with formal definitions and a proper characterization of the notion of tautology—in effect the familiar mechanism of truth tables is used here. Then, in the next two chapters, we turn to proof theoretic methods—axiom systems, natural deduction, resolution, and semantic tableaux. Each of these amounts to a set of rules, which, if followed correctly, must determine the tautologyhood of a given propositional sentence. Thus, we wind up with several algorithms, algorithms based on resolution, on tableaux, etc. For each of these, we must prove formally that it meets our specifications. In propositional logic, a proof procedure (algorithm) that proves only tautologies is called *sound.* If it proves every tautology, it is called *complete.* So, for each kind of proof procedure, we must establish soundness and completeness. Finally, in the next chapter, we consider implementation issues for some of these proof procedures.

Truth tables themselves are implementable, so what is the need for other kinds of mechanisms? First, truth tables are always inefficient, while on a case-by-case basis, other mechanisms can be much better. But more fundamentally, the truth table mechanism is limited to the classical, propositional setting. Axiom systems, natural deduction, and so on, all extend readily to encompass classical first-order logic (allowing quantifiers), and even various non-classical logics. We will carry out the extension to the first-order setting, but we will not consider non-classical logics here at all. Since the classical propositional case is the simplest of all, it is best to get a thorough understanding of the various proof mechanisms here first, before moving on to more complicated things.

2.2 Propositional Logic—Syntax

Just as with programming languages, each logic has its syntax and its semantics. In the case of classical propositional logic, these are rather simple. We begin with syntax, defining the class of formulas.

We assume we have an infinite list, P_1, $P_2, \ldots$, of *propositional letters.* Informally, we can think of propositional letters as expressing propositions, without analyzing how. Formally, we may think of these as letters of an infinite (countable) alphabet. The only real requirement is that they should be distinct and recognizable. In general we will informally use P, $Q, \ldots$ for propositional letters, rather than P_1, $P_2, \ldots$.

A formula will be a word built up from propositional letters using propositional connectives. Propositional connectives can be zero-place (constants), one-place, two-place (binary), three-place, and so on. There are naturally two constants, since classical logic is a two-valued logic. We will denote these by $\top$ and $\bot$, for *true* and *false*, respectively. Negation ($\neg$) is the only one-place connective of interest. Three-or-more-place connectives rarely occur in practice. Different works on logic tend to differ on what is taken as a binary connective. Some may consider only $\wedge$ and $\vee$, while others may allow $\supset$, $\equiv$, and maybe more (or less). We leave the choice open for now and give a kind of generic definition. The issue will be considered again when we discuss semantics. The only other symbols we will need, to form propositional formulas, are left and right parentheses.

Definition 2.2.1 A (propositional) *atomic formula* is a propositional letter, $\top$ or $\bot$.

Definition 2.2.2 The set of *propositional formulas* is the smallest set $\mathbf{P}$ such that

1. If A is an atomic formula, $A \in \mathbf{P}$.
2. $X \in \mathbf{P} \Rightarrow \neg X \in \mathbf{P}$.
3. If $\circ$ is a binary symbol, then $X, Y \in \mathbf{P} \Rightarrow (X \circ Y) \in \mathbf{P}$.

As an example, if $\wedge$ and $\vee$ are binary symbols, then $\neg((P_1 \wedge P_2) \vee \neg(P_3 \wedge \neg\top))$ is a propositional formula. You should try showing this.

The definition of propositional formula presupposes the existence of a (unique) *smallest* set meeting conditions 1 through 3, where smallest means is a subset of all others. But the existence of such a set is easily established. First, there are sets meeting conditions 1 through 3, for example, the universal set consisting of all words formed from the symbols we are using. Next, it is easy to verify that the intersection of any family of sets meeting conditions 1 through 3 is another set meeting these conditions. Now, just form the intersection of the (nonempty) family of all sets meeting conditions 1 through 3. This will be the smallest such set.

From time to time we will define other concepts using this same technique. Defining things this way yields useful immediate consequences. One is a method of proof, the other is a method of definition.

Theorem 2.2.3 **(Principle of Structural Induction)**
Every formula of propositional logic has a property, $\mathbf{Q}$, provided:

Basis step *Every atomic formula has property $\mathbf{Q}$.*

Induction steps

If X has property $\mathbf{Q}$ so does $\neg X$.

If X and Y have property $\mathbf{Q}$ so does $(X \circ Y)$, where $\circ$ is a binary symbol.

Proof Let $\mathbf{Q}^*$ be the set of propositional formulas that have property $\mathbf{Q}$. The basis step and the induction steps say that $\mathbf{Q}^*$ meets conditions 1 through 3 of the definition of propositional formula. Since the set $\mathbf{P}$ of propositional formulas is the smallest set meeting these conditions, it follows that $\mathbf{P}$ is a subset of $\mathbf{Q}^*$, and hence every propositional formula has property $\mathbf{Q}$. $\square$

The other basic principle we need will allow us to define functions on the class of propositional formulas.

Theorem 2.2.4 **(Principle of Structural Recursion)** *There is one, and only one, function f defined on the set $\mathbf{P}$ of propositional formulas such that:*

> **Basis step** *The value of f is specified explicitly on atomic formulas.*
>
> **Recursion steps**
>
> *The value of f on $\neg X$ is specified in terms of the value of f on X.*
>
> *The value of f on $(X \circ Y)$ is specified in terms of the values of f on X and on Y, where $\circ$ is a binary symbol.*

We omit the proof of this theorem, which is similar to that of structural induction, but which requires consideration of existence principles for functions, and more generally sets, and so would take us rather far afield.

Syntax is really part of the subject matter of automata theory. It is immediate from the form of the definition that propositional formulas are a context free language, and hence much is known about them [28]. The main thing we need is that we have an unambiguous grammar. We state this as a theorem and leave the proof as a series of exercises. In fact, the theorem is needed for the proof of the Principle of Structural Recursion, Theorem 2.2.4.

Theorem 2.2.5 **(Unique Parsing)** *Every propositional formula is in exactly one of the following categories:*

1. *atomic*
2. *$\neg X$, for a unique propositional formula X*
3. *$(X \circ Y)$ for a unique binary symbol $\circ$ and unique propositional formulas X and Y*

Exercise 2.2.3 shows that every propositional formula falls into at least one of the three categories listed in the Unique Parsing Theorem. It is more work to demonstrate uniqueness, however. The job is easier in a restricted setting in which there are no negations. So, temporarily, by a *restricted formula*, we mean a propositional formula that does not contain any negation symbols ($\neg$). It is trivial that a restricted formula can not fall into both categories 1 and 3 of the Unique Parsing Theorem. Then Exercise 2.2.6 establishes the theorem for restricted formulas. Finally, Exercise 2.2.8 establishes the full version.

We will occasionally need the notion of *subformula*. Informally, a subformula of a formula is a substring that, itself, is a formula. The following is a more rigorous characterization:

Definition 2.2.6 *Immediate subformulas* are defined as follows:

1. An atomic formula has no immediate subformulas
2. The only immediate subformula of $\neg X$ is X
3. For a binary symbol $\circ$, the immediate subformulas of $(X \circ Y)$ are X and Y.

Definition 2.2.7 Let X be a formula. The set of *subformulas* of X is the smallest set $\mathbf{S}$ that contains X and contains, with each member, the immediate subformulas of that member. X is called an *improper subformula* of itself.

Exercises

2.2.1. For a propositional formula X, let $b(X)$ be the number of occurrences of binary symbols in X. Prove using structural induction that, for every formula X, $b(X) =$ number of left parentheses in $X =$ number of right parentheses in X. (Hence every propositional formula has the same number of left as right parentheses.)

2.2.2. Using structural recursion, we define a function d on the set of propositional formulas as follows: If P is atomic, $d(P) = 0$. $d(\neg X) = d(X) + 1$. $d((X \circ Y)) = d(X) + d(Y) + 1$. For a propositional formula X, $d(X)$ is called the *degree* of X. Assuming $\supset$ and $\vee$ are binary symbols, calculate $d((\neg P \supset \neg(Q \vee R)))$.

2.2.3. Suppose F were a propositional formula that fell into none of the three categories stated in the Unique Parsing Theorem. Let $\mathbf{P}^\circ$ be the set $\mathbf{P}$ of propositional formulas, with F removed. Show that $\mathbf{P}^\circ$ still meets the conditions of the definition of propositional formula. Since $\mathbf{P}^\circ$ is a proper subset of $\mathbf{P}$, and $\mathbf{P}$ was the smallest set meeting the conditions, we have a contradiction.

2.2.4. Show that every proper initial segment of a restricted formula has more left than right parentheses. (Hint: Use Structural Induction and Exercise 2.2.1.)

2.2.5. Show that no proper initial segment of a restricted formula is itself a restricted formula. (Hint: Use Exercise 2.2.4.)

2.2.6. Using Exercise 2.2.5 show that, if a restricted formula is of the form $(X \circ Y)$ and also $(U \bullet V)$, where $\circ$ and $\bullet$ are binary symbols, then: $X = U$, $Y = V$, and $\circ = \bullet$.

2.2.7. Show the following:

1. No propositional formula consists entirely of negation symbols

2. A proper initial segment of a propositional formula is either a string of negation symbols or contains more left than right parentheses

3. No proper initial segment of a propositional formula is itself a propositional formula.

2.2.8. Using Exercise 2.2.7, prove the full version of the Unique Parsing Theorem.

2.2.9. Assume $\supset$, $\equiv$, and $\downarrow$ are binary symbols. Determine the set of subformulas of the following:

1. $((P \supset Q) \equiv (Q \downarrow \neg R))$.

2. $((\top \supset P) \supset Q)$.

2.3 Propositional Logic—Semantics

Classical logic is two-valued. We take as the space of *truth values* the set $\mathbf{Tr} = \{\mathbf{t}, \mathbf{f}\}$. Here $\mathbf{t}$ and $\mathbf{f}$ are simply two distinct objects. We will, of course, think of $\mathbf{t}$ as representing truth and $\mathbf{f}$ as representing falsehood. Next, we must say how to interpret each of the operation symbols from the previous section by an operation on $\mathbf{Tr}$. Negation is easy: From now on we assume we have a mapping $\neg : \mathbf{Tr} \to \mathbf{Tr}$, given by $\neg(\mathbf{t}) = \mathbf{f}$, and $\neg(\mathbf{f}) = \mathbf{t}$. (We will generally follow the custom of using the same notation for an operation symbol of the formal language defined in Section 2.2, and for an operation on $\mathbf{Tr}$. This should cause no confusion and will simplify terminology somewhat.)

Binary connectives are a more complicated affair. It is clear that there are 16 different two-place functions from $\mathbf{Tr}$ to itself. But not all of these are of interest. One of them is the trivial map that sends everything to $\mathbf{t}$. Another is the identically $\mathbf{f}$ map. (These correspond to our language constants $\top$ and $\bot$.) Ruling these out, we still have 14 maps. One of these is the map $f(x, y) = x$, and another is $g(x, y) = \neg x$, identity and negation with respect to the first input. Two more functions behave similarly with respect to the second input. Tossing these out, we are down to 10 binary operations. These 10 are all of genuine interest, though some play a more important role than others. For reasons that will be discussed later, we divide the 10 into two categories, *primary* and *secondary.* We give their definitions, and the symbols by which we will denote them, in Table 2.1.

Among the Primary Connectives, $\wedge$ (and), $\vee$ (or), and $\supset$ (implies) are probably the most familiar. $\subset$ is backward implication, which is used in Prolog, but for which the common Prolog notation is :- or $\longleftarrow$. $\not\supset$ and $\not\subset$ are simply $\supset$ and $\subset$ negated. $\uparrow$ is conveniently read "not both" or "NAND," while $\downarrow$ may be read "neither-nor" or "NOR." In fact, $\uparrow$ and

		Primary								Secondary	
		$\wedge$	$\vee$	$\supset$	$\subset$	$\uparrow$	$\downarrow$	$\not\supset$	$\not\subset$	$\equiv$	$\not\equiv$
t	t	t	t	t	t	f	f	f	f	t	f
t	f	f	t	f	t	t	f	t	f	f	t
f	t	f	t	t	f	t	f	f	t	f	t
f	f	f	f	t	t	t	t	f	f	t	f

TABLE 2.1. Primary and Secondary Connectives

$\downarrow$ are $\wedge$ and $\vee$ negated. Finally, for the Secondary Connectives, $\equiv$ (if and only if) is probably familiar, while $\not\equiv$ is $\equiv$ negated. In fact, $\not\equiv$ is the standard exclusive-or.

From now on we assume that for each of the Primary and Secondary Connectives we have a corresponding binary operation symbol in the formal language of propositional logic. To make the relationship between symbol and operation more memorable, we will overload our notation. From now on, $\wedge$ will be a binary operation *symbol* of our formal language and will also denote an *operation* on **Tr**, according to Table 2.1. This double usage should cause no difficulty, as context will make clear which is meant.

It is well-known that $\neg$ and $\wedge$ form a *complete set of connectives* in that all others can be defined using them. For instance, assuming x and y range over **Tr**, $(x \vee y) = \neg(\neg x \wedge \neg y)$ and $(x \supset y) = \neg(x \wedge \neg y)$. By all others, we mean exactly that: all two-place, three-place, ... operations on **Tr** can be defined from $\neg$ and $\wedge$ alone. Exercises 2.3.1 and 2.3.5 ask for a proof of this. Two of the binary connectives are complete by themselves. Exercise 2.3.2 asks you to show there are at least two; Exercise 2.3.4 asks you to show there at most two, which is harder.

Exercises

2.3.1. Show that all binary connectives can be defined using $\neg$ and any one of the Primary Connectives.

2.3.2. Show that all binary connectives can be defined using either $\uparrow$ or $\downarrow$ alone.

2.3.3. Show that no Primary Connective can be defined using $\neg$ and the Secondary Connectives.

2.3.4. Show that $\uparrow$ and $\downarrow$ are the only binary connectives that are complete by themselves.

2.3.5. Show that all three-place, four-place,... operations on **Tr** can be defined using $\neg$, $\wedge$, and $\vee$ (and hence using $\neg$ and any Primary Connective, by Exercise 2.3.1).

2.4 Boolean Valuations

Now we connect syntax and semantics. We are interested in those propositional formulas that must be true because of their logical structure, without reference to the meanings of the propositional letters involved. Such formulas are called *tautologies*; we give a proper definition later. Of course, we must say how an operation *symbol*, $\wedge$ say, is to be interpreted. We will interpret it as naming the operation on $\mathbf{Tr}$ that we designated by $\wedge$ in Table 2.1. More generally, as we remarked in Section 2.3, we follow the convention of using the same notation to refer to a binary operation on $\mathbf{Tr}$ and to be a binary operation symbol of our propositional language.

For reading ease, from now on we will generally omit the outer set of parentheses in a propositional formula, writing $X \vee \neg X$, for instance, instead of $(X \vee \neg X)$. This does not change the official definition.

Definition 2.4.1 A *Boolean valuation* is a mapping v from the set of propositional formulas to the set $\mathbf{Tr}$ meeting the conditions:

1. $v(\top) = \mathbf{t}$; $v(\bot) = \mathbf{f}$.
2. $v(\neg X) = \neg v(X)$.
3. $v(X \circ Y) = v(X) \circ v(Y)$, for any of the binary operations $\circ$ of Table 2.1.

It turns out Boolean valuations are easy to specify; the following two propositions provide a simple mechanism. We leave the proofs to you, as exercises.

Proposition 2.4.2 *For each mapping f from the set of propositional letters to the set* $\mathbf{Tr}$, *there is a Boolean valuation v that agrees with f on the propositional letters.*

Proposition 2.4.3 *If v_1 and v_2 are two Boolean valuations that agree on a set S of propositional letters, then v_1 and v_2 agree on every propositional formula that contains only propositional letters from S.*

Example Consider the Boolean valuation v such that on the propositional letters P, Q, and R, we have $v(P) = \mathbf{t}$, $v(Q) = \mathbf{f}$ and $v(R) = \mathbf{f}$, and to all other propositional letters v assigns $\mathbf{f}$. By Proposition 2.4.2, there is a Boolean valuation that meets these conditions, and by Proposition 2.4.3, there is exactly one. Now:

$$\begin{aligned}
v((P \uparrow \neg Q) \supset R) &= v(P \uparrow \neg Q) \supset v(R) \\
&= (v(P) \uparrow v(\neg Q)) \supset v(R) \\
&= (v(P) \uparrow \neg v(Q)) \supset v(R) \\
&= (\mathbf{t} \uparrow \neg \mathbf{f}) \supset \mathbf{f} \\
&= (\mathbf{t} \uparrow \mathbf{t}) \supset \mathbf{f} \\
&= \mathbf{f} \supset \mathbf{f} \\
&= \mathbf{t}.
\end{aligned}$$

We are interested in those propositional formulas that can't help being true. The following makes this rather vague notion precise.

Definition 2.4.4 A propositional formula X is a *tautology* if $v(X) = \mathbf{t}$ for every Boolean valuation v.

It is *decidable* whether or not a formula X is a tautology. We have to check the behavior of X under every Boolean valuation. By Proposition 2.4.2, a Boolean valuation is determined by its action on propositional letters, and by Proposition 2.4.3 we only need to consider this action on propositional letters that actually appear in X. If X contains n different propositional letters, we will need to check 2^n different cases. *Truth tables* provide a convenient technique for doing this. We do not describe truth tables here, as you are probably already familiar with them.

Definition 2.4.5 A set S of propositional formulas is *satisfiable* if some Boolean valuation v maps every member of S to $\mathbf{t}$.

To be a tautology, every line of a truth table must make a formula true. To be satisfiable, at least one line must do this. This is inexact, however, since formulas are tautologies, but *sets* of formulas are satisfiable. An infinite set can be satisfiable, but a direct truth table verification of this would not be possible. There is a useful and simple connection between the two notions: X is a tautology if and only if $\{\neg X\}$ is not satisfiable.

For classical propositional logic, there is an elegant notion of *duality* that can help give some insight into the structure of the logic.

Definition 2.4.6 Suppose $\circ$ and $\bullet$ are two binary operations on $\mathbf{Tr}$. We say $\bullet$ is the *dual* of $\circ$ if $\neg(x \circ y) = (\neg x \bullet \neg y)$. If $\bullet$ is dual to $\circ$, we also say the corresponding binary operation symbol $\bullet$ is dual to the binary symbol $\circ$.

For example, $\vee$ is the dual of $\wedge$ because $\neg(X \wedge Y) = (\neg X \vee \neg Y)$. Duality is symmetric, so we can simply refer to two binary connectives as duals of each other. We leave the proof of this to you, as well as the determination of which connectives are duals.

Definition 2.4.7 For a propositional formula X, we write X^d for the result of replacing every occurrence of $\top$ in X with an occurrence of $\bot$, and conversely, and replacing every occurrence of a binary symbol in X with an occurrence of its dual. We refer to X^d as the dual formula of X.

It is not hard to see that if $\bullet$ and $\circ$ are dual binary symbols then for any formulas X and Y, $(X \bullet Y)^d = (X^d \circ Y^d)$. Next we give exercises concerning this notion. As a special case of Exercise 2.4.12, for instance, it follows that $(P \wedge Q) \supset P$ is a tautology because $P \supset (P \vee Q)$ is.

Exercises

2.4.1. (To establish Proposition 2.4.2.) Show that if f is a mapping from the set of propositional letters to **Tr**, then f can be extended to a Boolean valuation v. (Hint: use Structural Recursion.)

2.4.2. (To establish Proposition 2.4.3.) Show that if v_1 and v_2 are two Boolean valuations that agree on a set S of propositional letters (which may not include all propositional letters), then v_1 and v_2 agree on all propositional formulas that contain only propositional letters from S. (Hint: use Structural Induction.)

2.4.3. Which of the following are, and which are not tautologies?

1. $((P \supset Q) \supset P) \supset P$
2. $((P \supset Q) \wedge (Q \supset R)) \vee ((R \supset Q) \wedge (Q \supset P))$
3. $(\neg(P \wedge Q) \wedge (P \equiv \bot)) \not\equiv (Q \equiv \bot)$
4. $((P \supset Q) \uparrow (P \supset Q)) \equiv (P \wedge \neg Q)$
5. $((P \supset Q) \wedge (Q \equiv \bot)) \supset (P \equiv \bot)$
6. $((P \supset Q) \wedge (Q \equiv \top)) \supset (P \equiv \top)$
7. $(P \not\equiv Q) \equiv (Q \not\equiv P)$
8. $(P \not\equiv (Q \not\equiv R)) \equiv ((P \not\equiv Q) \not\equiv R)$

2.4.4. Show that for any propositional formulas X and Y, and any Boolean valuation v:

1. $v(X \equiv Y) = \mathrm{t}$ if and only if $v(X) = v(Y)$.

2. $v(X \not\equiv Y) = \mathbf{t}$ if and only if $v(X) \neq v(Y)$.

3. $v(X \supset Y) = \mathbf{t}$ if and only if $(v(X) = \mathbf{f}$ or $v(Y) = \mathbf{t})$ if and only if $(v(X) = \mathbf{t}$ implies $v(Y) = \mathbf{t})$.

2.4.5. Let $\mathcal{F}$ be a family of Boolean valuations. We define a new Boolean valuation $\bigwedge \mathcal{F}$ by specifying it on propositional letters as follows: For a propositional letter P, $\bigwedge \mathcal{F}(P) = \mathbf{t}$ if and only if $v(P) = \mathbf{t}$ for every $v \in \mathcal{F}$. Show that for A_i and B propositional letters, if $(A_1 \wedge \ldots \wedge A_n) \supset B$ is true under every Boolean valuation in $\mathcal{F}$ then it is true under $\bigwedge \mathcal{F}$. (Propositional formulas of this form are called *propositional Horn clauses.* Loosely, this exercise says that the truth of propositional Horn clauses is preserved under the intersection of Boolean valuations.)

2.4.6. Suppose v_1, v_2 are Boolean valuations and every propositional letter that v_1 makes true, v_2 also makes true.

1. Prove that if X is any formula that contains only the connectives $\wedge$ and $\vee$, and if $v_1(X) = \mathbf{t}$, then $v_2(X) = \mathbf{t}$.

2. Must the converse be true? That is, if X contains only $\wedge$ and $\vee$, and $v_2(X) = \mathbf{t}$, must $v_1(X) = \mathbf{t}$? Justify your answer.

2.4.7. Suppose $F(A_1, \ldots, A_n)$ is a propositional formula whose propositional letters are among $A_1, \ldots, A_n$, and let $X_1, \ldots, X_n$ be n arbitrary propositional formulas. We denote by $F(X_1, \ldots, X_n)$ the result of simultaneously replacing all occurrences of A_i by X_i $(i = 1, \ldots, n)$ in $F(A_1, \ldots, A_n)$. (For example, if $F(A, B) = (A \wedge \top) \vee \neg B$, then $F(P \supset Q, P \vee R) = ((P \supset Q) \wedge \top) \vee \neg(P \vee R)$.) Show: if $F(A_1, \ldots, A_n)$ is a tautology, so is $F(X_1, \ldots, X_n)$.

2.4.8. Prove that if $\bullet$ is the dual connective of $\circ$, then $\circ$ is also the dual connective of $\bullet$.

2.4.9. For each primary and secondary connective, determine its dual.

2.4.10. For each propositional formula X, let $\overline{X}$ be the result of replacing every occurrence of a propositional letter P in X with $\neg P$. Prove that for a propositional formula X, if X is a tautology so is $\overline{X}$.

2.4.11. Using the notation of the previous exercise, prove that for every propositional formula X, and for every Boolean valuation v, $v(\neg\overline{X}) = v(X^d)$. (Hint: Structural Induction.)

2.4.12. Prove that if the propositional formula $X \supset Y$ is a tautology, so is $Y^d \supset X^d$.

2.5 The Replacement Theorem

Exercise 2.4.4 says that the binary connective $\equiv$ has a close relationship with equality. In this section we establish a result analogous to the familiar substitution property of equality that says one can substitute equals for equals. The result is fundamental, intuitive, and we generally use it without explicitly saying so.

We find notation like that used in Exercise 2.4.7 convenient. When we write $F(P)$, we mean F is a propositional formula, and occurrences of the propositional letter P play a special role. Then later, when we write $F(X)$ for some propositional formula X, we mean the formula that results from F when all occurrences of P are replaced by occurrences of the formula X. For example, if $F(P)$ is the propositional formula $(P \supset Q) \vee \neg P$, and if X is $(P \wedge R)$, then $F(X)$ is $((P \wedge R) \supset Q) \vee \neg(P \wedge R)$. Note that, as in this example, the replacement formula X may introduce fresh occurrences of the propositional letter P. We do not require that P actually have occurrences in $F(P)$, and this is an important point. $F(X)$ is the result of replacing all occurrences of P by occurrences of X, if there are any.

Theorem 2.5.1 **(Replacement Theorem, Version One)** *Let $F(P)$, X and Y be propositional formulas, and v be a Boolean valuation. If $v(X) = v(Y)$ then $v(F(X)) = v(F(Y))$.*

Proof Suppose we call a propositional formula $F(P)$ *good* if

$$v(X) = v(Y) \text{ implies } v(F(X)) = v(F(Y)) \text{ (for all } X, Y \text{ and } v). \qquad (*)$$

We must show that all propositional formulas are good, and we can use Structural Induction.

Basis Step Suppose $F(P)$ is atomic. Then there are two cases, either $F(P) = P$ or $F(P) \neq P$. If $F(P) = P$, then $F(X) = X$ and $F(Y) = Y$, and $(*)$ is trivially true. If $F(P) \neq P$, then $F(X) = F(Y) = F(P)$, and $(*)$ is again trivially true. Thus all atomic formulas are good.

Induction Step Suppose $G(P)$ and $H(P)$ are both good propositional formulas (induction hypothesis), and $F(P) = G(P) \circ H(P)$ where $\circ$ is some binary symbol; we show $F(P)$ is good. Well, if $v(X) = v(Y)$, then

$$\begin{aligned}
v(F(X)) &= v(G(X) \circ H(X)) && \text{(def. of } F\text{)}\\
&= v(G(X)) \circ v(H(X)) && \text{(def. of Boolean valuation)}\\
&= v(G(Y)) \circ v(H(Y)) && \text{(ind. hypothesis)}\\
&= v(G(Y) \circ H(Y)) && \text{(def. of Boolean valuation)}\\
&= v(F(Y)) && \text{(def. of } F\text{)}
\end{aligned}$$

Thus, in this case $F(P)$ is good.

The induction step involving negation is established similarly. Now it follows by the Principle of Structural Induction that all propositional formulas are good, and we are done. □

Theorem 2.5.2 **(Replacement Theorem, Version Two)** *If $X \equiv Y$ is a tautology, so is $F(X) \equiv F(Y)$.*

Proof Suppose $X \equiv Y$ is a tautology. Let v be an arbitrary Boolean valuation; we must show that $v(F(X) \equiv F(Y)) = \mathbf{t}$. Since $X \equiv Y$ is a tautology, $v(X \equiv Y) = \mathbf{t}$, so $v(X) = v(Y)$ by Exercise 2.4.4. Then $v(F(X)) = v(F(Y))$ by Theorem 2.5.1, hence $v(F(X) \equiv F(Y)) = \mathbf{t}$ by Exercise 2.4.4 again. □

Example Suppose we let $F(A)$ be the formula $[(\neg P \vee Q) \wedge (R \uparrow \neg A)]$. Since $(\neg P \vee Q) \equiv (P \supset Q)$ is a tautology, by Theorem 2.5.2 so is $F(\neg P \vee Q) \equiv F(P \supset Q)$, or $[(\neg P \vee Q) \wedge (R \uparrow (\neg P \vee Q))] \equiv [(\neg P \vee Q) \wedge (R \uparrow (P \supset Q))]$. Here is an informal but useful way of describing this: we have replaced *one occurrence* of $\neg P \vee Q$ by its equivalent, $P \supset Q$, in $(\neg P \vee Q) \wedge (R \uparrow (\neg P \vee Q))$ to turn it into its equivalent, $(\neg P \vee Q) \wedge (R \uparrow (P \supset Q))$.

We illustrate the uses of this theorem by showing every propositional formula can be put into *negation normal form.*

Definition 2.5.3 A propositional formula X is in *negation normal form* if the only negation symbols in X occur in front of propositional letters.

Proposition 2.5.4 *Every propositional formula can be put into a negation normal form. More precisely, there is an algorithm that will convert a propositional formula X into a propositional formula Y, where Y is in negation normal form and $X \equiv Y$ is a tautology.*

Since negation normal form will not play a major role in the development here, we do not give a formal proof of this theorem but only sketch the ideas. Say we wish to put X into negation normal form. If we are not already done, X must contain a subformula of the form $\neg Z$ where Z is not a propositional letter. Choose such a subformula and, using the Replacement Theorem, substitute for it an equivalent formula in which the negation symbols occur "further inside." For instance, if we have a subformula $\neg(U \wedge V)$ of X, we can replace it with $\neg U \vee \neg V$, since $\neg(U \wedge V) \equiv (\neg U \vee \neg V)$ is a tautology (recall that $\wedge$ and $\vee$ are duals). If we have $\neg\neg U$, we can replace it with U, since $\neg\neg U \equiv U$ is a tautology. If we have $\neg\bot$, we can replace it with $\top$, and so on.

Exercises

2.5.1. Put the following formulas into negation normal form:

1. $\neg(\neg(P \supset Q) \lor \neg R)$.
2. $\neg((P \uparrow Q) \equiv R)$.
3. $\neg(P \not\equiv Q) \land \neg(P \equiv R)$.

2.5.2. Put the propositional formulas of the previous exercise into a negation normal form in which the only binary connectives are $\land$ and $\lor$.

2.5.3. In propositional logic, assume all formulas are built up from propositional letters using only $\neg$ and $\lor$. Also assume P_0 is a fixed propositional letter. We define two functions v and w from triples of formulas to formulas as follows. First, on a propositional letter L:

$$v(X, Y, L) = \begin{cases} X & \text{if } L = P_0 \\ L & \text{otherwise} \end{cases}$$

$$w(X, Y, L) = \begin{cases} Y & \text{if } L = P_0 \\ L & \text{otherwise} \end{cases}$$

Then, on non-atomic formulas:

$$v(X, Y, \neg Z) = \neg w(X, Y, Z)$$
$$w(X, Y, \neg Z) = \neg v(X, Y, Z)$$
$$v(X, Y, Z \lor W) = v(X, Y, Z) \lor v(X, Y, W)$$
$$w(X, Y, Z \lor W) = w(X, Y, Z) \lor w(X, Y, W)$$

1. Let $Z = P_0 \lor \neg(Q \lor P_0)$ (where Q is a propositional letter different from P_0). Compute $w(X, Y, \neg Z) \lor w(X, Y, Z)$, and show it is a tautology, provided $\neg X \lor Y$ is.
2. Show, by structural induction on Z that, whenever $\neg X \lor Y$ is a tautology, so is $w(X, Y, \neg Z) \lor w(X, Y, Z)$.

2.5.4. It follows from Exercise 2.4.7 that, if $F(P)$ is a tautology, so are both $F(\top)$ and $F(\bot)$. Now show the converse: if both $F(\top)$ and $F(\bot)$ are tautologies, so is $F(P)$.

2.5.5. Devise a decision procedure for being a tautology, based on the previous exercise. Compare the result with the truth table method.

2.5.6. Let X be a formula that contains no occurrences of $\top$ or $\bot$ and no connectives except $\equiv$. Show that X is a tautology if and only if X contains an even number of occurrences of each propositional letter. Hint: First show that the connective $\equiv$ is associative and commutative and that $(Z \equiv (P \equiv P)) \equiv Z$ is a tautology.

2.6 Uniform Notation

Works on logic tend to adopt a small number of connectives as basic and take others as defined. Generally, this has the virtue of cutting down on the number of cases that must be considered when discussing techniques and methods. Exercise 2.3.1 is what makes it possible to proceed this way. But instead, we will use a device that is due to R. M. Smullyan [48], *uniform notation*, that allows us to have a very large set of basic connectives while still not having to do unnecessary work in proving theorems about our logic, or in writing automated theorem provers. The method works well for all the Primary Connectives, but not for the Secondary Connectives.

- From now on, $\equiv$ and $\not\equiv$ will be treated as defined, in terms of negation and the Primary Connectives.

Of course, we still have results like the Replacement Theorem 2.5.1, 2.5.2 available, even though $\equiv$ is a defined connective. For our basic binary connectives, we take all of the Primary Connectives.

We group all propositional formulas of the forms $(X \circ Y)$ and $\neg(X \circ Y)$ (where $\circ$ is a Primary Connective) into two categories, those that act *conjunctively*, which we call *α-formulas*, and those that act *disjunctively*, which we call *β-formulas.* For each α-formula we define two *components*, which we denote α_1 and α_2. Similarly, we define components β_1 and β_2 for each β-formula. This is done in Table 2.2.

Conjunctive			Disjunctive		
α	α_1	α_2	β	β_1	β_2
$X \wedge Y$	X	Y	$\neg(X \wedge Y)$	$\neg X$	$\neg Y$
$\neg(X \vee Y)$	$\neg X$	$\neg Y$	$X \vee Y$	X	Y
$\neg(X \supset Y)$	X	$\neg Y$	$X \supset Y$	$\neg X$	Y
$\neg(X \subset Y)$	$\neg X$	Y	$X \subset Y$	X	$\neg Y$
$\neg(X \uparrow Y)$	X	Y	$X \uparrow Y$	$\neg X$	$\neg Y$
$X \downarrow Y$	$\neg X$	$\neg Y$	$\neg(X \downarrow Y)$	X	Y
$X \not\supset Y$	X	$\neg Y$	$\neg(X \not\supset Y)$	$\neg X$	Y
$X \not\subset Y$	$\neg X$	Y	$\neg(X \not\subset Y)$	X	$\neg Y$

TABLE 2.2. α- and β-Formulas and Components

The following accounts for what we said about α-formulas being conjunctive and β-formulas being disjunctive:

Proposition 2.6.1 *For every Boolean valuation v, and for all α- and β-formulas:*

$$v(\alpha) = v(\alpha_1) \wedge v(\alpha_2)$$
$$v(\beta) = v(\beta_1) \vee v(\beta_2)$$

Corollary 2.6.2 *For every α and β, $\alpha \equiv (\alpha_1 \wedge \alpha_2)$ and $\beta \equiv (\beta_1 \vee \beta_2)$ are tautologies.*

Next, we have an alternative to the original version of structural induction, Theorem 2.2.3. We use the first version to establish this one, and it is this that will be of primary use from now on.

Theorem 2.6.3 **(Principle of Structural Induction)** *Every formula of propositional logic has a property, $\mathbf{Q}$, provided:*

> **Basis step** *Every atomic formula and its negation has property $\mathbf{Q}$.*
>
> **Induction steps**
>
> *If X has property $\mathbf{Q}$, so does $\neg\neg X$.*
>
> *If α_1 and α_2 have property $\mathbf{Q}$, so does α.*
>
> *If β_1 and β_2 have property $\mathbf{Q}$, so does β.*

Proof Suppose $\mathbf{Q}$ is a property that meets these conditions. Say we call a propositional formula X *good*, provided both X and $\neg X$ have property $\mathbf{Q}$. If we show every propositional formula is good, it will follow that every formula has property $\mathbf{Q}$. And we can use the original Principle of Structural Induction from Section 2.2 to show this.

If X is atomic, X is good by the basis step.

Next we show that the set of good formulas is closed under negation. Suppose X is good. Then both X and $\neg X$ have property $\mathbf{Q}$. Since X has property $\mathbf{Q}$, by the first of the induction steps, $\neg\neg X$ also has property $\mathbf{Q}$. Thus both $\neg X$ and $\neg\neg X$ have property $\mathbf{Q}$, so $\neg X$ is good.

Suppose X and Y are good, and $\circ$ is a Primary Connective; we show $(X \circ Y)$ is good. If $(X \circ Y)$ is an α-formula, $\neg(X \circ Y)$ is a β-formula; if $(X \circ Y)$ is a β-formula, $\neg(X \circ Y)$ is an α. In either case, both α_1 and β_1 are among X, $\neg X$. X is good by hypothesis, hence $\neg X$ is also good by the preceding paragraph. Then both X and $\neg X$ have property $\mathbf{Q}$; in other words, both α_1 and β_1 have property $\mathbf{Q}$. Both α_2 and β_2 are among Y, $\neg Y$, and by a similar argument both have property $\mathbf{Q}$. Then, by the last two of the induction steps, both α and β have property $\mathbf{Q}$, and it follows that $(X \circ Y)$ is good.

Now, by the original Principle of Structural Induction, Theorem 2.2.3, every propositional formula is good, hence every propositional formula has property $\mathbf{Q}$, and we are done. □

There is also a version of Structural Recursion that uses uniform notation. We state it, but omit the proof.

Theorem 2.6.4 **(Principle of Structural Recursion)** *There is one, and only one, function f defined on the set* $\mathbf{P}$ *of propositional formulas such that:*

Basis step *The value of f is specified explicitly on atomic formulas and their negations.*

Recursion steps

The value of f on $\neg\neg X$ is specified in terms of the value of f on X.

The value of f on α is specified in terms of the values of f on α_1 and α_2.

The value of f on β is specified in terms of the values of f on β_1 and β_2.

In Exercise 2.2.2, we defined the notion of degree. Now we define a notion we call *rank*, which relates better to uniform notation than degree does. The theorem just stated is needed here.

Definition 2.6.5 The function r on the set $\mathbf{P}$ of propositional formulas is defined as follows: First, for atomic formulas and their negations. If A is a propositional letter, $r(A) = r(\neg A) = 0$. $r(\top) = r(\bot) = 0$. $r(\neg\top) = r(\neg\bot) = 1$. Next, the recursion steps: $r(\neg\neg Z) = r(Z) + 1$; $r(\alpha) = r(\alpha_1) + r(\alpha_2) + 1$; $r(\beta) = r(\beta_1) + r(\beta_2) + 1$. The *rank* of a formula X is the number $r(X)$.

Exercises

2.6.1. Compute the degree and the rank of the following:

1. $(P \supset Q) \supset (Q \downarrow \neg R)$.
2. $(\top \supset P) \supset Q$.
3. $Q \supset (\top \supset P)$.

2.6.2. Suppose we define the *depth* $h(X)$ of a propositional formula X as follows: If A is a propositional letter, $h(A) = h(\neg A) = 0$. $h(\top) = h(\bot) = 0$. $h(\neg\top) = h(\neg\bot) = 1$. And, using recursion, $h(\neg\neg Z) = h(Z) + 1$; $h(\alpha) = \max\{h(\alpha_1), h(\alpha_2)\} + 1$; $h(\beta) = \max\{h(\beta_1), h(\beta_2)\} + 1$.

1. Prove that for any propositional formula X, $r(X) \leq 2^{h(X)} - 1$.
2. Prove that for every non-negative integer n there is a formula X of depth n whose rank is $2^n - 1$.

2.6.3. Prove Proposition 2.6.1.

2.7 König's Lemma

We digress from formal logic briefly to present a result about trees that will be of fundamental use throughout this work. Its statement and proof are easy, but the consequences are profound. We do not give a definition of tree here; we assume you are familiar with the notion already. All our trees will be rooted; we will not say so each time. We do not require a tree to be finite.

Definition 2.7.1 A tree is *finitely branching* if each node has a finite number of children (possibly 0). A tree is *finite* if it has a finite number of nodes; otherwise it is *infinite*. Likewise, a branch is *finite* if it has a finite number of nodes; otherwise it is *infinite*.

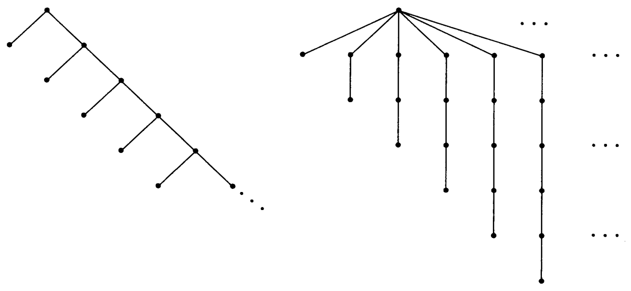


FIGURE 2.1. Examples of Trees

Examples of trees are given in Figure 2.1. In each case the displayed pattern is assumed to continue. The tree on the left is finitely branching but infinite. It has an infinite branch. The tree on the right is not finitely branching, every branch is finite, but branches are arbitrarily long. König's Lemma says that if we don't allow infinite branching, the left-hand example shows the only way a tree can be infinite.

Theorem 2.7.2 **(König's Lemma)** *A tree that is finitely branching but infinite must have an infinite branch.*

Proof Suppose **T** is a finitely branching, infinite tree. We will show **T** has an infinite branch. By a descendant of a node, we mean a child of it, or a child of a child, or a child of a child of a child, etc. Let us (temporarily) call a node of **T** *good* if it has infinitely many descendants.

The root node of **T** is good, since every other node of **T** is its descendant, and **T** is infinite.

Suppose N is a good node of **T**. Since **T** is finitely branching, N has a finite set of children; say $C_1, \ldots, C_n$. If child C_i were not good, it would have a finite number of descendants, say c_i. If none of $C_1, \ldots, C_n$ were good, N would have $c_1 + \cdots + c_n + n$ descendants, contradicting the assumption that N had infinitely many descendants. Consequently, some C_i must be good. In brief, we have showed that a good node must have a good child.

Now, the root node of $\mathbf{T}$ is good. It has a good child; pick one. It, in turn, has a good child; pick one. And so on. Since the process never stops, in this way we trace out an infinite branch of $\mathbf{T}$. □

The following is an amusing application of König's Lemma (see Smullyan [49]). We are going to play a game involving balls labeled with positive integers. There is no restriction on how many balls can have the same number or on how big the numbers can be. Suppose, at the start, we have a box containing just one ball. We are allowed to remove it and replace it with as many balls as we please, all having lower numbers. And this is also what we do at each stage of the game: Select a ball in the box, remove it, and replace it with balls having lower numbers. For instance, we could remove a ball labeled 27 and replace it with 1,000,000 26s. The problem is to show that, no matter how we proceed, eventually we must remove all balls from the box.

Exercises

2.7.1. Use König's Lemma to show we must empty the box out. (Hint: Suppose we create a tree as follows: The nodes are all the balls that have ever been in the box. The root node is the ball that is in the box at the start. The children of a ball are the balls we replace it with.)

2.8 Normal Forms

If propositional formulas are standardized, it is sometimes easier to establish their tautologyhood or their satisfiability. In fact, most automated theorem-proving techniques convert formulas into some kind of normal form as a first step. We have already met one normal form, negation normal form, but it does not have enough structure for many purposes. In this section we introduce two more normal forms, both built on the following idea: Since the behavior of all Primary Connectives can be described in conjunctive/disjunctive terms, we will eliminate all other binary connectives in favor of just conjunctions and disjunctions. The conjunction operation $\land$ is commutative and associative. Consequently, in a conjunction of many items, placement of parentheses and order does not really matter and similarly for disjunction. So, we begin by introducing generalized conjunction and disjunction notation.

Definition 2.8.1 Let $X_1, X_2, \ldots, X_n$ be a list of propositional formulas (possibly empty). We define two new types of propositional formulas as follows:

> $[X_1, X_2, \dots, X_n]$ is the *generalized disjunction* of X_1, X_2,…, X_n,
>
> $\langle X_1, X_2, \dots, X_n \rangle$ is the *generalized conjunction* of X_1, X_2, …, X_n.

If v is a Boolean valuation, then in addition to the original conditions, we also require

> $v([X_1, X_2, \dots, X_n]) = \mathbf{t}$ if v maps some member of the list X_1, X_2,…, X_n to $\mathbf{t}$, $v([X_1, X_2, \dots, X_n]) = \mathbf{f}$ otherwise;
>
> $v(\langle X_1, X_2, \dots, X_n \rangle) = \mathbf{t}$ if v maps every member of the list X_1, X_2,…, X_n to $\mathbf{t}$, $v(\langle X_1, X_2, \dots, X_n \rangle) = \mathbf{f}$ otherwise.

The case of the empty list is a little tricky. For every Boolean valuation v, $v([\,]) = \mathbf{f}$ because, for v to map $[\,]$ to $\mathbf{t}$, it must map some member of $[\,]$ to $\mathbf{t}$, and there aren't any members. Consequently, $[\,] \equiv \bot$ is a tautology. Similarly, $\langle\,\rangle \equiv \top$ is a tautology. One- and two-member lists are more straightforward: $[X] \equiv X$ and $\langle X \rangle \equiv X$ are both tautologies, as are $[X, Y] \equiv X \vee Y$ and $\langle X, Y \rangle \equiv X \wedge Y$.

To keep special terminology at a minimum, we will use the term *conjunction* to mean either a binary conjunction, as in earlier sections, or a generalized conjunction. Context will generally make our meaning clear; if it does not, we will say so. We reserve the term *ordinary formula* to mean one without generalized conjunctions and disjunctions (and, for that matter, not containing secondary connectives, except as abbreviations). The **Replacement Theorems**, 2.5.1 and 2.5.2, extend to include generalized conjunctions and disjunctions. We make use of this, but omit the proof. The essential items needed are left to you as an exercise.

Resolution theorem proving is built around the notion of a clause. We are about to define this. At the same time we define what we call a *dual clause*. This terminology is not standard, but it will play for semantic tableaux the role that clauses play for resolution.

Definition 2.8.2 A *literal* is a propositional letter or the negation of a propositional letter, or a constant, $\top$ or $\bot$.

Definition 2.8.3 A *clause* is a disjunction $[X_1, X_2, \dots, X_n]$ in which each member is a literal. A *dual clause* is a conjunction $\langle X_1, X_2, \dots, X_n \rangle$ in which each member is a literal. A propositional formula is in *conjunctive normal form* or is in *clause form* or is a *clause set* if it is a conjunction $\langle C_1, C_2, \dots, C_n \rangle$ in which each member is a clause. A propositional formula is in *disjunctive normal form* or is in *dual clause form* or is a *dual clause set* if it is a disjunction $[D_1, D_2, \dots, D_n]$ in which each member is a dual clause.

For example, $\langle [P, \neg Q], [\top, R, \neg S], [\,] \rangle$ is in clause form. Under a Boolean valuation, it will be treated as a conjunction, each term of which is a disjunction of literals, and similarly for dual clause form. Notice, by the way, that all conjunctive and all disjunctive normal form formulas are also in negation normal form.

Theorem 2.8.4 **(Normal Form)** *There are algorithms for converting an ordinary propositional formula into clause form and into dual clause form.*

The proof of this will, more or less, occupy the rest of the section. We begin with the algorithm for converting a formula into clause, or conjunctive normal form. Say the ordinary formula to be converted is X. We describe a sequence of steps, each of which produces a conjunction, whose members are disjunctions. The last step of the sequence is a conjunctive normal form. The algorithm is non-deterministic; at each stage there are choices to be made of what to do next. We will show the algorithm works no matter what choices are made.

Step 1. Start with $\langle [X] \rangle$.

Now, having completed step n, producing $\langle D_1, D_2, \ldots, D_k \rangle$, where the members D_i are disjunctions, if we do not yet have conjunctive normal form, go on to the following:

Step $n+1$. Select a member, D_i, which contains some non-literal; select a non-literal member, say N, and:

If N is $\neg\top$ replace N with $\bot$.
If N is $\neg\bot$ replace N with $\top$.
If N is $\neg\neg Z$ replace N with Z.
If N is a β-formula, replace N with the two formula sequence β_1, β_2.
If N is an α-formula, replace the *disjunction* D_i with two disjunctions, one like D_i but with α replaced by α_1 and one like D_i but with α replaced by α_2.

This constitutes a complete description of the algorithm. We can present it somewhat more perspicuously if we first introduce some reduction rules. We give these in Table 2.3.

These rules are to be thought of as rewrite rules for conjunctions of disjunctions. More specifically, if S is a conjunction, with a member D that is a disjunction, and if N is a member of D, then if N is $\neg\neg Z$, N may be replaced with Z, and similarly for the cases where N is $\neg\top$ and $\neg\bot$. If N is β, N may be replaced with the two formulas β_1 and β_2. Finally, if N is α, then D itself may be replaced with the two disjunctions

$$\begin{array}{ccccc} \dfrac{\neg\neg Z}{Z} & \dfrac{\neg\top}{\bot} & \dfrac{\neg\bot}{\top} & \dfrac{\beta}{\begin{array}{c}\beta_1\\ \beta_2\end{array}} & \dfrac{\alpha}{\alpha_1 \mid \alpha_2} \end{array}$$

TABLE 2.3. Clause Set Reduction Rules

D_1 and D_2, where D_1 is like D except that α has been replaced with α_1, and D_2 is like D but with α replaced with α_2. Now, the key fact concerning these rules can be easily stated and verified.

Lemma 2.8.5 **(Conjunctive Rewrite)** *If S is a conjunction of disjunctions, and one of the clause set reduction rules is applied to S, producing S^*, then $S \equiv S^*$ is a tautology.*

Proof This follows easily from Exercise 2.8.1, the fact that $\alpha \equiv (\alpha_1 \wedge \alpha_2)$ and $\beta \equiv (\beta_1 \vee \beta_2)$ are tautologies, and the Replacement Theorem. □

Using the Clause Set Reduction Rules, the algorithm given previously can be stated in a simpler schematic form.

Clause Form Algorithm To convert the ordinary propositional formula X to clause form,

```
Let $S$ be $\langle [X] \rangle$
While some member of $S$ contains a non-literal do
    select a member $D$ of $S$ containing a non-literal
    select a non-literal $N$ of $D$
    apply the appropriate clause set reduction rule
        to $N$ in $D$ producing a new $S$
    end.
```

Example The following is a conversion of

$$(P \supset (Q \supset R)) \supset ((P \supset Q) \supset (P \supset R))$$

into clause form using the Clause Form Algorithm. Make sure you can justify the steps. Note that there is considerable redundancy in the final result. This can be eliminated, but we do not do so now.

1. $\langle [(P \supset (Q \supset R)) \supset ((P \supset Q) \supset (P \supset R))] \rangle$

2. $\langle[\neg(P \supset (Q \supset R)), ((P \supset Q) \supset (P \supset R))]\rangle$
3. $\langle[\neg(P \supset (Q \supset R)), \neg(P \supset Q), (P \supset R)]\rangle$
4. $\langle[\neg(P \supset (Q \supset R)), \neg(P \supset Q), \neg P, R]\rangle$
5. $\langle[P, \neg(P \supset Q), \neg P, R], [\neg(Q \supset R), \neg(P \supset Q), \neg P, R]\rangle$
6. $\langle[P, P, \neg P, R], [P, \neg Q, \neg P, R], [\neg(Q \supset R), \neg(P \supset Q), \neg P, R]\rangle$
7. $\langle[P, P, \neg P, R], [P, \neg Q, \neg P, R], [Q, \neg(P \supset Q), \neg P, R],$
   $[\neg R, \neg(P \supset Q), \neg P, R]\rangle$
8. $\langle[P, P, \neg P, R], [P, \neg Q, \neg P, R], [Q, P, \neg P, R], [Q, \neg Q, \neg P, R],$
   $[\neg R, \neg(P \supset Q), \neg P, R]\rangle$
9. $\langle[P, P, \neg P, R], [P, \neg Q, \neg P, R], [Q, P, \neg P, R], [Q, \neg Q, \neg P, R],$
   $[\neg R, P, \neg P, R], [\neg R, \neg Q, \neg P, R]\rangle$

Merely giving an algorithm is not enough; we must verify that it works the way we think it should. Specifically, we must consider *correctness* and *termination.* Correctness says that if the algorithm halts, then it halts with the right answer. The termination question is, Under what circumstances will the algorithm halt? We will have to deal with such issues frequently, so we take some extra space now to introduce general methods that have been developed in the area of program verification.

Informally, a *loop invariant* for a while-loop W is an assertion A having the property that, if it is true, and the body of the loop W is executed, then A is still true. Since the execution of the loop body can alter values of variables, the 'meaning' of A afterward may not be the same as before. For A to be a loop invariant, it is the truth of A that must be preserved. Here is a useful form of mathematical induction, specially tailored for program verification.

Induction Principle for While Loops *If A is true at the first entry into a while loop, and if A is a loop invariant for that while loop, then A will be true when the loop terminates (if ever).* ▪

Say we execute the Clause Form Algorithm, starting with the propositional formula X. Let A be the assertion: S is a conjunction, whose members are disjunctions, and $S \equiv X$ is a tautology. The algorithm makes an assignment of an initial value to S of $\langle[X]\rangle$, and this trivially makes assertion A true at the first entry into the while loop. It follows from the Conjunctive Rewrite Lemma that A is a loop invariant. Then A must be true at termination as well. But the termination condition is that only literals are present. It follows that at termination S must be in conjunctive normal form, and equivalent to X; that is, correctness of the Clause Form Algorithm is established.

The Clause Form Algorithm permits choices to be made in the while-loop body, and hence there may be several different ways of executing it. In other words, it has a *non-deterministic* character. What we mean by termination for a non-deterministic algorithm must be carefully specified. One possibility is that at least one way of executing the algorithm must terminate. This is what is generally thought of when non-deterministic algorithms are considered. We will refer to this as *weak termination.* One reason non-deterministic algorithms are viewed with some disfavor is that they often only weakly terminate: We have termination if the right choices are made at each step, otherwise not. First-order theorem-proving algorithms are generally of this nature, which is one reason why the subject is so difficult. But in the present case, we can do much better: No matter what choices are made in executing the Clause Form Algorithm, it must terminate. We will refer to this as *strong termination.* Strong termination allows us to impose a wide variety of heuristics on what choices to make, without affecting termination. It is a very nice state of affairs, indeed.

Our proof of strong termination is based on König's Lemma. In fact, the application of König's Lemma at the end of Section 2.7 to show termination of the game involving numbered balls is exactly what we need here. In Definition 2.6.5 we defined the rank of an ordinary propositional formula. Suppose we extend this and define the rank of a generalized disjunction of propositional formulas to be the sum of the ranks of the individual propositional formulas. Now associate with each generalized conjunction S whose members are generalized disjunctions a box B of numbered balls in the following simple way. For each generalized disjunction D in S, if the rank of D is positive, put a ball in B labeled with the rank of D. We leave it to you to check that each pass through the loop in the Clause Form Algorithm corresponds exactly to taking a ball from the box B associated with S and replacing it with either two balls with lower numbers (in the α-case) or with one ball with a lower number (in all other cases). By Exercise 2.7.1, no matter how we proceed, eventually the box must empty out. The only way this can happen is if S has been converted into a generalized conjunction all of whose members have rank 0, which means only literals are present. But this is the termination condition for the loop.

We have now finished the proof of the conjunctive half of the Normal Form Theorem. With this behind us, the disjunctive half can be dealt with more swiftly. We begin with a new set of reduction rules, given in Table 2.4, suitable for converting formulas into dual clause form. Essentially, α and β have switched roles from the Clause Set Reduction Rules.

$$\begin{array}{ccccc} \dfrac{\neg\neg Z}{Z} & \dfrac{\neg\top}{\bot} & \dfrac{\neg\bot}{\top} & \dfrac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} & \dfrac{\beta}{\beta_1 \mid \beta_2} \end{array}$$

TABLE 2.4. Dual Clause Set Reduction Rules

This time the rules are to be thought of as rewrite rules for disjunctions of conjunctions. If S is a disjunction, with a member C, which is a conjunction, and if N is a member of C, then if N is $\neg\neg Z$, N may be replaced with Z, and similarly for the cases where N is $\neg\top$ and $\neg\bot$. If N is α, N may be replaced with the two formulas α_1 and α_2. And if N is β, then C itself may be replaced with the two conjunctions C_1 and C_2, where C_1 is like C except that β has been replaced with β_1, and C_2 is like C but with β replaced with β_2.

Lemma 2.8.6 **(Disjunctive Rewrite)** *If S is a disjunction of conjunctions, and one of the Dual Clause Set Reduction Rules is applied to S, producing S^*, then $S \equiv S^*$ is a tautology.*

Dual Clause Form Algorithm To convert the ordinary propositional formula X to dual clause form,

```
Let $S$ be $[\langle X\rangle]$
While some member of $S$ contains a non-literal do
    select a member $C$ of $S$ containing a non-literal
    select a non-literal $N$ of $C$
    apply the appropriate dual clause set reduction rule
        to $N$ producing a new $S$
    end.
```

Exercises

2.8.1. Show that the following are tautologies (n or k may be 0):

1. $[A_1, \ldots, A_n, (U \vee V), B_1, \ldots, B_k] \equiv [A_1, \ldots, A_n, U, V, B_1, \ldots, B_k]$.
2. $[A_1, \ldots, A_n, (U \wedge V), B_1, \ldots, B_k] \equiv$ $([A_1, \ldots, A_n, U, B_1, \ldots, B_k] \wedge [A_1, \ldots, A_n, V, B_1, \ldots, B_k])$.
3. $\langle A_1, \ldots, A_n, (U \wedge V), B_1, \ldots, B_k\rangle \equiv \langle A_1, \ldots, A_n, U, V, B_1, \ldots, B_k\rangle$.

4. $\langle A_1, \ldots, A_n, (U \vee V), B_1, \ldots, B_k\rangle \equiv$
$(\langle A_1, \ldots, A_n, U, B_1, \ldots, B_k\rangle \vee \langle A_1, \ldots, A_n, V, B_1, \ldots, B_k\rangle)$.

2.8.2. The duality principle extends. Show the following are tautologies:

1. $\neg[X_1, \ldots, X_n] \equiv \langle \neg X_1, \ldots, \neg X_n\rangle$.
2. $\neg\langle X_1, \ldots, X_n\rangle \equiv [\neg X_1, \ldots, \neg X_n]$.

2.8.3. Show that if $X \equiv Y$ is a tautology, so are

1. $[A_1, \ldots, A_n, X, B_1, \ldots, B_k] \equiv [A_1, \ldots, A_n, Y, B_1, \ldots, B_k]$.
2. $\langle A_1, \ldots, A_n, X, B_1, \ldots, B_k\rangle \equiv \langle A_1, \ldots, A_n, Y, B_1, \ldots, B_k\rangle$.

2.8.4. Apply the Clause Form Algorithm and convert the following into clause form:

1. $P \vee \neg P$.
2. $P \wedge \neg P$.
3. $(P \downarrow Q) \uparrow \neg(P \downarrow Q)$.
4. $\neg((P \supset (Q \supset R)) \supset ((P \supset Q) \supset (P \supset R)))$.
5. $(P \subset (Q \uparrow R)) \downarrow \neg P$.

2.8.5. Prove Lemma 2.8.6.

2.8.6. Apply the Dual Clause Form Algorithm to convert the formulas of Exercise 2.8.4.

2.8.7. Establish the correctness and strong termination of the Dual Clause Form Algorithm.

2.9 Normal Form Implementations

Now that we have algorithms for converting formulas into clause and dual clause forms, we want to embody these algorithms in programs. We have chosen the language Prolog for this job because it lets us get to the heart of the matter with a minimum of preliminary detail. We do not discuss the general ideas of programming in Prolog; several readily available books do this. A recommended standard here is provided by Clocksin and Mellish [10]; another more advanced work is Sterling and Shapiro [51].

Prolog allows one to define 'operators.' Essentially, this is a matter of convenient syntax. For example, we can specify that the name and will always be used as an infix binary connective, and hence p and q will be meaningful. Prolog also allows one to specify an order of precedence and whether infix operators will be left or right associative. In the implementation that follows we specify neg to be a prefix operator, which we use to stand for $\neg$ (we do not use not, because this already has a meaning in most Prologs). For the Primary Connectives, we use and for $\wedge$, or for $\vee$, imp for $\supset$, revimp for $\subset$, uparrow for $\uparrow$, downarrow for $\downarrow$, notimp for $\not\supset$, and notrevimp for $\not\subset$. We assume all occurrences of $\equiv$ and $\not\equiv$ have been translated away. neg is defined to have a precedence of 140. The Primary Connectives are all defined to be infix, right associative, and are given precedences of 160. All this is done at the beginning of the program, using the op predicate. The precedence numbers selected may need to be changed for different Prolog implementations, and the syntax appropriate for the op predicate varies somewhat too. Apart from this, the program should work as written in any 'Edinburgh syntax' Prolog.

Before giving the Prolog program, a word about its abstract structure may help. Suppose we call a directed graph *well-founded* if, for each node a, every path starting at a is finite. Starting at any node in such a graph, at a, say, there is at least one maximal path, which begins at a and ends at a node with only incoming edges, and we can find such a path by simply following edges until we can't do so any further. Now, suppose we have clauses for a Prolog predicate that generates the edges of a well-founded, directed graph; say the predicate edge(X,Y) is true exactly when X and Y are nodes and there is an edge from X to Y. Then there is a simple, and standard, way of writing a graph search program to find maximal paths. We just add the following clauses:

```
path(X, Y) :- edge(X, Z), path(Z, Y).
path(X, X).
```

Now consider the following graph: The nodes are disjunctions of conjunctions. There is an edge from node One to node Two if the application of a single Dual Clause Set Reduction Rule will turn One into Two. (This corresponds to a single pass through the while loop of the Dual Clause

Form Algorithm.) The Dual Clause Form Algorithm is correct, and this amounts to saying that if we start at a node and follow edges until we reach a node with only incoming edges, that node will be a dual clause form of the node at which we started. Also, the Dual Clause Form Algorithm strongly terminates, and this amounts to saying that the graph we have created is well-founded.

Now, the Prolog program is easily described. The `singlestep` predicate generates edges in the graph just described. We want to find maximal paths. The predicate that corresponds to `path` is now called `expand`, for obvious reasons. Finally, there is a 'driver' predicate, `dualclauseform`, that simply takes an ordinary formula X, turns it into the equivalent of $[\langle X\rangle]$, and calls on the `expand` predicate. For convenience both generalized conjunctions and generalized disjunctions are represented as lists; context can easily determine which is meant. So, finally, to use the program for converting a formula `x` to dual clause form, enter the query `dualclauseform(x,V)`, where `V` is a Prolog variable. Prolog will return a value for `V` that is a list of the dual clauses making up the dual clause form.

```
/*  Dual Clause Form Program

  Propositional operators are: neg, and, or, imp, revimp,
    uparrow, downarrow, notimp and notrevimp.
*/

?-op(140, fy, neg).
?-op(160, xfy, [and, or, imp, revimp, uparrow, downarrow,
    notimp, notrevimp]).

/*  member(Item, List) :- Item occurs in List.
*/

member(X, [X | _]).
member(X, [_ | Tail]) :- member(X, Tail).

/*  remove(Item, List, Newlist) :-
      Newlist is the result of removing all occurrences of
      Item from List.
*/

remove(X, [ ], [ ]).
remove(X, [X | Tail], Newtail) :-
  remove(X, Tail, Newtail).
remove(X, [Head | Tail], [Head | Newtail]) :-
```

```
  remove(X, Tail, Newtail).

/*  conjunctive(X) :- X is an alpha formula.
*/

conjunctive(_ and _).
conjunctive(neg(_ or _)).
conjunctive(neg(_ imp _)).
conjunctive(neg(_ revimp _)).
conjunctive(neg(_ uparrow _)).
conjunctive(_ downarrow _).
conjunctive(_ notimp _).
conjunctive(_ notrevimp _).

/*  disjunctive(X) :- X is a beta formula.
*/

disjunctive(neg(_ and _)).
disjunctive(_ or _).
disjunctive(_ imp _).
disjunctive(_ revimp _).
disjunctive(_ uparrow _).
disjunctive(neg(_ downarrow _)).
disjunctive(neg(_ notimp _)).
disjunctive(neg(_ notrevimp _)).

/*  unary(X) :- X is a double negation,
      or a negated constant.
*/

unary(neg neg _).
unary(neg true).
unary(neg false).

/*  components(X, Y, Z) :- Y and Z are the components
      of the formula X, as defined in the alpha and
      beta table.
*/

components(X and Y, X, Y).
components(neg(X and Y), neg X, neg Y).
components(X or Y, X, Y).
components(neg(X or Y), neg X, neg Y).
components(X imp Y, neg X, Y).
components(neg(X imp Y), X, neg Y).
components(X revimp Y, X, neg Y).
components(neg(X revimp Y), neg X, Y).
components(X uparrow Y, neg X, neg Y).
components(neg(X uparrow Y), X, Y).
components(X downarrow Y, neg X, neg Y).
components(neg(X downarrow Y), X, Y).
components(X notimp Y, X, neg Y).
components(neg(X notimp Y), neg X, Y).
components(X notrevimp Y, neg X, Y).
components(neg(X notrevimp Y), X, neg Y).

/*  component(X, Y) :- Y is the component of the
      unary formula X.
*/

component(neg neg X, X).
component(neg true, false).
component(neg false, true).

/*  singlestep(Old, New) :-  New is the result of applying
      a single step of the expansion process to Old, which
      is a generalized disjunction of generalized
      conjunctions.
*/

singlestep([Conjunction | Rest], New) :-
  member(Formula, Conjunction),
  unary(Formula),
  component(Formula, Newformula),
  remove(Formula, Conjunction, Temporary),
  Newconjunction = [Newformula | Temporary],
  New = [Newconjunction | Rest].

singlestep([Conjunction | Rest], New) :-
  member(Alpha, Conjunction),
  conjunctive(Alpha),
  components(Alpha, Alphaone, Alphatwo),
  remove(Alpha, Conjunction, Temporary),
  Newcon = [Alphaone, Alphatwo | Temporary],
  New = [Newcon | Rest].

singlestep([Conjunction | Rest], New) :-
  member(Beta, Conjunction),
  disjunctive(Beta),
  components(Beta, Betaone, Betatwo),
  remove(Beta, Conjunction, Temporary),
  Newconone = [Betaone | Temporary],
  Newcontwo = [Betatwo | Temporary],
  New = [Newconone, Newcontwo | Rest].

singlestep([Conjunction|Rest], [Conjunction|Newrest]) :-
  singlestep(Rest, Newrest).

/*  expand(Old, New) :- New is the result of applying
      singlestep as many times as possible, starting
      with Old.
*/

expand(Dis, Newdis) :-
  singlestep(Dis, Temp),
  expand(Temp, Newdis).

expand(Dis, Dis).

/*  dualclauseform(X, Y) :- Y is the dual clause form of X.
*/

dualclauseform(X, Y) :- expand([[X]], Y).
```

Exercises

2.9.1$^{\mathbf{P}}$. Write a Prolog program for converting a propositional formula into clause form.

2.9.2$^{\mathbf{P}}$. Write a Prolog program implementing the algorithm for converting a propositional formula into negation normal form that was presented at the end of Section 2.5.

2.9.3$^{\mathbf{P}}$. Write a Prolog program for translating away occurrences of $\equiv$ and $\not\equiv$.

2.9.4$^{\mathbf{P}}$. Write a Prolog program implementing the degree function defined in Exercise 2.2.2 and another implementing the rank function defined in Definition 2.6.5.

3 Semantic Tableaux and Resolution

3.1 Propositional Semantic Tableaux

We will present several proof procedures for propositional logic in this chapter and the next. Two of them are especially suitable for automation: resolution [42] and semantic tableaux [48]. Of these, resolution is closely connected with conjunctive normal or clause forms, while the semantic tableaux system is similarly connected with disjunctive normal or dual clause forms. We discuss semantic tableaux rules in this section and resolution in Section 3.3. We begin with a general description that is suited to either hand or machine implementation and give a Prolog implementation in Section 3.2.

Both resolution and tableaux are *refutation* systems. To prove a formula X, we begin with $\neg X$ and produce a contradiction. The procedure for doing this involves expanding $\neg X$ so that inessential details of its logical structure are cleared away. In tableaux proofs, such an expansion takes the form of a tree, where nodes are labeled with formulas. The idea is that each branch should be thought of as representing the conjunction of the formulas appearing on it and the tree itself as representing the disjunction of its branches.

To begin, we restate the Dual Clause Set Reduction Rules from Section 2.8, but now we call them *Tableau Expansion Rules.* They are given in Table 3.1

Next we say something about how the rules in Table 3.1 are intended to be applied. Basically, they allow us to turn a tree with formulas as node labels into another such tree. Suppose we have a finite tree **T** with

$$\begin{array}{ccccc}
\dfrac{\neg\neg Z}{Z} & \dfrac{\neg\top}{\bot} & \dfrac{\neg\bot}{\top} & \dfrac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} & \dfrac{\beta}{\beta_1 \mid \beta_2}
\end{array}$$

TABLE 3.1. Tableau Expansion Rules

nodes labeled by propositional formulas. Select a branch θ and a non-literal formula occurrence X on θ. If X is $\neg\neg Z$, lengthen θ by adding a node labeled Z to its end. Similarly, if X is $\neg\top$, add $\bot$, and if X is $\neg\bot$, add $\top$. If X is α, add a node to the end of θ labeled α_1 and another node after that labeled α_2. Finally, if X is β, add left and right children to the final node of θ, and label one β_1 and the other β_2. Call the resulting tree $\mathbf{T}^*$. We say $\mathbf{T}^*$ *results from* $\mathbf{T}$ *by the application of a Tableau Expansion Rule.* If it is necessary to be more specific, we may say $\mathbf{T}^*$ results from the application of the α-rule, or whichever, to formula occurrence X on branch θ.

Now we define the notion of a tableau. Our definition is a little more general than we need at the moment since we allow *finite sets* of formulas at the start. The added generality will be of use when we come to prove completeness. The definition is a recursive one.

Definition 3.1.1 Let $\{A_1, \ldots, A_n\}$ be a finite set of propositional formulas.

1. The following one-branch tree is a tableau for $\{A_1, \ldots, A_n\}$:

$$\begin{array}{c} A_1 \\ A_2 \\ \vdots \\ A_n \end{array}$$

2. If $\mathbf{T}$ is a tableau for $\{A_1, \ldots, A_n\}$ and $\mathbf{T}^*$ results from $\mathbf{T}$ by the application of a Tableau Expansion Rule, then $\mathbf{T}^*$ is a tableau for $\{A_1, \ldots, A_n\}$.

Example Figure 3.1 shows a tableau for $\{P \downarrow (Q \vee R), \neg(Q \wedge \neg R)\}$. The numbers are not an official part of the tableau but have been added to make talking about it easier. In this tree, 1 and 2 make up the set the tableau is for; 3 and 4 are from 2 by the β-rule; 5 is from 4 by the $\neg\neg$ rule; 6 and 7 are from 1 by the α-rule; 8 and 9 are from 7 by the α-rule. Notice that we never applied the α rule to 1 on the right-hand branch. Also, we chose to apply a rule to 2 before we did to 1.

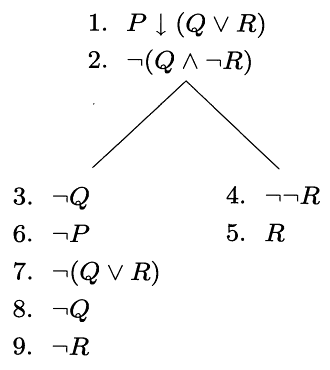


FIGURE 3.1. Tableau for $\{P \downarrow (Q \vee R), \neg(Q \wedge \neg R)\}$

Definition 3.1.2 A branch θ of a tableau is called *closed* if both X and $\neg X$ occur on θ for some propositional formula X, or if $\bot$ occurs on θ. If A and $\neg A$ occur on θ where A is atomic, or if $\bot$ occurs, θ is said to be *atomically closed.* A tableau is (atomically) closed if every branch is (atomically) closed.

As we remarked earlier, both tableau and resolution-style proofs are *refutation* arguments. That is, a proof of X amounts to a refutation of $\neg X$.

Definition 3.1.3 A *tableau proof* of X is a closed tableau for $\{\neg X\}$. X is a *theorem* of the tableau system if X has a tableau proof. We will write $\vdash_{pt} X$ to indicate that X has a propositional tableau proof.

Example Figure 3.2 shows a tableau proof (with numbers added for reference) of $[(P \supset (Q \supset R)) \supset ((P \vee S) \supset ((Q \supset R) \vee S))]$. In it, 1 is the negation of the formula to be proved; 2 and 3 are from 1 by α; 4 and 5 are from 3 by α; 6 and 7 are from 5 by α; 8 and 9 are from 2 by β. 10 and 11 are from 4 by β. Reading from left to right, the branches are closed because of 8 and 10, 7 and 11, and 6 and 9. Notice that on one of the branches closure was on a non-atomic formula.

Of course, we must establish that the tableau procedure does what we want. To be precise, we must show *soundness*: anything provable is a tautology (Section 3.4). And we must show *completeness*: all tautologies have proofs (Section 3.7). Indeed, we will show a particularly strong version that says, as long as we eventually apply every Tableau Expansion Rule once to every non-literal formula occurrence on every branch, we

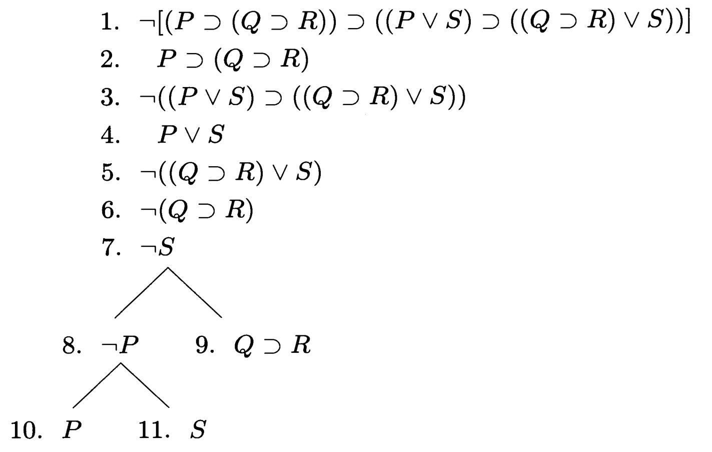

1. $\neg[(P \supset (Q \supset R)) \supset ((P \vee S) \supset ((Q \supset R) \vee S))]$
2. $P \supset (Q \supset R)$
3. $\neg((P \vee S) \supset ((Q \supset R) \vee S))$
4. $P \vee S$
5. $\neg((Q \supset R) \vee S)$
6. $\neg(Q \supset R)$
7. $\neg S$
8. $\neg P$
9. $Q \supset R$
10. P
11. S

FIGURE 3.2. Proof of $[(P \supset (Q \supset R)) \supset ((P \vee S) \supset ((Q \supset R) \vee S))]$

will find a proof if one exists. We will also show that testing for closure can be restricted to the level of literals without affecting completeness.

Tableau proofs can be very much shorter than truth table verifications. As a trivial example, if X is a propositional formula with n propositional variables, a truth table for $X \vee \neg X$ will have 2^n lines, but a closed tableau begins with $\neg(X \vee \neg X)$, proceeds with the α rule to add $\neg X$ and $\neg\neg X$, and is closed at this point (though it is not *atomically* closed). Further, the tableau method extends easily to handle quantifiers, while the truth table method does not.

The tableau rules are non-deterministic—they say what we may do, not what we must do. They allow us to choose which formula to work with next, on which branches. They allow us to skip formulas or use them more than once. And they allow us to close branches at the atomic level or at a more complex level if we can. People generally find this freedom useful, and often judicious choices of rule applications can produce shorter proofs than might be expected. On the other hand, when it comes to incorporating the tableau system in a deterministic computer program, some standardized order of rule applications must be imposed, and various limitations on the basic tableau freedom will be necessary. One limitation turns out to be very fundamental, and we discuss it now. As we remarked, in constructing tableaux we are allowed to use formulas over and over. For instance, if α occurs on a branch, we can add α_1 and α_2, and later we can add α_1 and α_2 again, since α is still present. For certain non-classical logics, this ability to reuse formulas is essential (see

Fitting [16]). But if we are allowed to reuse formulas, how do we know when we should give up on a proof attempt? After all, there will always be something we can try; if it didn't work before, maybe it will now. Fortunately, for classical propositional logic, it is never necessary to use a formula more than once on any branch. This makes the task of implementing tableaux easier. On the other hand, it makes the proof of tableau completeness somewhat more work. If we allow the reuse of formulas, completeness of the tableau system can easily be proved by a general method, based on the Model Existence Theorem from Section 3.6, and this same method allows us to prove the completeness of many other types of proof procedures. If we impose a no-reuse restriction, the easy general methods fail us, and we must introduce other techniques. Since the restriction is so important, we introduce some special terminology for it.

Definition 3.1.4 A tableau is *strict* if in its construction no formula has had a Tableau Expansion Rule applied to it twice on the same branch.

In constructing strict tableaux by hand, we might keep track of which formulas have had rules applied to them by simply checking them off as they are used. But a formula occurrence may be common to several branches, and we may have applied a rule to it on only one of them. An easy way of dealing with this is to check the occurrence off but add fresh occurrences at the ends of those branches where we have not used it.

A strictness restriction is of more importance for machine implementation. In our implementation we will represent a tableau as a list of its branches, and a branch as a list of its formulas. (This is the same data structure we used for the Dual Clause Form Algorithm implementation.) Using this representation a formula occurrence that is common to several branches turns into multiple occurrences, in several lists. Then a strict tableau construction is easy to keep track of: When we use a formula on a branch, or list, we simply remove it. Notice that now Tableau Expansion Rule applications become identical with Dual Clause Set Reduction Rule applications. There are some important differences, however. The procedure here may produce a closed tableau before all possible Expansion Rules have been applied. In particular, we do not have to apply the appropriate Tableau Expansion Rule on every branch that goes through a given formula occurrence. This means that, by being clever about when to check for closure and about which Tableau Expansion Rules to apply, and when to apply them, we may be able to produce a short proof instead of a long one. In other words, there is considerable scope for *heuristics.*

$(P \land (Q \supset (R \lor S))) \supset (P \lor Q)$ is a theorem. Figure 3.3 gives two different tableau proofs for this formula. Clearly, the right-hand proof is shorter

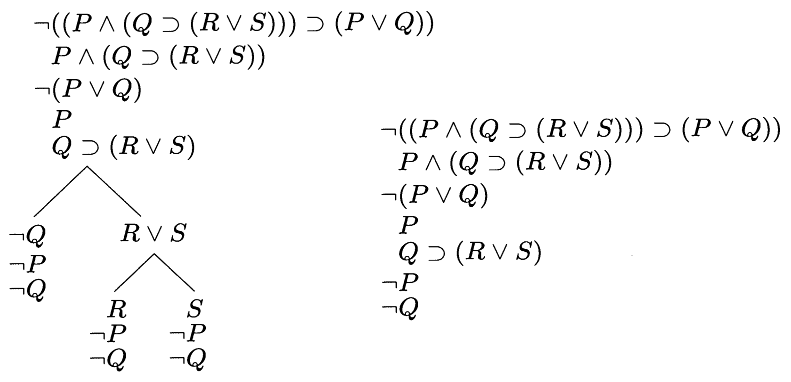


FIGURE 3.3. Two Tableau Proofs of $(P \wedge (Q \supset (R \vee S))) \supset (P \vee Q)$

than the left. And much more dramatic examples can be given. Now, having raised the issue of heuristics, we abandon it for the time being.

Exercises

3.1.1. Give tableau proofs of the following:

1. $((P \supset Q) \wedge (Q \supset R)) \supset \neg(\neg R \wedge P)$.
2. $(\neg P \supset Q) \supset ((P \supset Q) \supset Q)$.
3. $((P \supset Q) \supset P) \supset P$.
4. $(P \uparrow P) \uparrow P$.
5. $\neg((P \downarrow Q) \downarrow (P \vee Q))$.
6. $(\neg P \downarrow \neg Q) \subset \neg(P \uparrow Q)$.
7. $((((A \supset B) \supset (\neg C \supset \neg D)) \supset C) \supset E) \supset ((E \supset A) \supset (D \supset A))$.
8. $(P \uparrow (Q \uparrow R)) \uparrow ((P \uparrow (R \uparrow P)) \uparrow ((S \uparrow Q) \uparrow ((P \uparrow S) \uparrow (P \uparrow S))))$.
9. $(P \uparrow (Q \uparrow R)) \uparrow (((S \uparrow R) \uparrow ((P \uparrow S) \uparrow (P \uparrow S))) \uparrow (P \uparrow (P \uparrow Q)))$.

3.1.2. The exclusive-or connective, $\not\equiv$, can be defined in terms of the Primary Connectives:

$$(P \not\equiv Q) = ((P \wedge \neg Q) \vee (\neg P \wedge Q)).$$

1. Using this definition, give a tableau proof that $\not\equiv$ is commutative.
2. Similarly, give a tableau proof showing that $\not\equiv$ is associative.

3.2 Propositional Tableaux Implementations

In Section 3.7 we will prove completeness of the tableau procedure, and in Section 3.8 we will show it remains complete even when several restrictions are imposed. These restrictions are critical for successful implementation, so we discuss them briefly now.

First, all tableaux in this section will be strict. Second, the tableau rules allow testing for branch closure at any time; we will show it is enough to test for *atomic* closure only, and only after all possible Tableau Expansion Rules have been applied.

Finally, the tableau rules are non-deterministic, but we will show they have a certain *strong* completeness property. The order in which the rules are applied doesn't matter, as long as we eventually try everything once. This means we still have considerable freedom for the imposition of heuristics.

Incidentally, we represent a tableau itself as a list of its branches, and a branch as a list of its formulas. If we discover that a branch is closed, we remove it from the list. Then the empty list of branches represents a closed tableau.

If we apply all the Tableau Expansion Rules we can, removing the formulas to which they have been applied, what we are doing is producing a Dual Clause Form expansion. We already have the Dual Clause program of Section 2.9, so all we have to do is add a test for closure. Consequently, a tableau theorem prover can be produced very simply as follows: Begin with the program from Section 2.8, and remove the Prolog clause for `dualclauseform`, which we will not need now. Then add the following clauses that test a tableau for closure:

```
/*  closed(Tableau) :- every branch of Tableau contains a
        contradiction.
*/

closed([Branch | Rest]) :-
  member(false, Branch),
  closed(Rest).

closed([Branch | Rest]) :-
  member(X, Branch),
  member(neg X, Branch),
  closed(Rest).

closed([ ]).
```

Now, all we have to do is expand, using the expand predicate from the earlier program, then test the result for closure. The following will do the job.

```
/*  test(X) :- create a complete tableau expansion
        for neg X, and see if it is closed.
*/

test(X) :-
  expand([[neg X]], Y),
  closed(Y).
```

The `test` predicate has an efficiency problem (though not a logical one). If we use `test` on a formula that is provable, the call on `expand` will succeed, then the call on `closed` will also succeed, all as expected. On the other hand, if we try this on something that is *not* provable, the call on `expand` will still succeed, but the call on `closed` will fail, causing backtracking to `expand`. But `expand` can succeed in several ways (essentially since dual clause forms are not unique), and `closed` will never succeed. Consequently, before the program terminates in failure, it will be forced to run through many ways of expanding into dual clause form. Since we know it is enough to try only one such expansion, much unnecessary work is being done. Clearly, this is a proper place for Prolog's cut `!`, which prevents backtracking. The `test` clause should be replaced by the following, which will be more efficient in cases involving a non-theorem:

```
test(X) :-
  expand([[neg X]], Y),
  !,
  closed(Y).
```

It would be a little nicer if the program could respond with polite messages, "yes, it's a theorem" or "no, it's not a theorem," say. This minor improvement is most simply handled using an if–then–else construction. This exists in some Prologs and is easily implemented in the rest. The following clauses are taken from Sterling and Shapiro [51].

```
 /*  if_then_else(P, Q, R) :-
        either P and Q, or not P and R.
*/

if_then_else(P, Q, R) :- P, !, Q.

if_then_else(P, Q, R) :- R.
```

Now, here is an improved version of the `test` predicate.

```
/*  test(X) :- create a complete tableau expansion
```

```
        for neg X, if it is closed, say we have a
        proof, otherwise, say we don't.
*/

test(X) :-
  expand([[neg X]], Y),
  if_then_else(closed(Y), yes, no).

yes :- write('Propositional tableau theorem'), nl.

no :- write('Not a propositional tableau theorem'), nl.
```

Notice that we no longer have a cut after expand. What follows it now is not closed, which might or might not succeed, but if_then_else, which is written so that it always succeeds. Thus we not only get nicer responses from our program; we have also made it a little more structured.

A notable inefficiency is still present, however. Say we want to prove the tautology $(P \wedge Q) \vee \neg(P \wedge Q)$. A tableau for this will begin with $\neg((P \wedge Q) \vee \neg(P \wedge Q))$, then using the α-rule, it will continue with the two formulas $\neg(P \wedge Q)$ and $\neg\neg(P \wedge Q)$. At this point the tableau is closed, since one of these formulas is the negation of the other. But if we use our program to find a proof, work continues beyond this point, since the program must produce a complete expansion into dual clause form before checking for closure. In this case not much extra work is required, but we could just as well have asked for a proof of $F \vee \neg F$ where F is a formula of great complexity. For such cases a version of the program that checks for closure frequently might be desirable. It is easy to modify our program so that it checks for closure after each application of a Tableau Expansion Rule. First, remove the earlier Prolog clauses for expand, and replace them by the following clauses for expand_and_close:

```
/*  expand_and_close(Tableau) :-
      some expansion of Tableau closes.
*/

expand_and_close(Tableau) :-
  closed(Tableau).

expand_and_close(Tableau) :-
  singlestep(Tableau, Newtableau), !,
  expand_and_close(Newtableau).
```

We noted the use of Prolog's cut, to prevent retries of tableau expansions. We made use of the same device here, but this time we pass a cut after every application of a single Tableau Expansion Rule. Just as before, the program would be correct without this cut but would be less efficient. Now, finally, replace the clause for `test` used by the following.

```
/*  test(X) :- create a tableau expansion for neg X,
         if it is closed, say we have a proof,
         otherwise, say we don't.
*/

test(X) :-
  if_then_else(expand_and_close([[neg X]]), yes, no).
```

Which is the better program version? It depends. If one is trying to prove a formula like $F \vee \neg F$, where F is very complicated, the version that tests for closure frequently is clearly better. But for tautologies whose proofs have no short-cuts, a program that tests often for closure would be wasting time. Such tests are expensive, after all. So, which is better depends on what you know about the formula. In other words, *heuristics* play a role.

There is yet one more inefficiency left in the final version of the implementation. Suppose a tableau construction has been carried out to the point where there are five branches, four of which have closed. The predicate called `expand_and_close` will test the entire tableau for closure, find it is not closed, then apply a Tableau Expansion Rule, and test the entire tableau once again for closure. But there were four closed branches; they will still be closed and need not have been checked again. As written, these unnecessary closure checks must be made. A better version of the program would remove a branch from the tableau whenever it has been discovered to be closed, thus avoiding useless labor.

Exercises

3.2.1$^{\mathbf{P}}$. Write a modified version of the Prolog propositional tableau program that removes closed branches from the tableau as it generates it, thus avoiding redundant tests for closure.

3.2.2$^{\mathbf{P}}$. The connectives $\equiv$ and $\not\equiv$ are not primary, and so the tableau system implemented above cannot treat formulas containing them. They can be added easily if we "cheat" a little, as follows. Think of $X \equiv Y$ as an α, with α_1 being $X \supset Y$ and α_2 being $Y \supset X$. Likewise think of $\neg(X \equiv Y)$ as a β with β_1 being $\neg(X \supset Y)$ and β_2 being $\neg(Y \supset X)$. The connective $\not\equiv$ is treated similarly.

Modify the tableau implementation of this section so that $\equiv$ and $\not\equiv$ can appear in formulas, using the device just outlined.

3.3 Propositional Resolution

Tableau proofs are connected with the notion of Dual Clause Form. In the same way, resolution proofs are related to Clause Forms. Indeed, most common versions of resolution begin with a complete conversion to clause form, followed by applications of what is called the Resolution Rule. But, just as tableaux can close before a full conversion to Dual Clause Form has been carried out, so too, the Resolution Rule can be applied before we have reached Clause Form. Thus, we will begin by describing a kind of non-clausal resolution; we discuss the more conventional version later.

Tableau proofs are presented as trees, where a branch stands for the conjunction of the formulas on it, and the tree itself stands for the disjunction of its branches. Thus, trees are convenient ways of displaying generalized disjunctions of generalized conjunctions. Resolution involves the dual notion: generalized conjunctions of generalized disjunctions. This time trees are not convenient for graphical representation. Instead we represent generalized disjunctions in the usual way, listing the disjuncts within square brackets. And we represent a conjunction of disjunctions by simply listing its members in a sequence, one disjunction to a line. Then what takes the place of Tableau Expansion Rules are rules for adding new lines, new disjunctions, to a sequence. We begin by restating the Clause Set Reduction Rules, but this time we call them *Resolution Expansion Rules*, Table 3.2.

$$\frac{\neg\neg Z}{Z} \qquad \frac{\neg\top}{\bot} \qquad \frac{\neg\bot}{\top} \qquad \frac{\beta}{\begin{array}{c}\beta_1\\ \beta_2\end{array}} \qquad \frac{\alpha}{\alpha_1 \mid \alpha_2}$$

TABLE 3.2. Resolution Expansion Rules

The rules in Table 3.2 are intended to specify which disjunctions follow from which. Suppose we have a disjunction D containing a non-literal formula occurrence X. If X is $\neg\neg Z$, then a disjunction follows that is like D except that it contains an occurrence of Z where D contained $\neg\neg Z$. Similarly, if X is $\neg\top$ or $\neg\bot$. If X is β, a disjunction follows that is like D except that it contains occurrences of both β_1 and β_2 where D contained β. If X is α, two disjunctions follow, one like D but with α replaced by α_1, the other with α replaced by α_2. In each case we say the new disjunction (or disjunctions) follows from D by the application of a Resolution Expansion Rule.

Example The following is a sequence of disjunctions in which each, after the first two, follows from earlier lines by the application of a Resolution Expansion Rule:

1. $[P \downarrow (Q \wedge R)]$
2. $[\neg(Q \vee (P \supset Q))]$
3. $[\neg P]$
4. $[\neg(Q \wedge R)]$
5. $[\neg Q, \neg R]$
6. $[\neg Q]$
7. $[\neg(P \supset Q)]$

Here 3 and 4 are from 1 by α; 5 is from 4 by β; and 6 and 7 are from 2 by α.

For tableaux we distinguished between strict and non-strict. We do the same thing here and for the same reasons.

Definition 3.3.1 We call a sequence of Resolution Expansion Rule applications *strict* if every disjunction has at most one Resolution Expansion Rule applied to it.

The easiest way to ensure strictness is to check off a disjunction whenever we apply a rule to it. The previous example is, in fact, strict.

Strict Resolution Expansion Rule applications allow no formula reuse, non-strict ones do. Just as with tableaux, completeness of the non-strict version of resolution can be proved easily and by general methods; completeness of the strict version is more work and requires special techniques. But again, just as with tableaux, the strict version is by far the one best suited for implementation.

Resolution Expansion Rules are familiar, under the name *Clause Set Reduction Rules.* Now we introduce a rule of quite a different nature, the *Resolution Rule.*

Definition 3.3.2 Suppose D_1 and D_2 are two disjunctions, with X occurring as a member of D_1 and $\neg X$ as a member of D_2. Let D be the result of the following:

1. Deleting all occurrences of X from D_1
2. Deleting all occurrences of $\neg X$ from D_2
3. Combining the resulting disjunctions

We say D is the result of *resolving* D_1 *and* D_2 *on* X. We also refer to D as the *resolvent* of D_1 and D_2, with X being the formula *resolved on.* We also allow a trivial special case of resolution. If F is a disjunction containing $\bot$, and D is the result of deleting all occurrences of $\bot$ from F, we call D the *trivial resolvent* of F.

Example The result of resolving $[P, Q \supset R]$ and $[A \wedge B, \neg P]$ on P is $[Q \supset R, A \wedge B]$. The result of resolving $[A \wedge B]$ and $[\neg(A \wedge B)]$ on $A \wedge B$ is the empty clause $[\,]$. The trivial resolution of $[P, Q \uparrow R, \bot]$ is $[P, Q \uparrow R]$.

Propositional Resolution Rule D follows from the disjunctions D_1 and D_2 by the Resolution Rule if D is the result of resolving D_1 and D_2 on some formula X. If X is atomic, we say this is an *atomic* application of the Resolution Rule. Likewise, D follows from D_1 by a *trivial* application of the Resolution Rule if D is the trivial resolvent of D_1.

There is no analog of strictness for the Resolution Rule. If we are not allowed to use disjunctions more than once in Resolution Rule applications, completeness can not be proved.

Next we define the notion of a Resolution Expansion. The definition is a recursive one and a bit more general than we need just now.

Definition 3.3.3 Let $\{A_1, A_2, \ldots, A_n\}$ be a finite set of propositional formulas.

1. $$\begin{array}{l} [A_1] \\ [A_2] \\ \vdots \\ [A_n] \end{array}$$

   is a *Resolution Expansion* for $\{A_1, A_2, \ldots, A_n\}$.

2. If S is a Resolution Expansion for $\{A_1, A_2, \ldots, A_n\}$ and D results from some line or lines of S by the application of a Resolution Expansion Rule or the Resolution Rule, then S with D added as a new last line is also a Resolution Expansion for $\{A_1, A_2, \ldots, A_n\}$.

Recall from Section 2.8 that $[\,]$ is always **f** under any Boolean valuation. If we think of a resolution expansion as the conjunction of its lines, any Resolution Expansion containing $[\,]$ must also evaluate to **f** under every Boolean valuation.

Definition 3.3.4 We call a Resolution Expansion containing the empty clause *closed.*

Resolution, like the tableau system, is a refutation system. To prove X we attempt to refute its negation.

Definition 3.3.5 A *resolution proof* of X is a closed resolution expansion for $\{\neg X\}$. X is a *theorem* of the resolution system if X has a resolution proof. We will write $\vdash_{pr} X$ to indicate that X has a propositional resolution proof.

Example The following is a (strict) resolution proof of $((P \wedge Q) \vee (R \supset S)) \supset ((P \vee (R \supset S)) \wedge (Q \vee (R \supset S)))$:

1. $[\neg(((P \wedge Q) \vee (R \supset S)) \supset ((P \vee (R \supset S)) \wedge (Q \vee (R \supset S))))]$
2. $[(P \wedge Q) \vee (R \supset S)]$
3. $[\neg((P \vee (R \supset S)) \wedge (Q \vee (R \supset S)))]$
4. $[P \wedge Q, R \supset S]$
5. $[P, R \supset S]$
6. $[Q, R \supset S]$
7. $[\neg(P \vee (R \supset S)), \neg(Q \vee (R \supset S))]$
8. $[\neg P, \neg(Q \vee (R \supset S))]$
9. $[\neg(R \supset S), \neg(Q \vee (R \supset S))]$
10. $[\neg P, \neg Q]$
11. $[\neg P, \neg(R \supset S)]$
12. $[\neg(R \supset S), \neg Q]$
13. $[\neg(R \supset S), \neg(R \supset S)]$
14. $[P, \neg Q]$
15. $[\neg Q]$
16. $[R \supset S]$
17. $[\,]$

In this, 2 and 3 are from 1 by α; 4 is from 2 by β; 5 and 6 are from 4 by α; 7 is from 3 by β; 8 and 9 are from 7 by α; 10 and 11 are from 8 by α; and 12 and 13 are from 9 by α. Now 14 is by the Resolution Rule on $R \supset S$ in 5 and 12; 15 is by Resolution on P in 10 and 14; 16 is by Resolution on Q in 6 and 15; and 17 is by Resolution on $R \supset S$ in 13 and 16. Note that not all Resolution Rule applications are at the atomic level. This is not the only resolution proof for this formula. You might try finding others.

As we remarked earlier, the resolution system is complete even if all Resolution Rule applications are atomic and follow all Resolution Expansion Rules, applications of which are strict. Further, just as with tableaux, as long as all possible Resolution Expansion Rules and all possible Resolution Rule applications get made in a proof attempt, a proof must be found if one exists. We will prove all this later, but use can be made of it now in implementing a resolution theorem prover.

Exercises

3.3.1. Redo Exercise 3.1.1, but giving resolution instead of tableau proofs.

3.3.2. Redo Exercise 3.1.2 on the exclusive-or connective, using resolution instead of tableaux.

3.3.3$^{\mathbf{P}}$. Write a resolution theorem prover in Prolog. Use the tableau program in Section 3.2 as a starting point.

3.4 Soundness

A proof procedure for propositional logic is called *sound* if it can prove only tautologies. In effect this is a *correctness* issue; we want theorem-proving algorithms to give no incorrect answers. We will show soundness in some detail for the tableau system and leave the resolution version as a series of exercises. Incidentally, if we show the basic tableau or resolution system is sound, it remains sound no matter what restrictions we impose, because restrictions can only have the effect of making it impossible to prove certain things. Consequently, we prove soundness with no restrictions. It follows that resolution and tableaux with strictness requirements are also sound, for instance.

Any algorithm based on the tableau system will have the general form: Begin with some initial tableau, then keep applying Tableau Expansion Rules in some order until a closed tableau is generated. Our proof of soundness is based on the following simple idea: We define what it means for a tableau to be satisfiable, then we show that satisfiability is a loop invariant. From this, soundness will follow easily. The definition of satisfiability is straightforward, once we remember the connection between tableaux and disjunctions of conjunctions.

Definition 3.4.1 A *set* S of propositional formulas is *satisfiable* if some Boolean valuation maps every member of S to $\mathbf{t}$. A tableau *branch* θ is satisfiable if the set of propositional formulas on it is satisfiable. A *tableau* $\mathbf{T}$ is satisfiable if at least one branch of $\mathbf{T}$ is satisfiable.

Proposition 3.4.2 *Any application of a Tableau Expansion Rule to a satisfiable tableau yields another satisfiable tableau.*

Proof Suppose $\mathbf{T}$ is a satisfiable tableau, and a Tableau Expansion Rule is applied to formula occurrence X on branch θ of $\mathbf{T}$, producing a tableau $\mathbf{T}^*$. We must show $\mathbf{T}^*$ is also a satisfiable tableau. The proof has several cases and subcases, all of which are simple. Since $\mathbf{T}$ is satisfiable, $\mathbf{T}$ has at least one satisfiable branch. Choose one, say it is branch τ.

Case 1 $\tau \neq \theta$. Then since a rule was applied only to θ, τ is still a branch of $\mathbf{T}^*$, hence $\mathbf{T}^*$ is satisfiable.

Case 2 $\tau = \theta$. Then θ itself is satisfiable, say the Boolean valuation v maps all formulas on θ to $\mathbf{t}$. Now we have subcases depending on which Rule was applied to the formula occurrence X.

Subcase 2a $X = \alpha$. Then θ was extended with α_1 and α_2 to produce $\mathbf{T}^*$. Since α occurs on θ, $v(\alpha) = \mathbf{t}$. By Proposition 2.6.1, $v(\alpha) = v(\alpha_1) \wedge v(\alpha_2)$; hence, v must map both α_1 and α_2 to $\mathbf{t}$. Consequently, v maps every formula on the extension of θ in $\mathbf{T}^*$ to $\mathbf{t}$, and thus, $\mathbf{T}^*$ is satisfiable.

Subcase 2b $X = \beta$. Then left and right children were added to the last node of θ, one labeled β_1 and one labeled β_2, to produce $\mathbf{T}^*$. Since β occurs on θ, $v(\beta) = \mathbf{t}$. But $v(\beta) = v(\beta_1) \vee v(\beta_2)$, hence one of β_1 or β_2 must be mapped to $\mathbf{t}$ by v. It follows that either every formula on the left-hand branch extending θ is mapped to $\mathbf{t}$ by v, or else every formula on the right-hand branch extending θ is mapped to $\mathbf{t}$. In either event, $\mathbf{T}^*$ has a satisfiable branch and hence is a satisfiable tableau.

The other subcases, corresponding to X being $\neg\neg Z$, $\neg\top$ or $\neg\bot$, are treated by straightforward arguments. We omit these. □

Proposition 3.4.3 *If there is a cl osed tableau for a set S, then S is not satisfiable.*

Proof Suppose there is a closed tableau for S, but S is satisfiable; we derive a contradiction. The construction of a closed tableau for S begins with an initial tableau consisting of a single branch whose nodes are labeled with members of S. Since S is satisfiable, this initial tableau is satisfiable. By Proposition 3.4.2, every subsequent tableau we construct must also be satisfiable, including the final closed tableau. But there are no closed, satisfiable tableaux. □

Theorem 3.4.4 **(Propositional Tableau Soundness)**
If X has a tableau proof, then X is a tautology.

Proof A tableau proof of X is a closed tableau for $\{\neg X\}$. By the preceding proposition, if there is a closed tableau for $\{\neg X\}$, then $\{\neg X\}$ is not a satisfiable set. It follows that X is a tautology. □

Next we turn to resolution. Essentially, we simply apply ideas that are dual to those we used for tableaux.

Definition 3.4.5 A resolution expansion is *satisfiable* if some Boolean valuation maps every line of it to **t**.

If we think of a tableau as a graphical representation of a generalized disjunction (of branches) of generalized conjunctions (of formulas on a branch), then satisfiability for a tableau simply means some Boolean valuation maps it to **t**. In a similar way, if we think of a resolution expansion as a generalized conjunction, of its disjunctions, satisfiability again means some Boolean valuation maps it to **t**.

Proposition 3.4.6 *Any application of a Resolution Expansion Rule or the Resolution Rule to a satisfiable Resolution Expansion yields another satisfiable Resolution Expansion.*

Proposition 3.4.7 *If there is a closed Resolution Expansion for a set S, then S is not satisfiable.*

Theorem 3.4.8 **(Propositional Resolution Soundness)**
If X has a resolution proof, then X is a tautology.

Exercises

3.4.1. Show that a closed tableau is not satisfiable.

3.4.2. Prove Proposition 3.4.6.

3.4.3. Prove Proposition 3.4.7.

3.4.4. Prove Theorem 3.4.8.

3.5 Hintikka's Lemma

We must show completeness of both the propositional tableau and resolution systems. That is, we must show that each tautology actually has a proof in these systems. There are many ways of doing this. For instance, the similarity between tableau construction in Section 3.1 and the Dual Clause Form Algorithm from Section 2.8 is more than coincidence. If an attempted tableau proof does not terminate in closure, it will terminate with a Dual Clause Form, and a proof of completeness can be based on this. The problem is that such a method does not extend readily to first-order logic. So rather than using this here, we take a more abstract approach that does generalize well, though it may seem like overkill for the propositional case. We begin by proving the propositional version of a Lemma that is due to Hintikka; we will see more elaborate versions of it in later chapters.

Definition 3.5.1 A set $\mathbf{H}$ of propositional formulas is called a *propositional Hintikka set*, provided the following:

1. For any propositional letter A, not both $A \in \mathbf{H}$ and $\neg A \in \mathbf{H}$.
2. $\bot \notin \mathbf{H}$; $\neg\top \notin \mathbf{H}$.
3. $\neg\neg Z \in \mathbf{H} \Rightarrow Z \in \mathbf{H}$.
4. $\alpha \in \mathbf{H} \Rightarrow \alpha_1 \in \mathbf{H}$ and $\alpha_2 \in \mathbf{H}$.
5. $\beta \in \mathbf{H} \Rightarrow \beta_1 \in \mathbf{H}$ or $\beta_2 \in \mathbf{H}$.

For example, the empty set is trivially a Hintikka set. The set of all propositional variables is a Hintikka set. The set $\{P \wedge (\neg Q \supset R), P, (\neg Q \supset R), \neg\neg Q, Q\}$ is a Hintikka set. Notice that conditions 3 through 5 all say, if certain formulas belong, so must certain simpler ones. Hintikka sets are also called *downward saturated.* The main result concerning these sets is due to Hintikka [27]

Proposition 3.5.2 **(Hintikka's Lemma)** *Every propositional Hintikka set is satisfiable.*

Proof Let $\mathbf{H}$ be a Hintikka set. We produce a Boolean valuation mapping every member of $\mathbf{H}$ to $\mathbf{t}$. As we observed in Section 2.4, every mapping from the set of propositional letters to $\mathbf{Tr}$ extends to a unique Boolean valuation. Well, let f be the mapping defined as follows: For a propositional letter A, $f(A) = \mathbf{t}$ if $A \in \mathbf{H}$; $f(A) = \mathbf{f}$ if $\neg A \in \mathbf{H}$; otherwise $f(A)$ is arbitrary, say for definiteness $f(A) = \mathbf{f}$ if neither A nor $\neg A$ is in $\mathbf{H}$. Note that condition 1 ensures that f is well-defined. Now let v be the Boolean valuation extending f. v maps every member of $\mathbf{H}$ to $\mathbf{t}$ (we leave the verification as an exercise). □

Exercises

3.5.1. The definition requires that a propositional Hintikka set be consistent at the atomic level. Prove by Structural Induction that, if $\mathbf{H}$ is a propositional Hintikka set, and if X is any propositional formula, then not both $X \in \mathbf{H}$ and $\neg X \in \mathbf{H}$.

3.5.2. Complete the proof of Proposition 3.5.2. More specifically, let us say a propositional formula X has property Q provided $X \in \mathbf{H} \Rightarrow v(X) = \mathbf{t}$. In other words, X has property Q, provided either X is not in $\mathbf{H}$ or else $v(X) = \mathbf{t}$. Now use Structural Induction 2.6.3, and show every propositional formula has property Q.

3.6 The Model Existence Theorem

Hintikka's Lemma connects syntax and semantics. In this section we state and prove a more complicated and more powerful theorem that also relates syntax and semantics. The proof contains the essence of a 'standard' completeness argument. With the argument given abstractly once and for all, completeness of resolution and tableaux formulations will be easy consequences and so will the completeness of several other proof procedures for classical propositional logic. It is partly because we can deal with several systems at once that we go to the trouble of doing this work at the level of abstraction we have chosen.

Most completeness proofs make use of the notion of *consistency*, which is relative to a particular proof procedure, such as resolution. A set of formulas is generally called consistent if no contradiction follows from it using the machinery of the proof procedure. Then various features of consistency are used to construct a Boolean valuation. By looking carefully at such constructions, one can identify those features of consistency that are essential. An *abstract consistency property* is something having these features, and the Model Existence Theorem is the assertion that these features are sufficient for the construction of a suitable Boolean valuation. (Boolean valuations are simplified versions, sufficient for propositional logic, of the first-order *models* that will be introduced in Chapter 5. This accounts for the name of the theorem.)

There is a minor technical point before we get down to business. Instead of talking about a consistency *property*, say $\mathcal{C}$, of sets of formulas, we talk about the *collection* of all sets having property $\mathcal{C}$. In fact, we identify this collection with $\mathcal{C}$ itself. Thus, an abstract consistency property is defined to be a collection $\mathcal{C}$ of sets of formulas, meeting certain closure conditions. If a set S is in the collection $\mathcal{C}$, we can refer to S as *$\mathcal{C}$-consistent.*

Definition 3.6.1 Let $\mathcal{C}$ be a collection of sets of propositional formulas. We call $\mathcal{C}$ a *propositional consistency property* if it meets the following conditions for each $S \in \mathcal{C}$:

1. for any propositional letter A, not both $A \in S$ and $\neg A \in S$.
2. $\bot \notin S$; $\neg \top \notin S$.
3. $\neg\neg Z \in S \Rightarrow S \cup \{Z\} \in \mathcal{C}$.
4. $\alpha \in S \Rightarrow S \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$.
5. $\beta \in S \Rightarrow S \cup \{\beta_1\} \in \mathcal{C}$ or $S \cup \{\beta_2\} \in \mathcal{C}$.

For instance, item 5 says that if S is $\mathcal{C}$-consistent and contains β, then it remains $\mathcal{C}$-consistent when one of β_1 or β_2 is added, and similarly for the other items.

Theorem 3.6.2 **(Propositional Model Existence)** *If $\mathcal{C}$ is a propositional consistency property, and $S \in \mathcal{C}$, then S is satisfiable.*

Proof The basic idea of the proof is to show that any member S of a Propositional Consistency Property can be enlarged to another member that is a Hintikka set, which will be satisfiable by Hintikka's Lemma. If S is finite this is easy. If $\alpha \in S$, α_1 and α_2 can be added, by item 4 of the definition, to produce another member of the consistency property, and similarly for other kinds of formulas. So, just keep adding what we need to produce a Hintikka set; the process must terminate because S is finite. It is a good exercise for you to carry out the details of this sketchy argument, under the assumption that S is finite. But if S is infinite, things are not so simple. We can, one by one, add the formulas we want to have in a Hintikka set, but the process need not terminate. Instead, we may find ourselves producing an infinite sequence of larger and larger members of the consistency property, and what we want is the limit (chain union) of this sequence. But there is no guarantee that Propositional Consistency Properties are closed under limits. Consequently, the proof that follows begins with an argument that Propositional Consistency Properties can always be extended to ones that are closed under limits. This portion of the proof requires several preliminary results, whose verification we leave to you as exercises.

1. Call a Propositional Consistency Property *subset closed* if it contains, with each member, all subsets of that member. Every Propositional Consistency Property can be extended to one that is subset closed.
2. Call a Propositional Consistency Property $\mathcal{C}$ of *finite character* provided $S \in \mathcal{C}$ if and only if every finite subset of S belongs to $\mathcal{C}$. Every Propositional Consistency Property of finite character is subset closed.

3. A Propositional Consistency Property that is subset closed can be extended to one of finite character.

Finally, being of finite character is enough to guarantee the existence of limits. More precisely, suppose $\mathcal{C}$ is a Propositional Consistency Property of finite character, and S_1, S_2, S_3, $\ldots$ is a sequence of members of $\mathcal{C}$ such that $S_1 \subseteq S_2 \subseteq S_3 \subseteq \ldots$. Then $\bigcup_i S_i$ is a member of $\mathcal{C}$.

The argument for this goes as follows. Since $\mathcal{C}$ is of finite character, to show $\bigcup_i S_i \in \mathcal{C}$, it is enough to show every finite subset of $\bigcup_i S_i$ is in $\mathcal{C}$. So, suppose $\{A_1, \ldots, A_k\} \subseteq \bigcup_i S_i$; we show $\{A_1, \ldots, A_k\} \in \mathcal{C}$. For each $i = 1, \ldots, k$, $A_i \in S_{n_i}$ for some smallest integer n_i. Let $N = \max\{n_1, \ldots, n_k\}$. It is easy to see each $A_i \in S_N$. But $S_N \in \mathcal{C}$, and $\mathcal{C}$ is subset closed, hence $\{A_1, \ldots, A_k\} \in \mathcal{C}$.

Now for the heart of the proof. Suppose S belongs to a Propositional Consistency Property $\mathcal{C}$. By Items 1 and 3, every Propositional Consistency Property can be extended to one that is of finite character. We may assume this has already been done, and so $\mathcal{C}$ is of finite character.

Since the list of propositional letters is countable, the entire set of propositional formulas is countable as well. This is a standard result of set theory, and we do not prove it here. Let X_1, X_2, X_3, $\ldots$ be an enumeration of all propositional formulas in some fixed order. We define a sequence, S_1, S_2, S_3, $\ldots$ of members of $\mathcal{C}$ as follows:

$$S_1 = S$$
$$S_{n+1} = \begin{cases} S_n \cup \{X_n\} & \text{if } S_n \cup \{X_n\} \in \mathcal{C} \\ S_n & \text{otherwise} \end{cases}$$

Then every $S_n \in \mathcal{C}$, and also S_n is a subset of S_{n+1}. Finally, let $\mathbf{H} = S_1 \cup S_2 \cup S_3 \cup \ldots$. Trivially, $\mathbf{H}$ extends S. Also, since $\mathcal{C}$ is of finite character, it is closed under chain unions, and hence $\mathbf{H} \in \mathcal{C}$.

$\mathbf{H}$ is maximal in $\mathcal{C}$; that is, if $\mathbf{H} \subseteq K$ for some $K \in \mathcal{C}$, then $\mathbf{H} = K$. Reasons: Suppose $\mathbf{H}$ is a proper subset of K, where $K \in \mathcal{C}$. Then for some propositional formula X_n, we have $X_n \in K$ but $X_n \notin \mathbf{H}$. Since $X_n \notin \mathbf{H}$, $X_n \notin S_{n+1}$, which implies $S_n \cup \{X_n\} \notin \mathcal{C}$. But $S_n \cup \{X_n\} \subseteq K$, since $S_n \subseteq \mathbf{H}$, and $\mathbf{H} \subseteq K$, and also $X_n \in K$. Since $\mathcal{C}$ is subset closed, $S_n \cup \{X_n\} \in \mathcal{C}$, and we have a contradiction.

$\mathbf{H}$ is a Hintikka set. Reasons: Suppose $\alpha \in \mathbf{H}$; we show $\alpha_1, \alpha_2 \in \mathbf{H}$. Since $\alpha \in \mathbf{H}$ and $\mathbf{H} \in \mathcal{C}$, $\mathbf{H} \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$. But this set extends $\mathbf{H}$, which is maximal, hence it must be identical with $\mathbf{H}$, which means $\alpha_1, \alpha_2 \in \mathbf{H}$. The other conditions are verified similarly.

Now, by Hintikka's Lemma, **H** is satisfiable, hence, so is S, since $S \subseteq$ **H**. □

Mathematically, it is often of interest to work with languages having an *uncountable* set of propositional letters. The Propositional Model Existence Theorem is still true when using such languages, though the proof we gave will not work. This proof explicitly makes use of countability. Alternate proofs based on the Axiom of Choice or Zorn's Lemma can be given instead. We do not do so here.

We illustrate the power of the Model Existence Theorem by proving two of the fundamental theorems of propositional logic. Both are semantic in nature; neither mentions a proof procedure. The first is the Compactness Theorem; we will need it later.

Theorem 3.6.3 **(Propositional Compactness)** *Let S be a set of propositional formulas. If every finite subset of S is satisfiable, so is S.*

Proof Assume every finite subset of S is satisfiable. Let $\mathcal{C}$ be the following collection of sets of propositional formulas: Put a set W in $\mathcal{C}$, provided every finite subset of W is satisfiable. Trivially, S is in $\mathcal{C}$. We claim $\mathcal{C}$ is a Propositional Consistency Property. Once this is shown, satisfiability of S follows immediately from the Propositional Model Existence Theorem.

Suppose $W \in \mathcal{C}$, but both A and $\neg A$ are in W, where A is a propositional letter. Then $\{A, \neg A\}$ is a finite subset of W, but it is not a satisfiable set. Consequently, we can not have both A and $\neg A$ in W.

Suppose $W \in \mathcal{C}$ and $\alpha \in W$. We show every finite subset of $W \cup \{\alpha_1, \alpha_2\}$ is satisfiable and hence that $W \cup \{\alpha_1, \alpha_2\}$ is in $\mathcal{C}$. Now, a finite subset of $W \cup \{\alpha_1, \alpha_2\}$ may or may not include α_1 and α_2. If it includes neither, it is a finite subset of W alone and hence is satisfiable because $W \in \mathcal{C}$. The argument for the cases where it includes one of α_1 or α_2 is similar to the argument for the case where it includes both, so we consider only that one. Suppose we have the set $W_0 \cup \{\alpha_1, \alpha_2\}$, where W_0 is a finite subset of W. Now, $W_0 \cup \{\alpha\}$ is also a finite subset of W, hence it is satisfiable. But any Boolean valuation mapping every member of $W_0 \cup \{\alpha\}$ to **t** must map α to **t**, hence both α_1 and α_2 must also be mapped to **t**. That is, $W_0 \cup \{\alpha, \alpha_1, \alpha_2\}$ is satisfiable, hence so is its subset $W_0 \cup \{\alpha_1, \alpha_2\}$.

The rest of the proof is similar and is left as an exercise. □

Our second application of the Model Existence Theorem is Craig's Interpolation Theorem. This result has important model-theoretic consequences, and we will consider it more fully once first-order logic has been introduced.

Definition 3.6.4 A formula Z is an *interpolant* for the implication $X \supset Y$ if every propositional letter of Z also occurs in both X and Y and if $X \supset Z$ and $Z \supset Y$ are both tautologies.

For example, $(P \vee (Q \wedge R)) \supset (P \vee \neg\neg Q)$ has $P \vee Q$ as an interpolant; $(P \wedge \neg P) \supset Q$ has $\bot$ as an interpolant.

Theorem 3.6.5 **(Craig Interpolation)** *If $X \supset Y$ is a tautology, then it has an interpolant.*

Proof We write $\langle S \rangle$, as usual, to denote the conjunction of the members of S. Call a finite set S *Craig consistent*, provided there is a partition of S into subsets S_1 and S_2 (that is, $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$) such that $\langle S_1 \rangle \supset \neg\langle S_2 \rangle$ has no interpolant. Let $\mathcal{C}$ be the collection of all Craig-consistent sets. $\mathcal{C}$ is a Propositional Consistency Property (Exercise 3.6.5).

Now we show the theorem in its contrapositive form. Suppose $X \supset Y$ has no interpolant. Let S be the set $\{X, \neg Y\}$, and consider the partition $S_1 = \{X\}$, $S_2 = \{\neg Y\}$. If $\langle\{X\}\rangle \supset \neg\langle\{\neg Y\}\rangle$ had an interpolant Z, then Z would also be an interpolant for $X \supset Y$, hence it does not have an interpolant. Then S is Craig consistent, and so by the Model Existence Theorem, S is satisfiable. It follows that $X \supset Y$ is not a tautology. □

Exercises

3.6.1. Show that every Propositional Consistency Property can be extended to one that is subset closed. Hint: Let $\mathcal{C}$ be a Propositional Consistency Property. Let $\mathcal{C}^+$ consist of all subsets of members of $\mathcal{C}$, and show $\mathcal{C}^+$ is also a Propositional Consistency Property.

3.6.2. Show that every Propositional Consistency Property of finite character is subset closed.

3.6.3. Show that a Propositional Consistency Property that is subset closed can be extended to one of finite character. Hint: Let $\mathcal{C}$ be a Propositional Consistency Property that is subset closed. Let $\mathcal{C}^+$ consist of those sets S all of whose finite subsets are in $\mathcal{C}$. Show that $\mathcal{C}^+$ is a Propositional Consistency Property and extends $\mathcal{C}$.

3.6.4. Finish the proof of the Propositional Compactness Theorem by showing in detail that $\mathcal{C}$ is a Propositional Consistency Property.

3.6.5. Complete the proof of Theorem 3.6.5 by showing that the collection of Craig-consistent sets is a Propositional Consistency Property.

3.6.6. Show that if $X \supset Y$ is a tautology and X and Y have no propositional letters in common, then one of $\neg X$ or Y is a tautology.

3.6.7. Let $\mathcal{C}$ be a Propositional Consistency Property and let B be a set of propositional formulas. We say $\mathcal{C}$ is B *compatible* if, for each $S \in \mathcal{C}$ and for each $X \in B$, $S \cup \{X\} \in \mathcal{C}$. Prove that if $\mathcal{C}$ is a propositional consistency property that is B compatible, and if $S \in \mathcal{C}$, then $S \cup B$ is satisfiable.

3.7 Tableau and Resolution Completeness

Now that the Model Existence Theorem is available, completeness results are easy to prove, at least for the non-strict versions of tableau and resolution. In the next section we take up the completeness of tableau and resolution with restrictions that are useful for implementation. At that point the Model Existence Theorem can no longer be used.

Definition 3.7.1 A finite set S of propositional formulas is *tableau consistent* if there is no closed tableau for S.

Lemma 3.7.2 *The collection of all tableau consistent sets is a Propositional Consistency Property.*

Proof We must establish that the conditions of Definition 3.6.1 are met. Items 1 and 2, requiring consistency at the atomic level, are trivial. For the closure conditions 2 through 5, all are rather similar, so we treat only one. It is easiest to work in the contrapositive direction. Suppose $\alpha \in S$, but $S \cup \{\alpha_1, \alpha_2\}$ is not tableau consistent; we show that S is not tableau consistent either.

Since $S \cup \{\alpha_1, \alpha_2\}$ is not tableau consistent, there is a closed tableau for $S \cup \{\alpha_1, \alpha_2\}$. α Is one of the members of S; say $S = \{\alpha, X_1, \ldots, X_n\}$. Then we have a closed tableau that looks like the following.

$$\begin{array}{c} \alpha \\ X_1 \\ \vdots \\ X_n \\ \alpha_1 \\ \alpha_2 \\ \text{rest of closed tableau} \end{array}$$

To show S itself is not tableau consistent, we must produce a closed tableau beginning with α, $X_1, \ldots, X_n$. But this is easy. Start with these formulas, apply the α-rule to add α_1 and α_2, and then continue the tableau construction exactly as before. □

Theorem 3.7.3 **(Completeness for Propositional Tableaux)**
If X is a tautology, X has a tableau proof.

Proof We show the contrapositive. If X does not have a tableau proof, there is no closed tableau for $\{\neg X\}$. Then $\{\neg X\}$ is tableau consistent, hence satisfiable by the Propositional Model Existence Theorem 3.6.2, and so X is not a tautology. □

If we change the definition of tableau consistency for a finite set S to no tableau for S is *atomically* closed, then nothing essential changes in the proofs we have given, because condition 1 of Definition 3.6.1 required only no contradictions at the atomic level. Thus, we have the stronger result: If X is a tautology, X has a tableau proof in which the tableau is atomically closed.

The Model Existence Theorem, as it stands, is still not enough to get us the completeness of *strict* tableaux. It is possible to strengthen the Model Existence Theorem for this purpose, but the strengthened version still does not apply readily to resolution. Consequently, we leave this approach to you in the Exercises and treat strict versions of tableaux and resolution by quite different techniques, in the next section. In the rest of this section, we use the Model Existence Theorem to prove completeness of resolution *without a strictness requirement.* It will be convenient to first introduce some special terminology.

Definition 3.7.4 Let S be a set of disjunctions. A *resolution derivation* from S is a sequence of disjunctions, each of which is a member of S, or comes from an earlier term in the sequence by one of the Resolution Expansion Rules, or comes from earlier terms by the Resolution Rule. We say a disjunction D is *resolution derivable* from S if D is the last line of a resolution derivation from S.

If $\{A_1, \dots, A_n\}$ is a set of formulas, a resolution expansion for this set, and a resolution derivation from $\{[A_1], \dots, [A_n]\}$ amount to the same thing. The notion of resolution derivation is more general though, since it allows us to start with any family of generalized disjunctions.

Definition 3.7.5 Let X be a propositional formula. We say both disjunctions $[X, A_1, \dots, A_n]$ and $[A_1, \dots, A_n]$ are *X-enlargements* of $[A_1, \dots, A_n]$. If S is a set of disjunctions and S^* is the result of replacing each member of S by an X-enlargement, we say S^* is an X-enlargement of S.

Example $\{[A_1, A_2, X], [B_1, B_2, B_3, X]\}$ and $\{[A_1, A_2], [B_1, B_2, B_3, X]\}$ are both X-enlargements of $\{[A_1, A_2], [B_1, B_2, B_3]\}$.

Lemma 3.7.6 *Suppose S_1 and S_2 are sets of disjunctions, and S_2 is an X-enlargement of S_1. If the disjunction D_1 is resolution derivable from S_1, then there is an X-enlargement D_2 of D_1 that is resolution derivable from S_2.*

Proof The informal idea is quite simple: carry-along occurrences of X at appropriate points in the resolution derivation from S_1. A formal proof is by induction on the lengths of resolution expansions. Length 1 is trivial.

Suppose the result is known for resolution derivations from S_1 of length $< n$, and we now have a resolution derivation of length n. Say the last line came from an earlier line using the β-Resolution Expansion Rule. Then the resolution derivation from S_1 looks like:

$$\begin{array}{c} \vdots \\ [\beta, A_1, \ldots, A_k] \\ \vdots \\ [\beta_1, \beta_2, A_1, \ldots, A_k]\,. \end{array}$$

Since $[\beta, A_1, \ldots, A_k]$ occurs at a line earlier than line n, by the induction hypothesis, there is a resolution derivation from S_2 ending with an X-enlargement, one of $[\beta, A_1, \ldots, A_k]$ or $[X, \beta, A_1, \ldots, A_k]$. In the first case, $[\beta_1, \beta_2, A_1, \ldots, A_k]$ follows by the β-rule, and in the second case, $[X, \beta_1, \beta_2, A_1, \ldots, A_k]$ follows, still by the β-rule. Either way, the result is established for line n in the β-case.

This takes care of one case. There are several more, depending on the rule used to add the n^{th} line. We leave the other cases as an exercise. □

Definition 3.7.7 A finite set S of propositional formulas is *resolution consistent* if there is no closed resolution expansion for S.

An equivalent version of this definition follows: $\{X_1, \ldots, X_n\}$ is resolution consistent if there is no resolution derivation of the empty clause from $\{[X_1], \ldots, [X_n]\}$.

Lemma 3.7.8 *The collection of all resolution consistent sets is a Propositional Consistency Property.*

Proof Again we must establish that the conditions of Definition 3.6.1 are met. Items 1 and 2, requiring consistency at the atomic level, are straightforward and are the only ones that directly involve the Resolution Rule. Of the closure conditions 3 through 5, we consider only the α-case and the β-case. As with tableaux, it is easiest to show the contrapositive.

Suppose $\alpha \in S$ and $S \cup \{\alpha_1, \alpha_2\}$ is not resolution consistent. We show that S itself is not resolution consistent. Say $S = \{X_1, \ldots, X_n, \alpha\}$. Since $S \cup \{\alpha_1, \alpha_2\}$ is not resolution consistent, there is a resolution derivation, call it D, of [ ] from $\{[X_1], \ldots, [X_n], [\alpha], [\alpha_1], [\alpha_2]\}$. Now, a verification that S is not resolution consistent can easily be produced as follows. Start a resolution derivation with $[X_1], \ldots, [X_n], [\alpha]$. Apply the α-rule to add $[\alpha_1]$ and $[\alpha_2]$, and then continue with the steps of D.

Next, suppose $\beta \in S$ and neither $S \cup \{\beta_1\}$ nor $S \cup \{\beta_2\}$ are resolution consistent. We show S is also not resolution consistent. This time things are just a little trickier.

Say $S = \{\beta, X_1, \ldots, X_n\}$. Applying the β-rule to a resolution derivation beginning with $[X_1], \ldots, [X_n], [\beta]$ allows us to add $[\beta_1, \beta_2]$. It follows that to show S is not resolution consistent, it is enough to show there is a resolution derivation of [ ] from $[X_1], \ldots, [X_n], [\beta], [\beta_1, \beta_2]$.

Since $S \cup \{\beta_1\}$ is not resolution consistent there is a resolution derivation of [ ] from $\{[X_1], \ldots, [X_n], [\beta], [\beta_1]\}$. Then by Lemma 3.7.6, there is a derivation from $\{[X_1], \ldots, [X_n], [\beta], [\beta_1, \beta_2]\}$ of either [ ] or $[\beta_2]$. In the first case we are done immediately; the second possibility requires a little more work. Since $S \cup \{\beta_2\}$ is not resolution consistent, there is a resolution derivation, call it D, of [ ] from $\{[X_1], \ldots, [X_n], [\beta], [\beta_2]\}$. All these disjunctions already occur as lines in the derivation from $\{[X_1], \ldots, [X_n], [\beta], [\beta_1, \beta_2]\}$ that we have produced thus far, and we can use these lines again, since we are not imposing a strictness requirement, so simply continue by adding the lines of D to produce a derivation of [ ] directly. □

Theorem 3.7.9 **(Completeness for Propositional Resolution)**
If X is a tautology, X has a resolution proof.

Proof Exactly as with tableaux: If X does not have a resolution proof, there is no closed resolution expansion for $\{\neg X\}$. Then $\{\neg X\}$ is resolution consistent, hence satisfiable by the Propositional Model Existence Theorem 3.6.2, so X is not a tautology. □

Just as before, we can change the definition of resolution consistency for a finite set S as follows: There is no closed resolution expansion for S in which all applications of the Propositional Resolution Rule are atomic. This does not affect the proof of Theorem 3.7.9, and so we have the stronger result: If X is a tautology, X has a resolution proof in which all applications of the Propositional Resolution Rule are atomic.

Exercises

3.7.1. Call a set U of propositional formulas *upward closed* if

1. $Z \in U \Rightarrow \neg\neg Z \in U$.
2. $\alpha_1 \in U$ and $\alpha_2 \in U \Rightarrow \alpha \in U$.
3. $\beta_1 \in U$ or $\beta_2 \in U \Rightarrow \beta \in U$.

Show that any set S has a smallest upward closed extension. (We call this the *upward closure* of S and denote it S^u.)

3.7.2. Using the notation of Exercise 3.7.1, show that for any sets of propositional formulas:

1. $(S^u)^u = S^u$.
2. $S_1 \subseteq S_2 \Rightarrow S_1^u \subseteq S_2^u$.
3. $S_1 \subseteq S_2^u \Rightarrow S_1^u \subseteq S_2^u$.
4. If L is a literal, $L \in S^u \Leftrightarrow L \in S$.
5. If $\neg\neg Z \notin S$ then $Z \in S^u \Leftrightarrow \neg\neg Z \in S^u$.
6. If $\alpha \notin S$ then $\alpha \in S^u \Leftrightarrow \alpha_1 \in S^u$ and $\alpha_2 \in S^u$.
7. If $\beta \notin S$ then $\beta \in S^u \Leftrightarrow \beta_1 \in S^u$ or $\beta_2 \in S^u$.

3.7.3. Let $\mathcal{C}$ be a collection of sets of propositional formulas. We call $\mathcal{C}$ a *strict* propositional consistency property if it meets the following conditions for each $S \in \mathcal{C}$:

1. For any propositional letter A, not both $A \in S$ and $\neg A \in S$.
2. $\bot \notin S$; $\neg\top \notin S$.
3. $\neg\neg Z \in S \Rightarrow S^- \cup \{Z\} \in \mathcal{C}$, where S^- is S with $\neg\neg Z$ removed.
4. $\alpha \in S \Rightarrow S^- \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$, where S^- is S with α removed.
5. $\beta \in S \Rightarrow S^- \cup \{\beta_1\} \in \mathcal{C}$ or $S^- \cup \{\beta_2\} \in \mathcal{C}$, where S^- is S with β removed.

Show the following: If $\mathcal{C}$ is a strict propositional consistency property, and $S \in \mathcal{C}$, then S is satisfiable.

Hint: Suppose $\mathcal{C}$ is a strict propositional consistency property. Let $\mathcal{C}^u = \{S^u \mid S \in \mathcal{C}\}$, and let $\mathcal{C}^*$ be the subset closure of $\mathcal{C}^u$. Show $\mathcal{C}^*$ is a propositional consistency property. Exercise 3.7.2 will be useful.

3.7.4. Use Exercise 3.7.3, and show every tautology has a *strict* tableau proof.

3.7.5. Complete the proof of Lemma 3.7.6

3.8 Completeness With Restrictions

We now know that every tautology has tableau and resolution proofs. But for implementation purposes, we need more than that; we need a way of finding a proof if one exists. Not every implementation of these proof procedures must do so. As a trivial example, in a tableau construction, if we have $\neg\neg Z$ on a branch, we are allowed to add Z. But there is nothing to say we can't apply this same rule a second time to $\neg\neg Z$, adding another occurrence of Z, then a third, and a fourth, and so on. This would be particularly stupid, but it is permitted. And if an implementation proceeds this way, it can run forever without finding a proof, though one may exist. But we have not yet established that forbidding reuse of formulas will leave us with a complete proof procedure. Indeed, it does not happen with certain non-classical logics, and there are complications even with first-order classical logic. Fortunately, in the classical propositional setting, strict tableau and resolution systems are complete. We prove this now. Our proofs will not use the Model Existence Theorem.

We begin with tableaux, whose completeness is rather easy to prove. Then we go on to resolution, which will require more work.

Recall, the strictness restriction is as follows: A Tableau Expansion Rule can be applied to a formula on a branch only once. Suppose we call a formula occurrence *used* on a branch provided it is not a literal, and a Tableau Expansion Rule has been applied to it on that branch. Then the restriction is as follows: We are forbidden to apply any rule to a used formula. We will prove a strong form of completeness that says any proof attempt that does everything, without violating the restriction on reusing used formula occurrences, must find a proof if one exists. We will even show the stronger result that a proof in which each branch is *atomically* closed must be found. It is this that justifies the tableau implementations in Section 3.2.

Theorem 3.8.1 *Suppose X is a tautology. A strict tableau construction process for $\{\neg X\}$ that is continued until every non-literal formula occurrence on every branch has been used must terminate and do so in an atomically closed tableau.*

Proof Construct a strict sequence of tableaux for $\{\neg X\}$, and continue until no further Tableau Expansion Rules are applicable. No matter how this is done, the process must terminate (see Exercise 3.8.1). Let $\mathbf{T}$ be the final tableau produced.

Suppose $\mathbf{T}$ is not atomically closed. Let θ be a branch of $\mathbf{T}$ that is not atomically closed. If $\neg\neg Z$ occurs on θ, since every non-literal formula occurrence has been used, Z must also occur on θ. If α occurs on θ, both α_1 and α_2 must occur. Likewise, if β occurs on θ, one of β_1 or β_2

must occur. It follows that the set of formulas occurring on θ, used or not, is a Hintikka set. By Hintikka's Lemma 3.5.2, this set is satisfiable. It includes $\neg X$ (since $\neg X$ occurs on every branch of $\mathbf{T}$). Hence, some Boolean valuation maps $\neg X$ to $\mathbf{t}$, so X is not a tautology. □

Corollary 3.8.2 *The tableau system provides a decision procedure for being a tautology.*

For resolution we need a dual counterpart of Hintikka's Lemma. Things are naturally more complicated, since we now must deal with sets of *clauses* rather than with sets of *formulas*. After some preliminary work, we introduce a notion of *Robinson set* and prove the analog we need.

Definition 3.8.3 Let $\mathbf{C}$ be a set of clauses. We say $\mathbf{C}$ is *resolution saturated*, provided the result of applying the Propositional Resolution Rule to members of $\mathbf{C}$ always produces another member of $\mathbf{C}$.

Proposition 3.8.4 *If* $\mathbf{C}$ *is resolution saturated and unsatisfiable, then the empty clause is in* $\mathbf{C}$.

Proof According to Exercise 3.8.2, we can assume, without loss of generality, that $\mathbf{C}$ contains no occurrences of $\bot$, and so any application of the Propositional Resolution Rule to members of $\mathbf{C}$ is nontrivial. So, from now on, assume $\mathbf{C}$ is a fixed resolution saturated set that is unsatisfiable and contains no occurrences of $\bot$.

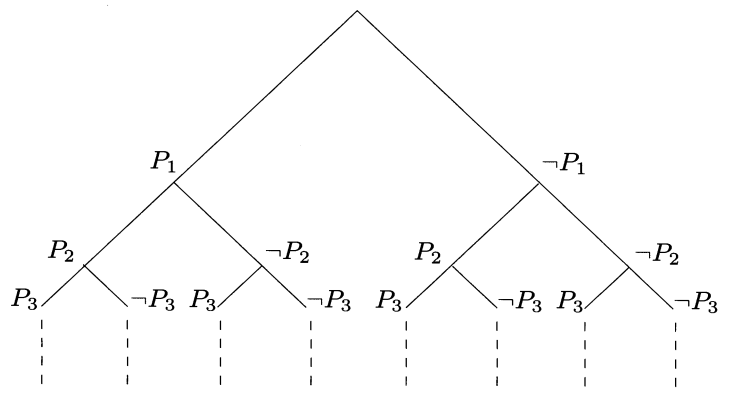


FIGURE 3.4. The Semantic Tree

The set of propositional letters is countable; let P_1, P_2, P_3,... be a listing of them. Now, consider the complete binary tree $\mathcal{T}$ displayed in

Figure 3.4. Following Robinson [43], where this proof has its origins, we call $\mathcal{T}$ a *semantic tree.* In $\mathcal{T}$ there is no label on the root node, and otherwise at level n, all left children are labeled P_n and all right children are labeled $\neg P_n$. Strictly speaking, there are many semantic trees, depending on the order in which the propositional letters are listed. In fact this makes no difference; any semantic tree will do for our purposes as well as any other. We keep the ordering of propositional letters fixed, use the tree displayed, and refer to it as *the* semantic tree.

A Boolean valuation is completely determined by its action on propositional letters, and each branch of $\mathcal{T}$ in effect assigns a value to each propositional letter, since each letter or its negation, but not both, appears on each branch. Consequently, there is a one-to-one correspondence between Boolean valuations and branches of $\mathcal{T}$: Pair up branch θ with the unique Boolean valuation v_θ assigning to every literal on θ the value $\mathbf{t}$.

A *path* in a tree is a sequence of nodes starting at the root and proceeding from parent to child; possibly terminating, possibly not. Maximal paths are called *branches.* Every branch is a path, though not every path is a branch. Also the following notation is handy (and standard): For a propositional letter P, let $\overline{P} = \neg P$ and $\overline{\neg P} = P$.

Let θ be a path in $\mathcal{T}$. We say θ *contradicts* a clause C if, for each literal $L \in C$, $\overline{L}$ occurs as a label on θ. Call a path $\mathbf{C}$-*closed* if it contradicts some clause in the resolution saturated set $\mathbf{C}$. Call a node N of $\mathcal{T}$ a *failure node* if the path from the origin to N is $\mathbf{C}$-closed. (Note that if a node is a failure node, so are its children.)

It is our intention to show the root node of $\mathcal{T}$ is a failure node. Since the only clause that the trivial path from the origin to itself can contradict is the empty clause, it will follow that the empty clause must be in $\mathbf{C}$, and we will be done.

Every branch of $\mathcal{T}$ must be $\mathbf{C}$-closed, because otherwise there would be some branch θ of $\mathcal{T}$ that did not contradict any clause in $\mathbf{C}$, and then it is easy to see the Boolean valuation v_θ corresponding to θ would satisfy $\mathbf{C}$.

Let θ be a branch of $\mathcal{T}$. Since θ is $\mathbf{C}$-closed, there is some clause $C \in \mathbf{C}$ that θ contradicts. Since C is finite, there must be a finite initial segment of θ that contradicts C. In other words, every branch of $\mathcal{T}$ has a finite initial segment that is $\mathbf{C}$-closed, and so every branch of $\mathcal{T}$ contains a failure node. Now let $\mathcal{T}^*$ be the subtree of $\mathcal{T}$ in which every descendant of a failure node has been deleted. If we can show that $\mathcal{T}^*$ is the trivial tree, consisting of just the root node, we will be done. To do this we suppose otherwise, and derive a contradiction.

Every branch of $\mathcal{T}^*$ is finite, and $\mathcal{T}^*$ is binary. It follows from König's Lemma 2.7.2 that $\mathcal{T}^*$ itself is finite. Since $\mathcal{T}^*$ is finite, it has a finite number of branches, and so there is one of maximal length. Since we are assuming $\mathcal{T}^*$ is nontrivial, a maximal length branch of $\mathcal{T}^*$ must end at a successor node. Say M is a maximal length branch, of the form θ, L, where L is a left child and θ is the path from the root of $\mathcal{T}^*$ to the parent of L. (The argument if M ends with a right child is similar.) Note that by the construction of $\mathcal{T}^*$, L is a failure node, θ, L is **C**-closed, but θ is not.

Let R be the right sibling of L in $\mathcal{T}$. R is also a failure node, for if it were not, the shortest **C**-closed path beginning with θ, R would be longer than θ, L, but θ, L is of maximal length in $\mathcal{T}^*$. Since θ is not **C**-closed, it follows that θ, R must also be a branch of $\mathcal{T}^*$.

Now, say the literal that labels node L is P and so $\neg P$ labels node R. Each branch θ, L and θ, R is **C**-closed and so contradicts some clause in **C**. Say θ, L contradicts C_L and θ, R contradicts C_R. It must be that $\neg P$ occurs in C_L, for if it did not, θ itself would already contradict C_L, but θ is not **C**-closed. Likewise, P must occur in C_R. Now, let C be the result of resolving C_L and C_R on P. It is easy to see that θ contradicts C. But **C** is resolution saturated, so $C \in \mathbf{C}$ and hence θ is **C**-closed after all.

We arrived at a contradiction by assuming $\mathcal{T}^*$ was nontrivial. Consequently, it is trivial, the root node of $\mathcal{T}$ is a failure node, and we are done. □

Definition 3.8.5 Let $\mathcal{R}$ be a set of disjunctions. We call $\mathcal{R}$ a *propositional Robinson set* if

1. For any member D of $\mathcal{R}$ that contains a non-literal, the results of applying at least one Resolution Expansion Rule to D are also in $\mathcal{R}$.
2. $\mathcal{R}$ is closed under the application of the Resolution Rule *to clauses.*
3. $\mathcal{R}$ does not contain the empty clause.

Theorem 3.8.6 *A propositional Robinson set is satisfiable.*

Proof Let $\mathcal{R}$ be a propositional Robinson set. If **C** is the set of clauses in $\mathcal{R}$, **C** is resolution saturated by part 2 of the definition. Then **C** is satisfiable by Proposition 3.8.4. Let v be a Boolean valuation satisfying **C**. We claim v satisfies the entire of $\mathcal{R}$.

In Definition 2.6.5 the notion of the rank of a propositional formula was introduced, and this was extended to generalized disjunctions in Section 2.6.5 by setting the rank of a disjunction to be the sum of the ranks of the individual propositional formulas. Note that under this definition the rank of a clause is 0.

We already know that v maps every member of $\mathcal{R}$ of rank 0 to $\mathbf{t}$. This is the start of an induction argument; we leave the rest to you in Exercise 3.8.3. □

Theorem 3.8.7 *Suppose X is a tautology. Strictly construct a sequence of resolution expansions for $\{\neg X\}$, beginning with applications of Resolution Expansion Rules, until no further applications are possible, followed by applications of the Resolution Rule to clauses until no new clauses are added. Such a process must terminate and do so with a closed resolution expansion.*

Proof We leave proof of termination to you as an exercise. Let $\mathcal{R}$ be the set consisting of those generalized disjunctions that occur at any stage of the process. If no closed resolution expansion is produced, $\mathcal{R}$ will be a Robinson set, hence satisfiable by Theorem 3.8.6. Since $\{[\neg X]\} \in \mathcal{R}$, $\{[\neg X]\}$ is satisfiable, and it follows that X is not a tautology. □

Resolution as restricted in the Theorem is, in fact, the traditional version and amounts to a first-stage conversion to clause form, then a second stage consisting entirely of Resolution Rule applications to clauses.

Exercises

3.8.1. Complete the proof of Theorem 3.8.1 by showing termination.

3.8.2. Let $\mathbf{C}$ be a set of clauses that is resolution saturated. Let $\mathbf{C}^\circ$ be the result of deleting all occurrences of $\bot$ from the clauses of $\mathbf{C}$. Prove the following:

1. $\mathbf{C}^\circ$ is resolution saturated.
2. $\mathbf{C}$ is satisfiable if and only if $\mathbf{C}^\circ$ is satisfiable.
3. $[\,] \in \mathbf{C}$ if and only if $[\,] \in \mathbf{C}^\circ$.

3.8.3. Complete the proof of Theorem 3.8.6 by doing the induction step. That is, show for each n each disjunction in $\mathcal{R}$ of rank n maps to $\mathbf{t}$ under v.

3.8.4. Complete the proof of Theorem 3.8.7 by proving termination.

3.9 Propositional Consequence

Often, instead of wanting to know whether something is a tautology, we want to know whether it follows from other formulas. Typically we ask: Is something a consequence of certain axioms? This is a question that becomes especially interesting when there are infinitely many axioms.

Definition 3.9.1 We say a propositional formula X is a *propositional consequence* of a set S of propositional formulas, and we write $S \models_p X$, provided X maps to $\mathbf{t}$ under every Boolean valuation that maps every member of S to $\mathbf{t}$.

Thus, $S \models_p X$ if X has to be true whenever the members of S are. The notion of propositional consequence directly generalizes what we have been studying: X is a tautology if and only if $\emptyset \models_p X$. Generally, we write $\models_p X$ instead of $\emptyset \models_p X$. The definition of propositional consequence allows the set S to be infinite, but in fact in any given instance, only a finite amount of the information in S will be needed.

Theorem 3.9.2 *$S \models_p X$ if and only if there is a finite subset S_0 of S such that $S_0 \models_p X$.*

Proof Suppose there is a finite subset S_0 of S such that $S_0 \models_p X$. Then $S \models_p X$ by Exercise 3.9.2, part 2. Conversely, suppose $S \models_p X$. Then by Exercise 3.9.2, part 3, $S \cup \{\neg X\}$ is not satisfiable. By the Propositional Compactness Theorem 3.6.3, some finite subset of $S \cup \{\neg X\}$ is not satisfiable. If a set is not satisfiable, neither is any extension of it, so we can assume $S \cup \{\neg X\}$ has a finite subset that is not satisfiable, and that subset includes $\neg X$. Such a set is of the form $S_0 \cup \{\neg X\}$ where S_0 is a finite subset of S. Now by Exercise 3.9.2, part 3 again, $S_0 \models_p X$. □

In principle then, for an infinite set S, to determine whether $S \models_p X$, we could systematically go through all finite subsets S_0 of S to determine whether $S_0 \models_p X$, and for finite sets S_0, Exercise 3.9.3 can be used to convert the problem to one on which resolution or tableaux can be used. A better way is to modify the resolution and tableau systems to allow the direct use of premises.

Definition 3.9.3 Let S be a set of formulas.

1. The *S-introduction rule for tableaux*: Any member X of S can be added to the end of any tableau branch. We write $S \vdash_{pt} X$ if there is a closed propositional tableau for $\{\neg X\}$, allowing the S-introduction rule for tableaux.
2. The *S-introduction rule for resolution*: $[X]$ can be added as a line to a resolution expansion, for any X in S. We write $S \vdash_{pr} X$ if there is a closed propositional resolution expansion for $\{\neg X\}$, allowing the S-introduction rule for resolution.

Theorem 3.9.4 **(Strong Soundness and Completeness)**
For any set S of propositional formulas, and any propositional formula X: $S \models_p X$ iff $S \vdash_{pt} X$ iff $S \vdash_{pr} X$.

Proof Recall the notion of satisfiability (Definition 2.4.5). Now modify that as follows: A tableau is *S-satisfiable* if every branch is S-satisfiable, and a branch is S-satisfiable if there is some Boolean valuation that maps every formula on the branch to $\mathbf{t}$ *and also maps every member of S to* $\mathbf{t}$. (Thus, $\emptyset$-satisfiability is equivalent to the standard version of satisfiability for tableaux.) Now, just as in Section 3.4, one can show that every Tableau Expansion Rule, and the S-introduction rule too, turns an S-satisfiable tableau into another S-satisfiable tableau. There are no closed S-satisfiable tableaux. Consequently if there is a closed tableau for $\{\neg X\}$ allowing the S-introduction rule, the initial tableau can not be S-satisfiable. This implies that $S \cup \{\neg X\}$ is not satisfiable, and hence $S \models_p X$. We have shown that $S \vdash_{pt} X$ implies $S \models_p X$, a soundness result.

For the completeness direction, the proof using the Model Existence Theorem adopts readily, though it applies only to tableaux without a strictness requirement. For a given formula X, call a set S of formulas *X--tableau inconsistent* if $S \vdash_{pt} X$; otherwise, call S *X--tableau consistent.* Now we need the following facts, whose proofs we leave to you:

1. For each X, the collection of X–tableau consistent sets is a propositional consistency property.

2. If S is X–tableau consistent, so is $S \cup \{\neg X\}$.

Now, suppose we do not have that $S \vdash_{pt} X$. Then S is X–tableau consistent. By item 2, $S \cup \{\neg X\}$ is also X–tableau consistent. But then by the Propositional Model Existence Theorem 3.6.2, and item 1, $S \cup \{\neg X\}$ is satisfiable, and hence we do not have that $S \models_p X$ by Exercise 3.9.2, part 3.

The part of the proof involving Resolution is similar and is left to you. □

Once we have Strong Soundness and Completeness, an alternative proof of Theorem 3.9.2 is possible. Suppose $S \models_p X$. By Strong Completeness we have $S \vdash_{pt} X$, and so there is a closed tableau for $\{\neg X\}$ using the S-introduction rule. A closed tableau must be finite, and so only a finite subset of S was actually used in the tableau, say it is the subset S_0. Then the tableau also shows that $S_0 \vdash_{pt} X$, and hence by Strong Soundness, $S_0 \models_p X$. We could have used resolution proofs just as well. The basic idea is that, whatever our proof procedure, a proof is a finite object and so can contain only a finite amount of information.

In the proof of Strong Soundness and Completeness, we defined a notion of X consistency for an arbitrary formula X. It is not hard to see that $\bot$ consistency is equivalent to tableau consistency, as defined in Section 3.7. The proof we gave does not apply if strictness conditions are imposed, but the result still holds, and proofs along the general lines of those in the preceding section are possible.

Since we are now dealing with deductions from sets of formulas that may be infinite, we can not expect implementations to always terminate. It can be shown that a *fair* implementation must succeed in finding a derivation if any exists. Loosely, a fair implementation is one that eventually applies any rule that is applicable. In particular it must eventually introduce each particular member of S, using the S-introduction rule.

Exercises

3.9.1. Prove the following:

1. $\{A \vee \neg B, B \vee \neg C, C \vee \neg D\} \models_p D \supset A$.
2. If $S = \{A_1 \supset A_2, A_2 \supset A_3, A_3 \supset A_4, \ldots\}$ then $S \models_p A_1 \supset A_n$ for each n.

3.9.2. Prove the following:

1. If $A, \neg A \in S$, then for any X, $S \models_p X$.
2. If $S \models_p X$ and $S \subseteq S^*$, then $S^* \models_p X$.
3. $S \models_p X$ if and only if $S \cup \{\neg X\}$ is not satisfiable.

3.9.3. Prove that $\langle A_1, \ldots, A_n \rangle \supset X$ is a tautology if and only if $\{A_1, \ldots, A_n\} \models_p X$.

3.9.4. Complete the proof of Theorem 3.9.4 by showing strong soundness and completeness of resolution.

3.9.5. Give an alternate proof of Theorem 3.9.4 by using Exercise 3.6.7.

4 Other Propositional Proof Procedures

4.1 Hilbert Systems

We have, so far, concentrated on tableaux and resolution as theorem-proving mechanisms. Since these are much better suited to automation than other approaches, the emphasis on them will continue throughout the book. But a wide variety of theorem-proving formalisms have been developed, based on different insights into the processes by which one recognizes that a formula expresses a logical truth. In this chapter we take a brief look at three formalisms in widespread use: axiom systems, natural deduction, and Gentzen sequents. We also consider the Davis-Putnam method, which, like tableaux and resolution, is especially suitable for automation purposes. This section is devoted to axiom systems, also called Hilbert systems or Frege systems. We will use the name *Hilbert systems.*

One can think of both the tableau and resolution mechanisms as involving a kind of backward reasoning. We start with the formula we are trying to prove, negate it, then break the result down into simpler and simpler parts until we arrive at an obvious contradiction. By contrast a Hilbert system embodies forward-reasoning principles. To prove a formula, one starts with known tautologies, derives immediate consequences, immediate consequences of the immediate consequences, and so on, until the desired formula is reached. Since the number of possible immediate consequences can grow explosively as work progresses, this approach does not lend itself to proof automation, or even to proof discovery for human beings. Still, once a Hilbert system proof of some tautology has been found, it is often easy to follow and explain to others, and it may provide insights that tableau or resolution arguments lack.

Classical logic is not the only logic of interest. For certain non-classical logics, only Hilbert-style formulations are known to exist. For this reason, for historical reasons, and for reasons just mentioned, Hilbert systems are widespread, and should be familiar to everyone who uses formal logic.

All Hilbert systems, for whatever logic, have the following features in common. Certain formulas are designated as *axioms.* Some *rules of derivation* or *rules of inference* are specified. These rules all say that some formulas 'follow from' others.

Definition 4.1.1 A *proof* in a Hilbert system is a finite sequence X_1, X_2, ..., X_n of formulas such that each term is either an axiom or follows from earlier terms by one of the rules of inference.

A more general notion than proof will be of use here, the notion of *derivation.*

Definition 4.1.2 A *derivation* in a Hilbert system from a set S of formulas is a finite sequence X_1, X_2,..., X_n of formulas such that each term is either an axiom, *or is a member of* S, or follows from earlier terms by one of the rules of inference.

A proof is simply a derivation from the empty set of formulas. It is customary to display proofs and derivations by writing a list of the formulas, one formula to a line. Consequently, from now on we will refer to a *line* of a proof or derivation, rather than to a term of it.

Definition 4.1.3 X is a *theorem* of a Hilbert system if X is the last line of a proof. X is a *consequence* of a set S if X is the last line of a derivation from S.

We write $S \vdash_{ph} X$ to symbolize that X has a derivation from S in the propositional Hilbert system called h. Instead of $\emptyset \vdash_{ph} X$, we will write $\vdash_{ph} X$; this corresponds to X being a theorem.

So far we have given general characteristics of Hilbert systems. If we are interested in classical propositional logic, we will also want every axiom to be a tautology and every rule of inference to produce only tautologies from tautologies. But even so, we have not been sufficiently restrictive. If we take as axioms all tautologies, and no rules of inference, we have a system meeting these conditions, and in it every tautology has a one-line proof. Clearly, this is not a very interesting system. Generally, the additional assumption is made that there are only a finite number of axioms, or else that there are an infinite number, but only a finite number of forms they can take on. For instance, $P \supset P$, $(P \wedge Q) \supset (P \wedge Q)$, and $\neg Q \supset \neg Q$ all have the common form $X \supset X$. If we want these, and other formulas of this form, to be axioms, it is customary to say $X \supset X$ is an *axiom scheme*, and any formula of this form is an axiom. In general, we will allow our axioms to be infinite in number, but they must be specifiable by a finite number of axiom schemes. Incidentally, we will adopt the convention that P, Q,... are *propositional letters*, while X, Y,... are arbitrary *formulas*. This makes the presentation of axiom schemes somewhat simpler. It is a convention we used informally earlier in the paragraph.

We also want a finite number of rules of inference, and we want them to be structural in the same sense that axiom schemes are. For instance, one possible rule of inference follows: If X and Y are arbitrary formulas, the formula $X \wedge Y$ follows from X and Y. Such a rule would allow us to obtain $P \wedge Q$ from P and Q, and also $(P \supset R) \wedge \neg Q$ from $P \supset R$ and $\neg Q$. Such a rule is stated schematically as follows:

$$\frac{X \quad Y}{X \wedge Y}.$$

As it happens, this will not be a rule of inference adopted here, but it illustrates the kind of thing we intend.

One rule of inference that is often used, but that we will not adopt, is a *rule of substitution.* Such a rule states that a formula Y follows from a formula X, provided Y is the result of uniformly replacing the propositional letters of X by arbitrary formulas. For example, $(P \wedge Q) \supset (P \wedge Q)$ follows from $P \supset P$ by this rule. Exercise 2.4.7 shows that the rule of substitution meets the necessary condition of producing tautologies from tautologies. If a rule of substitution is used, then a finite set of axiom schemes can be replaced by a finite set of axioms. For various technical reasons, we find it simpler to use axiom schemes and omit a rule of substitution.

The most common rule of inference is *Modus Ponens.* We adopt this as the only rule of inference in this section.

Modus Ponens

$$\frac{X \quad X \supset Y}{Y}$$

Modus Ponens is stated in terms of the connective $\supset$, and this connective will play a special role in the axiom schemes we have chosen. Indeed, the first two schemes involve only this connective.

Axiom Scheme 1 $X \supset (Y \supset X)$

Axiom Scheme 2 $(X \supset (Y \supset Z)) \supset ((X \supset Y) \supset (X \supset Z))$

More axiom schemes will come, but already we can give an example of a proof in the system we have thus far.

Example $P \supset P$ is a theorem. The following is a proof.

1. $(P \supset ((P \supset P) \supset P)) \supset ((P \supset (P \supset P)) \supset (P \supset P))$
2. $P \supset ((P \supset P) \supset P)$
3. $(P \supset (P \supset P)) \supset (P \supset P)$
4. $P \supset (P \supset P)$
5. $P \supset P$

Here 1 is an instance of Axiom Scheme 2, taking X to be P, Y to be $P \supset P$, and Z to be P; 2 is an instance of Axiom Scheme 1, taking X to be P and Y to be $P \supset P$; 3 follows from 1 and 2 by Modus Ponens; 4 is an instance of axiom scheme 1, taking X and Y to be P. Finally, 5 follows from 3 and 4 by Modus Ponens.

If we had written W in place of P in this proof, we would have a proof outline that shows how to construct a proof of $W \supset W$ for any particular formula we might want to put in place of W. We will call such an outline a *proof scheme.* Clearly, it is more efficient to give proof schemes rather than proofs, and this is what we will generally do from now on.

A very important result (that is due independently to Tarski and to Herbrand) can be established about the system thus far constructed. It says that there is a proof of $X \supset Y$, provided there is a derivation of Y from $\{X\}$. The proof of this result is as important as its statement; the proof is constructive and shows how to turn a derivation of Y from $\{X\}$ into a formal proof of $X \supset Y$. Since such derivations are often easier to discover, this means we can shortcut much of the work of producing proofs in a Hilbert system.

Theorem 4.1.4 **(Deduction Theorem)** *In any Hilbert system h with at least Axiom Schemes 1 and 2, and with Modus Ponens as the only rule of inference,* $S \cup \{X\} \vdash_{ph} Y$ *if and only if* $S \vdash_{ph} (X \supset Y)$.

Proof The argument from right to left is trivial; we concentrate on the other direction. Suppose $S \cup \{X\} \vdash_{ph} Y$; say $Z_1, Z_2, \ldots, Z_n$ is a derivation of Y from $S \cup \{X\}$ (let us call it *Derivation One*). In it, each line is either one of the axioms, a member of $S \cup \{X\}$, or comes from earlier lines by Modus Ponens; and $Z_n = Y$. We show how to convert this into a derivation (call it *Derivation Two*) showing that $S \vdash_{ph} (X \supset Y)$.

First, prefix each formula of Derivation One with $X \supset$, forming the sequence: $X \supset Z_1$, $X \supset Z_2, \ldots$, $X \supset Z_n$. This sequence ends with the desired formula $X \supset Y$, since $Z_n = Y$, and so it is a preliminary version of Derivation Two. But it is not necessarily a legal derivation. To turn it into one, we insert some extra lines, as follows.

If Z_i is an axiom or a member of S, then in the Derivation Two candidate, just before $X \supset Z_i$, insert the formulas Z_i (an axiom or a member of S) and $Z_i \supset (X \supset Z_i)$ (an axiom). Note that $X \supset Z_i$ follows from these by Modus Ponens.

If Z_i is the formula X, then in the Derivation Two candidate, insert the steps of a proof of $X \supset X$ just before $X \supset Z_i (= X \supset X)$.

If Z_i comes from earlier terms of Derivation One by Modus Ponens, then there must be Z_j and Z_k with $j, k < i$ and where $Z_k = Z_j \supset Z_i$. In the Derivation Two candidate, there will be corresponding lines $X \supset Z_j$ and $X \supset Z_k = X \supset (Z_j \supset Z_i)$. Now, insert just before $X \supset Z_i$ the formulas $(X \supset (Z_j \supset Z_i)) \supset ((X \supset Z_j) \supset (X \supset Z_i))$ (an axiom) and $(X \supset Z_j) \supset (X \supset Z_i)$ (which follows from earlier terms by Modus Ponens). Now $X \supset Z_i$ also follows from earlier terms by Modus Ponens.

Call the resulting sequence *Derivation Two.* It is easy to see it constitutes a derivation of $X \supset Y$ from S. □

Example $(P \supset (Q \supset R)) \supset (Q \supset (P \supset R))$ is a theorem, and it is easy to argue for this using the Deduction Theorem, as follows. First, $\{P \supset (Q \supset R), Q, P\} \vdash_{ph} R$; in fact, the following is a derivation:

1. $P \supset (Q \supset R)$
2. P
3. $Q \supset R$
4. Q
5. R

Using the Deduction Theorem, it follows that $\{P \supset (Q \supset R), Q\} \vdash_{ph} P \supset R$, and again $\{P \supset (Q \supset R)\} \vdash_{ph} Q \supset (P \supset R)$, and finally, $\vdash_{ph} (P \supset (Q \supset R)) \supset (Q \supset (P \supset R))$.

Now we want to add axiom schemes to introduce the other (primary) connectives. We make use of uniform notation here, to give a compact presentation.

Axiom Scheme 3 $\bot \supset X$

Axiom Scheme 4 $X \supset \top$

Axiom Scheme 5 $\neg\neg X \supset X$

Axiom Scheme 6 $X \supset (\neg X \supset Y)$

Axiom Scheme 7 $\alpha \supset \alpha_1$

Axiom Scheme 8 $\alpha \supset \alpha_2$

Axiom Scheme 9 $(\beta_1 \supset X) \supset ((\beta_2 \supset X) \supset (\beta \supset X))$

Axiom Scheme 6 can be weakened by requiring that X be atomic. It can be shown that each instance of the unrestricted version is a consequence in the resulting Hilbert system.

Example $(\neg X \supset X) \supset X$ is a theorem (for any choice of formula X). In Axiom Scheme 9, if we take β to be $\neg X \supset X$, then the scheme reads $(\neg\neg X \supset X) \supset ((X \supset X) \supset ((\neg X \supset X) \supset X))$. $\neg\neg X \supset X$ is Axiom Scheme 5, and $X \supset X$ is provable. Then $(\neg X \supset X) \supset X$ follows using Modus Ponens.

For the rest of this section, h is the Hilbert system with Axiom Schemes 1 through 9 and the rule of Modus Ponens. It is time to establish soundness and completeness for the Hilbert system h. Soundness is easy. Every axiom is a tautology (a fact that is easily checked). Also, if X and $X \supset Y$ are tautologies, so is Y, hence, the rule of Modus Ponens produces only tautologies from tautologies. It follows easily that every line of a proof is a tautology; in particular the last line. This argument extends easily to derivations as well; we do not give details. Thus, we have the following:

Theorem 4.1.5 **(Strong Hilbert Soundness)**
If $S \vdash_{ph} X$, then $S \models_p X$.

To show completeness we use our favorite tool, the Model Existence Theorem 3.6.2, and we follow a pattern that worked for tableaux and resolution.

Definition 4.1.6 Let X be a propositional formula. Call a set S of formulas *X--Hilbert inconsistent* if $S \vdash_{ph} X$; call S *X--Hilbert consistent* otherwise.

Lemma 4.1.7 *For each formula X, the collection of all X--Hilbert consistent sets is a propositional consistency property.*

Proof Several separate items must be checked. We consider a few in detail and leave the rest to you.

Suppose S is X–Hilbert consistent; we show $\neg\top \notin S$. Or rather, suppose $\neg\top \in S$; we show S is X–Hilbert inconsistent. Well briefly, since $\neg\top \in S$, $S \vdash_{ph} \neg\top$. Since $\neg\top \supset \top$ is an axiom (Axiom Scheme 4), it follows that $S \vdash_{ph} \top$. Finally, $\top \supset (\neg\top \supset X)$ is an axiom (Axiom Scheme 6), and hence $S \vdash_{ph} X$.

We also check the β-condition, and once again it is simplest to show the contrapositive. Suppose $S\cup\{\beta_1\}$ and $S\cup\{\beta_2\}$ are X–Hilbert inconsistent; we show $S \cup \{\beta\}$ is X–Hilbert inconsistent. Well, we are given that $S\cup\{\beta_1\} \vdash_{ph} X$, so by the Deduction Theorem, $S \vdash_{ph} \beta_1 \supset X$. Similarly $S \vdash_{ph} \beta_2 \supset X$. Also $(\beta_1 \supset X) \supset ((\beta_2 \supset X) \supset (\beta \supset X))$ is an instance of Axiom Scheme 9, so it follows that $S \vdash_{ph} \beta \supset X$, and hence that $S \cup \{\beta\} \vdash_{ph} X$. □

Theorem 4.1.8 **(Strong Hilbert Completeness)**
If $S \models_p X$, then $S \vdash_{ph} X$.

Proof As usual, we show the contrapositive. Suppose we do not have that $S \vdash_{ph} X$. Then S is X–Hilbert consistent. It follows that $S \cup \{\neg X\}$ is also X–Hilbert consistent, for if not, $S \cup \{\neg X\} \vdash_{ph} X$, hence $S \vdash_{ph} \neg X \supset X$, and it would follow that $S \vdash_{ph} X$, since, as we have shown, $(\neg X \supset X) \supset X$ is a theorem. Now, by Lemma 4.1.7, and the Propositional Model Existence Theorem 3.6.2, $S \cup \{\neg X\}$ is satisfiable, and hence, we do not have that $S \models_p X$. □

There are many different Hilbert systems for classical propositional logic besides the one we have been considering. Frege's system took implication and negation as the only connectives, had Modus Ponens and Substitution as rules of inference, and had the following six axioms: $P \supset (Q \supset P)$, $(R \supset (Q \supset P)) \supset ((R \supset Q) \supset (R \supset P))$, $(R \supset (Q \supset P)) \supset (Q \supset (R \supset P))$, $(Q \supset P) \supset (\neg P \supset \neg Q)$, $\neg\neg P \supset P$ and $P \supset \neg\neg P$. The very influential system of *Principia Mathematica* [56] took negation and disjunction as connectives, with implication defined. The rules of inference were Substitution and a version of Modus Ponens: from X and $\neg X \vee Y$ to conclude Y. The axioms were the following: $(P\vee P) \supset P$, $Q \supset (P\vee Q)$, $(P\vee Q) \supset (Q\vee P)$, $(P\vee(Q\vee R)) \supset (Q\vee(P\vee R))$, and $(Q \supset R) \supset ((P \vee Q) \supset (P \vee R))$. In fact, the fourth of these axioms can be derived from the rest, though it was some time before this was discovered. As a kind of extreme in this area, consider the Hilbert system with only the $\uparrow$ connective, the rule of inference: Z follows from $X \uparrow (Y \uparrow Z)$ and X; and the formula in part 8 of Exercise 3.1.1 as its only axiom scheme. This is sound and complete (the other connectives can all be defined from $\uparrow$).

We gave the axioms we did to introduce all the Primary Connectives at once and to prove completeness with a minimum of work. Consequently we have a rather large number of axiom schemes (Schemes 7 through 9 are really shorthand patterns, with a different actual scheme for each connective.) In fact, all connectives can be defined from $\neg$ and $\supset$, and if we choose to do so, the variety of axiom schemes can be considerably minimized. Exercises 4.1.8 and 4.1.9 show ways of doing this.

Exercises

4.1.1. Using the proof of the Deduction Theorem, convert the derivation showing $\{P \supset (Q \supset R), Q, P\} \vdash_{ph} R$ into a direct proof of $(P \supset (Q \supset R)) \supset (Q \supset (P \supset R))$.

4.1.2. Give a proof in Hilbert system h of $(\neg X \supset \bot) \supset X$.

4.1.3. We can give an alternate definition of Hilbert system theorem as follows. A *theorem* is a member of the smallest set of formulas that contains all the axioms and that contains Y whenever it contains X and $X \supset Y$. Prove the two definitions of theorem are equivalent.

4.1.4. Suppose we place a restriction on Axiom Scheme 6, that X must be atomic. Show by induction on the rank of X that $X \supset (\neg X \supset Y)$ has a proof in the resulting Hilbert system, for arbitrary X.

4.1.5. Show (without using Completeness) that $Z \supset \neg\neg Z$ is a theorem of the Hilbert system h. Hint: Use Axiom Scheme 9, with $\beta = \neg Z \supset \neg Z$ and $X = Z \supset \neg\neg Z$.

4.1.6. Interestingly enough, the two axiom schemes for implication, Axiom Schemes 1 and 2, together with Modus Ponens, do *not* characterize the implication of classical logic. Here is one way of showing this. We move from the two-valued version of $\supset$ given in Table 2.1 to a three-valued version by introducing a "middle" truth value, $\mathbf{m}$ (read it as "maybe"). Now, use the following table.

$\supset$	f	m	t
f	t	t	t
m	f	t	t
t	f	m	t

Define a *3-valuation* to be a mapping v from the set of propositional formulas to the set $\{\mathbf{f}, \mathbf{m}, \mathbf{t}\}$ such that $v(X \supset Y) = v(X) \supset v(Y)$, where $\supset$ on the right is given by the table. Call a propositional formula X a *3-tautology* if $v(X) = \mathbf{t}$ for every 3-valuation v. Now show the following:

1. Every instance of Axiom Schemes 1 and 2 is a 3-tautology.

2. If X and $X \supset Y$ are 3-tautologies, so is Y.
3. Every formula provable using Axiom Schemes 1 and 2 and Modus Ponens is a 3-tautology.
4. *Pierce's law*, $((P \supset Q) \supset P) \supset P$, is not a 3-tautology.
5. Pierce's law is a classical tautology.

(Remarks. It can be shown that Modus Ponens, Axiom Schemes 1 and 2, *and* Pierce's law do axiomatize classical implication. Without Pierce's law, what we get is the implication of a well-known non-classical logic, *intuitionistic logic.* This exercise continues in Exercise 4.2.2.)

4.1.7. Complete the proof of Lemma 4.1.7.

4.1.8. Consider the Hilbert system with $\neg$ and $\supset$ as primitive and with Modus Ponens and Axiom Schemes 1, 2, 5, 6, and $(\neg X \supset X) \supset X$. This system is complete. Show this by giving proofs in this system of the following:

1. $(\neg Y \supset \neg X) \supset (X \supset Y)$.
2. $(X \supset Y) \supset (\neg Y \supset \neg X)$.
3. $\neg(X \supset Y) \supset X$ (Axiom Scheme 7).
4. $\neg(X \supset Y) \supset \neg Y$ (Axiom Scheme 8).
5. $(\neg X \supset Z) \supset ((Y \supset Z) \supset ((X \supset Y) \supset Z))$ (Axiom Scheme 9).

4.1.9. Consider the Hilbert system with $\neg$ and $\supset$ as primitive, and with Modus Ponens, Axiom Schemes 1, 2, and $(\neg Y \supset \neg X) \supset ((\neg Y \supset X) \supset Y)$. Show this system is complete by using Exercise 4.1.8 and giving proofs of the following:

1. $(\neg X \supset X) \supset X$.
2. $\neg\neg X \supset X$ (Axiom Scheme 5).
3. $X \supset (\neg X \supset Y)$ (Axiom Scheme 6).

4.2 Natural Deduction

Natural deduction systems constitute another family of proof mechanisms, intended to formalize the kind of reasoning people do in informal arguments. They are based on the idea of *subordinate proofs*, in which one derives conclusions from premises, then *discharges* those premises to produce assumption-free results. We will give an example of such a proof shortly, but first we introduce some typical rules and a mechanism for displaying subordinate proofs. Many mechanisms exist in the literature; we will simply write them in boxes, with the first line inside a box being the particular assumption made in that subordinate proof, and the first line below the box being the result of discharging the assumption.

A typical rule of many natural deduction systems follows: If one can derive Y from X as an assumption, then one can discharge the assumption X and conclude that one has proved $X \supset Y$. This is given schematically in Figure 4.1.

$$\boxed{\begin{array}{c} X \\ \vdots \\ Y \end{array}}$$

$$X \supset Y$$

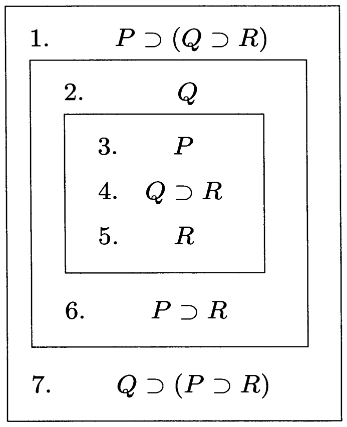
FIGURE 4.1. A Natural Deduction Rule for Implication

Another typical example is the Modus Ponens Rule. But this must be formulated with some care. We do not want to use it to derive a conclusion from an assumption made earlier in a proof that has now been discharged. If we think of a proof as being constructed in *stages*, it is simple to say which formulas we are allowed to consider at each stage.

Definition 4.2.1 The formulas *active* at a stage in a proof are those occurring in boxes that have not closed by this stage.

Now the Modus Ponens Rule becomes the following: From X and $X \supset Y$, conclude Y, provided both X and $X \supset Y$ are active.

Note that our rules so far have been paired: one for *introducing* the connective $\supset$ and one for using it, in effect, *eliminating* it. This is a common and important principle behind the organization of natural deduction systems.



8. $(P \supset (Q \supset R)) \supset (Q \supset (P \supset R))$

FIGURE 4.2. Proof of $(P \supset (Q \supset R)) \supset (Q \supset (P \supset R))$

Example We have enough machinery to give a simple example of a natural deduction proof. The argument in Figure 4.2 should be compared with the Hilbert system proof in Section 4.1, which used the Deduction Theorem 4.1.4. Here 1 through 3 are assumptions, each starting a subordinate proof; 4 is from 1 and 3 by Modus Ponens (note that at this point no boxes have closed, so 1 and 3 are both active); and 5 is from 2 and 4), again by Modus Ponens. Now a box is closed, assumption 3 is discharged, to conclude 6. Note that formulas 3 through 5 are no longer active. Two more assumption discharges produce 7 and 8.

Many different natural deduction systems are described in the literature. Prawitz [36], for instance, has a particularly elegant one, together with an important analysis of its proof theory. The system we have chosen to give does not actually have the rules just considered, though it has some that are very close. And it contains redundancies. It is designed to make use of uniform notation so that all Primary Connectives can be brought in smoothly and to lead to a quick proof of completeness.

Constant Rules $\dfrac{\bot}{X}$ $\qquad \dfrac{}{\top}$

Negation Rules

$$\frac{\begin{array}{c} X \\ \neg X \end{array}}{\bot} \qquad \begin{array}{c} \boxed{\begin{array}{c} X \\ \vdots \\ \bot \end{array}} \\ \neg X \end{array} \qquad \begin{array}{c} \boxed{\begin{array}{c} \neg X \\ \vdots \\ \bot \end{array}} \\ X \end{array}$$

Primary Connective Rules

α**E** $$\frac{\alpha}{\alpha_1} \qquad \frac{\alpha}{\alpha_2}$$

α**I** $$\frac{\begin{array}{c} \alpha_1 \\ \alpha_2 \end{array}}{\alpha}$$

β**E** $$\frac{\begin{array}{c} \neg\beta_1 \\ \beta \end{array}}{\beta_2} \qquad \frac{\begin{array}{c} \neg\beta_2 \\ \beta \end{array}}{\beta_1}$$

β**I** $$\begin{array}{c} \boxed{\begin{array}{c} \neg\beta_1 \\ \vdots \\ \beta_2 \end{array}} \\ \beta \end{array} \qquad \begin{array}{c} \boxed{\begin{array}{c} \neg\beta_2 \\ \vdots \\ \beta_1 \end{array}} \\ \beta \end{array}$$

We have used **E** and **I** in rule names to suggest *elimination* and *introduction*. Note that one of the Constant Rules is a no-premise rule. Also some of the rules have two premises; in these, order of premises does not matter. In applying such rules, the premises must be active. Before we consider examples of proofs of formulas in this system, it will be useful to introduce the notion of a derived rule. Loosely, a rule is *derived* if its addition does not change the strength of the system. More precisely, a rule is derived if we can translate any use of it away. Two very useful derived rules in this natural deduction system are the following:

$$\frac{\neg\neg X}{X} \qquad \frac{X}{\neg\neg X}$$

To show the first of these rules is derived, suppose line (n) in a proof is the formula $\neg\neg X$. Then, without using the proposed new negation rules, we may proceed as follows:

$$\begin{array}{ll} & \vdots \\ (n) & \neg\neg X \\ \multicolumn{2}{|l|}{\begin{array}{ll} (n+1) & \neg X \\ (n+2) & \bot \end{array}} \\ (n+3) & X \end{array}$$

Here line $n+1$ is an assumption. Line $n+2$ follows from n and $n+1$ by the first of the official negation rules. Then line $n+3$ follows using the third negation rule. Thus, we have added X without using any rules other than the basic ones; consequently, a rule allowing us to add X directly when $\neg\neg X$ is present is a derived rule. The other double negation rule can be shown to be derived in a similar way.

From now on, we will use the rules for introducing and eliminating double negations frequently, often without comment.

In the βE rules, suppose we take β to be $X \supset Y$, so that $\beta_1 = \neg X$ and $\beta_2 = Y$. Then the rules become the following:

$$\frac{\begin{array}{c}\neg\neg X \\ X \supset Y\end{array}}{Y} \qquad \frac{\begin{array}{c}\neg Y \\ X \supset Y\end{array}}{\neg X}$$

The second of these is a common rule, called *Modus Tollens.* The first is almost Modus Ponens and has the effect of it when used in conjunction with a double negation rule. Likewise the βI rules become the following:

$$\begin{array}{c} \boxed{\begin{array}{c}\neg\neg X \\ \vdots \\ Y\end{array}} \\ X \supset Y \end{array} \qquad \begin{array}{c} \boxed{\begin{array}{c}\neg Y \\ \vdots \\ \neg X\end{array}} \\ X \supset Y \end{array}$$

The first of these is almost the rule for introducing implication that we considered earlier; again, a double negation rule is also needed. The second embodies the principle of contraposition.

Example Figure 4.3 contains a proof of $\neg(P \wedge Q) \supset (\neg P \vee \neg Q)$. In it 1 and 2 are assumptions. Taking β to be $\neg(P \wedge Q)$, the first βE rule says; from $\neg\neg P$ and $\neg(P \wedge Q)$, conclude $\neg Q$. Using this, 3 follows from 1 and 2. Now, taking β to be $\neg P \vee \neg Q$, the first βI rule says; conclude $\neg P \vee \neg Q$ from a derivation of $\neg Q$ from $\neg\neg P$. This allows us to derive 4. Finally, 5 follows, again using βI.

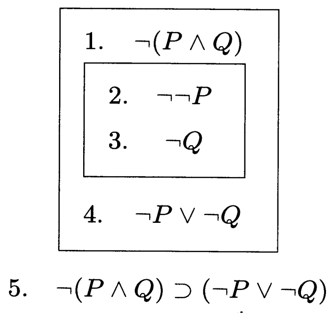


FIGURE 4.3. A Natural Deduction Proof of $\neg(P \wedge Q) \supset (\neg P \vee \neg Q)$

Just as with earlier proof mechanisms, we can introduce a notion of derivation, as well as of proof.

Definition 4.2.2 A natural deduction *derivation* of X from a set S of formulas meets the conditions for being a proof of X but also allows the following additional rule: At any stage, any member of S may be used as a line. We write $S \vdash_{pn} X$ to indicate there is a derivation of X from S in the propositional natural deduction system of this section.

Theorem 4.2.3 **(Natural Deduction Soundness)**
If $S \vdash_{pn} X$, then $S \models_p X$.

Proof Unfinished Hilbert system proofs are, themselves, proofs (though of something else). An unfinished natural deduction proof is not a natural deduction proof, since there will be premises not yet discharged, boxes not yet closed. A soundness proof must take this into account.

Suppose we have a possibly unfinished natural deduction derivation, from a set S. Say the last line contains the formula Z, and at this stage the assumptions $A_1, \ldots, A_k$ are still active, not having been discharged. We *associate* with this incomplete derivation the assertion $S \cup \{A_1, \ldots, A_k\} \models_p Z$. Now, if it could be shown that the assertion associated with every derivation from S (incomplete or not) is always correct, the soundness result would follow immediately. This is done by an induction on the lengths of incomplete derivations. We leave the details to you as Exercise 4.2.3. □

For completeness we proceed much as we did with earlier systems.

Definition 4.2.4 Let X be a propositional formula. Call a set S *X--natural deduction inconsistent* if $S \vdash_{pn} X$; otherwise call S *X--natural deduction consistent.*

Lemma 4.2.5 *For each formula X, the collection of all X--natural deduction consistent sets is a propositional consistency property.*

Theorem 4.2.6 **(Strong Natural Deduction Completeness)** *If $S \models_p X$, then $S \vdash_{pn} X$.*

Exercises

4.2.1. Give natural deduction proofs of the following:

1. $X \supset (Y \supset X)$.
2. $(X \supset (Y \supset Z)) \supset ((X \supset Y) \supset (X \supset Z))$.
3. $(\neg Y \supset \neg X) \supset ((\neg Y \supset X) \supset Y)$.
4. $((X \supset Y) \supset X) \supset X$.
5. $(X \wedge (Y \vee Z)) \supset ((X \wedge Y) \vee (X \wedge Z))$.
6. $(X \supset Y) \supset (\neg(Y \vee Z) \supset \neg(X \vee Z))$.
7. $(P \uparrow P) \uparrow P$.
8. $\neg((P \downarrow Q) \downarrow (P \vee Q))$.
9. $(\neg P \downarrow \neg Q) \subset \neg(P \uparrow Q)$.

4.2.2. At the beginning of the section, we considered two simple natural deduction rules for implication that did not become part of our "official" system. Continuing Exercise 4.1.6, we ask you to show that these two rules do not characterize the implication of classical logic. First, we re-formulate them using the following notation. We write $S \vdash X$ to mean X is deducible, using the two implication rules, when S is the set of premises. With this notation, the rules can be stated equivalently as follows:

$$\frac{S \vdash X \quad S \vdash X \supset Y}{S \vdash Y} \qquad \frac{S, X \vdash Y}{S \vdash X \supset Y} \qquad \frac{}{S, X \vdash X}$$

Now, use the 3-valued system of Exercise 4.1.6, define an ordering by setting $\mathbf{f} < \mathbf{m} < \mathbf{t}$, and extend 3-valuations to sets of formulas by taking $v(S) = \min\{v(X) \mid X \in S\}$ (as a special case, $v(\emptyset) = \mathbf{t}$). Call $S \vdash X$ *3-valid* if, for every 3-valuation v, $v(S) \leq v(X)$. Now, show the following:

1. $S, X \vdash X$ is 3-valid.
2. If $S \vdash X$ and $S \vdash X \supset Y$ are 3-valid, so is $S \vdash Y$.
3. If $S, X \vdash Y$ is 3-valid, so is $S \vdash X \supset Y$.
4. If $\emptyset \vdash X$ is provable in this system, X is 3-valid.
5. Pierce's law is not provable in this system.

(Remark. It can be shown that this system is equivalent to the axiom system considered in Exercise 4.1.6.)

4.2.3. Complete the proof of the Soundness Theorem 4.2.3.

4.2.4. Verify Lemma 4.2.5, and use it to prove the Strong Completeness Theorem.

4.3 The Sequent Calculus

The Gentzen sequent calculus can be looked at as an intermediary between semantic tableaux and natural deduction systems. Historically, both a natural deduction system and the sequent calculus can be found in Gentzen's fundamental paper [22]. Indeed, the sequent calculus itself is sometimes referred to in the current literature as a natural deduction system, though we find this confusing and will not do so here.

It will be simplest if we limit the binary connectives to those actually considered by Gentzen: $\wedge$, $\vee$, and $\supset$. Once the *signed tableau* system has been introduced, it will be a simple matter for you to devise rules for the other connectives. So for the rest of this section, formulas are limited to these binary connectives (and, of course, $\neg$, $\top$, and $\bot$).

Definition 4.3.1 A *sequent* is a pair $\langle \Gamma, \Delta \rangle$ of finite sets of formulas.

This definition is a technical one. In practice we use the suggestive notation introduced by Gentzen: the sequent $\langle \Gamma, \Delta \rangle$ will be written $\Gamma \rightarrow \Delta$. The arrow suggests a kind of implication, and that is, indeed, the intention. We also introduce some useful notational abbreviations. Instead of writing $\{A_1, \ldots, A_n\} \rightarrow \{B_1, \ldots, B_k\}$, we will write $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_k$. For a single formula X, and sets Γ and Δ of formulas, we will write $\Gamma, X \rightarrow \Delta$ instead of $\Gamma \cup \{X\} \rightarrow \Delta$, and so on. Likewise, we will write $\rightarrow \Delta$ for $\emptyset \rightarrow \Delta$, and similarly for other occurrences of $\emptyset$. Generally, we follow the convention that capital Latin letters stand for formulas, while capital Greek letters stand for finite sets of formulas.

We have made a minor departure from Gentzen in our definition of sequent. For Gentzen a sequent was a pair of *lists* of formulas, not sets of formulas. Gentzen included structural rules for rearranging lists and for dealing with repetitions. By using sets we have avoided this, though the issue becomes significant if a computer representation of sequent is to be chosen.

As we remarked, the arrow should be thought of as a kind of implication, 'follows from'. Think of a sequent as asserting the following: If all the formulas on the left of the arrow are true, then at least one of the formulas on the right is also true. Technically, we extend Boolean valuations to sequents as follows:

Definition 4.3.2 $v(\Gamma \rightarrow \Delta) = \mathbf{t}$ if $v(X) = \mathbf{f}$ for some $X \in \Gamma$ or $v(Y) = \mathbf{t}$ for some $Y \in \Delta$.

Note that under this definition, $v(\rightarrow) = \mathbf{f}$ and $v(\rightarrow X) = v(X)$.

In the sequent calculus, certain very simple sequents are taken as axioms, and there are rules for deriving new sequents from old. The resulting system is not simply a Hilbert system, though. For one thing, the arrow is not a connective but a 'metalogical' symbol. Thus, while $P \rightarrow (Q \supset R)$ is legal, $P \rightarrow (Q \rightarrow R)$ is not. This restriction allows the axioms and rules of the sequent calculus to be particularly simple, and a deep analysis of formal proofs is possible. Now, the axioms and rules are as follows:

Axioms $X \rightarrow X$

$\bot \rightarrow$

$\rightarrow \top$

Structural Rule, Thinning If $\Gamma_1 \subseteq \Gamma_2$ and $\Delta_1 \subseteq \Delta_2$ then

$$\frac{\Gamma_1 \rightarrow \Delta_1}{\Gamma_2 \rightarrow \Delta_2}$$

Negation Rules

$$\frac{\Gamma \rightarrow \Delta, X}{\Gamma, \neg X \rightarrow \Delta} \qquad \frac{\Gamma, X \rightarrow \Delta}{\Gamma \rightarrow \Delta, \neg X}$$

Conjunction Rules

$$\frac{\Gamma, X, Y \rightarrow \Delta}{\Gamma, X \wedge Y \rightarrow \Delta} \qquad \frac{\Gamma \rightarrow \Delta, X \quad \Gamma \rightarrow \Delta, Y}{\Gamma \rightarrow \Delta, X \wedge Y}$$

Disjunction Rules

$$\frac{\Gamma, X \to \Delta \quad \Gamma, Y \to \Delta}{\Gamma, X \vee Y \to \Delta} \qquad \frac{\Gamma \to \Delta, X, Y}{\Gamma \to \Delta, X \vee Y}$$

Implication Rules

$$\frac{\Gamma \to \Delta, X \quad \Gamma, Y \to \Delta}{\Gamma, X \supset Y \to \Delta} \qquad \frac{\Gamma, X \to \Delta, Y}{\Gamma \to \Delta, X \supset Y}$$

The rule for introducing a conjunction onto the left-hand side of a sequent has a single premise, while the one for introducing a conjunction onto the right-hand side has two. To conclude $\Gamma \to \Delta, X \wedge Y$, we need both $\Gamma \to \Delta, X$ and $\Gamma \to \Delta, Y$, and similarly for the other connectives.

Definition 4.3.3 A *proof* is a tree labeled with sequents (generally written with the root at the bottom) meeting the following conditions. If node N is labeled with $\Gamma \to \Delta$, then if N is a leaf node, $\Gamma \to \Delta$ must be an axiom; and if N has children, their labels must be the premises from which $\Gamma \to \Delta$ follows by one of the sequent calculus rules. The label on the root node is the sequent that is proved. Finally, a formula X is a *theorem* of the sequent calculus if the sequent $\to X$ has a proof.

1. $P \to P$ 2. $Q \to Q$
3. $P, Q \to P$ 4. $P, Q \to Q$
5. $P, Q \to P \wedge Q$
6. $Q \to P \wedge Q, \neg P$
7. $\to P \wedge Q, \neg P, \neg Q$
8. $\neg(P \wedge Q) \to \neg P, \neg Q$
9. $\neg(P \wedge Q) \to \neg P \vee \neg Q$
10. $\to \neg(P \wedge Q) \supset (\neg P \vee \neg Q)$

FIGURE 4.4. Sequent Calculus Proof of $\neg(P \wedge Q) \supset (\neg P \vee \neg Q)$

Example Figure 4.4 displays a proof of $\neg(P \wedge Q) \supset (\neg P \vee \neg Q)$, arranged as a tree with 10 as root and 1 and 2 as leaves. In it, 1 and 2 are axioms, from which 3 and 4 follow, respectively, by thinning; 5 follows from 3 and 4 using a conjunction rule. Then 6 through 8 follow by negation rules. Finally 9 follows from 8 using a disjunction rule and 10 from 9 by an implication rule.

Soundness is easily established. Each axiom is a tautology, and each rule produces sequents that are tautologies from sequents that are tautologies. It follows that only tautologous sequents can be proved. We thus have the following:

Theorem 4.3.4 **(Sequent Calculus Soundness)** *If X is a theorem of the sequent calculus, X is a tautology.*

Completeness takes work, though once again we can use the Model Existence Theorem 3.6.2. We first need to associate sequents with sets of formulas.

Definition 4.3.5 $\neg S = \{\neg X \mid X \in S\}$

Definition 4.3.6 Let S be a finite set of formulas. An *associated sequent* for S is a sequent $\Gamma \longrightarrow \neg\Delta$, where Γ, Δ is a partition of S, that is, $\Gamma \cap \Delta = \emptyset$ and $\Gamma \cup \Delta = S$.

For example, if $S = \{X \supset Y, \neg X, X \wedge Y\}$, then $X \supset Y \longrightarrow \neg\neg X, \neg(X \wedge Y)$ and $\neg X, X \wedge Y \longrightarrow \neg(X \supset Y)$ are both associated sequents.

Lemma 4.3.7 *If any associated sequent for S has a proof, every associated sequent does.*

Definition 4.3.8 A finite set S of formulas is *sequent inconsistent* if any (equivalently, every) associated sequent has a proof. S is *sequent consistent* if it is not sequent inconsistent.

Proposition 4.3.9 *The collection of sequent consistent sets is a propositional consistency property.*

We leave the proof of this proposition to you. From it completeness follows easily.

Theorem 4.3.10 **(Sequent Calculus Completeness)** *If X is a tautology, then X is a theorem of the sequent calculus.*

Proof Suppose X is not a theorem. Then the sequent $\longrightarrow X$ is not provable. It follows that $\{\neg X\}$ is sequent consistent, for otherwise $\neg X \longrightarrow$ would be provable, and hence $\longrightarrow X$, by Exercise 4.3.3. Now, by Proposition 4.3.9 and the Model Existence Theorem, $\{\neg X\}$ is satisfiable, and so X is not a tautology. $\square$

Relationships between the sequent calculus and natural deduction go back to Gentzen. Loosely, one can imagine the sequent calculus as providing a kind of 'specification language' for a natural deduction system. Think of the sequent $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_k$ as asserting the following: One of $B_1, \ldots, B_k$ can be obtained as a line in a natural deduction proof whenever all of $A_1, \ldots, A_n$ are active as premises. Now one wants a natural deduction system for which the sequent calculus axioms are true and such that true premises for a sequent calculus rule ensure a true conclusion. For example, the first of the sequent calculus rules for conjunction would be true of any natural deduction system that contained a rule: From $X \wedge Y$ one can derive both X and Y. Since the sequent calculus is complete, any natural deduction system meeting the sequent calculus specifications would also be complete. We do not pursue this idea further here but recommend Gentzen's original, and very readable paper [22].

Smullyan introduced two kinds of semantic tableaux [48], *signed* and *unsigned.* We have been using the unsigned version so far in this book and will continue to do so. The signed version has no advantages for classical theorem proving, but there are natural modifications of it that provide proof mechanisms for intuitionistic [16] and many-valued logics [7, 19] and that are not available without signs. What is relevant now is that there is a direct connection between the Gentzen sequent calculus and Smullyan's signed tableau system. We briefly sketch the Smullyan system, and illustrate the connection with an example, though we do not formally prove the relationship.

First, two new symbols are introduced, T and F, called *signs.* A *signed formula* is a formula prefixed with a sign, such as $T\ X \wedge Y$ or $F\ \neg X$. Think of $T\ Z$ as asserting that Z is true, and $F\ Z$ as asserting that Z is false. Formally, Boolean valuations are extended to signed formulas by $v(T\ Z) = v(Z)$ and $v(F\ Z) = \neg v(Z)$. Note that there is a formal distinction between $F\ Z$ and $T\ \neg Z$, just as there is between $X \rightarrow Y$ and $X \supset Y$.

Next, the tableau rules we have been using are replaced with corresponding signed versions. For example, in the original system we had a rule: From $\neg(X \supset Y)$, obtain X and $\neg Y$. The signed version follows: From $F\ X \supset Y$, obtain $T\ X$ and $F\ Y$. Similarly, there is an unsigned rule: From $X \supset Y$, branch to $\neg X$ and Y. That corresponds to the following: from $T\ X \supset Y$, branch to $F\ X$ and $T\ Y$. We leave it to you to formulate the various signed versions. Indeed, an α-, β-classification, and uniform rule formulations are straightforward.

Finally, a tableau branch is closed if it contains both $T\ X$ and $F\ X$, or if it contains $T\ \bot$, or if it contains $F\ \top$. A closed tableau for $\{F\ X\}$ constitutes a proof of X.

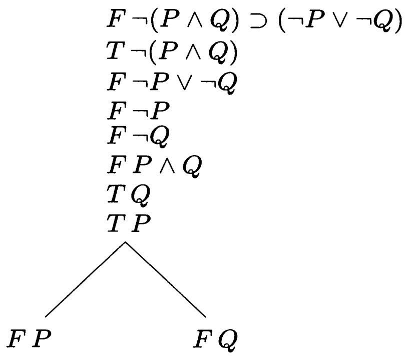


FIGURE 4.5. Signed Tableau Proof of $\neg(P \wedge Q) \supset (\neg P \vee \neg Q)$

One can pair signed formulas with sequents in the following straightforward way. If a formula X occurs on the left-hand side of an arrow, associate with it $T\,X$; if X occurs on the right-hand side, associate with it $F\,X$. Note the role of the signs here. If we used unsigned formulas, with negation playing the role of F, we could not tell if we had a formula $\neg X$, because $\neg X$ occurred on the left-hand side of an arrow, or because X occurred on the right-hand side. Use of the signs avoids this ambiguity. Now, each of the sequent calculus rules corresponds exactly to one of the tableau system rules, though we must keep in mind that sequent trees are written with the root at the bottom, while tableaux are written with the root at the top. Consequently, in the correspondence, rules must be turned over. As an example, the sequent calculus rule introducing $X \supset Y$ on the left-hand side of an arrow and the tableau rule for $T\,X \supset Y$ clearly correspond.

$$\frac{\Gamma \rightarrow \Delta, X \qquad \Gamma, Y \rightarrow \Delta}{\Gamma, X \supset Y \rightarrow \Delta} \qquad\qquad \frac{T\,X \supset Y}{F\,X \mid T\,Y}$$

Using this correspondence, the sequent calculus proof in Figure 4.4 and the signed tableau proof in Figure 4.5 clearly also correspond. (Remember to invert things when going from one system to the other.)

We have not spelled out the details of this correspondence but have relied on your general intuition and understanding. Formal details may be found in Smullyan [48].

Exercises

4.3.1. Give sequent calculus proofs of the following:

1. $X \supset (Y \supset X)$.
2. $(X \supset (Y \supset Z)) \supset ((X \supset Y) \supset (X \supset Z))$.
3. $(\neg Y \supset \neg X) \supset ((\neg Y \supset X) \supset Y)$.
4. $((X \supset Y) \supset X) \supset X$.
5. $(X \wedge (Y \vee Z)) \supset ((X \wedge Y) \vee (X \wedge Z))$.
6. $(X \supset Y) \supset (\neg(Y \vee Z) \supset \neg(X \vee Z))$.

4.3.2. Give a rigorous proof of the soundness of the sequent calculus.

4.3.3. Show that if $\Gamma \longrightarrow \Delta, \neg X$ is a theorem of the sequent calculus, so is $\Gamma, X \longrightarrow \Delta$. Likewise, if $\Gamma, \neg X \longrightarrow \Delta$ is a theorem, so is $\Gamma \longrightarrow \Delta, X$. Hint: Use induction on the number of sequents in the proof tree.

4.3.4. Prove Lemma 4.3.7. Hint: Use Exercise 4.3.3.

4.3.5. Prove Proposition 4.3.9.

4.4 The Davis-Putnam Procedure

In 1960 the Davis-Putnam procedure was introduced [14]. This was intended to be a theorem-proving technique suitable for automation, covering classical propositional and first-order logic. The first-order version was not as efficient as resolution, which was introduced soon after, because the notion of unification was missing, but the propositional version is still among the fastest. We present it here for its own sake and as a theorem-proving algorithm that is not hard to implement and experiment with.

The Davis-Putnam procedure, like resolution, is a refutation method. To prove X, start with $\neg X$ and derive a contradiction. The first phase is a conversion to clause form, just as with resolution. This has been discussed in Sections 2.8 and 2.9. Essentially, then, the Davis-Putnam procedure is a test for the unsatisfiability of a clause form.

A clause is a disjunction of literals. A clause set is a conjunction of clauses. We now need a more complicated object, a disjunction of clause sets. We call these ***blocks*** (the term was not used in Davis and Putnam [14]). Part of the strategy in applying Davis-Putnam methods will be to keep the number of clause sets in a block to a minimum, preferably one.

Definition 4.4.1 A *block* is a disjunction of clause sets.

The procedure involves the mechanical transformation of blocks into blocks, via simple rewriting rules. Syntactically, all these transformations have the same general pattern: Replace some clause set in a block by one or more others. Semantically, each transformation should not affect satisfiability: The transformed block must be satisfiable if and only if the original block was satisfiable. The intention is to produce a block that is obviously satisfiable or not. There are two families of rules. The first are preliminary rules that are not strictly necessary but that can considerably speed up later steps. Then there are the primary rules that are the essence of the Davis-Putnam procedure. We begin with a discussion of the preliminary rules.

Preliminary Step 1 Remove any repetitions from the clauses in a block, and (maybe) arrange the literals into some standard order.

It is easy to see that this step can not affect satisfiability. The same applies to the next; we omit any formal verification.

Definition 4.4.2 For a propositional letter P, we set $\overline{P} = \neg P$ and $\overline{\neg P} = P$. The literals L and $\overline{L}$ are *complementary* literals.

Preliminary Step 2 Delete any clause that contains both a literal and its complement. Delete any clause that contains $\top$. Delete every occurrence of $\bot$.

Now we come to the main transformations. We state them, and we either prove the necessary semantic facts about each or leave them as exercises. Then we give examples, and finally, we prove soundness and completeness.

One-Literal Rule Suppose $\mathbf{B}$ is a block containing the clause set S, and S in turn contains the one-literal clause $[L]$. Modify $\mathbf{B}$ by changing S as follows: Remove from S all clauses containing L, and delete all occurrences of $\overline{L}$ from the remaining clauses of S. (When talking about applications of this rule, we say it has been used *on* the literal L.)

To prove that the One-Literal Rule has no effect on satisfiability of blocks, it is enough to prove the modifications it makes to clause sets have this property.

Proposition 4.4.3 *Suppose S is a clause set that contains the one-literal clause $[L]$. Let S^* be like S except that all clauses containing L have been removed, and from the remaining clauses, all occurrences of $\overline{L}$ have been deleted. Then S is satisfiable if and only if S^* is satisfiable.*

Proof For the sake of simple notation, we use a Prolog-style convention when writing clauses. $[A \mid B]$ denotes a clause whose first item is A, with B being the clause consisting of the rest of the members. Now, suppose S is the clause set $\langle [A], [A \mid C_1], \ldots, [A \mid C_n], [\neg A \mid D_1] \ldots, [\neg A \mid D_k], E_1, \ldots, E_j \rangle$, where the literal L is the propositional letter A, and the only occurrences of A and $\neg A$ are the ones indicated. Then S^* is the clause set $\langle D_1, \ldots, D_k, E_1, \ldots, E_j \rangle$.

Suppose first that S is satisfiable. Say the Boolean valuation v maps every member of S to $\mathbf{t}$. Then in particular, $v(A) = \mathbf{t}$. Also $v([\neg A \mid D_1]) = \mathbf{t}$, but since $v(\neg A) = \mathbf{f}$, it must be that $v(D_1) = \mathbf{t}$, and similarly for each D_i. And of course, $v(E_1) = \mathbf{t}, \ldots, v(E_j) = \mathbf{t}$. Hence, each member of S^* maps to $\mathbf{t}$ under v, so S^* is satisfiable.

Next suppose that S^* is satisfiable; say the Boolean valuation v maps every member of S^* to $\mathbf{t}$. We define a new Boolean valuation w by specifying it on propositional letters. For every propositional letter P, except for A, $w(P) = v(P)$. And $w(A) = \mathbf{t}$. It is obvious that on any clause K that contains no occurrences of A or $\neg A$, $v(K) = w(K)$. Hence $w(E_1) = \mathbf{t}, \ldots, w(E_j) = \mathbf{t}$, since $v(E_1) = \mathbf{t}, \ldots, v(E_j) = \mathbf{t}$. Further, since $v(D_1) = \mathbf{t}$, $w(D_1) = \mathbf{t}$ and hence $w([\neg A \mid D_1]) = \mathbf{t}$. Similarly, for $[\neg A \mid D_2], \ldots, [\neg A \mid D_k]$. Finally, by design, $w(A) = \mathbf{t}$, and hence $w([A \mid C_1]) = \mathbf{t}, \ldots, w([A \mid C_n]) = \mathbf{t}$. And of course, $w([A]) = \mathbf{t}$. Thus, w maps every member of S to $\mathbf{t}$, so S is satisfiable. $\square$

Now we give the remaining rules, leaving their semantic properties as exercises.

Affirmative-Negative Rule Suppose $\mathbf{B}$ is a block containing clause set S, some clauses in S contain the literal L, and no clauses in S contain $\overline{L}$. Modify $\mathbf{B}$ by removing from S all clauses containing L. (We will say this rule has been used *on* the literal L.)

Definition 4.4.4 A clause C_1 *subsumes* a clause C_2 if every literal in C_1 also occurs in C_2.

The idea behind subsumption is simple. If C_1 subsumes C_2, then if C_1 is unsatisfiable, so is C_2. In testing for unsatisfiability, if one clause subsumes another, we can ignore the one that is subsumed.

Subsumption Rule Suppose $\mathbf{B}$ is a block containing the clause set S, and S contains clauses C_1 and C_2 where C_1 subsumes C_2. Modify $\mathbf{B}$ by removing clause C_2 from S.

Splitting Rule Suppose $\mathbf{B}$ is a block containing the clause set S, and some clauses in S contain the literal L while others contain $\overline{L}$. (There may also be clauses with neither.) Let S_L be the clause set that results when all clauses in S containing L are removed, and all occurrences of $\overline{L}$ are deleted. Likewise, let $S_{\overline{L}}$ be the clause set that results when all clauses in S containing $\overline{L}$ are removed, and all occurrences of L are deleted. Modify $\mathbf{B}$ by replacing the clause set S by the two clause sets S_L and $S_{\overline{L}}$. (When talking about applications of this rule, we say we have *split on* the literal L.)

This completes the presentation of the Davis-Putnam Rules. In discussing their use, the following terminology is handy:

Definition 4.4.5 Let $\mathbf{B}$ be a block. A *Davis-Putnam derivation* for $\mathbf{B}$ is a finite sequence of blocks $\mathbf{B}_1$, $\mathbf{B}_2, \ldots$, $\mathbf{B}_n$, where $\mathbf{B}_1 = \mathbf{B}$, and otherwise each block in the sequence comes from its predecessor using one of the four rewriting rules. A derivation *succeeds* if it ends with a block in which each clause set contains the empty clause. A derivation *fails* if it ends with a block in which some clause set itself is empty.

To *prove* a formula X in this system, begin with $\neg X$, convert this to a clause set S, form the block $[S]$, perhaps simplify this using the two preliminary rules, then show there is a Davis-Putnam derivation that succeeds.

Example We apply the technique to the formula $(P \equiv Q) \vee (P \equiv \neg Q)$.

1. Negate the formula:
   $\neg((P \equiv Q) \vee (P \equiv \neg Q))$.
2. Convert to a clause set, and form the corresponding block:
   $[\langle [P, \neg P], [\neg P, \neg Q], [P, Q], [Q, \neg Q], [P, \neg P], [P, \neg Q],$
   $[\neg P, Q], [Q, \neg Q]\rangle]$.
3. Apply Preliminary Rule 2 (four times):
   $[\langle [\neg P, \neg Q], [P, Q], [P, \neg Q], [\neg P, Q]\rangle]$.
4. Use the Splitting Rule, splitting on the literal P:
   $[\langle [\neg Q], [Q]\rangle,\ \langle [Q], [\neg Q]\rangle]$.
5. Apply the One-Literal Rule to each clause set in the block:
   $[\langle [\,]\rangle,\ \langle [\,]\rangle]$.
6. Each clause set in the final block contains the empty clause, so the derivation has succeeded.

Example We attempt to prove the formula $(P \vee Q) \supset (P \wedge Q)$.

1. Negate the formula:
   $\neg((P \vee Q) \supset (P \wedge Q))$.
2. Convert to a clause set, and form the corresponding block:
   $[\langle [P, Q], [\neg P, \neg Q] \rangle]$.
3. Use the Splitting Rule, splitting on the literal P:
   $[\langle [\neg Q] \rangle, \langle [Q] \rangle]$.
4. Apply either the One-Literal Rule or the Affirmative-Negative Rule to each clause set in the block:
   $[\langle\ \rangle, \langle\ \rangle]$.
5. Some clause set in the block is empty (in fact, both are), so the derivation has failed.

Now we must establish soundness and completeness for the procedure. And for once we will not use the Model Existence Theorem. We did not specify in what order the rules were to be applied; there is a certain degree of non-determinism here. We show the strong result that this non-determinism has no effect on success or failure; we have a decision procedure.

Theorem 4.4.6 *Suppose an attempt is made to prove the formula X using the Davis-Putnam procedure, and the attempt is continued until no rule is applicable. Any such attempt must terminate, and it must do so in a success or in a failure. If it terminates in a success, X is a tautology; if it terminates in a failure, X is not a tautology.*

Proof First, we argue that if a proof attempt terminates it must do so in a success (with every clause set containing the empty clause) or in a failure (with some clause set itself empty). We actually establish the contrapositive. Suppose we have a block $\mathbf{B}$ that contains some clause set S that is not empty and that does not contain the empty clause; we show we do not have termination. Since S is not empty, we can choose a clause C from it. And since S does not contain the empty clause, C itself is not empty, so we can choose a literal L from it. If $\overline{L}$ does not occur in any clause in S, we can apply the Affirmative-Negative Rule. If $\overline{L}$ does occur in some clauses in S, we can apply the Splitting Rule (or perhaps the One-Literal Rule). Either way, we do not have termination.

Next we argue that every proof attempt must terminate. But this is easy. Every rule except for Subsumption reduces the number of distinct literals that occur in clause sets. Since we began with a finite number, these rules can be applied only a finite number of times. And the Subsumption Rule removes clauses, which are finite in number, so it too can be applied only a finite number of times.

Finally, we argue that termination in success means that X is a tautology, while termination in failure means it is not. X is a tautology if and only if $\{\neg X\}$ is not satisfiable, and our proof attempt begins by converting $\neg X$ to a clause set S, and forming the block $[S]$. Since every Boolean valuation will assign the same truth value to $\neg X$ and $[S]$, X is a tautology if and only if $[S]$ is not satisfiable. Since the rules do not affect satisfiability, $[S]$ will be satisfiable if and only if the final block in the derivation is satisfiable. But a block containing the empty clause set is satisfiable, while a block in which every clause set contains the empty clause is not. □

As we presented them, the Davis-Putnam Rules are non-deterministic. According to the theorem just proved, any order of rule application will produce a proof if one is obtainable. But of course, some attempts will be speedier than others. It is easy to see that the Splitting Rule multiplies the number of cases to be considered, so its use should be postponed. At the other extreme, the One-Literal Rule simply cuts down on the number of clauses we need to consider without introducing any other complications, so it should be used in preference to any other rule. In fact, the order in which we presented the rules is also a good order in which to apply them.

Exercises

4.4.1. Prove an application of the Affirmative-Negative Rule will not affect the satisfiability of a block.

4.4.2. Prove an application of the Subsumption Rule will not affect the satisfiability of a block.

4.4.3. Prove an application of the Splitting Rule will not affect the satisfiability of a block.

4.4.4. Use the Davis-Putnam procedure to test the following for being tautologies:

1. $(P \supset (Q \supset R)) \supset ((P \supset Q) \supset (P \supset R))$.
2. $((P \supset Q) \supset (P \supset R)) \supset (P \supset (Q \supset R))$.
3. $(P \wedge Q) \equiv \neg(\neg P \vee \neg Q)$.
4. $(P \supset Q) \vee (Q \supset R)$.

4.4.5. Take the exclusive-or connective $\not\equiv$ to be defined by:

$$A \not\equiv B = (A \wedge \neg B) \vee (\neg A \wedge B).$$

Give a Davis-Putnam proof that $\not\equiv$ is commutative, that is, prove $(A \not\equiv B) \equiv (B \not\equiv A)$. Similarly, give a proof that $\not\equiv$ is associative. Compare these with a tableau or resolution version.

4.4.6. Prove that the One-Literal Rule is not necessary; any tautology can be verified without using this rule.

4.4.7$^{\mathbf{P}}$. Implement the Davis-Putnam procedure.

4.5 Computational Complexity

If we have a mechanism for showing that formulas are tautologies, it is natural to ask how complicated it is to use. More precisely, as tautologies get more complex, what happens to the complexity of their verifications. For truth tables this is easy to see. A truth table for a formula with n propositional letters will have 2^n lines, so complexity of verification using truth tables is exponential in n. For other proof procedures such questions are less straightforward. In this section we very briefly survey some recent results and give references to the literature. The proofs of these results are too complicated to be given here.

The 1979 paper by Cook and Reckhow [11] gives a thorough analysis of computational complexity for Hilbert systems (which are called Frege systems there), based partly on Reckhow [37]. The complexity of a formula is measured by the number of its symbols and that of a proof by the number of its lines. It is shown that if we have two Hilbert systems, using possibly different but complete sets of connectives, then there is a translation of proofs in one system into proofs in the other that increases proof length by at most a fixed polynomial in the length of the original proof. In other words, any two Hilbert systems are equivalent in complexity, up to a polynomial factor. And this result extends to include natural deduction systems as well. On the other hand, Urquhart [53] shows this does not extend to resolution.

An obvious problem with Hilbert systems, and with natural deduction systems as well, is that while a tautology may have a short proof, it may not be easy to find one. In a Hilbert system a short proof of Y may begin with short proofs of X and $X \supset Y$, followed by an application of Modus Ponens. One can think of X and $X \supset Y$ as Lemmas for the proof of Y. But finding an appropriate formula X when given the task of proving Y may be nontrivial. And certainly, $X \supset Y$ is more complicated than Y, so for a portion of the proof, formula complexity goes up instead of down. Consequently, most, but not all, attempts at automated theorem proving have relied on mechanisms that build proofs entirely out of parts of the formula being proved and do not require outside lemmas. Both resolution and tableaux have this important property (called *analytic* in Smullyan [48]). While this apparently simplifies the problem of finding proofs, a recent result of Haken says that the complexity of proofs in resolution systems still grows exponentially in the complexity of the formulas being proved.

In a resolution system it is possible to begin by doing all the Resolution Reduction Rule applications first, leaving only the Resolution Rule itself to be applied. Most implemented theorem provers proceed this way, and Haken's analysis [24] starts from this point. So, let S be a set of clauses. A *resolution refutation* of S is a sequence of clauses, each a member of S or following from earlier clauses in the sequence by the Resolution Ruleply the total number of clauses. The complexity of the set S is the number of symbols. Haken showed there is a particular sequence of formulas, P_1, P_2, P_3,..., each a tautology, such that the complexity of $\neg P_n$ when converted to clause form is of the order of n^3, but the shortest resolution refutation of it is of complexity at least c^n (for a fixed $c > 1$). In other words, for this family of examples, resolution is exponentially bad.

The formulas P_n that Haken used all express 'pigeonhole' principles. These were used earlier [11] in the analysis of Hilbert systems, and their intuitive content is easy to grasp. If we have $n + 1$ pigeons but only n pigeonholes, some pigeonhole must contain two pigeons. More mathematically stated, if f is a function from a set with $n + 1$ members to a set with n members, there must be two elements i and j of the domain such that $f(i) = f(j)$. We want to capture the essential content of this principle by a formula of propositional logic; the result will be P_n.

Let n be fixed; we show how to formulate P_n. First, we have $n+1$ pigeons and n pigeonholes. Let us introduce a family of $n(n + 1)$ propositional letters, $H_{1,1}$, $H_{1,2}$, $H_{2,1}$,..., $H_{i,j}$,... where i ranges from 1 to $n + 1$ and j from 1 to n. Think of $H_{i,j}$ as saying pigeon i is in pigeonhole j. Now, to say pigeon i is in some pigeonhole, we need the formula $H_{i,1} \vee H_{i,2} \vee \ldots \vee H_{i,n}$, more compactly written $\bigvee_{j=1}^{n} H_{i,j}$. Then, to say pigeon 1 is in some pigeonhole, and so is pigeon 2, and so on, we need a conjunction of formulas like this, for $i = 1, 2, \ldots, n + 1$. Briefly, $\bigwedge_{i=1}^{n+1} \bigvee_{j=1}^{n} H_{i,j}$ expresses what we want.

Next, to say two pigeons are in pigeonhole k, we say 1 and 2 are there, or else 1 and 3 are there, or else 2 and 3, etc. This becomes

$$\bigvee_{i=1}^{n+1} \bigvee_{j=i+1}^{n+1} (H_{i,k} \wedge H_{j,k}).$$

Then saying some pigeonhole has two pigeons requires a disjunction of formulas like this, for $k = 1, 2, \ldots, n$:

$$\bigvee_{k=1}^{n} \bigvee_{i=1}^{n+1} \bigvee_{j=i+1}^{n+1} (H_{i,k} \wedge H_{j,k}).$$

Finally, combining all this, P_n itself becomes the following:

$$\bigwedge_{i=1}^{n+1} \bigvee_{j=1}^{n} H_{i,j} \supset \bigvee_{k=1}^{n} \bigvee_{i=1}^{n+1} \bigvee_{j=i+1}^{n+1} (H_{i,k} \wedge H_{j,k})$$

It is not hard to show that, for each n, P_n is a tautology. To do this we can use the informal pigeonhole principle which is, after all, mathematically correct. The argument goes as follows:

Let v be a Boolean valuation. We wish to show $v(P_n) = \mathbf{t}$. If v maps the left-hand side of the implication of P_n to $\mathbf{f}$, we are done, so now suppose $v(\bigwedge_{i=1}^{n+1} \bigvee_{j=1}^{n} H_{i,j}) = \mathbf{t}$. Then for each $i \in \{1, 2, \ldots, n+1\}$, there is some $j \in \{1, 2, \ldots, n\}$ such that $v(H_{i,j}) = \mathbf{t}$. Define a function $f : \{1, 2, \ldots, n+1\} \to \{1, 2, \ldots, n\}$ such that $f(i)$ is the least j for which $v(H_{i,j}) = \mathbf{t}$. Think of f as mapping pigeons to pigeonholes. By the informal pigeonhole principle, there must be i, j with $i < j$ such that $f(i) = f(j) = k$ for some k. Then $v(H_{i,k}) = v(H_{j,k}) = \mathbf{t}$, so v maps the right-hand side of the implication of P_n to $\mathbf{t}$, and we are done.

While each P_n is a tautology and expresses a mathematical fact that is easily grasped, resolution proofs of P_n grow so quickly with n as to become unmanageable.

Haken's method of proof was applied by Urquhart [53] to a different family, S_n, of formulas based on graph properties. In this case the clauses have a size that varies with n, rather than with n^3 as happened with P_n. It is still the case that the minimal resolution proof length of S_n is bounded below by an exponential in n. But now it is possible to show that each S_n has a short proof in a Hilbert system. From this follows the result cited earlier that, although Hilbert systems are equivalent to each other from a complexity point of view, at least up to a polynomial factor, this equivalence does not extend to resolution.

The status of the pigeonhole principle in Hilbert systems is complex. Cook and Reckhow [11] showed P_n has a Hilbert system proof whose length grows exponentially with n, while Ajtai [1] showed that there can be no Hilbert system proof of P_n in which the number of symbols is bounded by a polynomial in n, and the formulas have constant depth (deepness of subformula nesting).

Note that the results cited do not say that every complex tautology must have a complex resolution proof. But the existence of families like P_n and S_n guarantee that some do. As a matter of fact, a recent paper [9] shows that in a probabilistic sense, most randomly chosen sequences of formulas will be bad ones. We must be prepared to guide proofs by supplying heuristics, since otherwise astronomically long computational times can arise.

Hilbert systems are inappropriate for automated theorem proving. The same applies to natural deduction, since Modus Ponens is a rule of both. Resolution can be exponentially bad. Similar results apply to tableaux. There is no proof, but there is good reason to believe similar results will apply to any proposed proof procedure. Heuristics are a necessity, not a nicety.

Exercises

4.5.1. Give a resolution proof of P_2.

4.5.2. Give a tableau proof of P_2.

5 First-Order Logic

5.1 First-Order Logic—Syntax

In this chapter we present the syntax and semantics of classical first-order logic. We also state and prove a Model Existence Theorem, essentially a semantical result, asserting the existence of certain models. We use the theorem to establish some basic facts about first-order logic, such as compactness and Löwenheim-Skolem results. In the next chapter we introduce proof procedures for first-order logic, and then the Model Existence Theorem will find its primary application, in proving completeness. Further consequences will be found in Chapter 8, after we have considered the implementation of proof procedures. This section sets forth the syntax of first-order logic, which is a considerably more complicated business than it was in the propositional case.

We work with several different first-order languages, depending on what applications we have in mind. For instance, if we want to talk about rings, we will want a constant symbol to denote the zero of a ring; if we want to talk about rings with unity, we will need an additional constant symbol to denote the ring unit. Language features like these depend on the intended applications, but certain items are common to all our languages. We begin with them.

Propositional Connectives are the same as in propositional logic, with the Secondary Connectives thought of as defined and the Primary ones as basic. We also have the propositional constants, $\top$ and $\bot$.

Quantifiers

$\forall$ (for all, the universal quantifier)

$\exists$ (there exists, the existential quantifier)

Punctuation ')', '(', and ','

Variables v_1, v_2,... (which we write informally as x, y, z,...).

Now we turn to the parts that vary from language to language.

Definition 5.1.1 A *first-order language* is determined by specifying:

1. A finite or countable set $\mathbf{R}$ of *relation symbols*, or *predicate symbols*, each of which has a positive integer associated with it. If $P \in \mathbf{R}$ has the integer n associated with it, we say P is an n-place relation symbol.
2. A finite or countable set $\mathbf{F}$ of *function symbols*, each of which has a positive integer associated with it. If $f \in \mathbf{F}$ has the integer n associated with it, we say f is an n-place function symbol.
3. A finite or countable set $\mathbf{C}$ of *constant symbols.*

We use the notation $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ for the first-order language determined by $\mathbf{R}$, $\mathbf{F}$, and $\mathbf{C}$. If there is no danger of confusion, we may abbreviate $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ to just L. Sometimes it is useful to allow the same symbol to be used as an n-place function symbol and also as an m-place one; no confusion should arise, because the different uses can be told apart easily. Similar remarks apply to relation symbols. Further, sometimes it is useful to think of constant symbols as 0-place function symbols.

Having specified the basic element of syntax, the alphabet, we go on to more complex constructions.

Definition 5.1.2 The family of *terms* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ is the smallest set meeting the conditions:

1. Any variable is a term of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$.
2. Any constant symbol (member of $\mathbf{C}$) is a term of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$.
3. If f is an n-place function symbol (member of $\mathbf{F}$) and $t_1, \ldots, t_n$ are terms of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, then $f(t_1, \ldots, t_n)$ is a term of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$.

A term is *closed* if it contains no variables.

Example If f is a one-place function symbol, g is a two-place function symbol, a, b are constants, and x, y are variables, then the following are terms: $f(g(a,x))$; $g(f(x),g(x,y))$; $g(a,g(a,g(a,b)))$.

Earlier we defined the notion of subformula of a propositional formula. In essentially the same way, we can define the notion of a *subterm* of a term. We omit the formal definition, but we will use the concept from time to time. We may sometimes be informal about how we write terms. For instance, if $+$ is a two-place function symbol, we may write $x+y$ instead of $+(x,y)$. Think of $x+y$ as an unofficial way of designating the 'real' term. In Section 2.2 we established Principles of Structural Induction and Structural Recursion for the set of propositional formulas. The definition of term we gave is very similar to that of propositional formula, and similar principles apply, with similar justifications. Thus, we can show every term of $L(\mathbf{R},\mathbf{F},\mathbf{C})$ has a certain property by showing that variables and constant symbols have the property, and that $f(t_1,\ldots,t_n)$ has the property whenever each of $t_1,\ldots,t_n$ does, for each function symbol f. Similarly, we can define functions on the set of terms by specifying them outright on variables and constant symbols, and on $f(t_1,\ldots,t_n)$ based on the values assigned to $t_1,\ldots,t_n$. This observation applies equally well to the definition of formula of first-order logic, which we give shortly. From now on we make free use of these principles, not always with explicit mention.

Definition 5.1.3 An *atomic formula* of $L(\mathbf{R},\mathbf{F},\mathbf{C})$ is any string of the form $R(t_1,\ldots,t_n)$ where R is an n-place relation symbol (member of $\mathbf{R}$) and $t_1,\ldots,t_n$ are terms of $L(\mathbf{R},\mathbf{F},\mathbf{C})$; also $\top$ and $\bot$ are taken to be atomic formulas of $L(\mathbf{R},\mathbf{F},\mathbf{C})$.

Definition 5.1.4 The family of *formulas* of $L(\mathbf{R},\mathbf{F},\mathbf{C})$ is the smallest set meeting the following conditions:

1. Any atomic formula of $L(\mathbf{R},\mathbf{F},\mathbf{C})$ is a formula of $L(\mathbf{R},\mathbf{F},\mathbf{C})$.
2. If A is a formula of $L(\mathbf{R},\mathbf{F},\mathbf{C})$ so is $\neg A$.
3. For a binary connective $\circ$, if A and B are formulas of $L(\mathbf{R},\mathbf{F},\mathbf{C})$, so is $(A\circ B)$.
4. If A is a formula of $L(\mathbf{R},\mathbf{F},\mathbf{C})$ and x is a variable, then $(\forall x)A$ and $(\exists x)A$ are formulas of $L(\mathbf{R},\mathbf{F},\mathbf{C})$.

Example If R and g are two-place relation and function symbols respectively, then $(\forall x)(\forall y)(R(x,y)\supset(\exists z)(R(x,z)\wedge R(z,y)))$ and $(\forall x)(\exists y)R(f(x,y),z)$ are formulas. We will be informal about parentheses, and instead of the first formula, we may write

$$(\forall x)(\exists y)\{R(x,y) \supset (\exists z)[R(x,z) \wedge R(z,y)]\}.$$

Likewise, if $<$ is a two-place relation symbol, we may write $x < y$ instead of the official $<(x,y)$, and similarly in other cases. This is to improve readability, and should be thought of as an informal substitute for the 'real' formula.

In Chapter 2 we sketched a proof that the formulas of propositional logic could be uniquely parsed. A similar result holds in the first-order case. We neither prove this result nor state it properly, though we make tacit use of it. Likewise, the notion of *rank* (Definition 2.6.5) extends to the first-order setting.

Definition 5.1.5 The *rank* $r(X)$ of a first-order formula X is given as follows: $r(A) = r(\neg A) = 0$ for A atomic, other than $\top$ or $\bot$. $r(\top) = r(\bot) = 0$. $r(\neg\top) = r(\neg\bot) = 1$. $r(\neg\neg Z) = r(Z) + 1$. $r(\alpha) = r(\alpha_1) + r(\alpha_2) + 1$. $r(\beta) = r(\beta_1) + r(\beta_2) + 1$. $r(\gamma) = r(\gamma(x)) + 1$. $r(\delta) = r(\delta(x)) + 1$.

Next, we must distinguish between a formula like $(\forall x)P(x,y)$ and one like $(\forall x)(\exists y)P(x,y)$. In the first, the variable y is not in the scope of any quantifier, while in the second, every variable is covered by some quantifier. The notion of free and bound variable is what is needed here. In the first formula y has a free occurrence, which is not the case with the second. Note that the definition that follows relies on the Principle of Structural Recursion.

Definition 5.1.6 The *free-variable occurrences* in a formula are defined as follows:

1. The free-variable occurrences in an atomic formula are all the variable occurrences in that formula.
2. The free-variable occurrences in $\neg A$ are the free variable occurrences in A.
3. The free-variable occurrences in $(A \circ B)$ are the free-variable occurrences in A together with the free variable occurrences in B.
4. The free-variable occurrences in $(\forall x)A$ and $(\exists x)A$ are the free-variable occurrences in A, except for occurrences of x.

A variable occurrence is called *bound* if it is not free.

Definition 5.1.7 A *sentence* (also called a *closed formula*) of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ is a formula of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ with no free-variable occurrences.

Exercises

5.1.1. Identify the free-variable occurrences in the following:

1. $(\forall x)R(x, c) \supset R(x, c)$.
2. $(\forall x)(R(x, c) \supset R(x, c))$.
3. $(\forall x)[(\exists y)R(f(x, y), c) \supset (\exists z)S(y, z)]$.

5.2 Substitutions

Formulas of first-order logic may contain free variables that can be replaced by other, more complicated, terms. The notion of substituting a term for a variable plays a fundamental role in automated theorem proving. In this section we begin looking at properties of substitution.

For this section, let $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ be a fixed first-order language, and let $\mathbf{T}$ be the set of terms of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$. All our definitions are relative to this language. Also, substitutions are functions; we use algebraic notation, writing $t\sigma$ instead of $\sigma(t)$ to denote the result of applying the function σ to t.

Definition 5.2.1 A *substitution* is a mapping $\sigma : \mathbf{V} \to \mathbf{T}$ from the set of variables $\mathbf{V}$ to the set of terms $\mathbf{T}$.

Although substitutions are maps on variables, their actions are easily extended to all terms.

Definition 5.2.2 Let σ be a substitution. Then we set:

1. $c\sigma = c$ for a constant symbol c.
2. $[f(t_1, \ldots, t_n)]\sigma = f(t_1\sigma, \ldots, t_n\sigma)$ for an n-place function symbol f.

Example Suppose $x\sigma = f(x, y)$, $y\sigma = h(a)$, and $z\sigma = g(c, h(x))$. Then $j(k(x), y)\sigma = j(k(f(x, y)), h(a))$.

The result of applying a substitution to a term always produces another term. Also, if two substitutions agree on the variables of a term t, they will produce the same results when applied to t. We leave the verification of these items as exercises, and assume them in what follows:

Definition 5.2.3 Let σ and τ be substitutions. By the *composition* of σ and τ, we mean that substitution, which we denote by $\sigma\tau$, such that for each variable x, $x(\sigma\tau) = (x\sigma)\tau$.

The definition of composition concerns behavior on variables. In fact, using structural induction, one can prove that this extends to all terms.

Proposition 5.2.4 *For every term t, $t(\sigma\tau) = (t\sigma)\tau$.*

Proposition 5.2.5 *Composition of substitutions is associative, that is, $(\sigma_1\sigma_2)\sigma_3 = \sigma_1(\sigma_2\sigma_3)$.*

Proof Let v be any variable. We must show $v(\sigma_1\sigma_2)\sigma_3 = v\sigma_1(\sigma_2\sigma_3)$. But, $v(\sigma_1\sigma_2)\sigma_3 = [v(\sigma_1\sigma_2)]\sigma_3 = [(v\sigma_1)\sigma_2]\sigma_3$. Likewise $v\sigma_1(\sigma_2\sigma_3) = (v\sigma_1)(\sigma_2\sigma_3) = [(v\sigma_1)\sigma_2]\sigma_3$. □

Definition 5.2.6 The *support* of a substitution σ is the set of variables x for which $x\sigma \neq x$. A substitution has *finite support* if its support set is finite.

Proposition 5.2.7 *The composition of two substitutions having finite support is a substitution having finite support.*

We leave the proof of this to you. The identity substitution has finite support, and so the set of substitutions having finite support constitutes a semigroup under composition. Frequently we will only be interested in substitutions that have finite support. For such cases we have a convenient special notation.

Definition 5.2.8 Suppose σ is a substitution having finite support; say $\{x_1, \ldots, x_n\}$ is the support, and for each $i = 1, \ldots, n$, $x_i\sigma = t_i$. Our *notation* for σ is: $\{x_1/t_1, \ldots, x_n/t_n\}$. In particular, our notation for the identity substitution is $\{\ \}$.

Proposition 5.2.9 *Suppose σ_1 and σ_2 are two substitutions having finite support. Say $\sigma_1 = \{x_1/t_1, \ldots, x_n/t_n\}$ and $\sigma_2 = \{y_1/u_1, \ldots, y_k/u_k\}$. Then the composition $\sigma_1\sigma_2$ has notation*

$$\{x_1/(t_1\sigma_2), \ldots, x_n/(t_n\sigma_2), z_1/(z_1\sigma_2), \ldots, z_m/(z_m\sigma_2)\}$$

where $z_1, \ldots, z_m$ are those variables in the list $y_1, \ldots, y_k$ that are not also in the list $x_1, \ldots, x_n$. (We assume that if any item degenerates into x/x, it is dropped from the substitution notation.)

Example Suppose $\sigma_1 = \{x/f(x, y), y/h(a), z/g(c, h(x))\}$ and $\sigma_2 = \{x/b, y/g(a, x), w/z\}$. Then $\sigma_1\sigma_2 = \{x/f(b, g(a, x)), y/h(a), z/g(c, h(b)), w/z\}$.

Next, the action of substitution is extended to arbitrary formulas. Here things are more complicated, because variables in formulas can have both free and bound occurrences, and substitutions should not affect bound variable occurrences.

Definition 5.2.10 Let σ be a substitution. By σ_x we mean the substitution that is like σ except that it does not change the variable x. More precisely, for any variable y:

$$y\sigma_x = \begin{cases} y\sigma & \text{if } y \neq x \\ x & \text{if } y = x. \end{cases}$$

Definition 5.2.11 Substitution is extended to formulas as follows. Let σ be a substitution. Then:

1. For the atomic case:

$$[A(t_1, \ldots, t_n)]\sigma = A(t_1\sigma, \ldots, t_n\sigma)$$
$$\top\sigma = \top$$
$$\bot\sigma = \bot.$$

2. $[\neg X]\sigma = \neg[X\sigma]$.
3. $(X \circ Y)\sigma = (X\sigma \circ Y\sigma)$ for a binary symbol $\circ$.
4. $[(\forall x)\Phi]\sigma = (\forall x)[\Phi\sigma_x]$.
5. $[(\exists x)\Phi]\sigma = (\exists x)[\Phi\sigma_x]$.

Example Suppose $\sigma = \{x/a, y/b\}$. Then

$$\begin{aligned}[(\forall x)R(x,y) \supset (\exists y)R(x,y)]\,\sigma &= [(\forall x)R(x,y)]\,\sigma \supset [(\exists y)R(x,y)]\,\sigma \\ &= (\forall x)\,[R(x,y)]\,\sigma_x \supset (\exists y)\,[R(x,y)]\,\sigma_y \\ &= (\forall x)R(x,b) \supset (\exists y)R(a,y).\end{aligned}$$

One of the key facts about substitution in terms follows: For any term t, $(t\sigma)\tau = t(\sigma\tau)$. This result does not carry over to formulas. For example, let $\sigma = \{x/y\}$ and $\tau = \{y/c\}$ (here x and y are variables and c is a constant symbol). Then $\sigma\tau = \{x/c, y/c\}$. If $\Phi = (\forall y)R(x,y)$, then $\Phi\sigma = (\forall y)R(y,y)$, so $(\Phi\sigma)\tau = (\forall y)R(y,y)$. But $\Phi(\sigma\tau) = (\forall y)R(c,y)$, which is different. What is needed is some restriction that will ensure composition of substitutions behaves well.

Definition 5.2.12 A substitution being *free for a formula* is characterized as follows:

1. If A is atomic, σ is free for A.
2. σ Is free for $\neg X$ if σ is free for X.
3. σ Is free for $(X \circ Y)$ if σ is free for X and σ is free for Y.

4. σ Is free for $(\forall x)\Phi$ and $(\exists x)\Phi$ provided: σ_x is free for Φ, and if y is a free variable of Φ other than x, $y\sigma$ does not contain x.

Theorem 5.2.13 *Suppose the substitution σ is free for the formula X, and the substitution τ is free for $X\sigma$. Then $(X\sigma)\tau = X(\sigma\tau)$.*

Proof By structural induction on X. The atomic case is immediate, and is omitted.

We give the binary operation symbol case; negation is similar. Suppose the result is known for X and for Y, σ is free for $(X \circ Y)$, and τ is free for $(X \circ Y)\sigma$. We show $((X \circ Y)\sigma)\tau = (X \circ Y)(\sigma\tau)$.

Since σ is free for $(X \circ Y)$, σ is free for X, and σ is free for Y. Since τ is free for $(X \circ Y)\sigma = (X\sigma \circ Y\sigma)$, τ is free for $X\sigma$ and τ is free for $Y\sigma$. Then by the induction hypothesis, $(X\sigma)\tau = X(\sigma\tau)$ and $(Y\sigma)\tau = Y(\sigma\tau)$. Hence $((X \circ Y)\sigma)\tau = (X\sigma \circ Y\sigma)\tau = (X\sigma)\tau \circ (Y\sigma)\tau = X(\sigma\tau) \circ Y(\sigma\tau) = (X \circ Y)(\sigma\tau)$.

Finally, we give one quantifier case; the other is similar. Suppose the result is known for Φ, σ is free for $(\forall x)\Phi$, and τ is free for $[(\forall x)\Phi]\sigma$. We show that $([(\forall x)\Phi]\sigma)\tau = [(\forall x)\Phi](\sigma\tau)$.

Since σ is free for $(\forall x)\Phi$, σ_x is free for Φ. And since τ is free for $[(\forall x)\Phi]\sigma = (\forall x)[\Phi\sigma_x]$, τ_x is free for $\Phi\sigma_x$. Then by the induction hypothesis, $(\Phi\sigma_x)\tau_x = \Phi(\sigma_x\tau_x)$.

Next we show that $\Phi(\sigma_x\tau_x) = \Phi(\sigma\tau)_x$. To do this it is enough to show that if y is any free variable of Φ, then $y(\sigma_x\tau_x) = y(\sigma\tau)_x$. This is trivial if $y = x$, so now suppose $y \neq x$. Since $y \neq x$, $y\sigma = y\sigma_x$ and $y(\sigma\tau) = y(\sigma\tau)_x$. Also since σ is free for $(\forall x)\Phi$, $y\sigma$ does not contain x, hence $(y\sigma)\tau = (y\sigma)\tau_x$. Then, putting all this together, $y(\sigma_x\tau_x) = (y\sigma_x)\tau_x = (y\sigma)\tau_x = (y\sigma)\tau = y(\sigma\tau) = y(\sigma\tau)_x$.

Finally, $([(\forall x)\Phi]\sigma)\tau = ((\forall x)[\Phi\sigma_x])\tau = (\forall x)[(\Phi\sigma_x)\tau_x] = (\forall x)[\Phi(\sigma_x\tau_x)] = (\forall x)[\Phi(\sigma\tau)_x] = [(\forall x)\Phi](\sigma\tau)$. This concludes the proof. □

Exercises

5.2.1. Prove that if t is a term and σ is a substitution, then $t\sigma$ is a term. Use structural induction on t.

5.2.2. Prove that if σ and τ are two substitutions that agree on the variables of the term t, then $t\sigma = t\tau$. Use structural induction on t.

5.2.3. Prove Proposition 5.2.4 by structural induction on t.

5.2.4. Prove Proposition 5.2.7.

5.2.5. Prove Proposition 5.2.9 by showing the substitutions $\sigma_1\sigma_2$ and $\{x_1/(t_1\sigma_2), \ldots, x_n/(t_n\sigma_2), z_1/(z_1\sigma_2), \ldots, z_m/(z_m\sigma_2)\}$ have the same action on each variable.

5.2.6. Prove that if Φ is a formula and σ is a substitution, then $\Phi\sigma$ is a formula. Use structural induction on Φ.

5.2.7. Prove that if σ and τ are two substitutions that agree on the free variables of the formula Φ, then $\Phi\sigma = \Phi\tau$. Use structural induction on Φ.

5.2.8. Prove that a formula Φ is a sentence if and only if $\Phi\sigma = \Phi$ for every substitution σ.

5.3 First-Order Semantics

It is more complicated to give meaning to a formula of first-order logic than it was in the propositional case. We must say what we are talking about, what *domain* is involved for the quantifiers to quantify over. We must say how we are interpreting the constant, function, and relation symbols with respect to that domain, an *interpretation*. These two items specify a *model*. (Models are also called *structures*.) In fact, the notion of a model is relative to a first-order language, since that determines what symbols we have to interpret. And finally, since formulas may contain free variables, we must say what they stand for, that is, give an *assignment* of values to them.

Definition 5.3.1 A *model* for the first-order language $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ is a pair $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ where:

$\mathbf{D}$ is a nonempty set, called the *domain* of $\mathbf{M}$.

$\mathbf{I}$ is a mapping, called an *interpretation* that associates:

To every constant symbol $c \in \mathbf{C}$, some member $c^{\mathbf{I}} \in \mathbf{D}$.

To every n-place function symbol $f \in \mathbf{F}$, some n-ary function $f^{\mathbf{I}} : \mathbf{D}^n \to \mathbf{D}$.

To every n-place relation symbol $P \in \mathbf{R}$, some n-ary relation $P^{\mathbf{I}} \subseteq \mathbf{D}^n$.

Definition 5.3.2 An *assignment* in a model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ is a mapping $\mathbf{A}$ from the set of variables to the set $\mathbf{D}$. We denote the image of the variable v under an assignment $\mathbf{A}$ by $v^{\mathbf{A}}$.

Suppose we have an interpretation, which gives meanings to the constant and function symbols of the language, and we have an assignment, which gives values to variables. We have enough information to calculate values for arbitrary terms.

Definition 5.3.3 Let $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ be a model for the language $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, and let $\mathbf{A}$ be an assignment in this model. To each term t of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, we associate a value $t^{\mathbf{I},\mathbf{A}}$ in $\mathbf{D}$ as follows:

1. For a constant symbol c, $c^{\mathbf{I},\mathbf{A}} = c^{\mathbf{I}}$.
2. For a variable v, $v^{\mathbf{I},\mathbf{A}} = v^{\mathbf{A}}$.
3. For a function symbol f, $[f(t_1, \ldots, t_n)]^{\mathbf{I},\mathbf{A}} = f^{\mathbf{I}}(t_1^{\mathbf{I},\mathbf{A}}, \ldots, t_n^{\mathbf{I},\mathbf{A}})$.

This definition (which is by structural recursion) associates a value in $\mathbf{D}$ with each term of the language. If the term is closed (has no variables), its value does not depend on the assignment $\mathbf{A}$. For closed terms we often write $t^{\mathbf{I}}$ instead of $t^{\mathbf{I},\mathbf{A}}$ to emphasize this point.

Example Suppose the language L has a constant symbol 0, a one-place function symbol s, and a two-place function symbol $+$. We will write $+$ in infix position, $x+y$ instead of $+(x, y)$. Now $s(s(0)+s(x))$ and $s(x+s(x+s(0)))$ are terms of L. We consider several choices of model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ and assignment $\mathbf{A}$.

1. $\mathbf{D} = \{0, 1, 2, \ldots\}$, $0^{\mathbf{I}} = 0$, $s^{\mathbf{I}}$ is the successor function, and $+^{\mathbf{I}}$ is the addition operation. Then, if $\mathbf{A}$ is an assignment such that $x^{\mathbf{A}} = 3$, $(s(s(0) + s(x)))^{\mathbf{I},\mathbf{A}} = 6$ and $(s(x + s(x + s(0))))^{\mathbf{I},\mathbf{A}} = 9$. More generally, $(s(s(0)+s(x)))^{\mathbf{I},\mathbf{A}} = x^{\mathbf{A}}+3$ and $(s(x+s(x+s(0))))^{\mathbf{I},\mathbf{A}} = 2x^{\mathbf{A}} + 3$.
2. $\mathbf{D}$ is the collection of all words over the alphabet $\{a, b\}$, $0^{\mathbf{I}} = a$, $s^{\mathbf{I}}$ is the operation that appends a to the end of a word, and $+^{\mathbf{I}}$ is concatenation. Then, if $\mathbf{A}$ is an assignment such that $x^{\mathbf{A}} = aba$, $(s(s(0) + s(x)))^{\mathbf{I},\mathbf{A}} = aaabaaa$ and $(s(x + s(x + s(0))))^{\mathbf{I},\mathbf{A}} = abaabaaaaa$.
3. $\mathbf{D} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, $0^{\mathbf{I}} = 1$, $s^{\mathbf{I}}$ is the predecessor function, and $+^{\mathbf{I}}$ is the subtraction operation. Then $(s(s(0) + s(x)))^{\mathbf{I},\mathbf{A}} = -x^{\mathbf{A}}$ and $(s(x + s(x + s(0))))^{\mathbf{I},\mathbf{A}} = 0$.

Next, we associate a truth value with each formula of the language, relative to a model and an assignment. For this we need a preliminary piece of terminology.

Definition 5.3.4 Let x be a variable. The assignment $\mathbf{B}$ in the model $\mathbf{M}$ is an *x-variant* of the assignment $\mathbf{A}$, provided $\mathbf{A}$ and $\mathbf{B}$ assign the same values to every variable except possibly x.

Definition 5.3.5 Let $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ be a model for the language $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, and let $\mathbf{A}$ be an assignment in this model. To each formula Φ of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, we associate a truth value $\Phi^{\mathbf{I},\mathbf{A}}$ ($\mathbf{t}$ or $\mathbf{f}$) as follows:

1. For the atomic cases,
$$[P(t_1, \ldots, t_n)]^{\mathbf{I},\mathbf{A}} = \mathbf{t} \Longleftrightarrow \langle t_1^{\mathbf{I},\mathbf{A}}, \ldots, t_n^{\mathbf{I},\mathbf{A}} \rangle \in P^{\mathbf{I}},$$
$$\top^{\mathbf{I},\mathbf{A}} = \mathbf{t},$$
$$\bot^{\mathbf{I},\mathbf{A}} = \mathbf{f}.$$
2. $[\neg X]^{\mathbf{I},\mathbf{A}} = \neg[X^{\mathbf{I},\mathbf{A}}]$.
3. $[X \circ Y]^{\mathbf{I},\mathbf{A}} = X^{\mathbf{I},\mathbf{A}} \circ Y^{\mathbf{I},\mathbf{A}}$.
4. $[(\forall x)\Phi]^{\mathbf{I},\mathbf{A}} = \mathbf{t} \Longleftrightarrow \Phi^{\mathbf{I},\mathbf{B}} = \mathbf{t}$ for every assignment $\mathbf{B}$ in $\mathbf{M}$ that is an x-variant of $\mathbf{A}$.
5. $[(\exists x)\Phi]^{\mathbf{I},\mathbf{A}} = \mathbf{t} \Longleftrightarrow \Phi^{\mathbf{I},\mathbf{B}} = \mathbf{t}$ for some assignment $\mathbf{B}$ in $\mathbf{M}$ that is an x-variant of $\mathbf{A}$.

Just as with terms, if the formula Φ contains no free variables, its truth value will not depend on the particular assignment used, so we generally write $\Phi^{\mathbf{I}}$ instead of $\Phi^{\mathbf{I},\mathbf{A}}$, to emphasize this.

Definition 5.3.6 A formula Φ of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ is *true in the model* $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ for $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ provided, $\Phi^{\mathbf{I},\mathbf{A}} = \mathbf{t}$ for all assignments $\mathbf{A}$. A formula Φ is *valid* if Φ is true in all models for the language. A set S of formulas is *satisfiable* in $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$, provided there is some assignment $\mathbf{A}$ (called a *satisfying assignment*) such that $\Phi^{\mathbf{I},\mathbf{A}} = \mathbf{t}$ for all $\Phi \in S$. S is *satisfiable* if it is satisfiable in some model.

Example For the following, suppose we have a language L with a two-place relation symbol R and a two-place function symbol $\oplus$; also suppose we have a model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$.

1. Consider the sentence $(\exists y)R(x, y \oplus y)$. Suppose $\mathbf{D} = \{1, 2, 3, \ldots\}$, $\oplus^{\mathbf{I}}$ is the addition operation, and $R^{\mathbf{I}}$ is the equality relation. Then $(\exists y)R(x, y \oplus y)^{\mathbf{I},\mathbf{A}}$ is true if and only if $x^{\mathbf{A}}$ is an even number.
2. This time consider the sentence $(\forall x)(\forall y)(\exists z)R(x \oplus y, z)$, where again $\mathbf{D} = \{1, 2, 3, \ldots\}$ and $\oplus^{\mathbf{I}}$ is the addition operation, but $R^{\mathbf{I}}$ is the greater-than relation. It is easy to see that the sentence is true in $\mathbf{M}$ if, for every counting number x and every counting number y, there is a counting number z such that $x + y > z$. Since this is in fact the case, the sentence is true in $\mathbf{M}$.

3. Use the same sentence as in item 2, but this time use the model where $\mathbf{D} = \{1, 2, 3, \ldots\}$, $\oplus^{\mathbf{I}}$ is addition, but $R^{\mathbf{I}}$ is the greater-than-by-4-or-more relation. In this model the sentence is not true. Since there is a model in which it is not true, the sentence is not valid. Likewise, item 2 shows the negation of the sentence is not valid either.

4. The sentence is $(\forall x)(\forall y)\{R(x,y) \supset (\exists z)[R(x,z) \wedge R(z,y)]\}$, $\mathbf{D}$ is the set of real numbers, and $R^{\mathbf{I}}$ is the greater-than relation. The sentence is true in this model (it expresses the denseness of the reals). If we change $\mathbf{D}$ to the counting numbers, the sentence will be false in the model.

5. $(\forall x)(\forall y)[R(x,y) \supset R(y,x)]$. Nothing requires domains to be infinite. Consider the model where $\mathbf{D} = \{7, 8\}$ and $R^{\mathbf{I}}$ is the relation that holds of $\langle 7, 8\rangle$ but not of $\langle 7, 7\rangle$, $\langle 8, 8\rangle$, or $\langle 8, 7\rangle$. In this model the sentence is not true.

In formal theorem proving, we (generally) are interested in establishing that some formula is valid. We can turn arbitrary formulas into sentences by universally quantifying away any free variables. Exercise 5.3.2 says this preserves truth in models. Consequently, we need only concern ourselves with establishing the validity of sentences.

The definition of model, and hence of satisfiability, depends on the language used, but Exercise 5.3.5 shows this dependence is not really critical.

The following proposition will play a role in proving the soundness of proof procedures in the next chapter. Since its verification is not difficult, we leave it as an exercise.

Proposition 5.3.7 *Suppose t is a closed term, Φ is a formula of the first-order language L, and $\mathbf{M} = \langle \mathbf{D}, \mathbf{I}\rangle$ is a model for L. Let x be a variable, and let $\mathbf{A}$ be any assignment such that $x^{\mathbf{A}} = t^{\mathbf{I}}$. Then $[\Phi\{x/t\}]^{\mathbf{I},\mathbf{A}} = \Phi^{\mathbf{I},\mathbf{A}}$. More generally, if $\mathbf{B}$ is any x-variant of $\mathbf{A}$ then $[\Phi\{x/t\}]^{\mathbf{I},\mathbf{B}} = \Phi^{\mathbf{I},\mathbf{A}}$.*

Finally, we have the following important result, relating substitutions and models.

Proposition 5.3.8 *Suppose $\mathbf{M} = \langle \mathbf{D}, \mathbf{I}\rangle$ is a model for the language L, Φ is a formula in this language, $\mathbf{A}$ is an assignment in $\mathbf{M}$, and σ is a substitution that is free for Φ. Define a new assignment $\mathbf{B}$ by setting, for each variable v, $v^{\mathbf{B}} = (v\sigma)^{\mathbf{I},\mathbf{A}}$. Then $\Phi^{\mathbf{I},\mathbf{B}} = (\Phi\sigma)^{\mathbf{I},\mathbf{A}}$.*

Proof We first need to know that, for any term t, $t^{\mathbf{I},\mathbf{B}} = (t\sigma)^{\mathbf{I},\mathbf{A}}$. This is a simple structural induction on t, which we omit.

Now, the result is shown by structural induction on Φ. The atomic case follows immediately from the result just cited concerning substitution in terms. The various propositional cases are straightforward. The quantifier cases are the significant ones.

Suppose the result is known for formulas simpler than Φ, and $\Phi = (\exists x)\varphi$. Further, suppose $[\Phi\sigma]^{\mathbf{I},\mathbf{A}} = \mathbf{t}$. We show $\Phi^{\mathbf{I},\mathbf{B}} = \mathbf{t}$.

By our assumption, $[[(\exists x)\varphi]\sigma]^{\mathbf{I},\mathbf{A}} = [(\exists x)[\varphi\sigma_x]]^{\mathbf{I},\mathbf{A}} = \mathbf{t}$. Then for some x-variant $\mathbf{A}'$ of $\mathbf{A}$, $[\varphi\sigma_x]^{\mathbf{I},\mathbf{A}'} = \mathbf{t}$. Now, define a new assignment $\mathbf{B}'$ by setting, for each variable v, $v^{\mathbf{B}'} = (v\sigma_x)^{\mathbf{I},\mathbf{A}'}$. Since σ is free for $(\exists x)\varphi$, σ_x is free for φ. Then by the induction hypothesis, $[\varphi\sigma_x]^{\mathbf{I},\mathbf{A}'} = \varphi^{\mathbf{I},\mathbf{B}'}$, hence, $\varphi^{\mathbf{I},\mathbf{B}'} = \mathbf{t}$. It will follow from this that $[(\exists x)\varphi]^{\mathbf{I},\mathbf{B}} = \Phi^{\mathbf{I},\mathbf{B}} = \mathbf{t}$ once we show that $\mathbf{B}'$ is an x-variant of $\mathbf{B}$.

Since σ is free for $(\exists x)\varphi$, if v is any variable except x, $v\sigma$ does not contain x. Now, if $v \neq x$, $v^{\mathbf{B}'} = (v\sigma_x)^{\mathbf{I},\mathbf{A}'} =$ (since $v \neq x$) $(v\sigma)^{\mathbf{I},\mathbf{A}'} =$ (since $v\sigma$ cannot contain x, and $\mathbf{A}$ and $\mathbf{A}'$ agree on all variables except x) $(v\sigma)^{\mathbf{I},\mathbf{A}} = v^{\mathbf{B}}$ (by definition). Thus $\mathbf{B}'$ is an x-variant of $\mathbf{B}$.

We have now showed that, given the induction hypothesis, if $[\Phi\sigma]^{\mathbf{I},\mathbf{A}} = \mathbf{t}$ then $\Phi^{\mathbf{I},\mathbf{B}} = \mathbf{t}$. The argument in the converse direction is similar, as is the universal quantifier case. □

Exercises

5.3.1. Prove Proposition 5.3.7.

5.3.2. Show Φ is true in a model $\mathbf{M}$ if and only if $(\forall x)\Phi$ is true in $\mathbf{M}$.

5.3.3. Show X is valid if and only if $\{\neg X\}$ is not satisfiable.

5.3.4. Show $X \equiv Y$ is true in $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ if and only if $X^{\mathbf{I},\mathbf{A}} = Y^{\mathbf{I},\mathbf{A}}$ for all assignments $\mathbf{A}$.

5.3.5. Let L and L' be first-order languages, with every constant, function and relation symbol of L also a symbol of L'. Let S be a set of formulas of L. Show S is satisfiable in some model for the language L if and only if S is satisfiable in some model for the language L'.

5.3.6. Write a sentence Φ involving the two-place relation symbol R such that:

1. Φ Is true in $\langle \mathbf{D}, \mathbf{I} \rangle$ if and only if $R^{\mathbf{I}}$ is a reflexive relation on $\mathbf{D}$.
2. Φ Is true in $\langle \mathbf{D}, \mathbf{I} \rangle$ if and only if $R^{\mathbf{I}}$ is a symmetric relation on $\mathbf{D}$.
3. Φ Is true in $\langle \mathbf{D}, \mathbf{I} \rangle$ if and only if $R^{\mathbf{I}}$ is a transitive relation on $\mathbf{D}$.

4. Φ Is true in $\langle \mathbf{D}, \mathbf{I} \rangle$ if and only if $R^{\mathbf{I}}$ is an equivalence relation on $\mathbf{D}$.

5.3.7. Write a sentence Φ involving the two-place relation symbol R having both the following properties:

1. Φ Is not true in any model $\langle \mathbf{D}, \mathbf{I} \rangle$ with a one element domain.
2. if $\mathbf{D}$ has two or more elements, there is some interpretation $\mathbf{I}$ such that Φ is true in $\langle \mathbf{D}, \mathbf{I} \rangle$.

5.3.8. Write a sentence Φ involving the two-place relation symbol R having both the following properties:

1. Φ Is not true in any model $\langle \mathbf{D}, \mathbf{I} \rangle$ with a one- or a two-element domain.
2. If $\mathbf{D}$ has three or more elements, there is some interpretation $\mathbf{I}$ such that Φ is true in $\langle \mathbf{D}, \mathbf{I} \rangle$.

5.3.9. Write a sentence Φ involving the two-place relation symbol R having the following properties:

1. Φ Is not true in any model $\langle \mathbf{D}, \mathbf{I} \rangle$ with a finite domain.
2. If $\mathbf{D}$ is infinite, there is some interpretation $\mathbf{I}$ such that Φ is true in $\langle \mathbf{D}, \mathbf{I} \rangle$.

5.3.10. In the following P and R are relation symbols and c is a constant symbol. Demonstrate the validity of the following:

1. $(\forall x)P(x) \supset P(c)$.
2. $(\exists x)[P(x) \supset (\forall x)P(x)]$.
3. $(\exists y)(\forall x)R(x, y) \supset (\forall x)(\exists y)R(x, y)$.
4. $(\forall x)\Phi \equiv \neg(\exists x)\neg\Phi$.
5. Determine the status of $(\forall x)(\exists y)R(x, y) \supset (\exists y)(\forall x)R(x, y)$.

5.4 Herbrand Models

Assignments are almost substitutions, but not quite. The problem is as follows: An assignment maps variables to members of a domain $\mathbf{D}$, and the members of $\mathbf{D}$ probably will not be terms of the formal language we are using, so if we replace a variable in a formula by what an assignment maps it to, we will not get a formula as a result. Still, it is convenient to think of assignments and substitutions together, and in certain cases we can. The domain of a model can be anything we like; in particular its members *could* be terms of the language L. If we are in such a fortunate situation, assignments *are* substitutions.

Definition 5.4.1 A model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ for the language L is a *Herbrand model* if:

1. $\mathbf{D}$ is exactly the set of closed terms of L.
2. For each closed term t, $t^{\mathbf{I}} = t$.

It may seem that Herbrand models are rather special. In fact, they do play a special role in our completeness proofs. By design, an assignment $\mathbf{A}$ in a Herbrand model $\mathbf{M}$ is also a substitution for the language L, and conversely, and so for a formula Φ of L, both $\Phi^{\mathbf{I},\mathbf{A}}$ (where $\mathbf{A}$ is used as an assignment) and $\Phi\mathbf{A}$ (where $\mathbf{A}$ is used as a substitution) are meaningful. The connection between these two roles of $\mathbf{A}$ is a simple one and is given in the following two propositions. Before stating them, we note that for any formula Φ of L, and for any assignment $\mathbf{A}$ in a Herbrand model, thought of as a substitution, $\Phi\mathbf{A}$ has no free variables. Then a truth value for $\Phi\mathbf{A}$ depends only on the interpretation of the model. Similar observations apply to terms.

Proposition 5.4.2 *Suppose* $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ *is a Herbrand model for the language* L*. For any term* t *of* L*, not necessarily closed,* $t^{\mathbf{I},\mathbf{A}} = (t\mathbf{A})^{\mathbf{I}}$.

Proof By structural induction on t. We begin with the ground cases. Suppose t is a variable, v. Then $t^{\mathbf{I},\mathbf{A}} = v^{\mathbf{I},\mathbf{A}} = v^{\mathbf{A}}$, and $(t\mathbf{A})^{\mathbf{I}} = (v\mathbf{A})^{\mathbf{I}} = v\mathbf{A}$, since $\mathbf{I}$ is the identity on closed terms. And finally, $v^{\mathbf{A}} = v\mathbf{A}$. This completes the argument if t is a variable. Next suppose t is a constant symbol, c, of L. Then $t^{\mathbf{I},\mathbf{A}} = c^{\mathbf{I},\mathbf{A}} = c^{\mathbf{I}}$, and $(t\mathbf{A})^{\mathbf{I}} = (c\mathbf{A})^{\mathbf{I}} = c^{\mathbf{I}}$.

Suppose the result is known for the terms $t_1,\ldots,t_n$, and we have the term $t = f(t_1,\ldots,t_n)$. Then $t^{\mathbf{I},\mathbf{A}} = [f(t_1,\ldots,t_n)]^{\mathbf{I},\mathbf{A}} = f^{\mathbf{I}}(t_1^{\mathbf{I},\mathbf{A}}, \ldots, t_n^{\mathbf{I},\mathbf{A}})$. Also $(t\mathbf{A})^{\mathbf{I}} = [f(t_1,\ldots,t_n)\mathbf{A}]^{\mathbf{I}} = [f(t_1\mathbf{A},\ldots,t_n\mathbf{A})]^{\mathbf{I}} = f^{\mathbf{I}}((t_1\mathbf{A})^{\mathbf{I}}, \ldots, (t_n\mathbf{A})^{\mathbf{I}})$. Now we are done, by the induction hypothesis. □

Proposition 5.4.3 *Suppose* $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ *is a Herbrand model for the language* L*. For a formula* Φ *of* L*,* $\Phi^{\mathbf{I},\mathbf{A}} = (\Phi\mathbf{A})^{\mathbf{I}}$.

A nice feature of working with Herbrand models is that quantifier truth conditions become much simplified. The following states this formally, using our notation for substitutions having finite support.

Proposition 5.4.4 *Suppose Φ is a formula of L and $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ is a Herbrand model for L. Then:*

1. *$(\forall x)\Phi$ is true in $\mathbf{M} \Leftrightarrow \Phi\{x/d\}$ is true in $\mathbf{M}$ for every $d \in \mathbf{D}$.*
2. *$(\exists x)\Phi$ is true in $\mathbf{M} \Leftrightarrow \Phi\{x/d\}$ is true in $\mathbf{M}$ for some $d \in \mathbf{D}$.*

Exercises

5.4.1. Prove Proposition 5.4.3.

5.4.2. Prove Proposition 5.4.4.

5.5 First-Order Uniform Notation

Earlier we introduced a system of uniform notation for propositional logic. Here, again following Smullyan [48], we extend it to include quantifiers. All quantified formulas and their negations are grouped into two categories, those that act *universally*, which are called *γ-formulas*, and those that act *existentially*, which are called *δ-formulas*. For each variety and for each term t, an *instance* is defined. The groups and the notions of instance are given in Table 5.1.

Universal		Existential	
γ	$\gamma(t)$	δ	$\delta(t)$
$(\forall x)\Phi$	$\Phi\{x/t\}$	$(\exists x)\Phi$	$\Phi\{x/t\}$
$\neg(\exists x)\Phi$	$\neg\Phi\{x/t\}$	$\neg(\forall x)\Phi$	$\neg\Phi\{x/t\}$

TABLE 5.1. γ- and δ-Formulas and Instances

Essentially, all γ-formulas act universally and all δ-formulas act existentially. More precisely, both $\gamma \equiv (\forall y)\gamma(y)$ and $\delta \equiv (\exists y)\delta(y)$ are valid, provided y is a variable that is new to γ and δ.

When we come to prove tableau and resolution soundness, the following will play a central role:

Proposition 5.5.1 *Let S be a set of sentences, and γ and δ be sentences.*

1. *If $S \cup \{\gamma\}$ is satisfiable, so is $S \cup \{\gamma, \gamma(t)\}$ for any closed term t.*
2. *If $S \cup \{\delta\}$ is satisfiable, so is $S \cup \{\delta, \delta(p)\}$ for any constant symbol p that is new to S and δ.*

Proof We are working with sentences, which makes things slightly easier for us.

Part 1 Suppose $S \cup \{\gamma\}$ is satisfiable in the model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$. We show $S \cup \{\gamma, \gamma(t)\}$ is satisfiable in the same model. Since γ is true in this model, so is $(\forall x)\gamma(x)$ (where x is a variable new to γ). Then for every assignment $\mathbf{A}$, $[\gamma(x)]^{\mathbf{I},\mathbf{A}}$ is true. Now, let $\mathbf{A}$ be an assignment such that $x^{\mathbf{A}} = t^{\mathbf{I}}$. By Proposition 5.3.7, $[\gamma(t)]^{\mathbf{I},\mathbf{A}} = [\gamma\{x/t\}]^{\mathbf{I},\mathbf{A}} = [\gamma(x)]^{\mathbf{I},\mathbf{A}} = \mathbf{t}$.

Part 2 Suppose $S \cup \{\delta\}$ is satisfiable in $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$, and p is a constant symbol new to S and δ. We will show $S \cup \{\delta, \delta(p)\}$ is satisfiable, though not necessarily in the model $\mathbf{M}$.

Since δ is true in $\mathbf{M}$, so is $(\exists x)\delta(x)$ (x new to δ), and hence $[\delta(x)]^{\mathbf{I},\mathbf{A}}$ is true for some assignment $\mathbf{A}$. If we had $x^{\mathbf{A}} = p^{\mathbf{I}}$, we could complete the argument just as we did in part 1, but there is no reason to suppose this happens. So, we construct a new model $\mathbf{M}^* = \langle \mathbf{D}, \mathbf{J} \rangle$, having the same domain, with an interpretation $\mathbf{J}$ that is exactly the same as $\mathbf{I}$ on everything except p, and for that case we set $p^{\mathbf{J}} = x^{\mathbf{A}}$. Now, since the two models differ only with respect to the constant symbol p, sentences not containing p will behave the same in the two models. Then $S \cup \{\delta\}$ is satisfiable in $\mathbf{M}^*$, and $[\delta(x)]^{\mathbf{J},\mathbf{A}}$ is true as well. Since $x^{\mathbf{A}} = p^{\mathbf{J}}$, we have $[\delta(p)]^{\mathbf{J},\mathbf{A}} = [\delta\{x/p\}]^{\mathbf{J},\mathbf{A}} = [\delta(x)]^{\mathbf{J},\mathbf{A}} = \mathbf{t}$. Then $S \cup \{\delta, \delta(p)\}$ is satisfiable, but in $\mathbf{M}^*$. □

When Herbrand models are involved, things become especially simple.

Proposition 5.5.2 *Suppose L is a first-order language and $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ is a Herbrand model for L.*

1. *If γ is a formula of L, γ is true in $\mathbf{M}$ if and only if $\gamma(d)$ is true in $\mathbf{M}$ for every $d \in \mathbf{D}$.*
2. *If δ is a formula of L, δ is true in $\mathbf{M}$ if and only if $\delta(d)$ is true in $\mathbf{M}$ for some $d \in \mathbf{D}$.*

We can also extend the notion of structural induction to make use of first-order uniform notation. We omit the proof, which is similar to that of the corresponding propositional version.

Theorem 5.5.3 **(First-Order Structural Induction)**
Every formula of a first-order language L has property $\mathbf{Q}$, provided:

Basis step *Each atomic formula and its negation has property $\mathbf{Q}$.*

Induction steps

If X has property **Q** *so does* $\neg\neg X$

If α_1 *and* α_2 *have property* **Q** *so does* α.

If β_1 *and* β_2 *have property* **Q** *so does* β.

If $\gamma(t)$ *has property* **Q** *for each term* t, γ *has property* **Q**.

If $\delta(t)$ *has property* **Q** *for each term* t, δ *has property* **Q**.

There is a similar extension of structural recursion.

Theorem 5.5.4 **(First-Order Structural Recursion)**
There is one, and only one, function f defined on the set of formulas of L such that:

Basis step *The value of f is specified explicitly on atomic formulas and their negations.*

Recursion steps

The value of f on $\neg\neg X$ is specified in terms of the value of f on X.

The value of f on α is specified in terms of the values of f on α_1 and α_2.

The value of f on β is specified in terms of the values of f on β_1 and β_2.

The value of f on γ is specified in terms of the values of f on $\gamma(t)$.

The value of f on δ is specified in terms of the values of f on $\delta(t)$.

Exercises

5.5.1. Prove that if the variable x does not occur in the sentence γ, and if t is a closed term, then $\gamma(t) = \gamma(x)\{x/t\}$. (Similarly for δ sentences too, of course.)

5.5.2. Prove Proposition 5.5.2.

5.6 Hintikka's Lemma

In the propositional case, we were able to obtain many fundamental results as direct corollaries of a single theorem, the Model Existence Theorem. This happy state of affairs continues. And just as before, it is convenient to separate out part of the work by first proving a result that is due to Hintikka. Of course, our task is considerably more difficult now than it was earlier, since the notion of first-order model is more complex than that of Boolean valuation.

Definition 5.6.1 Let L be some first-order language. A set $\mathbf{H}$ of sentences of L is called a *first-order Hintikka set* (with respect to L), provided $\mathbf{H}$ is a propositional Hintikka set (Definition 3.5.1), and in addition:

6. $\gamma \in \mathbf{H} \Rightarrow \gamma(t) \in \mathbf{H}$ for every closed term t of L.
7. $\delta \in \mathbf{H} \Rightarrow \delta(t) \in \mathbf{H}$ for some closed term t of L.

Just as in the propositional case, the empty set is trivially a first-order Hintikka set. So are many finite sets; we leave it to you to produce examples. But if L is a language with an infinite set of closed terms (which will be the case if L has a function symbol and a constant symbol), then any Hintikka set containing a γ-sentence must be infinite. The following essentially says that if L is a nontrivial first-order language, then every Hintikka set with respect to L is satisfiable:

Proposition 5.6.2 **(Hintikka's Lemma)** *Suppose L is a language with a nonempty set of closed terms. If $\mathbf{H}$ is a first-order Hintikka set with respect to L, then $\mathbf{H}$ is satisfiable in a Herbrand model.*

Proof Let $\mathbf{H}$ be a first-order Hintikka set with respect to L. We begin by constructing a model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$, then we verify that $\mathbf{H}$ is satisfiable in it.

Let $\mathbf{D}$ be the collection of closed terms of L, which is a nonempty set by the conditions of the Proposition.

We specify $\mathbf{I}$ on constant and function symbols. For a constant symbol c of L, $c^{\mathbf{I}} = c$. If f is an n-place function symbol of L, and $t_1, \ldots, t_n$ are members of $\mathbf{D}$ (hence closed terms of L), $f^{\mathbf{I}}(t_1, \ldots, t_n)$ is the closed term $f(t_1, \ldots, t_n)$. It can now be verified that, for each closed term t of L, $t^{\mathbf{I}} = \mathbf{t}$ (Exercise 5.6.2).

Next we define $\mathbf{I}$ on relation symbols. Suppose R is an n-place relation symbol of L and $t_1, \ldots, t_n$ are members of $\mathbf{D}$ (hence closed terms of L). The relation $R^{\mathbf{I}}$ holds of $\langle t_1, \ldots, t_n \rangle$ if the sentence $R(t_1, \ldots, t_n)$ is a member of $\mathbf{H}$. This completes the definition of the model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$.

Finally we show, by structural induction, that for each sentence X of L, $X \in \mathbf{H}$ implies X is true in $\mathbf{M}$. Once this is shown, we are done. We begin with the atomic cases.

Suppose the atomic sentence $R(t_1, \ldots, t_n)$ is in $\mathbf{H}$. We must show, for each assignment $\mathbf{A}$, $[R(t_1, \ldots, t_n)]^{\mathbf{I},\mathbf{A}} = \mathbf{t}$, which will be the case provided $\langle t_1^{\mathbf{I},\mathbf{A}}, \ldots, t_n^{\mathbf{I},\mathbf{A}} \rangle \in R^{\mathbf{I}}$. But since $R(t_1, \ldots, t_n)$ is a sentence, each t_i is a closed term, so $t_i^{\mathbf{I},\mathbf{A}} = t_i^{\mathbf{I}} = t_i$. Thus, what we need to show reduces to $\langle t_1, \ldots, t_n \rangle \in R^{\mathbf{I}}$, and we have this, since $R(t_1, \ldots, t_n)$ is in $\mathbf{H}$. The other atomic cases concerning $\top$ and $\bot$ are trivial, and the negation of atomic case is straightforward.

The propositional cases are essentially as they were earlier and are left to you. We consider one quantifier case and leave the other as an exercise.

Suppose γ is a sentence of L, the result is known for simpler sentences of L, and $\gamma \in \mathbf{H}$; we show γ is true in $\mathbf{M}$. Since $\gamma \in \mathbf{H}$, we have $\gamma(t) \in \mathbf{H}$ for every closed term t, since $\mathbf{H}$ is a Hintikka set. By the induction hypothesis, and the fact that $\mathbf{D}$ is exactly the set of closed terms, $\gamma(t)$ is true in $\mathbf{M}$ for every $t \in \mathbf{D}$. Then γ is true in $\mathbf{M}$ by Proposition 5.5.2. □

Exercises

5.6.1. Prove that if $\mathbf{H}$ is a first-order Hintikka set with respect to L and if X is any sentence of L, then not both $X \in \mathbf{H}$ and $\neg X \in \mathbf{H}$.

5.6.2. Prove that, in the model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ constructed in the proof of Hintikka's Lemma, for each closed term t of L, $t^{\mathbf{I}} = t$ and hence $\mathbf{M}$ is a Herbrand model.

5.6.3. Complete the proof of Proposition 5.6.2 by doing the remaining cases.

5.7 Parameters

We have not yet presented any proof procedures for first-order logic; indeed, we will not do so until the next chapter. But still, we have some feeling for what formal proofs might be like, from seeing informal ones in books and lectures. There is an important feature of both formal and informal proofs that is easily overlooked when logic itself is not the subject. Suppose, in the course of a proof, that we have established $(\exists x)\Phi$. We might want to introduce a 'name' for an item having the property Φ, given that we have shown there is such an item. Of course we can't say anything like "let 3 be something for which property Φ holds," since 3 already has a standard meaning, and having property Φ might not fit with that meaning. More generally, we can't use any term that has already been assigned a role. The solution is to introduce a new collection of terms that are 'uncommitted,' so that we have them available for this purpose. In a classroom lecture you probably have heard a version of "let

c be something having the property Φ." The constant c was something previously unused, though this was probably taken for granted and never mentioned explicitly. We will do the same thing here. The new constant symbols we introduce will be called *constant parameters* or just *parameters*. As it happens, the introduction of parameters plays a role not only in proof procedures, but in the Model Existence Theorem as well.

Definition 5.7.1 Let $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ be a first-order language, which we abbreviate as L. Let **par** be a countable set of constant symbols that is disjoint from $\mathbf{C}$. We call the members of **par** *parameters*. And we use $L^{\mathbf{par}}$ as shorthand for the language $L(\mathbf{R}, \mathbf{F}, \mathbf{C} \cup \mathbf{par})$.

When we come to formal proofs, we will see that they are *of* sentences of L but may *use* sentences of $L^{\mathbf{par}}$. This is in keeping with informal mathematical practice in which we may introduce new constant symbols during the course of a proof, symbols that had no meaning when we stated the theorem being proved.

Exercises

5.7.1. Let p be a parameter. Show that $\Phi\{x/p\}$ is valid if and only if $(\forall x)\Phi(x)$ is valid.

5.8 The Model Existence Theorem

Once again we extend a result from the propositional to the first-order setting. And once again, the extension is considerably more difficult than the original. We begin with the definition of consistency property, continuing the earlier propositional definition. Several different versions of first-order consistency property are used in the literature. They are all rather similar and are used for the same purposes, but on small details they diverge at many points. Be warned. One minor piece of terminology first: We call a parameter *new* to a set of formulas if it does not occur in any formula of the set.

Definition 5.8.1 Let L be a first-order language, and let $L^{\mathbf{par}}$ be the extension containing an infinite set of additional constant symbols, parameters. Let $\mathcal{C}$ be a collection of sets of sentences of $L^{\mathbf{par}}$. We call $\mathcal{C}$ a *first-order consistency property* (with respect to L) if it is a propositional consistency property as in Definition 3.6.1 and, in addition, for each $S \in \mathcal{C}$:

6. $\gamma \in S \Rightarrow S \cup \{\gamma(t)\} \in \mathcal{C}$ for every closed term t of $L^{\mathbf{par}}$.
7. $\delta \in S \Rightarrow S \cup \{\delta(p)\} \in \mathcal{C}$ for some parameter p of $L^{\mathbf{par}}$.

Theorem 5.8.2 **(First-Order Model Existence)** *If C is a first-order consistency property with respect to L, S is a set of sentences of L, and $S \in C$, then S is satisfiable; in fact S is satisfiable in a Herbrand model (but Herbrand with respect to L^{par}).*

The rest of the section will be spent in proving this theorem. The argument has several distinct parts, much as it did in the propositional case. Recall, there we began by enlarging a consistency property to one that was of finite character. Once we had done this, we were able to extend any member to a maximal one, which turned out to be a Hintikka set. We follow a similar plan here, but the quantifier conditions make life just a little more difficult. Still, the proof closely follows that of the propositional version, and you should review the proof of Theorem 3.6.2 before going on.

Just as in the propositional setting, we call a first-order consistency property *subset closed* if it contains, with each member, all subsets of that member. We leave it to you to verify that every first-order consistency property can be extended to one that is subset closed.

In the propositional argument, the next step was to extend to a consistency property of finite character. We can not do that now, because we can not ensure the result meets condition 7. (Try it.) So we proceed in a more roundabout fashion. The basic intuition is that parameters, having no preassigned roles, are essentially interchangeable. That is, if p and q are parameters, wherever we use p, we could use q just as well. But introducing this idea forces us to modify the notion of consistency property somewhat.

Definition 5.8.3 An *alternate first-order consistency property* is a collection C meeting the conditions for a first-order consistency property, except that condition 7 has been replaced by

7′. $\delta \in S \Rightarrow S \cup \{\delta(p)\} \in C$ for every parameter p that is new to S.

This alternate version is both weaker and stronger than the original. It is stronger in the sense that it (generally) says lots of instances of a δ-sentence can be added to S, not just one. It is weaker in the sense that, if all parameters already occur in formulas of S, so that none are new, no instances at all can be added.

Definition 5.8.4 A *parameter substitution* is a mapping π from the set of parameters to itself (not necessarily 1-1 or onto). If π is a parameter substitution, we extend its action to formulas of L^{par} as follows: $\Phi\pi$ means replace each parameter occurring in Φ by its image under π. Likewise, the action of π is extended to sets of formulas by applying it to each member of the set.

Lemma 5.8.5 *Suppose $\mathcal{C}$ is a first-order consistency property that is closed under subsets. Define a collection $\mathcal{C}^+$ as follows: $S \in \mathcal{C}^+$, provided $S\pi \in \mathcal{C}$ for some parameter substitution π. Then:*

1. *$\mathcal{C}^+$ extends $\mathcal{C}$.*
2. *$\mathcal{C}^+$ is closed under subsets.*
3. *$\mathcal{C}^+$ is an alternate first-order consistency property.*

We leave the proof of this Lemma to you and continue with the argument for the Model Existence Theorem. As in the propositional version, we say an alternate first-order consistency property $\mathcal{C}$ is of *finite character*, provided $S \in \mathcal{C}$ if and only if every finite subset of S belongs to $\mathcal{C}$. Every alternate first-order consistency property that is subset closed can be extended to one of finite character. We leave the verification of this to you as well.

Now we have the necessary background out of the way. Suppose $\mathcal{C}$ is a first-order consistency property with respect to L, S is a set of sentences of L, and $S \in \mathcal{C}$. We construct a model in which the members of S are true. First, extend $\mathcal{C}$ to $\mathcal{C}^*$, an alternate first-order consistency property of finite character. We work with $\mathcal{C}^*$ (which also contains S), rather than with $\mathcal{C}$.

Since the language L^{par} has a countable alphabet, it follows that there are a countable number of sentences of the language. Let $X_1, X_2, X_3, \ldots$ be an enumeration of all sentences of L^{par}. Now we define a sequence $S_1, S_2, S_3, \ldots$ of members of $\mathcal{C}^*$, each of which leaves unused an infinite set of parameters. (Note: This is true of S, since S is a set of sentences of L and so contains no parameters.)

$$S_1 = S.$$

Having defined S_n, which leaves unused infinitely many parameters, we define S_{n+1} as follows:

If $S_n \cup \{X_n\} \notin \mathcal{C}^*$, let $S_{n+1} = S_n$.

If $S_n \cup \{X_n\} \in \mathcal{C}^*$, and X_n is not a δ sentence, let $S_{n+1} = S_n \cup \{X_n\}$.

If $S_n \cup \{X_n\} \in \mathcal{C}^*$, and $X_n = \delta$, infinitely many parameters will be new to $S_n \cup \{X_n\}$; choose one, say p, and let $S_{n+1} = S_n \cup \{X_n\} \cup \{\delta(p)\}$.

By construction, each $S_n \in \mathcal{C}^*$, and each S_n is a subset of S_{n+1}. Finally, let $\mathbf{H} = S_1 \cup S_2 \cup S_3 \cup \ldots$. $\mathbf{H}$ extends S.

Claim 1 $\mathbf{H} \in \mathcal{C}^*$. The proof is exactly as it was in the propositional case and depends on the fact that $\mathcal{C}^*$ is of finite character.

Claim 2 $\mathbf{H}$ is maximal in $\mathcal{C}^*$. Again, the proof is exactly as in the propositional case.

Claim 3 $\mathbf{H}$ is a first-order Hintikka set with respect to L^{par}. Here the argument is essentially the same as in the propositional case, except for the δ-condition. But we have taken care of that separately, during the construction of the S_n sequence.

Now by Hintikka's Lemma, $\mathbf{H}$, and hence S, is satisfiable in a model that is Herbrand with respect to L^{par}.

Exercises

5.8.1. Show that every first-order consistency property can be extended to one that is subset closed. (See Exercise 3.6.1.)

5.8.2. Prove Lemma 5.8.5.

5.8.3. Show that an alternate first-order consistency property that is subset closed can be extended to one of finite character. (See Exercise 3.6.3.)

5.9 Applications

For us, the main uses of the Model Existence Theorem will be in proving completeness. But there are many other important applications of it. In this section we consider some particularly fundamental ones, among them the Compactness Theorem and the Löwenheim-Skolem Theorem. We proved a propositional version of the Compactness Theorem earlier, Theorem 3.6.3. The Löwenheim-Skolem Theorem has no propositional analog.

Theorem 5.9.1 **(First-Order Compactness)** *Let S be a set of sentences of the first-order language L. If every finite subset of S is satisfiable, so is S.*

Proof Let $\mathcal{C}$ be a collection of sets of sentences of L^{par}, constructed as follows. Put a set W in $\mathcal{C}$, provided (1) infinitely many parameters are new to W, and (2) every finite subset of W is satisfiable. $S \in \mathcal{C}$; it meets condition 1 since its members are sentences of L, which contain no parameters, and it meets condition 2 by assumption. $\mathcal{C}$ is a first-order consistency property, hence the satisfiability of S follows immediately by the First-Order Model Existence Theorem 5.8.2. □

The Compactness Theorem is one of the most powerful tools in model theory. We illustrate its uses by giving an easy proof of the following remarkable result.

Corollary 5.9.2 *Let L be a first-order language. Any set S of sentences of L that is satisfiable in arbitrarily large finite models is satisfiable in some infinite model.*

Proof Suppose S is satisfiable in arbitrarily large finite models. Let R be a two-place relation symbol that is not part of the language L, and enlarge L to L' by adding R. From now on we work with L' with the understanding that, in any model, we can modify the interpretation of R without affecting the truth values of members of S, since R does not occur in members of S. In Exercise 5.3.7 you were asked to write a sentence involving R, call it A_2, such that A_2 is not true in any one element model, but can be made true in any domain with two or more things by suitably interpreting R. We can think of A_2 as saying there are at least 2 things. Similarly, in Exercise 5.3.8 you were asked to write a sentence, call it A_3, that informally says there are at least 3 things. The ideas behind those exercises can be continued, and for each n we can produce a sentence A_n that asserts there are at least n things. Now consider the set $S^* = S \cup \{A_2, A_3, \ldots\}$. Since S is satisfiable in arbitrarily large finite models, it follows easily that every finite subset of S^* is satisfiable. Then by the Compactness Theorem, the entire set S^* is satisfiable. But any model in which the entire of S^* is satisfiable can not be finite, hence S is satisfiable in some infinite model. $\square$

In Exercise 5.3.9 you were asked to produce a sentence that was not true in any finite model but could be made true in any infinite domain by choosing a suitable interpretation. The corollary shows that the dual of this is impossible: There is no sentence that can be made true in any finite domain but is not true in any model with an infinite domain. *Thus, the notion of being finite can not be captured using the machinery of classical first-order logic.*

Theorem 5.9.3 **(Löwenheim-Skolem)** *Let L be a first-order language, and let S be a set of sentences of L. If S is satisfiable, then S is satisfiable in a countable model.*

Proof Form a collection $\mathcal{C}$ of sets of sentences of $L^{\mathbf{par}}$ as follows: Put a set W in $\mathcal{C}$, provided infinitely many parameters are new to W and W is satisfiable. We leave it to you to check that $\mathcal{C}$ is a first-order consistency property. If S is satisfiable, then $S \in \mathcal{C}$, so by the Model Existence Theorem, S is satisfiable in a model that is Herbrand with respect to

L^{par}. Now the language L^{par} has a countable alphabet, hence a countable collection of closed terms. And it is the collection of closed terms that constitutes the domain of a Herbrand model. □

Like the Compactness Theorem, the Löwenheim-Skolem Theorem also has remarkable consequences. One example will have to suffice here. Suppose we try to characterize the real-number system with a set of first-order formulas S. That is, the intended model of S should have the real numbers as its domain, and any other model should be isomorphic to the intended one. But the Löwenheim-Skolem Theorem says that if S is satisfiable in the intended model, it is also satisfiable in some countable model. Since the reals are uncountable, there must be a model for S that is not isomorphic to the intended one. The real number system has no first-order characterization.

Theorem 5.9.4 **(Herbrand Model)** *A set S of sentences of L is satisfiable if and only if it is satisfiable in a model that is Herbrand with respect to L^{par}. A sentence X of L is valid if and only if X is true in all models that are Herbrand with respect to L^{par}.*

Exercises

5.9.1. Complete the proof of Theorem 5.9.1 by verifying that $\mathcal{C}$ is a first-order consistency property.

5.9.2. A graph is a structure $G = \langle V, E\rangle$ where E is a symmetric, binary relation on V (V is meant to suggest "vertex" and E "edge"). A subgraph of G is a structure $\langle V^*, E^*\rangle$ where V^* is a subset of V and $E^* = E \cap (V^* \times V^*)$. A graph G is *four colorable* if there are subsets R, G, B, Y (meant to suggest red, green, blue, and yellow) of V such that $(\forall x \in V)(R(x) \vee G(x) \vee B(x) \vee Y(x))$ and $(\forall x \in V)(\forall y \in V)\{E(x,y) \supset [(R(x) \supset \neg R(y)) \wedge (G(x) \supset \neg G(y)) \wedge (B(x) \supset \neg B(y)) \wedge (Y(x) \supset \neg Y(y))]\}$. Show that if every *finite* subgraph of a graph is four colorable, so is the entire graph.

5.9.3. Prove Theorem 5.9.4.

5.10 Logical Consequence

In general, knowing which first-order formulas are valid is not enough. We also want to know which formulas *follow from* which formulas. Such a notion was introduced in the propositional case in Section 3.9. Extending it to first-order logic is straightforward.

Definition 5.10.1 A sentence X is a *logical consequence* of a set S of sentences, provided X is true in every model in which all the members of S are true. If X is a logical consequence of S, we symbolize this by $S \models_f X$ (the subscript is to suggest *first*-order consequence).

Notice that we defined logical consequence only for *sentences.* The notion can be extended to cover arbitrary formulas, but doing so is not quite straightforward. If formulas other than sentences are allowed, we could take $S \models_f X$ to mean the following: For every model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$, if the members of S are true in $\mathbf{M}$, then X is true in $\mathbf{M}$ (recall, a formula with free variables is true in a model if it is true under every assignment). Or we could take it to have a different meaning: For every model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ and for every assignment $\mathbf{A}$, if the members of S are true in $\mathbf{M}$ under the assignment $\mathbf{A}$, then X is true in $\mathbf{M}$ under the assignment $\mathbf{A}$. If only sentences are allowed, these two notions coincide, since the truth value of a sentence in a model does not depend on which assignment is used. But in general, these are different. We have chosen to avoid the problem by considering only sentences. Exercises 5.10.1 and 5.10.2 show we loose no expressive power by doing so.

Perhaps the most important feature of logical consequence in first-order logic is that, for each particular sentence X, to establish that it is a consequence of a set S, only a finite amount of the information in S will be needed. This is the content of the following theorem. Other significant facts about consequence are contained in the exercises.

Theorem 5.10.2 *$S \models_f X$ if and only if $S_0 \models_f X$ for some finite set $S_0 \subseteq S$.*

Proof If $S_0 \models_f X$ for some finite $S_0 \subseteq S$, then $S \models_f X$ using Exercise 5.10.3, part 3. The converse direction is the more interesting. Suppose $S \models_f X$. Then $S \cup \{\neg X\}$ is not satisfiable. By the Compactness Theorem 5.9.1, there must be a finite subset that is not satisfiable. We can assume that finite subset contains $\neg X$ (since adding a sentence to an unsatisfiable set leaves it unsatisfiable). Hence, we have an unsatisfiable set of the form $S_0 \cup \{\neg X\}$, where S_0 is finite and $S_0 \subseteq S$. Then $S_0 \models_f X$, and the proof is complete. $\square$

Exercises

5.10.1. For a formula Φ with free variables $x_1, \ldots, x_n$, we write $\forall\Phi$ for the sentence $(\forall x_1) \ldots (\forall x_n)\Phi$. For a set S of formulas, we write $\forall S$ for $\{\forall\Phi \mid \Phi \in S\}$. Show the following are equivalent:

1. X is true in every model in which the members of S are true.
2. $\forall S \models_f \forall X$.

5.10.2. Suppose X is a formula and S is a set of formulas of L. Let $p_1, p_2, \ldots$ be a list of parameters (which do not occur in formulas of L). Let $v_1, v_2, \ldots$ be the list of variables of L, and let σ be the substitution such that $v_i\sigma = p_i$. Note that $X\sigma$ is a *sentence* of L^{par}, as are the members of $S\sigma$. Show the following are equivalent:

1. For any model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ and for any assignment $\mathbf{A}$, if the members of S are true in $\mathbf{M}$ under the assignment $\mathbf{A}$, then X is true in $\mathbf{M}$ under the assignment $\mathbf{A}$.
2. $S\sigma \models_f X\sigma$.

5.10.3. Prove the following:

1. X is valid if and only if $S \models_f X$ for every set S.
2. If $A \in S$ and $\neg A \in S$, then $S \models_f X$ for every X.
3. If $S \models_f X$ and $S \subseteq S^*$, then $S^* \models_f X$.
4. If $S \models_f X$, and $T \models_f Y$ for every $Y \in S$, then $T \models_f X$.
5. $S \cup \{X_1, X_2\} \models_f Y$ if and only if $S \cup \{X_1 \wedge X_2\} \models_f Y$.
6. $S \cup \{X\} \models_f Y$ if and only if $S \models_f X \supset Y$.
7. $\{X_1, \ldots, X_n\} \models_f Y$ if and only if $(X_1 \wedge \ldots \wedge X_n) \supset Y$ is valid.

6 First-Order Proof Procedures

6.1 First-Order Semantic Tableaux

In this chapter we show how to extend Hilbert system, natural deduction, Gentzen sequent, tableau, and resolution proof procedures from propositional to first-order logic. We begin with tableaux and resolution, in versions that are best suited to hand work. In the next chapter, we revise these to a form that is better for computer implementation. As we observed in Section 5.3, we can confine ourselves to proving sentences, since validity of arbitrary formulas can be reduced to that of sentences. We begin with a tableau system in this section, treating resolution in the next.

In Section 5.7 we discussed how, in informal proofs, new constant symbols are routinely introduced. The formal counterpart is *parameters*, constant symbols not part of the original language. We have already seen them in the definition of first-order consistency property. They play a role in our proof procedures exactly as one might expect. Proofs will be *of* sentences of L but will *use* sentences of $L^{\mathbf{par}}$, the extension of L by the addition of a countable list of new parameters.

Tableau proofs are closed trees, constructed exactly as in propositional logic (Section 3.1) but with two additional Tableau Expansion Rules, given in Table 6.1. In the γ-rule any closed term can be used, including one involving parameters. But in the δ-rule, we are required to use a parameter that has not been previously introduced in the tree construction, called a *new* parameter. Informally, it must be a parameter, so that it does not have an intended meaning in the original language L, and it must be new to the tree so that it has not acquired a meaning earlier in the course of the proof. (As a matter of fact, the parameter need only be new to the branch, not to the whole tableau. Our soundness proof for tableaux will apply to this stronger version.)

$\dfrac{\gamma}{\gamma(t)}$	$\dfrac{\delta}{\delta(p)}$
(for any closed term t of L^{par})	(for a new parameter p)

TABLE 6.1. First-Order Tableau Expansion Rules

Example $(\forall x)[P(x) \vee Q(x)] \supset [(\exists x)P(x) \vee (\forall x)Q(x)]$ is given a tableau proof in Figure 6.1. In this the propositional steps are as usual. Line 6 is from 5 by the δ-rule; the parameter c is new to the tableau at this point. Then 7 is from 4, and 8 is from 2 by the γ-rule, which allows any closed term of L^{par}, in particular c.

1. $\neg\{(\forall x)[P(x) \vee Q(x)] \supset [(\exists x)P(x) \vee (\forall x)Q(x)]\}$
2. $(\forall x)[P(x) \vee Q(x)]$
3. $\neg[(\exists x)P(x) \vee (\forall x)Q(x)]$
4. $\neg(\exists x)P(x)$
5. $\neg(\forall x)Q(x)$
6. $\neg Q(c)$
7. $\neg P(c)$
8. $P(c) \vee Q(c)$

$$\begin{array}{ccc} & 8. & \\ \swarrow & & \searrow \\ 9.\ P(c) & & 10.\ Q(c) \end{array}$$

FIGURE 6.1. Proof of $(\forall x)[P(x) \vee Q(x)] \supset [(\exists x)P(x) \vee (\forall x)Q(x)]$

If a tableau branch closes because it contains Z and $\neg Z$, we may say it closes *on* Z. As in the propositional case, nothing in the tableau rules requires closure on a formula that is atomic but in fact, if closure is possible, closure at the atomic level can always be managed.

As usual, the tableau rules are non-deterministic. They say what *may* be done, not what *must* be done. But unlike in the propositional case, it is possible now to work forever, continually doing something really new, without producing a closed tableau for a set S, even though a closed tableau for S may exist. The γ-rule is the source of this difficulty. As a trivial example, suppose we have a tableau branch containing both

$(\exists x)\neg P(x)$ and $(\forall y)P(y)$. We might apply the δ-rule to the first formula, adding $\neg P(c)$, where c is a new parameter. But then using the γ-rule on the second, we might add one after the other $P(t_1)$, $P(t_2)$,... where t_1, t_2,... are all distinct closed terms different from c. In this way we never stop working, we never repeat an earlier step, and we never produce the obvious closure. Of course, in this example what one should do is clear, but in more complex cases things are not so easy. Indeed, unlike in propositional logic, first-order logic has no decision procedure. There are ways of partially coping with this, but the basic lesson remains: "life is not impossible, but it is exponentially difficult, and sometimes worse."

We can introduce a notion of *strictness* as we did with propositional tableaux, so that formula reuse is forbidden. Things are more complicated now, however. If we have a sentence γ on a branch, and we add $\gamma(t_1)$, and later use γ again to add $\gamma(t_2)$ where t_1 and t_2 are different, should this be counted as a strictness violation or not? It turns out such things must be allowed in order to have a complete proof procedure. Allowing reuse of γ-sentences is related to the lack of a decision procedure for first-order logic. At any rate, strictness issues become really significant when implementation is being considered. Since that does not happen until the next chapter, we impose no such restrictions for now.

Finally, just as in the propositional case, good heuristics are useful. The following principles are easily seen to be helpful: When possible, apply propositional rules before quantifier rules; among quantifier cases, apply δ-rules before γ-rules. Beyond this, you are on your own.

In Section 3.9 we showed how the propositional tableau system could be extended to handle propositional consequence. The same modification can be applied to the first-order version, to capture the notion of logical consequence (Section 5.10). From now on we write $S \vdash_{ft} X$ (where the subscript is meant to suggest first-order tableaux), provided there is a closed first-order tableau for $\{\neg X\}$, allowing the *S-introduction rule*: At any time we can add any member of S to any unclosed branch.

Exercises

6.1.1. Give, or attempt to give, tableau proofs of the following (they are not all theorems):

1. $(\exists x)(\forall y)R(x,y) \supset (\forall y)(\exists x)R(x,y)$
2. $(\forall x)(\exists y)R(x,y) \supset (\exists y)(\forall x)R(x,y)$
3. $(\exists x)[P(x) \supset (\forall x)P(x)]$
4. $(\exists x)[P(x) \vee Q(x)] \supset [(\exists x)P(x) \vee (\exists x)Q(x)]$
5. $(\exists x)[P(x) \wedge Q(x)] \supset [(\exists x)P(x) \wedge (\exists x)Q(x)]$

6. $[(\exists x)P(x) \wedge (\forall x)Q(x)] \supset (\exists x)[P(x) \wedge Q(x)]$
7. $(\forall x)(\exists y)(\forall z)(\exists w)[R(x,y) \vee \neg R(w,z)]$
8. $(\exists x)(\forall y)[P(y) \uparrow (P(x) \uparrow Q(x))] \subset (\forall x)Q(x)$
9. $(\forall x)[P(x) \supset Q] \supset [(\exists x)P(x) \supset Q]$ (where x does not occur free in Q)
10. $(\forall x)[P(x) \supset Q] \supset [(\forall x)P(x) \supset Q]$ (where x does not occur free in Q)

6.1.2. For convenience, we give the following names to sentences:

$$\textit{trans} = (\forall x)(\forall y)(\forall z)\{[R(x,y) \wedge R(y,z)] \supset R(x,z)\}$$
$$\textit{sym} = (\forall x)(\forall y)[R(x,y) \supset R(y,x)]$$
$$\textit{ref} = (\forall x)R(x,x)$$
$$\textit{nontriv} = (\forall x)(\exists y)R(x,y)$$

1. Show $\{\textit{trans}, \textit{sym}\} \not\models_f \textit{ref}$ by producing a model in which *trans* and *sym* are true but *ref* is not.
2. Show $\{\textit{trans}, \textit{sym}, \textit{nontriv}\} \vdash_{ft} \textit{ref}$.

6.1.3. Prove the following:

1. $(\exists x)(\forall y)(\forall z)[(P(y) \supset Q(z)) \supset (P(x) \supset Q(x))]$
2. $(\exists x)(\forall y)(\forall z)[(P(y) \vee Q(z)) \supset (P(x) \vee Q(x))]$
3. $(\exists x)(\forall y)(\forall z)(\forall w)[(P(y) \vee Q(z) \vee R(w)) \supset (P(x) \vee Q(x) \vee R(x))]$

6.1.4. Prove, by structural induction, that if there exists a closed first-order tableau for a set S then there is a closed tableau for S in which all closures are on atomic sentences.

$\dfrac{\gamma}{\gamma(t)}$	$\dfrac{\delta}{\delta(c)}$
(for any closed term t of L^{par})	(for a new parameter c)

TABLE 6.2. First-Order Resolution Expansion Rules

6.2 First-Order Resolution

Just as we did with tableaux, we present a version of resolution that is best suited to hand calculation, and we postpone until the next chapter a variation that is better adapted to machines. Once again we are interested only in proving sentences, and once again we do not impose a strictness condition. The rules of this section should be thought of as continuing those in Section 3.3. Just as in the previous section, we prove sentences *of* a language L, but proofs *use* sentences of L^{par}. In Table 6.2, we add two new Resolution Expansion Rules, which have the same appearance as the Tableau Rules in the previous section. As expected, in the γ-rule any closed term can be used, including one involving parameters, but in the δ-rule we must use a parameter that has not been previously introduced in the resolution construction.

Example We give a resolution proof of $(\forall x)(P(x) \vee Q(x)) \supset ((\exists x)P(x) \vee (\forall x)Q(x))$.

1. $[\neg\{(\forall x)(P(x) \vee Q(x)) \supset ((\exists x)P(x) \vee (\forall x)Q(x))\}]$
2. $[(\forall x)(P(x) \vee Q(x))]$
3. $[\neg((\exists x)P(x) \vee (\forall x)Q(x))]$
4. $[\neg(\exists x)P(x)]$
5. $[\neg(\forall x)Q(x)]$
6. $[\neg Q(c)]$
7. $[\neg P(c)]$
8. $[P(c) \vee Q(c)]$
9. $[P(c), Q(c)]$
10. $[Q(c)]$
11. $[\,]$

Here 6 is from 5 by a δ-rule application, 7 is from 4, and 8 is from 2 by γ-rule applications.

The resolution system, like tableaux, can be extended to capture the notion of first-order logical consequence. The *S-Introduction Rule* for resolution from Section 3.9 applies just as it did in the propositional setting. We will write $S \vdash_{fr} X$ (where the subscript is meant to suggest first-order resolution), provided there is a closed resolution expansion for $\{\neg X\}$ allowing the following rule: At any time we can add $[Z]$ to a resolution expansion for any member Z of S.

Exercises

6.2.1. Give (or attempt to give) resolution proofs of the sentences in Exercise 6.1.1.

6.2.2. Redo Exercise 6.1.2 using resolution.

6.2.3. Redo Exercise 6.1.3 using resolution.

6.2.4. Prove that an application of the Resolution Rule involving a formula that is not atomic can be replaced by resolutions involving simpler formulas. Hence, by structural induction, only atomic resolutions need be used.

6.3 Soundness

In Section 5.3 satisfiability was defined for sets of first-order formulas. This definition is extended to the structures used in proofs, essentially as in Section 3.4.

Definition 6.3.1 A tableau *branch* is *satisfiable* if the set of first-order sentences on it is satisfiable. A *tableau* is *satisfiable* if some branch is satisfiable. Likewise, a *resolution expansion* is *satisfiable* if there is some model in which every disjunction in it is true.

Thinking of a tableau as a disjunction of conjunctions, this definition amounts to saying a tableau is satisfiable if there is some model in which it is true, and similarly for a resolution expansion, thinking of it as a conjunction of disjunctions.

Lemma 6.3.2

1. *If any Tableau Expansion Rule is applied to a satisfiable tableau, the result is another satisfiable tableau.*
2. *If any Resolution Expansion Rule, or the Resolution Rule is applied to a satisfiable resolution expansion the result is again satisfiable.*

Proof We show only part 1. Suppose $\mathbf{T}$ is a satisfiable tableau and some Tableau Expansion Rule is applied to $\mathbf{T}$, producing the tableau $\mathbf{T}^*$. We show $\mathbf{T}^*$ is satisfiable. The cases where $\mathbf{T}$ has a satisfiable branch other than the one on which the rule is applied, or the rule is a propositional one, are treated just as in the propositional case, and the arguments are not repeated here. (You probably should go back and look at the proof of Proposition 3.4.2 again.) This leaves the two quantifier cases, and Proposition 5.5.1 takes care of these. □

Theorem 6.3.3 **(Soundness)**

1. *If X has a tableau proof, then X is valid.*
2. *If X has a resolution proof, then X is valid.*

Proof (of part 1). Suppose X has a tableau proof but is not valid; we derive a contradiction. Since X is not valid, there is a model in which $\neg X$ is true. The construction of a tableau proof for X begins with the one-branch, one-node tableau labeled $\neg X$, which is a satisfiable tableau. By Lemma 6.3.2, every subsequent tableau is satisfiable, including the final closed tableau that constitutes the proof. But a closed tableau can not be satisfiable. □

Exercises

6.3.1. Prove Part 2 of Lemma 6.3.2 and then Part 2 of Theorem 6.3.3.

6.3.2. Prove the strong soundness of the tableau system. That is, if S is a set of sentences and X is a sentence of the language L, show that $S \vdash_{ft} X$ implies $S \models_f X$.

6.3.3. Prove the strong soundness of the resolution system. That is, if S is a set of sentences and X is a sentence of the language L, show that $S \vdash_{fr} X$ implies $S \models_f X$.

6.4 Completeness

Just as in the propositional case, completeness results are easy to establish with the Model Existence Theorem available. We begin with tableaux, then move on to resolution. For this section, let L be a fixed first-order language.

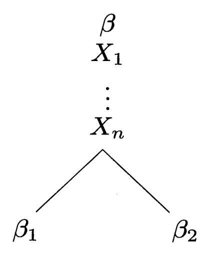
Definition 6.4.1 A finite set S of sentences of L^{par} is *tableau consistent* if there is no closed tableau for S.

Lemma 6.4.2 *The collection of all tableau-consistent sets is a first-order consistency property.*

Proof Most of the proof is a straightforward extension of the corresponding propositional version, Lemma 3.7.2. The β-case is slightly tricky though, and we discuss it in some detail.

Suppose S is a finite set of sentences of L^{par}, $\beta \in S$, but neither $S \cup \{\beta_1\}$ nor $S \cup \{\beta_2\}$ is tableau consistent; we show S is not tableau consistent. Since $S \cup \{\beta_1\}$ is not tableau consistent, there is a closed tableau, call it $\mathbf{T}_1$, for $S \cup \{\beta_1\}$. Similarly, there is a closed tableau, call it $\mathbf{T}_2$, for $S \cup \{\beta_2\}$. The trouble is, these tableaux may be incompatible with each other because we may have introduced a parameter via a δ-rule application in one tableau that had already been used in the other tableau. To get around this, we need the following simple observation. Suppose $\mathbf{T}$ is a correctly constructed tableau, c is a parameter used in the construction of $\mathbf{T}$, and d is a parameter that never appears in $\mathbf{T}$. Let $\mathbf{T}^*$ be like $\mathbf{T}$ but with every occurrence of c replaced by an occurrence of d. It is straightforward to check that $\mathbf{T}^*$ is still a correctly constructed tableau and is closed provided $\mathbf{T}$ is closed.

So, by renaming parameters in this way, we can assume the closed tableau $\mathbf{T}_1$ for $S \cup \{\beta_1\}$ and the closed tableau $\mathbf{T}_2$ for $S \cup \{\beta_2\}$ meet the condition that no δ-rule application in either introduces a parameter that has appeared in the other. Now $\mathbf{T}_1$ and $\mathbf{T}_2$ can be combined to produce a closed tableau for S as follows. Say $S = \{\beta, X_1, \ldots, X_n\}$. We begin a tableau for S by using the β-rule to produce the following:



Then we simply append to the left branch the tableau $\mathbf{T}_1$ and to the right branch the tableau $\mathbf{T}_2$, producing a closed tableau for S. □

Theorem 6.4.3 **(Completeness for First-Order Tableaux)**
If the sentence X of L is valid, X has a tableau proof.

Proof If X does not have a tableau proof, there is no closed tableau for $\{\neg X\}$. Then $\{\neg X\}$ is tableau consistent, hence satisfiable by the First-Order Model Existence Theorem, and so X is not valid. □

In the propositional case, we noted that the completeness proof continued to work if we required *atomic* closure. This still applies. We leave it to you to verify it.

Next we turn to resolution. We omit all details, since they are virtually the same as in the propositional case or are similar to the first-order treatment of tableaux. We carry over the notion of resolution derivation from Chapter 3. The following generalizes Lemma 3.7.6.

Lemma 6.4.4 *Suppose S_1 and S_2 are sets of disjunctions, and S_2 is an X-enlargement of S_1. If the disjunction D_1 is resolution derivable from S_1, then there is an X-enlargement D_2 of D_1 that is resolution derivable from S_2, provided X contains no parameters occurring in the derivation of D_1.*

Definition 6.4.5 A finite set S of sentences of L^{par} is *resolution consistent* if there is no closed resolution expansion for S.

Lemma 6.4.6 *The collection of all resolution-consistent sets is a first-order consistency property.*

Proof Just as in the propositional setting, Section 3.7, except that in the β-case some parameter renaming must be done, as in the tableau argument above. □

Theorem 6.4.7 **(Completeness for First-Order Resolution)**
If the sentence X of L is valid, X has a resolution proof.

Completeness can also be established under the stronger requirement that all applications of the Resolution Rule be on atomic formulas.

Exercises

6.4.1. Prove Lemma 6.4.6, and hence Theorem 6.4.7.

6.4.2. Prove the strong completeness of the tableau system. That is, if S is a set of sentences and X is a sentence of the language L, show that $S \models_f X$ implies $S \vdash_{ft} X$.

6.4.3. Prove the strong completeness of the resolution system. That is, if S is a set of sentences and X is a sentence of the language L, show that $S \models_f X$ implies $S \vdash_{fr} X$.

6.5 Hilbert Systems

In Chapter 4 we considered several proof procedures for propositional logic besides tableau and resolution. Generally, these extend easily to first-order logic. We show this for Hilbert systems now, and for natural deduction and the Gentzen sequent calculus next.

As in the propositional case, there is considerable leeway in formulating a Hilbert system, in the choice of axioms and rules. In addition we have chosen to present a system in which only sentences are used, though these may contain parameters. In other treatments free variables are often used in a similar way instead.

We add one axiom scheme and one rule of inference to our propositional Hilbert system. Uniform notation comes in, of course. We continue the numbering from Section 4.1.

Axiom Scheme 10 $\gamma \supset \gamma(t)$ for any closed term t of $L^{\mathbf{par}}$.

The rule we add is the following, generally known as Universal Generalization:

Universal Generalization

$$\frac{\Phi \supset \gamma(p)}{\Phi \supset \gamma}$$

provided p is a parameter that does not occur in the sentence $\Phi \supset \gamma$. In a derivation from a set S of sentences, p must not occur in S as well.

We write $S \vdash_{fh} X$ if there is a derivation of X from the set S in the first-order Hilbert system just presented. As usual, if $\emptyset \vdash_{fh} X$, we will call the derivation of X a *proof* and say X is a *theorem.*

There is a weaker version of the Universal Generalization Rule that reads as follows:

$$\frac{\gamma(p)}{\gamma}$$

where p is a parameter that does not occur in γ, or in the set of premises of a derivation. The use of this version is easily justified. In a proof or derivation, if we have a line $\gamma(p)$, we can easily get $\top \supset \gamma(p)$. Then $\top \supset \gamma$ follows by the Universal Generalization Rule as we gave it, from which we get γ using Modus Ponens. From now on we will use this simpler version of the Universal Generalization Rule, as convenient, without special comment.

Example The following is a proof (sketch) of the L sentence $(\forall x)(P(x) \land Q(x)) \supset (\forall x)P(x)$. In it p is a parameter.

1. $(\forall x)(P(x) \land Q(x)) \supset (P(p) \land Q(p))$
2. $(P(p) \land Q(p)) \supset P(p)$
3. $(\forall x)(P(x) \land Q(x)) \supset P(p)$
4. $(\forall x)(P(x) \land Q(x)) \supset (\forall x)P(x)$

Here 1 is an instance of Axiom Scheme 10; 2 is a tautology, hence provable in the propositional Hilbert system of Section 4.1; 3 follows from 1 and 2 by propositional logic. Finally 4 follows from 3 by the Universal Generalization Rule.

The Deduction Theorem 4.1.4 is an important tool for propositional Hilbert systems. Fortunately, it extends to the first-order setting.

Theorem 6.5.1 **(Deduction Theorem)** *In any first-order Hilbert system h with at least Axiom Schemes 1 and 2, and with Modus Ponens and Universal Generalization as the only rules of inference, $S \cup \{X\} \vdash_{ph} Y$ if and only if $S \vdash_{ph} (X \supset Y)$.*

Proof The proof is simply an extension of the proof of Theorem 4.1.4, which you should go back and read. We continue the notation and terminology from that proof, adding one more case corresponding to the Universal Generalization Rule.

Suppose Z_i comes from an earlier term Z_j of Derivation One by the Universal Generalization Rule. Say Z_j is $\Phi \supset \gamma(p)$ and Z_i is $\Phi \supset \gamma$. The parameter p can not occur in $\Phi \supset \gamma$, and since Derivation One is a derivation from the set $S \cup \{X\}$, p can not occur in S or in X either. Now, in the Derivation Two candidate, the corresponding lines are $(X \supset Z_j) = (X \supset (\Phi \supset \gamma(p)))$ and $(X \supset Z_i) = (X \supset (\Phi \supset \gamma))$. We must insert extra lines justifying the presence of $X \supset Z_i$, using $X \supset Z_j$. But this is easy. We have $X \supset (\Phi \supset \gamma(p))$. From it, by propositional logic, we can get $(X \land \Phi) \supset \gamma(p)$. By the Universal Generalization Rule applied to this (note that p does not occur in either Φ or X), we can get $(X \land \Phi) \supset \gamma$, and then by propositional manipulations again, we have $X \supset (\Phi \supset \gamma)$. □

Example Let S be the set $\{(\forall x)(P(x) \supset Q(x)), (\forall x)P(x)\}$. The following derivation shows that $S \vdash_{fh} (\forall x)Q(x)$:

1. $(\forall x)P(x)$

2. $P(p)$
3. $(\forall x)(P(x) \supset Q(x))$
4. $P(p) \supset Q(p)$
5. $Q(p)$
6. $(\forall x)Q(x)$

Here 1 and 3 are members of S; 2 and 4 follow from 1 and 3 by Axiom Scheme 10; 5 follows from 2 and 4 by Modus Ponens; 6 follows from 5 by Universal Generalization.

Now, two applications of the Deduction Theorem show that $(\forall x)(P(x) \supset Q(x)) \supset ((\forall x)P(x) \supset (\forall x)Q(x))$ is a theorem.

Theorem 6.5.2 **(Strong Hilbert Soundness)**
If $S \vdash_{fh} X$, then $S \models_f X$, where S is a set of sentences of L and X is a sentence of L.

Theorem 6.5.3 **(Strong Hilbert Completeness)**
If $S \models_f X$, then $S \vdash_{fh} X$, where S is a set of sentences of L and X is a sentence of L.

Exercises

6.5.1. Give (or attempt to give) Hilbert-style proofs of the sentences in Exercise 6.1.1.

6.5.2. Show that all instances of the scheme $\delta(t) \supset \delta$ have Hilbert system proofs, for any closed term t.

6.5.3. Show the following is a derived rule in the Hilbert system of this section:

$$\frac{\delta(p) \supset \Phi}{\delta \supset \Phi}$$

provided p is a parameter that does not occur in the sentence $\delta \supset \Phi$, and in a derivation from a set S of sentences, p does not occur in S as well.

6.5.4. Prove Theorem 6.5.2.

6.5.5. Prove Theorem 6.5.3 (use the Model Existence Theorem).

6.6 Natural Deduction and Gentzen Sequents

A propositional natural deduction system was given in Section 4.2. The addition of a few quantifier rules turns it into a first-order version. The propositional rules came in pairs, rules for introducing and rules for eliminating connectives. The quantifier rules can be given in the same paired format, and for theoretical investigations it is important to do so [36]. As it happens, though, the propositional system we gave becomes a complete first-order system through the addition of elimination rules only, so in the interests of simplicity, these are all we state.

Quantifier Rules

$$\gamma\mathbf{E} \quad \frac{\gamma}{\gamma(t)}$$

$$\delta\mathbf{E} \quad \frac{\delta}{\delta(p)}$$

In the δE rule, p must be a parameter that does not occur previously in the proof. In the γE rule, t can be any closed term of $L^{\mathbf{par}}$. Incidentally, the similarity of these rules and the tableau and resolution rules is no coincidence.

Figure 6.2 displays a natural deduction proof of $(\forall x)[P(x) \supset Q(x)] \supset [(\forall x)P(x) \supset (\forall x)Q(x)]$. In it, 1 through 3 are assumptions; 4 is from 3 by δE (p is a new parameter at this point); 5 is from 2 and 6 is from 1 by γE. Then 7 is from 5 and 6 by βE; 8 is from 4 and 7 by a negation rule; 9 is likewise by a negation rule. Finally 10 and 11 are by βI.

In Section 4.3 we gave a sequent calculus formulation of propositional logic. This too extends readily to a first-order version on the addition of quantifier rules.

γ-Rules

$$\frac{\Gamma, \gamma(t) \rightarrow \Delta}{\Gamma, \gamma \rightarrow \Delta} \qquad \frac{\Gamma \rightarrow \Delta, \gamma(p)}{\Gamma \rightarrow \Delta, \gamma}$$

δ-Rules

$$\frac{\Gamma \rightarrow \Delta, \delta(t)}{\Gamma \rightarrow \Delta, \delta} \qquad \frac{\Gamma, \delta(p) \rightarrow \Delta}{\Gamma, \delta \rightarrow \Delta}$$

As should be expected, in these rules t can be any closed term, but p must be a parameter that does not occur in any sentence below the line.

Example

$$\boxed{\begin{array}{ll} 1. & (\forall x)[P(x) \supset Q(x)] \\ & \boxed{\begin{array}{ll} 2. & (\forall x)P(x) \\ & \boxed{\begin{array}{ll} 3. & \neg(\forall x)Q(x) \\ 4. & \neg Q(p) \\ 5. & P(p) \\ 6. & P(p) \supset Q(p) \\ 7. & Q(p) \\ 8. & \bot \end{array}} \\ 9. & (\forall x)Q(x) \end{array}} \\ 10. & (\forall x)P(x) \supset (\forall x)Q(x) \end{array}}$$

$$11. \quad (\forall x)[P(x) \supset Q(x)] \supset [(\forall x)P(x) \supset (\forall x)Q(x)]$$

FIGURE 6.2. A Natural Deduction Proof of $(\forall x)[P(x) \supset Q(x)] \supset [(\forall x)P(x) \supset (\forall x)Q(x)]$

Exercises

6.6.1. Give (or attempt to give) natural deduction-style proofs of the sentences in Exercise 6.1.1.

6.6.2. Prove the soundness of the first-order natural deduction system.

6.6.3. Prove the completeness of the first-order natural deduction system.

6.6.4. Give (or attempt to give) sequent calculus-style proofs of the sentences in Exercise 6.1.1.

6.6.5. Prove the soundness of the first-order sequent calculus system.

6.6.6. Prove the completeness of the first-order sequent calculus system.

7

Implementing Tableaux and Resolution

7.1 What Next

In using either the resolution or the tableau system, the basic problem lies with the γ-rule. It allows us to introduce *any* closed term, but obviously some choices will be better than others. How do we decide? Our solution will be to postpone the choice, by going from γ to $\gamma(x)$, where x is a free variable whose value we will figure out later. But this introduces problems with the δ-rule, since we won't know which parameters are new if we haven't yet said what terms we have used in γ-rule applications. Our solution for this will be to introduce a more general notion of parameter, allowing *function* symbols. Then the δ-rule will become: from δ conclude $\delta(f(x_1, \ldots, x_n))$, where f is a new function parameter and $x_1, \ldots, x_n$ are the free variables introduced so far. Intuitively, this forces the term to be new because we can figure out what it is only after we have made choices for $x_1, \ldots, x_n$. The details will come later, but the basic issues confront us now. Both tableau and resolution proofs must be allowed to contain free variables, whose values must be chosen somehow.

In earlier chapters free variables did not occur in proofs, and the basic problem for tableaux, say, was whether branches were closed. Once free variables are allowed, the issue changes to the following: Can values for free variables be found that will result in closed branches? For instance, if a branch contains $P(t_1)$ and $\neg P(t_2)$, where t_1 and t_2 are terms containing free variables, we want to know whether some assignment of values to these free variables will result in t_1 and t_2 becoming identical and so yielding a contradiction. In other words, we want to solve the equation $t_1 = t_2$ in the space of terms. More generally, since tableaux almost always have several branches that must be closed, we will have the problem of solving a system of simultaneous equations. Similar considerations apply to resolution-style proofs. Algorithms exist for this problem, under the general heading of *Unification Algorithms.* It is such algorithms that we take up first.

7.2 Unification

Suppose we have two terms t and u, each containing variables. How do we decide whether there are any substitutions that make t and u identical, and if there are, how do we find them all? A substitution σ such that $t\sigma = u\sigma$ is called a *unifier* of t and u, so the problem is to determine the set of unifiers for t and u. Another way of expressing this sounds more algebraic: Determine the set of solutions for the equation $t = u$. In fact, we will often be interested in solving a *system* of equations. The relationship that this suggests with linear algebra is a real one and is discussed by Lassez and co-workers in [29].

Many different unification algorithms for solving the unification problem have been proposed. The earliest, by Herbrand [25], manipulated systems of equations. On the other hand, Robinson [42] worked with terms and substitutions directly. It was Robinson's paper [42] that introduced the term *unification*, and recognized the concept as fundamental for automated theorem proving. More recently, unification algorithms have been proposed that are more efficient, but they are harder to understand. We present an algorithm that follows Robinson's treatment fairly closely, in the interests of pedagogical simplicity. The report by Lassez and colleagues [29] provides references to other unification algorithms and a fuller study of theoretical issues than we can present here.

For the rest of this section, let L be a fixed first-order language. All reference to terms is to terms of L.

Definition 7.2.1 Let σ_1 and σ_2 be substitutions. We say σ_2 is *more general* than σ_1 if, for some substitution τ, $\sigma_1 = \sigma_2\tau$.

Example For the two substitutions $\sigma_1 = \{x/f(g(a, h(z))),\ y/g(h(x), b),\ z/h(x)\}$ and $\sigma_2 = \{x/f(g(x, y)),\ y/g(z, b)\}$, σ_2 is more general than σ_1, because $\sigma_1 = \sigma_2\tau$ where $\tau = \{x/a,\ y/h(z),\ z/h(x)\}$.

The idea is that σ_2 is more general than σ_1 if we can get the effect of σ_1 by first carrying out σ_2 and then making some further substitutions for variables. Every substitution is more general than itself, because $\sigma = \sigma\epsilon$, where ϵ is the identity substitution. Thus, we are using the term *more general* in a weak rather than a strict sense. The following shows the notion is transitive as well as reflexive.

Proposition 7.2.2 *If σ_3 is more general than σ_2 and σ_2 is more general than σ_1, then σ_3 is more general than σ_1.*

Proof Since σ_2 is more general than σ_1, there is a substitution τ such that $\sigma_1 = \sigma_2\tau$. Since σ_3 is more general than σ_2, there is a substitution η such that $\sigma_2 = \sigma_3\eta$. But then $\sigma_1 = \sigma_2\tau = (\sigma_3\eta)\tau = \sigma_3(\eta\tau)$, and so σ_3 is more general than σ_1. $\square$

Definition 7.2.3 Let t_1 and t_2 be two terms (this definition extends in the obvious way to more than two terms). A substitution σ is a *unifier*, for t_1 and t_2 provided $t_1\sigma = t_2\sigma$. t_1 and t_2 are *unifiable* if they have a unifier. A substitution σ is a *most general unifier* if it is a unifier and is more general than any other unifier.

Example The terms $f(y, h(a))$ and $f(h(x), h(z))$ are unifiable using the substitution $\{y/h(x), z/a\}$. Also, the substitution $\{x/k(w), y/h(k(w)), z/a\}$ is a unifier, but the first substitution is more general. The terms $f(x, x)$ and $f(a, b)$ are not unifiable (here x is a variable, and a and b are constant symbols).

It is possible for two terms to have several most general unifiers, but there are close relationships between them. This will play no role in what follows, so we consider the point only briefly and do not prove the strongest possible results.

Definition 7.2.4 A substitution η is a *variable renaming* for a set V of variables if

1. For each $x \in V$, $x\eta$ is a variable;
2. For $x, y \in V$ with $x \neq y$, $x\eta$ and $y\eta$ are distinct.

Definition 7.2.5 The *variable range* for a substitution σ is the set of variables that occur in terms of the form $x\sigma$, where x is a variable.

Proposition 7.2.6 *Suppose both σ_1 and σ_2 are most general unifiers of t_1 and t_2. Then there is a variable renaming η for the variable range of σ_1 such that $\sigma_1\eta = \sigma_2$.*

Proof σ_2 Is a unifier of t_1 and t_2, but σ_1 is most general, hence more general than σ_2. Then there is a substitution η such that $\sigma_2 = \sigma_1\eta$. Switching around the roles of σ_1 and σ_2, there is a substitution τ so that $\sigma_1 = \sigma_2\tau$. Then $\sigma_1 = \sigma_2\tau = (\sigma_1\eta)\tau$. We show η is a variable renaming for the variable range of σ_1.

Suppose y is in the variable range of σ_1; say it occurs in the term $x\sigma_1$. Then $y\eta$ can not be of the form $f(\cdots)$ where f is a function symbol, for if it were, $(x\sigma_1)\eta$ would be longer than $x\sigma_1$, and since no substitution can make a term shorter, $x\sigma_1\eta\tau$ would be at least as long as $x\sigma_1\eta$, longer than $x\sigma_1$, and hence unequal to $x\sigma_1$, which is impossible, since $\sigma_1 = (\sigma_1\eta)\tau$. Likewise, $y\eta$ can not be a constant symbol, for then there would be no way for τ to restore occurrences of y when applied to $x\sigma_1\eta$, and so $x\sigma_1\eta\tau$ and $x\sigma_1$ would again be different. It follows that $y\eta$ must be a variable, though it need not be y itself. Finally, it is easy to see that if y_1 and y_2 are distinct variables in the variable range of σ_1, $y_1\eta$ and $y_2\eta$ must be distinct variables. □

Now what we are heading for is an algorithm that can say of two terms whether or not they are unifiable and, if they are, will produce a most general unifier. For this purpose it is convenient to introduce some special terminology first.

We have been writing terms in conventional, linear fashion. It is also common to think of them as trees. For example, $f(g(x, y, h(a, k(b))))$ can be represented as the (labeled, ordered) tree displayed on the left-hand side in Figure 7.1, where the arguments of a function are displayed as the children of the node labeled with the function. In general, we will freely interchange tree terminology with that which we have been using. These are ordered trees, and sometimes it is useful to make explicit whether a node in a tree is the first child of its parent (first from the left, say), or the second, or whatever. An *augmented tree representation* of a term is a tree in which the label on each node has been augmented by an integer representing which child it is of its parent. For instance, the augmented version of the tree for the term $f(g(x, y, h(a, k(b))))$ is on the right in Figure 7.1.

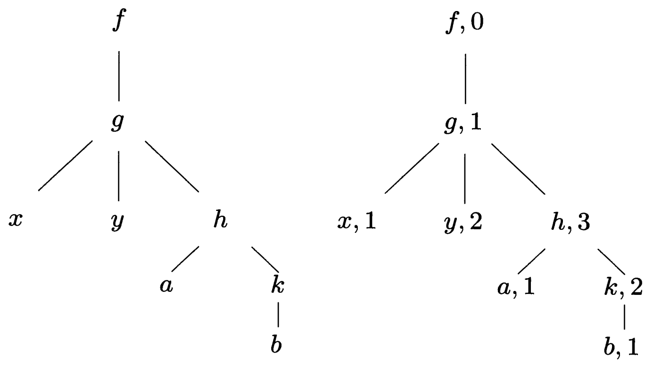


FIGURE 7.1. Terms as Trees

Now, given two terms that differ, we want to locate a place where they disagree, but we want the broadest disagreement possible. For instance, $f(g(a))$ and $f(h(b))$ differ on a and b, but they differed on g and h before we got inside to the level of a and b.

Definition 7.2.7 Let t_1 and t_2 be two terms. A *disagreement pair* for these terms is a pair, d_1, d_2, where d_1 is a subterm of t_1 and d_2 is a subterm of t_2 such that, thinking of terms as augmented trees, d_1 and d_2 have distinct labels at their roots, but the path from the root of t_1 down to the root of d_1, and the path from the root of t_2 down to the root of d_2 are the same.

Example Figure 7.2 shows a disagreement pair. Note that in both trees the paths from the root node down to the subterms constituting the disagreement pair are the same, $\langle f, 0\rangle$, $\langle h, 2\rangle$. In this example there is only one disagreement pair, though in other cases there could be more.

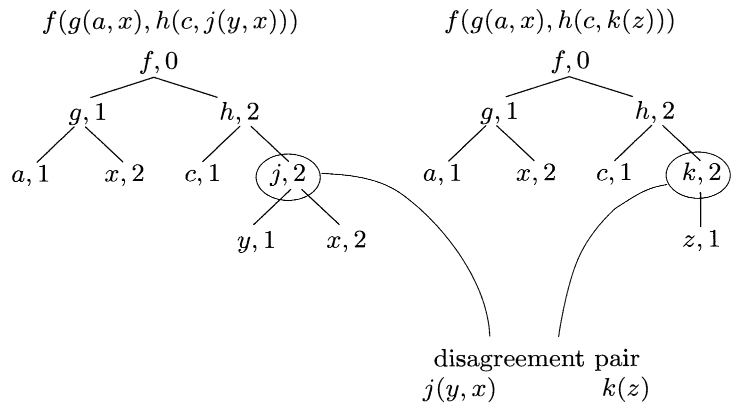


FIGURE 7.2. A Disagreement Pair

If two terms differ, there must be one or more disagreement pairs. Also if σ unifies distinct terms t_1 and t_2, it must unify each disagreement pair of these terms.

Now we state the Unification Algorithm that is due to Robinson. We give it in non-deterministic form, using a kind of pseudocode. The instruction FAIL means: terminate the algorithm and issue some kind of failure message. At the start t_1 and t_2 are terms that have been specified from the outside and that we wish to unify. σ Is a variable that takes substitutions as values; ϵ is the identity substitution.

Unification Algorithm

```
Let $\sigma := \epsilon$;
While $t_1\sigma \neq t_2\sigma$ do
    begin
    choose a disagreement pair, $d_1$, $d_2$ for $t_1\sigma$, $t_2\sigma$;
    if neither $d_1$ nor $d_2$ is a variable then FAIL;
    let $x$ be whichever of $d_1$, $d_2$ is a variable
        (if both are, choose one),
        and let $t$ be the other one of $d_1$, $d_2$;
    if $x$ occurs in $t$ then FAIL;
    let $\sigma := \sigma\{x/t\}$
    end.
```

We claim that (1) this algorithm always terminates; (2) if t_1 and t_2 are not unifiable, the algorithm will FAIL; (3) if t_1 and t_2 are unifiable, the algorithm will terminate without FAILure, and (4) the final value of σ will be a most general unifier for t_1 and t_2.

Proof that the algorithm terminates is easy. Let $S(\sigma)$ be the set of variables that occur in either $t_1\sigma$ or in $t_2\sigma$. Each pass through the while loop, if it does not terminate with FAIL, must decrease the size of $S(\sigma)$ by 1 (because x is replaced by a term t that cannot contain occurrences of x). Since t_1 and t_2 have only a finite number of variables, termination is ensured. Correctness requires more of an argument, but the following simple lemma provides the key item needed.

Lemma 7.2.8 *Suppose u and v are two terms that are unifiable, and τ is a unifier, $u\tau = v\tau$. Suppose also that x, t is a disagreement pair for u, v, where x is a variable. Then (1) x does not occur in t, and (2) $\{x/t\}\tau = \tau$.*

Proof Since τ unifies u and v, which has x, t as a disagreement pair, it must be that $t\tau = x\tau$. If x occurred in t, since x, t is a disagreement pair, x must be a proper part of t. Then for any substitution η, $x\eta$ would be a proper part of $t\eta$. But this is false for $\eta = \tau$. Hence, x does not occur in t and we have item 1. For item 2, to show two substitutions are equal, it is enough to show they have the same effect on each variable. If y is a variable other than x, both $y\{x/t\}\tau$ and $y\tau$ are trivially the same; the lemma hypotheses are not needed for this. Finally, $x\{x/t\}\tau = t\tau = x\tau$, so $\{x/t\}\tau$ and τ agree on x as well. □

Theorem 7.2.9 **(Unification Theorem)** *Let t_1 and t_2 be terms. If t_1 and t_2 are not unifiable, the Unification Algorithm will FAIL. If t_1 and t_2 are unifiable, the Unification Algorithm will terminate without FAILure, and the final value of σ will be a most general unifier for t_1 and t_2.*

Proof First, suppose t_1 and t_2 are not unifiable. We already showed that the algorithm must terminate. If it does not terminate in FAILure, but because of the while loop condition, then we must have produced a substitution σ such that $t_1\sigma = t_2\sigma$, hence t_1 and t_2 would have been unifiable. Consequently, the algorithm must terminate with FAIL.

Next, suppose t_1 and t_2 are unifiable. Let τ be any unifier for t_1 and t_2. We must show the algorithm terminates without FAILure (and hence the final value of σ is a unifier for t_1 and t_2), and we must show the final value of σ is more general than τ.

Consider the statement: $\tau = \sigma\tau$. When the while loop is first encountered, this statement is true, because σ is the identity substitution. If we show the statement is a loop invariant, then it will be true when the loop terminates. It will follow immediately that σ is more general than τ because there is a substitution η such that $\tau = \sigma\eta$, namely, $\eta = \tau$. If we also show the loop can not terminate because of FAILure, we are done.

Suppose we are at the beginning of the loop body, and for the current value of σ, $t_1\sigma \neq t_2\sigma$, and also $\tau = \sigma\tau$ is true. We first claim $t_1\sigma$ and $t_2\sigma$, though different terms, are unifiable; in fact, τ is a unifier. The verification is simple: $(t_1\sigma)\tau = t_1(\sigma\tau) = t_1\tau$ (by our hypothesis that $\tau = \sigma\tau$) $= t_2\tau$ (since τ unifies t_1 and t_2) $= t_2(\sigma\tau) = (t_2\sigma)\tau$. Since $t_1\sigma \neq t_2\sigma$, there must exist a disagreement pair for $t_1\sigma$ and $t_2\sigma$, say d_1, d_2. One of d_1, d_2 must be a variable, because otherwise $t_1\sigma$ and $t_2\sigma$ would not be unifiable; consequently, we do not exit the loop because of the first FAIL condition. Let x be one of d_1, d_2 that is a variable, and let t be the other. By part 1 of Lemma 7.2.8, x does not occur in t, so we do not exit the loop because of the second FAIL condition either. Then we must execute the assignment statement at the bottom of the loop. Let us denote the new value of σ by σ' and continue to use σ for the old value; thus, $\sigma' = \sigma\{x/t\}$. Then what we must show is the following: $\tau = \sigma'\tau$. But this is easy. Using part 2 of Lemma 7.2.8, and our assumption that $\tau = \sigma\tau$, we have $\sigma'\tau = \sigma\{x/t\}\tau = \sigma\tau = \tau$. This concludes the proof. □

The proof of the Unification Theorem actually shows something stronger about the Unification Algorithm than was stated. According to the definition of most general unifier, if σ is a most general unifier for t_1 and t_2, and if τ is any unifier, then $\tau = \sigma\eta$ for some η. But according to the proof, for the most general unifier constructed by the Unification Algorithm, we can take η to be τ itself—that is, $\tau = \sigma\tau$! A special case of this is particularly interesting. Since τ can be any unifier, we can take it to be σ, since a most general unifier is certainly a unifier. The result is that $\sigma = \sigma\sigma$.

Definition 7.2.10 A substitution σ is called *idempotent* if $\sigma = \sigma\sigma$.

Then the proof gives us the following, for free.

Corollary 7.2.11 *If t_1 and t_2 are unifiable, the Unification Algorithm terminates with a final value that is an idempotent most general unifier for them.*

It is not the case that most general unifiers must be idempotent, but idempotent ones have rather nice properties, so it is pleasant to have them. Here is one simple example of a nice feature. Earlier, when we first noted that the Unification Theorem proof gave more information than the statement of the theorem said, we began with a general fact about an arbitrary unifier τ, then narrowed it to the special case where $\tau = \sigma$. In fact, the general case follows easily.

Proposition 7.2.12 *Suppose σ is an idempotent most general unifier for t_1 and t_2, and τ is any unifier. Then $\tau = \sigma\tau$.*

Proof Since τ is a unifier and σ is a most general unifier, for some substitution η, $\tau = \sigma\eta$. And since σ is idempotent, $\sigma = \sigma\sigma$. But then, $\tau = \sigma\eta = \sigma\sigma\eta = \sigma\tau$. □

The Unification Algorithm as we stated it unifies exactly two terms. For convenience we will sometimes refer to this as *binary unification.* Often we will be interested in something more general. We introduce some names to make talking about the generalizations we need easier. We use the phrase *multiple unification* for the problem of finding a unifier for a set of more than two terms. We use *concurrent unification* for the problem of simultaneously unifying several pairs of terms. It is not necessary to invent completely new algorithms for these problems, since both can be reduced to binary unification.

Multiple Unification Suppose $\{t_0, t_1, t_2, \ldots, t_n\}$ is a set of terms. A *unifier* for the set is a substitution σ such that $t_0\sigma = t_1\sigma = \cdots = t_n\sigma$. As usual, a *most general unifier* is a unifier that is more general than any other unifier. The problem of multiple unification can be reduced to a sequence of binary unification problems in a rather simple way (note the use of *idempotent* most general unifiers here).

Suppose $\{t_0, t_1, t_2, \ldots, t_n\}$ has a unifier. Define a sequence of substitutions, each computed by binary unification, as follows.

σ_1 Is an idempotent most general unifier of t_0 and t_1.

σ_2 Is an idempotent most general unifier of $t_0\sigma_1$ and $t_2\sigma_1$.

σ_3 Is an idempotent most general unifier of $t_0\sigma_1\sigma_2$ and $t_3\sigma_1\sigma_2$.

etc.

σ_n Is an idempotent most general unifier of $t_0\sigma_1\sigma_2\cdots\sigma_{n-1}$ and $t_n\sigma_1\sigma_2\cdots\sigma_{n-1}$. *We claim $\sigma_1\sigma_2\cdots\sigma_{n-1}\sigma_n$ is a most general unifier for $\{t_0, t_1, \ldots, t_n\}$.*

We are assuming that the set $\{t_0, t_1, \ldots, t_n\}$ has a unifier. Let σ be some arbitrary unifier for the set. In particular, σ unifies t_0 and t_1, so an idempotent most general unifier σ_1 for them exists. Thus, the first step of the process is meaningful. However, it is not obvious that the rest of the sequence of unifiers is even well-defined, since the existence of each item after the first requires that a complicated pair of terms, involving substitutions, be unifiable, and this unifiability needs verification. In fact the sequence is well-defined, but to verify this we must show something stronger.

Suppose, for $i < n$, that σ_1, $\sigma_2, \ldots$, σ_i have been defined *and that* $\sigma_1\sigma_2\cdots\sigma_i\sigma = \sigma$. (This is true if $i = 1$, since σ_1 is idempotent.) Now, the terms $t_0\sigma_1\sigma_2\cdots\sigma_i$ and $t_{i+1}\sigma_1\sigma_2\cdots\sigma_i$ are unifiable, because σ is a unifier for the set $\{t_0, t_1, \ldots, t_n\}$, and

$$\begin{aligned}(t_0\sigma_1\sigma_2\cdots\sigma_i)\sigma &= t_0\sigma\\ &= t_{i+1}\sigma\\ &= (t_{i+1}\sigma_1\sigma_2\cdots\sigma_i)\sigma.\end{aligned}$$

Let σ_{i+1} be an idempotent most general unifier for these two terms. Since σ itself unifies them, by Proposition 7.2.12, $\sigma = \sigma_{i+1}\sigma$. But then

$$\begin{aligned}\sigma_1\sigma_2\cdots\sigma_i\sigma_{i+1}\sigma &= \sigma_1\sigma_2\cdots\sigma_i\sigma\\ &= \sigma,\end{aligned}$$

and we are ready to proceed with the next step.

Then, finally, $\sigma_1\sigma_2\cdots\sigma_n$ must exist. Since σ_1 unifies t_0 and t_1, so does $\sigma_1\sigma_2$, which also unifies t_0 and t_2, and so on. Thus, $\sigma_1\sigma_2\cdots\sigma_n$ unifies the whole set $\{t_0, t_1, \ldots, t_n\}$. We also know that $\sigma_1\sigma_2\cdots\sigma_n\sigma = \sigma$, and since σ was any unifier for the set, it follows that $\sigma_1\sigma_2\cdots\sigma_n$ is most general.

This is one way of proceeding: Reduce multiple unification to a sequence of binary unification problems. Another simple way is reducing it to a single, though more complex, binary unification problem. Let f be an arbitrary $n + 1$-place function symbol, and consider the pair of terms $f(t_0, t_1, \ldots, t_n)$ and $f(t_0, t_0, \ldots, t_0)$. Clearly, a most general unifier for $\{t_0, t_1, \ldots, t_n\}$ is also a most general unifier for these two and conversely (the argument earlier establishes that a most general unifier for the set exists, so we are justified in talking about it). Now just use binary unification with these terms.

As a slightly different variation, apply binary unification to the pair of terms $f(t_0, t_1, \ldots, t_{n-1}, t_n)$ and $f(t_1, t_2, \ldots, t_n, t_0)$.

Concurrent Unification Instead of saying the substitution σ unifies the terms t and u, it is sometimes convenient to say it is a *solution* of the equation $t = u$. We use this terminology here.

Suppose we have the equations $t_0 = u_0$, $t_1 = u_1, \ldots, t_n = u_n$. A *simultaneous solution* is a substitution σ that is a solution for each equation. A *most general solution* is a simultaneous solution that is more general than any other. This time we want to reduce the problem of finding a most general simultaneous solution to a sequence of binary unifications, much as we did in the case of multiple unification.

Suppose the system has a simultaneous solution. Construct the following sequence of substitutions:

σ_0 Is an idempotent most general unifier of t_0 and u_0.

σ_1 Is an idempotent most general unifier of $t_1\sigma_0$ and $u_1\sigma_0$.

etc.

σ_n Is an idempotent most general unifier of $t_n\sigma_0\sigma_1 \cdots \sigma_{n-1}$
and $u_n\sigma_0\sigma_1 \cdots \sigma_{n-1}$.

We claim $\sigma_1\sigma_2 \cdots \sigma_{n-1}\sigma_n$ is a most general solution for the system of equations.

As with multiple unification, if each item in this sequence can be constructed, then $\sigma_1\sigma_2 \cdots \sigma_{n-1}\sigma_n$ exists and is a simultaneous solution of the family of equations. Hence, if the family of equations has no solution, the construction must terminate early. We leave the rest to you in Exercise 7.2.5.

Exercises

7.2.1$^{\mathrm{P}}$. Write a Prolog program that will locate a disagreement pair (if one exists), when given two terms.

7.2.2. Find a most general simultaneous solution of the system

$$k(x_1, x_3) = k(g(x_2), j(x_4)) \text{ and } f(x_2, q(x_4, a)) = f(h(x_3, a), x_5).$$

7.2.3. Attempt to solve the system $x = f(y)$ and $y = g(x)$. Explain what goes wrong.

7.2.4. Show that the multiple unification problem can be reduced to the concurrent unification problem.

7.2.5. In the concurrent unification discussion, show that if the system of equations has a simultaneous solution, then the sequence of substitutions is well-defined; each σ_i exists, and $\sigma_1\sigma_2 \cdots \sigma_{n-1}\sigma_n$ is a most general solution.

7.3 Unification Implemented

Several alternatives to Robinson's unification algorithm are fundamentally different. The one we will implement is just a mild variation.

First, in the Unification Algorithm, the unifying substitution σ is produced by successively composing simple substitutions, often called *bindings*. We will not actually carry out the calculation of the composition; instead, we will remember each of the bindings separately. This saves time without loosing information. We will remember these substitutions by maintaining an *environment list*, consisting of items like $[x, t]$, which is intended to record that occurrences of x should be replaced by occurrences of t. In its turn, t itself may contain variables that have their bindings elsewhere in the environment list.

Second, we will not actually carry out the substitution σ on the terms t_1 and t_2 that we are unifying. The reason now is not time but space: If the variable x has many occurrences, and σ replaces x by some very big term t, we will wind up with many copies of t when the substitution is carried out, thus wasting space. Instead we will remember, when we see an x, that we should treat it as if it were t. In other words, whenever we see an x, we will look up its value in the environment list.

Finally, we will only partially use the environment list for σ anyway. Partial application is possible because we are maintaining an environment list that says things like, replace x by t, where t in turn may have other variables for which the environment list supplies values. We can think of x as replaced by t without necessarily replacing variables in t by their respective values. If under such a partial replacement we find a disagreement pair—say $f(\cdots)$ and $g(\cdots)$, where f and g are distinct function symbols—we know unification is impossible and we do not need to know what the arguments of f and g are. Thus, a partial application of substitution may be sufficient, and if it is not, we can always carry out more of it.

Now we present the algorithm, implemented in Prolog. We need some representation for the free variables of the language L; these should not be confused with Prolog's variables. For this purpose we use expressions of the form `var(···)`.

```
/*  variable(X) :- X is a free variable.
*/

variable(var(_)).
```

Next we consider `partialvalue(Term, Env, Result)`, which partially evaluates a term in an environment. If `Term` is not a free variable, or is a free variable without a binding in the environment `Env`, then `Result` is Term unchanged. Otherwise, the binding of Term in Env is found, and the partial evaluation process is applied in turn to that binding.

```
/*  partialvalue(X, Env, Y) :-
      Y is the partially evaluated value of term X
      in environment Env.
*/

partialvalue(X, Env, Y) :-
   member([X,Z], Env),
   partialvalue(Z, Env, Y).

partialvalue(X, Env, X).

/*  member(X, Y) :- X is a member of the list Y.
*/

member(X, [X | _]).
member(X, [_ | Y]) :- member(X, Y).
```

The next few predicates constitute the so-called *occurs check*. They determine whether a variable occurs in a term, when the term is evaluated in a particular environment.

```
/*  in(X, T, Env) :-
      X occurs in term T, evaluated in environment Env.
*/

in(X, T, Env) :-
   partialvalue(T, Env, U),
   ( X == U;
     not variable(U), not atomic(U), infunctor(X, U, Env)
   ).

infunctor(X, U, Env) :-
   U =.. [_|L],
   inlist(X, L, Env).

inlist(X, [T|_], Env) :-
   in(X, T, Env).

inlist(X, [_|L], Env) :-
   inlist(X, L, Env).
```

Finally, we give the unification predicate itself. It should be read as follows: unify(Term1, Term2, Env, Newenv) is true if the result of unifying Term1 and Term2 in the environment Env produces the environment Newenv. We have used Prolog's "or," written as ";," to make the program easier to read. Basically, to unify two terms in an environment, first their values are (partially) calculated in that environment. Next, if one of these values is a variable, it is bound to the other value, provided the occurs check is not violated. Finally, if both values turn out to be more complicated terms, these terms are broken up and unified component by component.

```
/*  unify(Term1, Term2, Env, Newenv) :-
      Unifying Term1 and Term2 in environment Env
      produces the new environment Newenv.
*/

unify(Term1, Term2, Env, Newenv) :-
   partialvalue(Term1, Env, Val1),
   partialvalue(Term2, Env, Val2),
   (
     (Val1==Val2, Newenv = Env);
     (variable(Val1), not in(Val1, Val2, Env),
        Newenv=[ [Val1, Val2] | Env] );
     (variable(Val2), not in(Val2, Val1, Env),
        Newenv=[ [Val2, Val1] | Env] );
     (not variable(Val1), not variable(Val2),
        not atomic(Val1), not atomic(Val2),
        unifyfunctor(Val1, Val2, Env, Newenv) )
   ).

/*  unifyfunctor(Fun1, Fun2, Env, Newenv) :-
      Unifying the functors Fun1 and Fun2 in
      environment Env produces environment Newenv.
*/

unifyfunctor(Fun1, Fun2, Env, Newenv) :-
   Fun1 =.. [FunSymb | Args1],
   Fun2 =.. [FunSymb | Args2],
   unifylist(Args1, Args2, Env, Newenv).

/*  unifylist(List1, List2, Env, Newenv) :-
      Starting in environment Env and
      unifying each term in List1 with the
      corresponding term in List2 produces
      the environment Newenv.
```

```
*/

unifylist([ ], [ ], Env, Env).

unifylist([Head1 | Tail1], [Head2 | Tail2], Env, Newenv) :-
   unify(Head1, Head2, Env, Temp),
   unifylist(Tail1, Tail2, Temp, Newenv).
```

By now you should have a reasonable understanding of the Robinson Unification Algorithm, so we should be able to suppress its visible operation and assume we can simply call on unification as needed. Those of you who know something about how Prolog works probably know that a built-in Unification Algorithm is central to its operations. Indeed, Prolog will unify two of its terms `t` and `u` when given the query `t = u`. It would be nice if we could make use of this to simplify our work: Use Prolog variables instead of expressions like `var(···)` and use Prolog's built-in unification, which is fast and efficient. Unfortunately, the Unification Algorithm in standard Prolog is incorrect, since it omits the occurs check in the interests of greater speed. Nonetheless, a good compromise is available: We can use the built-in algorithm but impose our own occurs check. From now on this is the course we take. The following program is from Sterling and Shapiro [51]. In effect, Prolog's own environment list takes the place of the one we constructed.

```
/*  unify(Term1, Term2) :-
       Term1 and Term2 are unified with the occurs check.
       See Sterling and Shapiro, The Art of Prolog.
*/

unify(X,Y) :-
     var(X), var(Y), X=Y.
unify(X,Y) :-
     var(X), nonvar(Y), not_occurs_in(X,Y), X=Y.
unify(X,Y) :-
     var(Y), nonvar(X), not_occurs_in(Y,X), Y=X.
unify(X,Y) :-
     nonvar(X), nonvar(Y), atomic(X), atomic(Y), X=Y.
unify(X,Y) :-
     nonvar(X), nonvar(Y),
     compound(X), compound(Y),
     term_unify(X,Y).

not_occurs_in(X,Y) :-
     var(Y), X \== Y.
not_occurs_in(X,Y) :-
```

```
     nonvar(Y), atomic(Y).
not_occurs_in(X,Y) :-
     nonvar(Y), compound(Y), functor(Y,F,N),
     not_occurs_in(N,X,Y).

not_occurs_in(N,X,Y) :-
     N>0, arg(N,Y,Arg), not_occurs_in(X,Arg), N1 is N-1,
     not_occurs_in(N1,X,Y).
not_occurs_in(0,X,Y) .

term_unify(X,Y) :-
     functor(X,F,N), functor(Y,F,N), unify_args(N,X,Y).

unify_args(N,X,Y) :-
     N>0, unify_arg(N,X,Y), N1 is N-1, unify_args(N1,X,Y).
unify_args(0,X,Y).

unify_arg(N,X,Y) :-
     arg(N,X,ArgX), arg(N,Y,ArgY), unify(ArgX,ArgY).

compound(X) :- functor(X,_,N), N>0.
```

Exercises

7.3.1$^{\mathbf{P}}$. Write a Prolog program that composes substitutions; a substitution can be represented in the same way that the environment list was represented.

7.3.2$^{\mathbf{P}}$. Write a version of our first unification program, but one that produces a substitution rather than an environment list.

7.3.3$^{\mathbf{P}}$. Write a Prolog implementation that will carry out concurrent unification.

7.4 Free-Variable Semantic Tableaux

The chief practical difficulty in implementing tableaux is centered in the γ-rule. It allows us to add to a branch containing γ the formula $\gamma(t)$ where t is any closed term. How do we know what is a good choice? Of course, it is possible to simply try all (infinitely many) choices in some systematic fashion. Early theorem provers essentially did this. But it is hopelessly inefficient. A better solution—one that we will adopt—is to add to a branch containing γ the formula $\gamma(x)$, where x is a new free variable, and later use unification to choose a useful value for x, one that will close a branch. This accounts for the title of the section; we consider tableaux that allow formulas with free variables. But this device, in turn, creates a problem with the δ-rule. If we have postponed our decision of what terms to use in γ-rule applications, how can we be sure the parameters used in δ-rule applications are new? We will get around this by using a more complicated notion of parameter than before. If δ occurs on a branch, we add $\delta(f(x_1, \dots, x_n))$ to the branch end, where f is a *new function symbol*, and $x_1, \dots, x_n$ are all the free variables occurring in δ. Then, no matter what values are eventually assigned to $x_1, \dots, x_n$, $f(x_1, \dots, x_n)$ must be different from any of them (the occurs check is needed here). These new function symbols are called *Skolem function symbols.* Of course the δ formula may have no free variables at all, and the Skolem function symbol we want must be 0-place. But 0-place function symbols are simply constant symbols—they are parameters in the old sense. All this leads to the following definition.

Definition 7.4.1 Let $L = L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ be a first-order language. As usual, let $\mathbf{par}$ be a countable set of constant symbols not in $\mathbf{C}$. Also let $\mathbf{sko}$ be a countable set of function symbols not in $\mathbf{F}$ (called *Skolem function symbols*), including infinitely many one-place, infinitely many two-place, and so on. By $L^{\mathbf{sko}}$ we mean the first-order language $L(\mathbf{R}, \mathbf{F} \cup \mathbf{sko}, \mathbf{C} \cup \mathbf{par})$.

We begin with a very general version of the free-variable tableau system. It is non-deterministic, and great flexibility in proof creation is allowed. Of course, various restrictions must be imposed to have a system that is practical to implement. Since our soundness proof will apply to the general version, it automatically applies to any version with restrictions.

Proofs will be *of sentences* of L but will *use formulas* of $L^{\mathbf{sko}}$. The general mechanism is exactly as in Chapter 6, except for the quantifier rules. Thus, the propositional Tableau Expansion Rules are the same, as are the rules for branch closure. The quantifier rules are replaced by those in Table 7.1. In these rules, x must be a free variable that does not also occur bound in the tableau; f must be a new Skolem function symbol, and $x_1, \dots, x_n$ must be all the free variables occurring in the δ-formula.

$\dfrac{\gamma}{\gamma(x)}$	$\dfrac{\delta}{\delta(f(x_1, \ldots, x_n))}$
(for an unbound variable x)	(for f new Skolem and $x_1, \ldots, x_n$ all the free variables of δ)

TABLE 7.1. Free-Variable Tableau Expansion Rules

Definition 7.4.2 Let σ be a substitution and $\mathbf{T}$ be a tableau. We extend the action of σ to $\mathbf{T}$ by setting $\mathbf{T}\sigma$ to be the result of replacing each formula X in $\mathbf{T}$ by $X\sigma$. We say σ is *free* for $\mathbf{T}$, provided σ is free for every formula in $\mathbf{T}$.

Tableau Substitution Rule If $\mathbf{T}$ is a tableau for the set S of sentences of L, and the substitution σ is free for $\mathbf{T}$, then $\mathbf{T}\sigma$ is also a tableau for S.

As usual, a free-variable tableau proof of X is a closed tableau for $\{\neg X\}$, though now the tableau is constructed using the propositional rules, the free-variable quantifier rules given, and the Tableau Substitution Rule.

Example The following is a one-branch tableau proof of $(\exists w)(\forall x)R(x, w, f(x, w)) \supset (\exists w)(\forall x)(\exists y)R(x, w, y)$.

1. $\neg[(\exists w)(\forall x)R(x, w, f(x, w)) \supset (\exists w)(\forall x)(\exists y)R(x, w, y)]$
2. $(\exists w)(\forall x)R(x, w, f(x, w))$
3. $\neg(\exists w)(\forall x)(\exists y)R(x, w, y)$
4. $(\forall x)R(x, a, f(x, a))$
5. $\neg(\forall x)(\exists y)R(x, v_1, y)$
6. $\neg(\exists y)R(b(v_1), v_1, y)$
7. $R(v_2, a, f(v_2, a))$
8. $\neg R(b(v_1), v_1, v_3)$

In this tableau proof 2 and 3 are from 1 by the α-rule; 4 is from 2 by the δ-rule; here a is a zero-place Skolem function symbol, i.e., a parameter; 5 is from 3 by the γ-rule, introducing the new free variable v_1; 6 is from 5 by the δ-rule, using the one-place Skolem function symbol b; 7 is from 4 using the γ-rule, introducing the new free variable v_2. Then 8 is from 6 using the γ-rule, introducing the new free variable v_3.

Now we apply the Tableau Substitution Rule, using the substitution $\sigma = \{v_1/a, v_2/b(a), v_3/f(b(a), a)\}$, producing a closed tableau by making 7 and 8 conflict. The substitution is trivially free, since the terms being substituted are all closed.

We have defined the free-variable tableau system in a very general way, because that is most desirable when it comes to proving soundness. But we have still not solved all the implementation problems. Before we introduced free variables into tableaux, the problem was, Which terms should we use in γ-rule applications? We have simply shifted the problem to, Which free substitutions should we apply? But now there is an obvious choice: We should apply substitutions that cause a branch to close, and we can determine these using unification.

We still have to meet the requirement that substitutions must be free for a tableau, and freeness is generally an expensive thing to check for. But there is one simple case where the check can be avoided. Suppose A and $\neg B$ are quantifier-free formulas that occur on a branch of a tableau, and suppose σ is a most general unifier for A and B, obtained using the Unification Algorithm. The only variables that an application of σ can introduce are those already present in A or B. These will all be free variables introduced by the γ-rule and so will be different than any bound variables. It follows easily that σ must be free for the tableau.

The simplest cases that meet the no-bound-variables condition are those involving *literals.* Thus, it makes sense to restrict substitution applications to most general substitutions that will cause a branch to *atomically* close. For convenience we give a name to this restricted version of substitution; in it, *MGU* stands for *most general unifier.*

MGU Atomic Closure Rule Suppose $\mathbf{T}$ is a tableau for the set S of sentences of L, and some branch of $\mathbf{T}$ contains A and $\neg B$, where A and B are atomic. Then $\mathbf{T}\sigma$ is also a tableau for S, *where σ is a most general unifier of A and B.*

We will prove the soundness of the free-variable system in full generality. Since the MGU Atomic Closure Rule is just a special case of the Tableau Substitution Rule, its addition to the system will not affect soundness. And we will prove the system is complete even when all applications of the Tableau Substitution Rule are restricted to MGU Atomic Closure Rule applications (and other restrictions are imposed as well). We make use of this in the tableau implementation given in the next section.

Remarks In the first edition of this book, the δ-free variable tableau expansion rule was stated differently. While correct, it led to less efficient tableau constructions. The version presented in this edition is due to Reiner Hähnle and Peter Schmitt [23], and independently to Wilfried Sieg (unpublished).

If we wish to check tableaux for non-atomic closure, we must use a version of unification that distinguishes between free and bound occurrences of variables. Such an algorithm has been developed, [45].

Exercises

7.4.1. Give tableau proofs of the following, using the rules of this section:

1. $(\exists x)(\forall y)R(x,y) \supset (\forall y)(\exists x)R(x,y)$.
2. $(\exists x)[P(x) \supset (\forall x)P(x)]$.
3. $(\forall x)(\forall y)[P(x) \wedge P(y)] \supset (\exists x)(\exists y)[P(x) \vee P(y)]$.
4. $(\forall x)(\forall y)[P(x) \wedge P(y)] \supset (\forall x)(\forall y)[P(x) \vee P(y)]$.
5. $(\forall x)(\exists y)(\forall z)(\exists w)[R(x,y) \vee \neg R(w,z)]$.
6. $(\exists x)(\forall y)[P(y) \uparrow (P(x) \uparrow Q(x))] \subset (\forall x)Q(x)$.

7.5 A Tableau Implementation

Most theorem provers that have been implemented are based on resolution, but there has always been interest in tableaux [35], and recently quite an efficient theorem prover using tableaux has appeared [34]. In this section we give an implementation based on the free-variable first-order tableau system. The implementation imposes constraints much like the propositional version did. There is a *strictness* condition, although it does not apply to γ-formulas for reasons discussed later. Also, we apply all Tableau Expansion Rules before considering branch closure. Finally, all Tableau Substitution Rule applications will involve most general unifiers that are designed to close branches. In fact, they will be applications of the MGU Atomic Closure Rule. In Section 7.8 we will show that the tableau system is complete even when restricted in these ways. Our implementation, in Prolog, follows the general pattern of the propositional version in Section 3.2, but it is considerably more complicated, and so we present it one part at a time, preceding each portion with a discussion of its key features.

We begin with a few minor utility routines, to get them out of the way. Note that the `remove` predicate is a little different than in the propositional case. We are using Prolog variables as the free variables of our formal language, so we must be careful not to introduce an accidental unification. We will remove a formula from a branch only if it is there, not if it can be unified with something that is there.

```
/*   member(Item, List) :- Item occurs in List.
*/

member(X, [X | _]).
member(X, [_ | Tail]) :- member(X, Tail).
```

```
/*   remove(Item, List, Newlist) :-
         Newlist is the result of removing all
         occurrences of Item from List.
*/

remove(X, [ ], [ ]).
remove(X, [Y | Tail], Newtail) :-
   X == Y,
   remove(X, Tail, Newtail).
remove(X, [Y | Tail], [Y | Newtail]) :-
   X \== Y,
   remove(X, Tail, Newtail).

/*   append(ListA, ListB, Newlist) :-
         Newlist is the result of appending ListA
         and ListB.
*/

append([ ], List, List).
append([Head | Tail], List, [Head | Newlist]) :-
   append(Tail, List, Newlist).
```

The propositional operators are exactly what they were in Section 2.9, and so their characterization is carried over without change.

```
/*   Propositional operators are: neg, and, or, imp,
       revimp, uparrow, downarrow, notimp
       and notrevimp.
*/

?-op(140, fy, neg).
?-op(160, xfy, [and, or, imp, revimp, uparrow, downarrow,
   notimp, notrevimp]).

/*   conjunctive(X) :- X is an alpha formula.
*/

conjunctive(_ and _).
conjunctive(neg(_ or _)).
conjunctive(neg(_ imp _)).
conjunctive(neg(_ revimp _)).
conjunctive(neg(_ uparrow _)).
conjunctive(_ downarrow _).
conjunctive(_ notimp _).
conjunctive(_ notrevimp _).

/*   disjunctive(X) :- X is a beta formula.
*/

disjunctive(neg(_ and _)).
disjunctive(_ or _).
disjunctive(_ imp _).
disjunctive(_ revimp _).
disjunctive(_ uparrow _).
disjunctive(neg(_ downarrow _)).
disjunctive(neg(_ notimp _)).
disjunctive(neg(_ notrevimp _)).

/*   unary(X) :- X is a double negation,
         or a negated propositional constant.
*/

unary(neg neg _).
unary(neg true).
unary(neg false).

/*   binary_operator(X) :- X is a binary operator.
*/

binary_operator(X) :-
   member(X, [and, or, imp, revimp, uparrow, downarrow,
   notimp, notrevimp]).

/*   components(X, Y, Z) :- Y and Z are the components of
         the formula X, as defined in the alpha and
         beta table.
*/

components(X and Y, X, Y).
components(neg(X and Y), neg X, neg Y).
components(X or Y, X, Y).
components(neg(X or Y), neg X, neg Y).
components(X imp Y, neg X, Y).
components(neg(X imp Y), X, neg Y).
components(X revimp Y, X, neg Y).
components(neg(X revimp Y), neg X, Y).
components(X uparrow Y, neg X, neg Y).
components(neg(X uparrow Y), X, Y).
components(X downarrow Y, neg X, neg Y).
components(neg(X downarrow Y), X, Y).
components(X notimp Y, X, neg Y).
components(neg(X notimp Y), neg X, Y).
components(X notrevimp Y, neg X, Y).
components(neg(X notrevimp Y), X, neg Y).

/*   component(X, Y) :- Y is the component of the
          unary formula X.
*/

component(neg neg X, X).
component(neg true, false).
component(neg false, true).
```

For quantifiers we make use of Prolog's functional notation. The formula $(\forall x)P(x)$ will be represented using a two-argument functor, as `all(x, p(x))`. Similarly, $(\exists x)p(x)$ will be represented as `some(x, p(x))`. Then a sentence like $(\exists x)(P(x) \supset (\forall x)P(x))$ will be written `some(x, p(x) imp all(x, p(x)))`.

```
/*   universal(X) :- X is a gamma formula.
*/

universal(all(_,_)).
universal(neg some(_,_)).

/*   existential(X) :- X is a delta formula.
*/

existential(some(_,_)).
existential(neg all(_,_)).

/*   literal(X) :- X is a literal.
*/

literal(X) :-
   not conjunctive(X),
   not disjunctive(X),
   not unary(X),
   not universal(X),
   not existential(X).
```

```
/*   atomicfmla(X) :- X is an atomic formula.
*/

atomicfmla(X) :-
   literal(X),
   X \= neg _.
```

α- And β-formulas have components, but γ- and δ-formulas have instances. The notion of instance presupposes that we know how to carry out the substitution of a term for a free variable in a formula. This is what the predicate sub does. It calls on the auxiliary predicate sub_, which does the real work; the main purpose of this division of labor is to provide an added level of safety by preventing accidental backtracking (note the cut in the body of sub). You should compare the clauses for sub_ that follow with the definition of substitution, Definition 5.2.11. The only mildly tricky part is that involving substitution at the atomic level. Here formulas and terms are broken down into lists, and the substitution is carried out by a simple list traversal.

```
/*   sub(Term, Variable, Formula, NewFormula) :-
        NewFormula is the result of substituting
        occurrences of Term for each free
        occurrence of Variable in Formula.
*/

sub(Term, Variable, Formula, NewFormula) :-
   sub_(Term, Variable, Formula, NewFormula) , !.

sub_(Term, Var, A, Term) :- Var == A.
sub_(Term, Var, A, A) :- atomic(A).
sub_(Term, Var, A, A) :- var(A).

sub_(Term, Var, neg X, neg Y) :-
   sub_(Term, Var, X, Y).

sub_(Term, Var, Binary_One, Binary_Two) :-
   binary_operator(F),
   Binary_One =.. [F, X, Y],
   Binary_Two =.. [F, U, V],
   sub_(Term, Var, X, U),
   sub_(Term, Var, Y, V).
sub_(Term, Var, all(Var, Y), all(Var, Y)).
sub_(Term, Var, all(X, Y), all(X, Z)) :-
   sub_(Term, Var, Y, Z).
sub_(Term, Var, some(Var, Y), some(Var, Y)).
sub_(Term, Var, some(X, Y), some(X, Z)) :-
   sub_(Term, Var, Y, Z).

sub_(Term, Var, Functor, Newfunctor) :-
   Functor =.. [F | Arglist],
   sub_list(Term, Var, Arglist, Newarglist),
   Newfunctor =.. [F | Newarglist].

sub_list(Term, Var, [Head | Tail], [Newhead | Newtail]) :-
   sub_(Term, Var, Head, Newhead),
   sub_list(Term, Var, Tail, Newtail).

sub_list(Term, Var, [ ], [ ]).
```

Now the notion of an instance of a γ- or δ-formula can be characterized easily.

```
/*  instance(F, Term, Ins) :-
          F is a quantified formula, and Ins is the result
          of removing the quantifier and replacing all
          free occurrences of the quantified variable by
          occurrences of Term.
*/

instance(all(X,Y), Term, Z) :- sub(Term, X, Y, Z).
instance(neg some(X,Y), Term, neg Z) :- sub(Term, X, Y, Z).
instance(some(X,Y), Term, Z) :- sub(Term, X, Y, Z).
instance(neg all(X,Y), Term, neg Z) :- sub(Term, X, Y, Z).
```

In using the δ-rule, we need a new function symbol each time the rule is applied. We take a Skolem function symbol to be `fun(n)`, where `n` is a number that is increased by 1 at each δ-rule application. We find it convenient to remember the current value of this number using a Prolog mechanism that is something like a global variable in a more conventional programming language. Specifically, we introduce a predicate, `funcount`, whose purpose is to remember the current Skolem function number. At the start of execution, there is a program clause `funcount(1)`. Then each time the δ-rule is used, the predicate `newfuncount` is called on.

This *retracts* the program clause funcount(n) and *asserts* in its place a new program clause funcount(n+1). This rather nonlogical use of Prolog could be avoided, but we believe no harm is done here. Since a user of this program will probably want to test several different formulas for theoremhood, we also include a reset predicate, which simply resets the value of funcount to 1.

```
/*   funcount(N) :- N is the current Skolem function index.
*/

funcount(1).

/*   newfuncount(N) :-
          N is the current Skolem function index, and as a
          side effect, the remembered value is incremented.
*/

newfuncount(N) :-
   funcount(N),
   retract(funcount(N)),
   M is N+1,
   assert(funcount(M)).

/*   reset :- the Skolem function index is reset to 1.
*/

reset :-
   retract(funcount(_)),
   assert(funcount(1)),
   !.
```

Once the problem of ensuring fresh Skolem functions has been addressed, the introduction of terms involving Skolem functions is easy. This is taken care of by the predicate sko_fun(X,Y).

```
/*   sko_fun(X, Y) :-
          X is a list of free variables, and Y is a
          previously unused Skolem function symbol
          applied to those free variables.
*/

sko_fun(Varlist, Skoterm) :-
```

```
   newfuncount(N),
   Skoterm =.. [fun | [N | Varlist]].
```

We are still not done with complications that are due to the δ-rule. In applying it we need to know what the free variables of a formula are. This can be done in two ways. First, we can figure out what they are when we need to. Second, we can keep track of them as we go along. Either way is acceptable—we use the second method, and leave the first to you as an exercise. Instead of placing formulas on tableau branches, in our program we use *notated formulas*, where a notated formula is a structure `n(Notation, Formula)` in which `Formula` is a formula in the usual sense and `Notation` is a list of free variables, intended to be those occurring in `Formula`. We introduce two utility predicates for dealing with this structure, mostly to make reading the clauses of `singlestep` somewhat easier.

```
/*   notation(Notated, Free) :-
          Notated is a notated formula, and Free is its
          associated free variable list.
*/

notation(n(X, Y), X).

/*   fmla(Notated, Formula) :-
          Notated is a notated formula, and Formula is its
          formula part.
*/

fmla(n(X, Y), Y).
```

In our propositional tableau implementation, we had a `singlestep` predicate, which carried out a single step of the tableau expansion. This was called on by an `expand` predicate, which applied `singlestep` until all possible Tableau Expansion Rules had been applied, after which we tested for closure. This simple approach no longer works now that quantifiers have been introduced. The γ-rule is the source of the difficulty. Recall that in its original form, without free variables, it allowed us to go from γ to $\gamma(t)$ for any closed term t, and such a rule can clearly be applied to the same formula several times, using a different closed term each time. Since there are infinitely many closed terms, we can never complete the process of expanding a tableau, and so we will never reach the stage at which we test for closure. Similar problems are encountered with the free-variable version, too. But after all, if there were some process that always produced a complete first-order tableau expansion in a finite number of steps, we would have a decision procedure for first-order logic, and a famous result known as *Church's Theorem* says there can be no such thing [8].

Our way out of this difficulty is to limit the number of applications of the γ-rule in a tableau proof. We have a user preset bound, known as the *Q-depth*, standing for quantifier depth. When the γ-rule has been applied the maximum allowed number of times, as specified by the value of Q-depth, it can not be applied any more. In this way a complete tableau expansion *to a given Q-depth* can be constructed in a finite number of steps, and we can then go on to the closure testing stage. Proofs are finite objects, so if a sentence X is provable, it has a proof in which some finite number of γ-rule applications have been made. Consequently, if X is valid it will be provable at some Q-depth. So in principle, by trying higher and higher values for Q-depth, a proof of any valid sentence will eventually be discovered. (Of course, this ignores problems of time and space complexity.) On the other hand, being invalid is equivalent to being unprovable at *every* Q-depth, and this is something no implementation can discover for us, since infinitely many proof attempts would have to be made.

We need to ensure that if there are several γ-formulas on a branch each gets its fair share of attention and that we do not use up our Q-depth allotment of γ-rule applications on a single formula. For this we use an idea of Smullyan [48], which amounts to treating each branch as a priority queue. When working with a branch, we always work from the top down. We pick the uppermost negation, or α-formula, or whatever, to apply a rule to. Now, if we have chosen a formula that is not a γ-formula, we remove it from the branch and add its components, or an instance. (Because of the way Prolog works, it is convenient to always add these to the top of the branch, though this is of no theoretical significance.) On the other hand, if we have chosen a γ-formula to work with, we remove it, add an appropriate instance to the branch top, and add a fresh occurrence of γ *to the branch end.* This guarantees that over time, each γ-formula on the branch will be considered, after which those that have been used once will be reused in their original order.

We have a similar problem ensuring that each *branch* also gets its fair share of γ-rule applications. Our solution here is a similar one. We treat a tableau representation, which is a list of notated branches, as a priority queue also. We work as long as possible on the first branch, then the second, and so on, unless we happen to apply a γ-rule. When a γ-rule is applied to a branch, that branch is moved to the end of the list of branches. This guarantees that the γ-rule applications allowed by the value of Q-depth are spread over all branches fairly.

Now we give the singlestep predicate clauses. These should be compared with those in Section 2.9. There are four arguments now, instead of two. Two of the arguments represent the tableau before and after a single Tableau Expansion Rule has been applied, just as in the propositional implementation. The two new arguments represent the available number of γ-rule applications before and after the rule application, the Q-depth. Of course, this plays a role only in a single program clause, the one corresponding to the γ-rule. The propositional cases are basically the same as in earlier chapters, except for the complications arising from the use of notated formulas. Notice that the list of free variables associated with a formula is passed along unchanged by every rule except for γ, when a new free variable is added. In addition, in applying the γ-rule, we must decrease the available Q-depth by 1 and reorder both the branch and the tableau itself. Note again that we are using Prolog's variables as free variables.

```
/*   singlestep(OldTableau, OldQdepth,
            NewTableau, NewQdepth) :-
          the application of one tableau rule to OldTableau,
          with a Q-depth of OldQdepth, will produce the
          tableau NewTableau, and will change the available
          Q-depth to NewQdepth.
*/

singlestep([OldBranch | Rest], Qdepth,
      [NewBranch | Rest], Qdepth) :-
   member(NotatedFormula, OldBranch),
   notation(NotatedFormula, Free),
   fmla(NotatedFormula, Formula),
   unary(Formula),
   component(Formula, NewFormula),
   notation(NewNotatedFormula, Free),
   fmla(NewNotatedFormula, NewFormula),
   remove(NotatedFormula, OldBranch, TempBranch),
   NewBranch = [NewNotatedFormula | TempBranch].

singlestep([OldBranch | Rest], Qdepth,
      [NewBranch | Rest], Qdepth) :-
   member(NotatedAlpha, OldBranch),
   notation(NotatedAlpha, Free),
   fmla(NotatedAlpha, Alpha),
   conjunctive(Alpha),
   components(Alpha, AlphaOne, AlphaTwo),
   notation(NotatedAlphaOne, Free),
   fmla(NotatedAlphaOne, AlphaOne),
   notation(NotatedAlphaTwo, Free),
   fmla(NotatedAlphaTwo, AlphaTwo),
   remove(NotatedAlpha, OldBranch, TempBranch),
   NewBranch =
     [NotatedAlphaOne, NotatedAlphaTwo | TempBranch].

singlestep([OldBranch | Rest], Qdepth,
      [NewBranchOne, NewBranchTwo | Rest], Qdepth) :-
   member(NotatedBeta, OldBranch),
   notation(NotatedBeta, Free),
   fmla(NotatedBeta, Beta),
   disjunctive(Beta),
   components(Beta, BetaOne, BetaTwo),
   notation(NotatedBetaOne, Free),
   fmla(NotatedBetaOne, BetaOne),
   notation(NotatedBetaTwo, Free),
   fmla(NotatedBetaTwo, BetaTwo),
   remove(NotatedBeta, OldBranch, TempBranch),
   NewBranchOne = [NotatedBetaOne | TempBranch],
   NewBranchTwo = [NotatedBetaTwo | TempBranch].

singlestep([OldBranch | Rest], Qdepth,
      [NewBranch | Rest], Qdepth) :-
   member(NotatedDelta, OldBranch),
   notation(NotatedDelta, Free),
   fmla(NotatedDelta, Delta),
   existential(Delta),
   sko_fun(Free, Term),
   instance(Delta, Term, DeltaInstance),
   notation(NotatedDeltaInstance, Free),
   fmla(NotatedDeltaInstance, DeltaInstance),
   remove(NotatedDelta, OldBranch, TempBranch),
   NewBranch = [NotatedDeltaInstance | TempBranch].

singlestep([OldBranch | Rest], OldQdepth,
      NewTree, NewQdepth) :-
   member(NotatedGamma, OldBranch),
   notation(NotatedGamma, Free),
   fmla(NotatedGamma, Gamma),
   universal(Gamma),
   OldQdepth > 0,
   remove(NotatedGamma, OldBranch, TempBranch),
   NewFree = [V | Free],
   instance(Gamma, V, GammaInstance),
   notation(NotatedGammaInstance, NewFree),
   fmla(NotatedGammaInstance, GammaInstance),
   append([NotatedGammaInstance | TempBranch],
     [NotatedGamma], NewBranch),
   append(Rest, [NewBranch], NewTree),
   NewQdepth is OldQdepth-1.

singlestep([Branch | OldRest], OldQdepth,
      [Branch | NewRest], NewQdepth) :-
   singlestep(OldRest, OldQdepth, NewRest, NewQdepth).
```

Now the expand predicate is just as it was in the propositional case, except that we have an extra argument to specify Q-depth.

```
/*   expand(Tree, Qdepth, Newtree) :-
          the complete expansion of the tableau Tree,
          allowing Qdepth applications of the gamma
          rule, is Newtree.
*/

expand(Tree, Qdepth, Newtree) :-
   singlestep(Tree, Qdepth, TempTree, TempQdepth),
   expand(TempTree, TempQdepth, Newtree).

expand(Tree, Qdepth, Tree).
```

Testing an expanded tableau for closure is more complicated than it was in the propositional case. A branch is counted as closed if it contains literals X and $\neg Y$, where X and Y unify. As noted earlier, we can not rely on Prolog's built-in unification, because it incorrectly omits the occurs check. Consequently, we need the unification clauses from Section 7.3. These are repeated here, for convenience.

```
/*   unify(Term1, Term2) :-
          Term1 and Term2 are unified with the occurs check.
          See Sterling and Shapiro,
          The Art of Prolog.
*/

unify(X,Y) :-
```

```
   var(X), var(Y), X=Y.
unify(X,Y) :-
   var(X), nonvar(Y), not_occurs_in(X,Y), X=Y.
unify(X,Y) :-
   var(Y), nonvar(X), not_occurs_in(Y,X), Y=X.
unify(X,Y) :-
   nonvar(X), nonvar(Y), atomic(X), atomic(Y), X=Y.
unify(X,Y) :-
   nonvar(X), nonvar(Y),
   compound(X), compound(Y),
   term_unify(X,Y).

not_occurs_in(X,Y) :-
   var(Y), X \== Y.
not_occurs_in(X,Y) :-
   nonvar(Y), atomic(Y).
not_occurs_in(X,Y) :-
   nonvar(Y), compound(Y), functor(Y,F,N),
   not_occurs_in(N,X,Y).

not_occurs_in(N,X,Y) :-
   N>0, arg(N,Y,Arg), not_occurs_in(X,Arg), N1 is N-1,
   not_occurs_in(N1,X,Y).
not_occurs_in(0,X,Y) .

term_unify(X,Y) :-
   functor(X,F,N), functor(Y,F,N), unify_args(N,X,Y).

unify_args(N,X,Y) :-
   N>0, unify_arg(N,X,Y), N1 is N-1, unify_args(N1,X,Y).
unify_args(0,X,Y).

unify_arg(N,X,Y) :-
   arg(N,X,ArgX), arg(N,Y,ArgY), unify(ArgX,ArgY).

compound(X) :- functor(X,_,N), N>0.
```

With unification available, the clauses for closure are straightforward. Note that we only test for *atomic closure* via the MGU Atomic Closure Rule. As we pointed out in the last section, this guarantees that all applications of the Tableau Substitution Rule are free ones.

```
/*   closed(Tableau) :- every branch of Tableau can be
```

```
        made to contain a contradiction, after a suitable
        free variable substitution.
*/

closed([Branch | Rest]) :-
   member(Falsehood, Branch),
   fmla(Falsehood, false),
   closed(Rest).

closed([Branch | Rest]) :-
   member(NotatedOne, Branch),
   fmla(NotatedOne, X),
   atomicfmla(X),
   member(NotatedTwo, Branch),
   fmla(NotatedTwo, neg Y),
   unify(X, Y),
   closed(Rest).

closed([ ]).
```

Now, exactly as in the propositional case, we simply expand (to a given Q-depth) using expand, and test for closure. We carry over the amenities that we used earlier.

```
/*   if_then_else(P, Q, R) :-
         either P and Q, or not P and R.
*/

if_then_else(P, Q, R) :- P, !, Q.

if_then_else(P, Q, R) :- R.

/*   test(X, Qdepth) :- create a complete tableau expansion
        for neg X, allowing Qdepth applications of the
        gamma rule.  Test for closure.
*/

test(X, Qdepth) :-
   reset,
   notation(NotatedFormula, [ ]),
   fmla(NotatedFormula, neg X),
   expand([[NotatedFormula]], Qdepth, Tree),
   if_then_else(closed(Tree), yes(Qdepth), no(Qdepth)).
```

```
yes(Qdepth) :-
   write('First-order tableau theorem at Q-depth '),
   write(Qdepth),
   write(.),
   nl.

no(Qdepth) :-
   write('Not a first-order tableau theorem at Q-depth '),
   write(Qdepth),
   write('.'),
   nl.
```

This is the end of the program. To use it with a formula `X`, select a reasonable Q-depth, `Q`, and issue the query `test(X, Q)`.

In Section 3.2 two versions of a propositional tableau implementation were given. The first, like the one in this section, began with a tableau expansion stage, followed by a closure testing stage. The second version applied a test for closure after each Tableau Expansion Rule application. Such variations are equally applicable in the first-order case as well. We leave them to you, as exercises.

Exercises

7.5.1^P. In the program of this section, at the stage of testing for closure, there may still be non-atomic formulas left, γ-formulas, for instance. This leads to a certain amount of inefficiency. Modify the program so that non-atomic formulas are removed before testing for closure.

7.5.2^P. Write a Prolog procedure that directly determines the free variables occurring in a formula. Then use your procedure and rewrite the Prolog program above so that formulas, and not notated formulas, occur on branches.

7.5.3^P. Modify the Prolog program of this section by incorporating an `expand_and_close` predicate, as in Section 3.2, in place of the `expand` predicate that was used.

7.5.4^P. Write a modified version of the Prolog implementation of this section that removes closed branches from the tableau as it generates it, thus avoiding redundant tests for closure.

7.5.5^P. In Section 7.3 we gave an implementation of the unification algorithm directly, not using Prolog's built-in version. In this, expressions of the form `var(···)` were used as variables. Rewrite the Prolog program of this section using these as free variables, rather than using Prolog variables as we did, and using the corresponding unification implementation from Section 7.3.

7.6 Free-Variable Resolution

We have seen how to turn the semantic tableau system with parameters, from the previous chapter, into a system suitable for automation, using unification. The same ideas apply to resolution. We sketch things and leave the actual implementation as an exercise. Just as in Section 7.4, proofs will be of sentences of a first-order language L, but will use formulas of L^{sko}. The formal system is much as in Section 6.2, except that the quantifier rules are replaced by free-variable versions and a substitution rule is added. We begin with the expansion rules, in Table 7.2. In these rules, x is a free variable that does not also occur bound in the resolution expansion; f is a new Skolem function symbol and $x_1, \ldots, x_n$ are all the free variables occurring in the δ-formula.

$\dfrac{\gamma}{\gamma(x)}$	$\dfrac{\delta}{\delta(f(x_1,\ldots,x_n))}$
(for an unbound variable x)	(for f new and $x_1,\ldots,x_n$ all the free variables of δ)

TABLE 7.2. Free-Variable Resolution Expansion Rules

Definition 7.6.1 Let σ be a substitution and $\mathbf{R}$ be a resolution expansion. We extend the action of σ to $\mathbf{R}$ by setting $\mathbf{R}\sigma$ to be the result of replacing each formula X in $\mathbf{R}$ by $X\sigma$. We say σ is *free* for $\mathbf{R}$, provided σ is free for every formula in $\mathbf{R}$.

Resolution Substitution Rule If $\mathbf{R}$ is a resolution expansion for the set S of sentences of L, and the substitution σ is free for $\mathbf{R}$, then $\mathbf{R}\sigma$ is also a resolution expansion for S.

As usual, a resolution proof of X is a closed resolution expansion for $\{\neg X\}$. Closure of a resolution expansion still means that it contains the empty clause. The key point is that the resolution-system rules are used in constructing the resolution expansion, instead of the quantifier rules from the previous chapter.

Example The following is a resolution proof of

$$(\exists w)(\forall x)R(x, w, f(x, w)) \supset (\exists w)(\forall x)(\exists y)R(x, w, y)$$

1. $[\neg((\exists w)(\forall x)R(x, w, f(x, w)) \supset (\exists w)(\forall x)(\exists y)R(x, w, y))]$
2. $[(\exists w)(\forall x)R(x, w, f(x, w))]$
3. $[\neg(\exists w)(\forall x)(\exists y)R(x, w, y)]$
4. $[(\forall x)R(x, a, f(x, a))]$
5. $[\neg(\forall x)(\exists y)R(x, v_1, y)]$
6. $[\neg(\exists y)R(b(v_1), v_1, y)]$
7. $[R(v_2, a, f(v_2, a))]$
8. $[\neg R(b(v_1), v_1, v_3)]$

Here 4 is from 2 by δ (with a being a new parameter); 5 is from 3 by γ (with v_1 a new free variable); 6 is from 5 by δ (with b as a new Skolem function symbol; 7 is from 4 by γ; and 8 is from 6 by γ. Now the substitution $\{v_1/a, v_2/b(a), v_3/f(b(a), a)\}$ is free for the resolution expansion. When it is applied, 7 becomes $[R(b(a), a, f(b(a), a))]$ and 8 becomes $[\neg R(b(a), a, f(b(a), a))]$. From these we get $[\,]$ by the Resolution Rule.

As expected, the problem in implementing this free-variable version of resolution comes from the Resolution Substitution Rule: What substitutions do we use? The obvious answer is to use a substitution that allows us to apply the Resolution Rule. That is, if one generalized disjunction in a resolution expansion contains X and another contains $\neg Y$, we should try to unify X and Y, then apply Resolution. And, as with tableaux, we can guarantee freeness of the unifying substitution if X and Y are atomic. But there is a hidden problem here. Suppose we have the following two clauses:

$$[P(x, f(y)), P(g(y), f(a)), Q(c, z)] \text{ and } [\neg P(g(a), z), R(x, a)].$$

We might choose to unify $P(x, f(y))$ and $P(g(a), z)$, in which case $\sigma = \{x/g(a), z/f(y)\}$ is a most general unifier. Applying it to both clauses, we get

$$[P(g(a), f(y)), P(g(y), f(a)), Q(c, f(y))] \text{ and } [\neg P(g(a), f(y)), R(g(a), a)],$$

and so the Resolution Rule yields

$$[P(g(y), f(a)), Q(c, f(y)), R(g(a), a)].$$

On the other hand, $\tau = \{x/g(a), z/f(a), y/a\}$ is also a unifier, though not most general, and when we apply it we get

$$[P(g(a), f(a)), P(g(a), f(a)), Q(c, f(a))] \text{ and } [\neg P(g(a), f(a)), R(g(a), a)].$$

So now the Resolution Rule gives us

$$[Q(c, f(a)), R(g(a), a)],$$

which is simpler! The use of a less general substitution got us nearer the goal of producing the empty clause.

There is nothing quite analogous for the tableau system. Of course what happened is that the substitution τ also happened to unify two formulas *within* one of the clauses, as well as unifying *across* clauses. Such a possibility must be taken into account if we are to have a complete proof procedure. Two general approaches have been introduced to deal with this.

Notice that in the example, the substitution τ, though not a most general unifier of $P(x, f(y))$ and $P(g(a), z)$, is a most general unifier of the three formulas $P(x, f(y))$, $P(g(a), z)$, and $P(g(y), f(a))$. This suggests the following rule:

General Literal Resolution Rule Suppose $\mathbf{R}$ is a resolution expansion for the set S of sentences of L, $\mathbf{R}$ contains the generalized disjunctions $[X_1, \ldots, X_n, Y_1, \ldots, Y_m]$ and $[\neg Z_1, \ldots, \neg Z_k, W_1, \ldots, W_p]$, where each X_i and Z_i is atomic and σ is a most general unifier of $\{X_1, \ldots, X_n, Z_1, \ldots, Z_k\}$. Then $\mathbf{R}^*\sigma$ is also a resolution expansion for S, where $\mathbf{R}^*$ is $\mathbf{R}$ with $[Y_1, \ldots, Y_m, W_1, \ldots, W_p]$ added.

This rule combines the Resolution Rule and the Resolution Substitution Rule, supplies a mechanism for choosing the substitutions, and ensures their freeness. We will show in Section 7.9 that the Resolution system is complete using this rule, with no other applications of Substitution. On the other hand, there is a certain simplicity in not bringing multiple unification into the picture. We could also consider the following special case of the rule just stated:

Binary Literal Resolution Rule Suppose $\mathbf{R}$ is a resolution expansion for the set S of sentences of L, $\mathbf{R}$ contains the generalized disjunctions $[X, Y_1, \ldots, Y_m]$ and $[\neg Z, W_1, \ldots, W_p]$, where X and Z are atomic, and σ is a most general unifier of X and Z. Then $\mathbf{R}^*\sigma$ is also a resolution expansion for S, where $\mathbf{R}^*$ is $\mathbf{R}$ with $[Y_1, \ldots, Y_m, W_1, \ldots, W_p]$ added.

As it happens, the system is *not* complete when only the binary version is used. You might try giving a proof of the valid sentence $(\forall x)(\forall y)[P(x) \vee P(y)] \supset (\exists x)(\exists y)[P(x) \wedge P(y)]$ to convince yourself of this. To restore completeness we also need the following.

Factoring Rule Suppose $\mathbf{R}$ is a resolution expansion for the set S of sentences of L, $\mathbf{R}$ contains the generalized disjunction $[X, Y, Z_1, \ldots, Z_n]$, where X and Y are literals, and σ is a most general unifier of X and Y. Then $\mathbf{R}\sigma$ is also a resolution expansion for S.

Now we have the basics set out, and we can make a few remarks about more specific implementation issues. In our tableau system implementation, we applied all Tableau Expansion Rules to a given Q-depth first, before testing for branch closure. The notion of Q-depth carries over directly to resolution, and similar ideas apply: Carry out all possible Resolution Expansion Rule applications to a given Q-depth first. Also these applications should be *strict*; remove any generalized disjunction to which a rule has been applied, unless it was a γ-rule; γ-rule applications should be fairly distributed in some manner. Finally, all applications of resolution, either Binary with Factoring, or General, can come last, and it is enough to apply them just to *clauses*. When we prove completeness in Section 7.8, our proof will apply to any implementation that meets these specifications.

Exercises

7.6.1. Give free-variable resolution proofs of the sentences in Exercise 7.4.1.

7.6.2. Give proofs using the General Literal Resolution Rule, and using the Binary Literal Resolution Rule and Factoring, of $(\forall x)(\forall y)[P(x) \vee P(y)] \supset (\exists x)(\exists y)[P(x) \wedge P(y)]$.

7.6.3$^{\mathbf{P}}$. Give an implementation of the resolution system in Prolog. You may use the tableau version in the previous section as a model, and you may use either General Resolution or Binary Resolution with Factoring.

7.7 Soundness

In the previous chapter we proved soundness of tableau and resolution systems in versions using constant parameters. Now we prove similar results for the systems of this chapter, which allow free variables and function parameters. The idea we followed before was to show that satisfiability of a tableau or resolution expansion is preserved whenever any of the rules is applied; in effect, that satisfiability is a loop invariant. This is still the idea, but we have the problem of how to treat free variables. Basically, we want to think of them as standing for anything—as if they were universally quantified. But the definition of satisfiability, Definition 5.3.6, treats them as if they were existentially quantified instead. To get around this, we introduce a variation on satisfiability, one that treats free variables as if they were universally quantified in some model.

For this section, L is a fixed first-order language. It is to sentences of L that we apply our theorem-proving methods.

Definition 7.7.1 Let $\mathbf{T}$ be a free-variable tableau.

1. Suppose $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ is a model for the language L^{sko} and $\mathbf{A}$ is an assignment in $\mathbf{M}$. We say $\mathbf{T}$ is $\langle \mathbf{M}, \mathbf{A} \rangle$*-satisfiable* provided there is some branch θ of $\mathbf{T}$ such that $X^{\mathbf{I},\mathbf{A}}$ is true for each formula X on θ.
2. We say $\mathbf{T}$ is $\forall$*-satisfiable* provided there is some model $\mathbf{M}$ such that $\mathbf{T}$ is $\langle \mathbf{M}, \mathbf{A} \rangle$-satisfiable for every assignment $\mathbf{A}$ in $\mathbf{M}$.

We can associate a closed formula with a free-variable tableau $\mathbf{T}$ in a simple way. First, think of each branch as the conjunction of its formulas. Next, think of the tree as the disjunction of its branches. Finally, take the universal closure of the result—that is, add enough universal quantifiers at the beginning of the formula to turn it into a closed formula. Call the result $X_{\mathbf{T}}$. The definition just given really amounts to this: $\mathbf{T}$ is $\forall$-satisfiable if and only if $X_{\mathbf{T}}$ is satisfiable in the usual sense. However, it is convenient to have a direct definition of the notion, avoiding the introduction of an auxiliary formula.

In a similar way a free-variable resolution expansion $\mathbf{R}$ can be converted into a closed formula—take the universal closure of the conjunction of its clauses, each of which is a disjunction. Satisfiability for the resulting formula is equivalent to the following directly defined notion.

Definition 7.7.2 Let $\mathbf{R}$ be a free-variable resolution expansion. We say $\mathbf{R}$ is $\forall$*-satisfiable* provided there is a model $\langle \mathbf{D}, \mathbf{I} \rangle$ such that, for each assignment $\mathbf{A}$ in the model, and for each generalized disjunction D in $\mathbf{R}$, $D^{\mathbf{I},\mathbf{A}}$ is true.

It is $\forall$-satisfiability that is a loop invariant for the construction of free-variable tableau or resolution proofs. This is the content of the following two lemmas:

Lemma 7.7.3

1. *If any propositional tableau expansion rule, or the Free-Variable γ- or δ-Rule, is applied to a $\forall$-satisfiable tableau, the result is another $\forall$-satisfiable tableau.*
2. *If any propositional resolution rule, or the Free-Variable γ- or δ-Rule, is applied to a $\forall$-satisfiable resolution expansion, the result is another $\forall$-satisfiable resolution expansion.*

Proof We give the argument for tableaux and leave resolution as an exercise. The propositional tableau rules are treated exactly as they were in Section 3.1, and the γ-rule is straightforward. We concentrate on the δ-rule.

Suppose θ is a branch of the tableau $\mathbf{T}$ and δ occurs on θ. Extend θ with $\delta(f(x_1,\ldots,x_n))$, producing θ', where f is a function symbol new to $\mathbf{T}$ and $x_1,\ldots, x_n$ are the free variables occurring in δ. Let $\mathbf{T}'$ be the tableau that results from $\mathbf{T}$ when θ is replaced with θ'. We show that if $\mathbf{T}$ is $\forall$-satisfiable, so is $\mathbf{T}'$.

Assume $\mathbf{T}$ is $\forall$-satisfiable. Then there is some model $\mathbf{M} = \langle\mathbf{D},\mathbf{I}\rangle$ such that $\mathbf{T}$ is $\langle\mathbf{M},\mathbf{A}\rangle$-satisfiable, for every assignment $\mathbf{A}$ in $\mathbf{M}$.

Given all this machinery, we define a *new* model $\mathbf{M}'$ as follows: The domain of $\mathbf{M}'$ is the same as that of $\mathbf{M}$, namely, $\mathbf{D}$. The interpretation $\mathbf{I}'$ of $\mathbf{M}'$ is the same as the interpretation $\mathbf{I}$ of $\mathbf{M}$ on all symbols except f. What is left is to specify a value for $f^{\mathbf{I}'}(d_1,\ldots,d_n)$, for each $d_1,\ldots, d_n \in \mathbf{D}$. We do this as follows: Let $\mathbf{A}$ be any assignment in $\mathbf{M}$ such that $x_1^{\mathbf{A}} = d_1,\ldots, x_n^{\mathbf{A}} = d_n$ (the behavior of $\mathbf{A}$ on variables not free in δ will not make a difference). If $\delta^{\mathbf{I},\mathbf{A}} = \mathbf{f}$, let $f^{\mathbf{I}'}(d_1,\ldots,d_n)$ be any arbitrarily chosen member of $\mathbf{D}$. On the other hand, suppose $\delta^{\mathbf{I},\mathbf{A}} = \mathbf{t}$. Speaking quite informally, since we are dealing with a δ-formula, which is existential in nature, if it is true in the model, there must be some member of the model domain that will serve as a true instance. To make this precise, let v be a new free variable. Then, since $\delta^{\mathbf{I},\mathbf{A}} = \mathbf{t}$, it must be that $\delta(v)^{\mathbf{I},\mathbf{B}} = \mathbf{t}$ for at least one $\mathbf{B}$ that is a v-variant of $\mathbf{A}$. Choose such an interpretation $\mathbf{B}$, and set $f^{\mathbf{I}'}(d_1,\ldots,d_n)$ to be $v^{\mathbf{B}}$.

Note that since the domains of $\mathbf{M}$ and $\mathbf{M}'$ are the same, an assignment in $\mathbf{M}$ is also an assignment in $\mathbf{M}'$. The point of the construction we just gave is to ensure the following easily checked fact: If $\delta^{\mathbf{I},\mathbf{A}}$ is true in the model $\mathbf{M}$, then $\delta(f(x_1,\ldots,x_n))^{\mathbf{I}',\mathbf{A}}$ is true in the model $\mathbf{M}'$.

Now, we will show that $\mathbf{T}'$ is $\langle \mathbf{M}', \mathbf{A}\rangle$ satisfiable for every assignment $\mathbf{A}$ in $\mathbf{M}'$, which will conclude the argument. Let $\mathbf{A}$ be some given assignment. The argument has two parts.

By assumption, all formulas on some branch of $\mathbf{T}$ are true in $\mathbf{M}$ under assignment $\mathbf{A}$. If that branch is not θ, then that branch is also a member of $\mathbf{T}'$, and its formulas are also true in $\mathbf{M}'$ under $\mathbf{A}$, since the two models differ only with respect to the symbol f, which was required to be new, hence, which can not occur on this branch. This is the easier half.

Now suppose all formulas on θ are true in $\mathbf{M}$ under assignment $\mathbf{A}$. Then all members of θ are also true in $\mathbf{M}'$ under $\mathbf{A}$, since, as earlier, the symbol f does not occur in $\mathbf{T}$ and hence does not occur in any formula of θ. But also, since δ is on θ, $\delta^{\mathbf{I},\mathbf{A}}$ is true in $\mathbf{M}$, so as noted, $\delta(f(x_1, \ldots, x_n)^{\mathbf{I}',\mathbf{A}}$ is true in the model $\mathbf{M}'$. Thus, all formulas on θ' are true in $\mathbf{M}'$ under $\mathbf{A}$.

This concludes the argument. □

Lemma 7.7.4

1. *If the Tableau Substitution Rule is applied to a $\forall$-satisfiable tableau, the result is another $\forall$-satisfiable tableau.*
2. *If the Resolution Substitution Rule is applied to a $\forall$-satisfiable resolution expansion, the result is another $\forall$-satisfiable resolution expansion.*

Proof Suppose the tableau $\mathbf{T}$ is $\forall$-satisfiable. Then for some model $\mathbf{M}$, $\mathbf{T}$ is $\langle \mathbf{M}, \mathbf{A}\rangle$-satisfiable for every assignment $\mathbf{A}$. We show the same is true for $\mathbf{T}\sigma$, where σ is free for $\mathbf{T}$.

Let $\mathbf{A}$ be an assignment in $\mathbf{M}$—we show $\mathbf{T}\sigma$ is $\langle \mathbf{M}, \mathbf{A}\rangle$-satisfiable. Define a new assignment $\mathbf{B}$ by setting, for each variable v, $v^{\mathbf{B}} = (v\sigma)^{\mathbf{I},\mathbf{A}}$. Since $\mathbf{B}$ is also an assignment in $\mathbf{M}$, there must be some branch θ of $\mathbf{T}$ whose formulas are all true in $\mathbf{M}$ under assignment $\mathbf{B}$. But now, using Proposition 5.3.8, it follows that for each formula X occurring in the tableau $\mathbf{T}$, $[X\sigma]^{\mathbf{I},\mathbf{A}} = X^{\mathbf{I},\mathbf{B}}$. Consequently, all members of branch $\theta\sigma$ of tableau $\mathbf{T}\sigma$ must be true in $\mathbf{M}$ under assignment $\mathbf{A}$.

This concludes the proof for tableaus. The resolution argument is similar. □

Theorem 7.7.5 **(Soundness)**

1. *If the sentence Φ has a free-variable tableau proof, Φ is valid.*
2. *If the sentence Φ has a free-variable resolution proof, Φ is valid.*

Proof As usual, suppose Φ has a tableau proof but is not valid. Since Φ is not valid, $\{\neg\Phi\}$ is satisfiable. For sentences, satisfiability and ground-satisfiability coincide, and so the tableau proof of Φ begins with a tableau that is ground-satisfiable. Then every subsequent tableau must be ground-satisfiable including the final closed one, which is impossible. The resolution argument is similar. □

Exercises

7.7.1. Prove part 2 of Lemma 7.7.3.

7.7.2. Prove part 2 of Lemma 7.7.4.

7.8 Free-Variable Tableau Completeness

Proving completeness of the free-variable tableau system is not difficult. The Model Existence Theorem 5.8.2 can be used. But we need something stronger than mere completeness. We need to show that the free-variable tableau system remains complete even when various restrictions are placed on allowed proofs. This will let us show completeness not just of the tableau system itself but of a family of implementations, including the one presented in Section 7.5.

The Tableau Substitution Rule allows us to make a free substitution at any time. This means we could devise an implementation in which Tableau Expansion Rule applications are mingled with substitutions and with tests for branch closure. There is a wide variety of possibilities. But two practical issues stand out: How do we decide what substitutions to make? And how do we test whether a substitution is free?

The problem of what substitutions to make has an obvious solution: Make those that will cause branches to close. We can write an implementation so that it selects two formulas on the same branch, A and $\neg B$, and produces a free substitution that unifies them, thus closing the branch. Since these formulas are not required to be literals, we may in this way be able to produce short proofs, proofs that do not apply tableau expansion rules to the point where every formula gets broken down. A notable inefficiency is hidden here, however. Suppose a tableau $\mathbf{T}$ has a branch θ containing A and $\neg B$, and σ is a free substitution that unifies A and B. If we apply σ, we may get a tableau $\mathbf{T}\sigma$ with a closed branch but in which no other branches can be closed. Applying σ may lead us to a dead end in our proof attempt. But there may be two other formulas C and $\neg D$ on θ, where C and D have a unifying substitution τ that is also free, so that $\mathbf{T}\tau$ does admit further work that will close its remaining branches. We see that it is not enough to select one pair of formulas on a branch and use them to develop a branch-closing substitution—we must be prepared to try *each* pair. And for each one of these pairs we try, we must be prepared to try each pair on the next branch, and so on. Clearly, this leads to exponentially explosive time expenditure. To make things worse, testing whether a substitution is free for a tableau is also expensive. We must, after all, verify whether it is free for every formula in the tableau, and this might be a very large number.

Part of the strategy in designing a reasonable general purpose theorem prover is to avoid, as far as possible, the kind of problems we have raised. In our implementation of Section 7.5, we only tested for branch closure at the *atomic* level by applying the MGU Atomic Closure Rule. This means we might miss the chance to close a branch early, before applying unnecessary tableau expansion rules. But it also means we can avoid the test for freeness of a substitution.

One of the restrictions under which we will prove tableau completeness, then, is that all substitutions are those required by the MGU Atomic Closure Rule. Be aware, though, that such a restriction is not built into the Tableau Substitution Rule. Other kinds of substitutions may be reasonable under other circumstances—given additional information about what is being proved, or given some human assistance with proofs, say. The restrictions we consider here are not the only ones that are possible or that are reasonable under all circumstances.

In the following we make use of the notion of *most general solution* of a family of *simultaneous equations*. This was defined in Section 7.2.

Definition 7.8.1 Suppose $\mathbf{T}$ is a tableau with branches $\theta_0, \theta_1, \ldots, \theta_n$, and for each i, A_i and $\neg B_i$ are a pair of literals on branch θ_i. If σ is a most general solution of the family of equations $A_0 = B_0$, $A_1 = B_1, \ldots, A_n = B_n$, we call σ a *most general atomic closure substitution.*

The discussion of concurrent unification in Section 7.2 shows that finding a solution to a family of equations reduces to repeated applications of ordinary binary unification. In the present case, a most general atomic closure substitution can be found (if one exists) by repeated applications of the MGU Atomic Closure Rule. So, if we prove tableau completeness under the restriction that all branch closures and substitutions must be those arising from a most general atomic closure substitution, we will also have completeness under a similar restriction to MGU Atomic Closure Rule applications.

The restriction to *most general* is not as strong as it seems, as the following shows. We call it a *Lifting Lemma* because it plays the role, for tableaux, that the well-known Lifting Lemma 7.9.2 plays for resolution.

Lemma 7.8.2 **(Lifting Lemma)** *Suppose* $\mathbf{T}$ *is a tableau, and* τ *is a substitution, free for* $\mathbf{T}$*, such that each branch of* $\mathbf{T}\tau$ *is* atomically *closed. Then there is a most general atomic closure substitution* σ *for* $\mathbf{T}$.

Proof Let θ be a branch of $\mathbf{T}$. Since $\mathbf{T}\tau$ is atomically closed, there must be formulas $P(t_1, \ldots, t_n)$ and $\neg P(u_1, \ldots, u_n)$ on θ such that $t_1\tau = u_1\tau, \ldots, t_n\tau = u_n\tau$. Associate the equations $t_1 = u_1, \ldots, t_n = u_n$ with the branch θ. Now, let $\mathcal{E}$ be the set of equations associated with any branch of $\mathbf{T}$. The substitution τ is a simultaneous solution of $\mathcal{E}$. By our discussion of concurrent unification in Section 7.2, a most general solution σ for $\mathcal{E}$ must exist. It is easy to see that σ is also a most general atomic closure substitution for $\mathbf{T}$. □

Next we turn to the problem of which tableau expansion rules to apply, when, and how. The rules are non-deterministic. We have free choice of what to do next. In any actual implementation, some algorithm must specify these choices. We need to know which algorithms will allow us to discover proofs and which will not. Keep in mind that special knowledge about the sentence being proved may suggest which tableau expansion rules are best to try. What we are interested in here is what must work, given no special knowledge. In the propositional case, a *strictness* assumption was imposed: Each formula could be used only once on a branch. That will still be the case, except for γ-formulas. On the other hand, with strictness imposed, propositional tableau constructions must terminate. In the first-order case, with single use not required of γ-formulas, termination may not always happen, and so *fairness* becomes an issue. We want to be sure every formula occurrence has its chance at having a rule applied to it. These issues are formalized in the notion of *fair tableau construction rule*, to be given shortly.

First, we have a minor issue to deal with. The Free-Variable γ-Rule allows us to introduce any free variable. Let x_1, x_2,... be an enumeration of all the variables of L that do not occur in the sentence we are attempting to prove; fix this list once and for all. For the rest of this section, we assume that when we work with the formula γ on a branch we simply add $\gamma(x_i)$ where x_i is the *first* free variable in this enumeration such that $\gamma(x_i)$ does not already appear on the branch. This makes γ-rule applications deterministic in a uniform way.

In the tableau implementation of Section 7.5, branches were rearranged from time to time, as a sort of priority queue, and formulas were deleted as well. Although this was useful for implementation purposes, it is not convenient now. But we can easily rephrase a description of what we are doing so that only the usual tableau rules are applied, but we remember as *side information* which formula occurrences should be treated as if they had been deleted, and in what order we should try formulas, and on which branches. We do not make precise this notion of side information. Intuition will serve quite well for now.

Definition 7.8.3 A *tableau construction rule* is a rule $\mathcal{R}$ that, when supplied with a tableau $\mathbf{T}$ and some side information, either 1 says no continuation is possible or 2 produces a new tableau $\mathbf{T}'$ and new side information, where $\mathbf{T}'$ results from $\mathbf{T}$ by the application of a single Tableau Expansion Rule to $\mathbf{T}$. We also assume that $\mathcal{R}$ specifies outright the appropriate side information to accompany the initial tableau of an attempt to prove the sentence X.

Example The following is a tableau construction rule, $\mathcal{R}$. For each tableau, side information consists of which formula occurrences have been used, on which branches; a priority ordering for formula occurrences on each branch; and a priority ordering for branches. The side information associated with an initial tableau is as follows: The sentence $\neg X$ is not used; the sentence has top priority on its branch; the single branch has top priority among branches. (You might have guessed this.)

Next, given a tableau $\mathbf{T}$ and side information about it, $\mathcal{R}$ says the following: If every non-literal formula occurrence has been used on every branch, no continuation is possible. Otherwise, select the branch θ of highest priority on which some non-literal is unused, and on it select the unused non-literal Z of highest priority. Apply the appropriate tableau expansion rule to Z on θ, producing the new tableau $\mathbf{T}'$. For the side information associated with $\mathbf{T}'$, add to the side information of $\mathbf{T}$ that the occurrence of Z on θ is now used. Give the lengthened branch (or branches) of $\mathbf{T}'$ the lowest priority, and on these branches, give the added formula occurrences lowest priorities.

Incidentally, this rule treats γ-formulas like all others; no reuse is allowed.

Definition 7.8.4 Let Φ be a sentence of L. We say a sequence of tableaux (and associated side information) $\mathbf{T}_1$, $\mathbf{T}_2$, $\mathbf{T}_3$,... (finite or infinite) is a *sequence for* Φ *constructed according to rule* $\mathcal{R}$ provided $\mathbf{T}_1$ is the tableau with one branch, that branch containing only Φ, and with the side information associated with it by $\mathcal{R}$, and provided each $\mathbf{T}_{i+1}$ and its side information result from $\mathbf{T}_i$ and its side information by an application of rule $\mathcal{R}$.

Some tableau construction rules are pretty dull. We might always select the same formula occurrence to work with over and over, for instance. This is clearly not fair to the other formulas.

Definition 7.8.5 A tableau construction rule $\mathcal{R}$ is *fair* provided, for any sentence Φ, the sequence $\mathbf{T}_1$, $\mathbf{T}_2$,..., of tableaux for Φ constructed according to $\mathcal{R}$ has the following properties, for each n:

1. Every non-literal formula occurrence in $\mathbf{T}_n$ eventually has the appropriate Tableau Expansion Rule applied to it, on each branch on which it occurs.

2. Every γ-formula occurrence in $\mathbf{T}_n$ has the γ-rule applied to it arbitrarily often, on each branch on which it occurs.

The tableau construction rule used in the implementation of Section 7.5 is fair; we leave it to you to convince yourself of this. The tableau construction rule used in the example just given is not fair, as γ-formulas only are worked with once.

Now we are ready to state the main result of this section: Not only is the free-variable tableau system complete, but it is complete when rather severely restricted.

Theorem 7.8.6 **(Completeness)** *Let $\mathcal{R}$ be any fair tableau construction rule. If X is a valid sentence of L, X has a proof in which the following apply:*

1. *All Tableau Expansion Rule applications come first and are according to rule $\mathcal{R}$.*
2. *A single Tableau Substitution Rule application follows, using a substitution σ that is a most general atomic closure substitution.*

Proof Let $\mathbf{T}_1$, $\mathbf{T}_2$, $\mathbf{T}_3$,... be the sequence of tableaux for $\neg X$ constructed according to rule $\mathcal{R}$, and suppose that no $\mathbf{T}_n$ has a most general atomic closure substitution. We show X is not valid.

We assume the sequence of tableaux is infinite. If it is finite, the argument is similar, but simpler. Since the sequence is infinite, and tree $\mathbf{T}_{n+1}$ extends tree $\mathbf{T}_n$ in an obvious way, we can think of what we are doing as constructing a sequence of approximations to an *infinite* tree. Let us call this tree $\mathbf{T}$.

Recall that we have fixed an ordering of all free variables, $x_1, x_2, \ldots$, and in applying the γ-rule, we said we would always use the first variable in this list that we had not used yet, with that formula, on that branch. Now we also assume we have a fixed enumeration $t_1, t_2, \ldots$, of all closed terms of the language $L^{\mathbf{par}}$. Let σ be the substitution given by $x_i\sigma = t_i$ for all i. σ Is free for $\mathbf{T}$, since only closed terms are substituted. Incidentally, this also means that in $\mathbf{T}\sigma$ only *sentences* appear.

We claim $\mathbf{T}\sigma$ is *not* atomically closed. The argument goes as follows: Suppose $\mathbf{T}\sigma$ were atomically closed. In $\mathbf{T}\sigma$, prune each branch by removing all sentences below an atomic contradiction, that is, all those below $\bot$, or below both of A and $\neg A$, where A is atomic. Call the resulting tree $(\mathbf{T}\sigma)^*$. In $(\mathbf{T}\sigma)^*$ every branch is finite, so by König's Lemma 2.7.2 $(\mathbf{T}\sigma)^*$ itself is finite. Then, for some n, $(\mathbf{T}\sigma)^*$ must be a subtree of $\mathbf{T}_n\sigma$. It follows that $\mathbf{T}_n\sigma$ itself is atomically closed, and so by Lemma 7.8.2, there is a most general atomic closure substitution for $\mathbf{T}_n$. Since this is not the case, $\mathbf{T}\sigma$ can not be atomically closed.

Since $\mathbf{T}\sigma$ is not atomically closed, it must have a branch $\theta\sigma$, without an atomic contradiction, where θ is some branch of the infinite tree $\mathbf{T}$. $\mathcal{R}$ is a fair tableau construction rule, and so if α occurs on θ, so will α_1 and α_2, and similarly for other formulas. Likewise, fair tableau construction rules return to γ-formulas arbitrarily often, so if γ occurs on θ, so will each of $\gamma(x_1)$, $\gamma(x_2)$,.... (This makes use of our special way of selecting which free variables to use in γ-rule applications.) It follows that the set of sentences on branch $\theta\sigma$ of $\mathbf{T}\sigma$ is a first-order Hintikka set (with respect to the language $L^{\mathbf{par}}$) and so is satisfiable by Hintikka's Lemma 5.6.2. $\neg X$ occurs on θ (indeed, on every branch of $\mathbf{T}$) and so $(\neg X)\sigma$ occurs on $\theta\sigma$. Since X is a *sentence* of L, $\neg X$ is not affected by σ, so $\neg X$ itself is on $\theta\sigma$; $\neg X$ is satisfiable; X is not valid. □

Exercises

7.8.1. Give a proof of the completeness of the free-variable tableau system without any restrictions, using the Model Existence Theorem.

7.8.2. Give a proper statement of the tableau construction rule used in the tableau implementation of Section 7.5.

7.8.3. State two additional fair tableau construction rules.

7.9 Free-Variable Resolution Completeness

The free-variable resolution system presented in Section 7.6 is very general. It allows for free substitutions at any time, and resolution on any formulas, not necessarily literals. Resolution, traditionally, has worked with clauses only, so we are allowing a kind of *non-clausal* resolution. Nonetheless, writing an implementation that can make adequate use of this freedom is a complicated matter. When we suggested you write an implementation of resolution, we outlined restrictions that, in effect, eliminated this non-clausal aspect. In this section we prove resolution completeness when such restrictions are imposed. This will ensure that your implementation is complete, provided it meets our specifications. Completeness of other implementations, based on different restrictions, is a different matter altogether. We confine ourselves here to restrictions that produce an equivalent of traditional resolution. The completeness proof we present is a cross between the completeness proof we gave earlier for propositional resolution, and the first-order tableau completeness proof of the previous section. It would be useful to go back and review Section 3.8.

The restrictions we impose on free-variable resolution are in the same spirit as those we imposed on tableaux: Apply all expansion rules first; make only those substitutions that will enable an application of the resolution rule; only resolve on literals. Nonetheless, details differ considerably from the tableau completeness proof.

As we noted in Section 7.6, we have a choice of adopting the General Literal Resolution Rule or else both the Binary Literal Resolution Rule and the Factoring Rule. Any application of the General Literal Resolution Rule can be replaced by applications of Factoring, followed by an application of the Binary Resolution Rule. So it is enough to show completeness using only the General Resolution Rule. Further, we will restrict its application to clauses only. (As originally stated, only the formulas being resolved on had to be literals.)

The Resolution Substitution Rule allowed us to make *any* free substitution. We now restrict ourselves to special ones, to most general unifiers. It is important to know this is no real restriction, at least when applied to clauses. The formal statement of this is called the *Lifting Lemma.* Before giving it we introduce some useful, but temporary, terminology and notation.

Definition 7.9.1 Let S be a set of clauses. By a *G-derivation from S*, we mean a resolution derivation from S in which only the General Literal Resolution Rule is used. We write $S \longrightarrow_n [\,]$ if there is a G-derivation from S of $[\,]$ with exactly n applications of General Literal Resolution.

Actually, we will want to apply trivial resolutions in derivations as well (removing $\bot$ from a clause). We have ignored this in the definition of G-derivation, for simplicity. It is not hard to see that if we allow trivial resolutions in G-derivations we can always arrange to do all of them first, so that the tail end of such a derivation will be a G-derivation in the narrower sense. Thus, nothing essential is lost by our concentration on nontrivial resolution steps. We continue this policy, ignoring trivial resolutions.

Since we are confining our attention to clause sets for now, only literals are present, and all substitutions are automatically free. This makes things a little simpler for us.

Theorem 7.9.2 **(Lifting Lemma)** *For every clause set S and every substitution ν, if $S\nu \longrightarrow_n [\,]$, then $S \longrightarrow_n [\,]$.*

Proof The argument is by induction on n. The initial case is trivial; if $n = 0$ and $S\nu \longrightarrow_n [\,]$, $S\nu$ itself must contain the empty clause, in which case so does S.

Now suppose the result is known for n. That is, for any S and any ν, if $S\nu \longrightarrow_n [\,]$, then $S \longrightarrow_n [\,]$. We show the result holds for $n + 1$ as well.

Suppose $S\nu \longrightarrow_{n+1} [\,]$; we show $S \longrightarrow_{n+1} [\,]$. We are given that there is a G-derivation of $[\,]$ from $S\nu$ with $n+1$ General Literal Resolution Rule applications. Consider the first such application in this G-derivation. There must be clauses $[A_1, \ldots, A_i, Y_1, \ldots, Y_j]$ and $[\neg B_1, \ldots \neg B_k, W_1, \ldots, W_p]$ in S, and a substitution σ that is a most general unifier for the set $\{A_1\nu, \ldots, A_i\nu,\ B_1\nu, \ldots,\ B_k\nu\}$, so that $[Y_1\nu, \ldots, Y_j\nu, W_1\nu, \ldots,\ W_p\nu]$ is added and σ is applied, giving $(S\nu)\sigma \cup \{[Y_1\nu, \ldots, Y_j\nu, W_1\nu, \ldots,\ W_p\nu]\sigma\}$, which we denote S'. Clearly $S' \longrightarrow_n [\,]$.

$\nu\sigma$ Is a unifier for the set $\{A_1, \ldots, A_i, B_1, \ldots, B_k\}$, so this set has a most general unifier, say η. Since η is more general than $\nu\sigma$, there must be a substitution μ such that $\eta\mu = \nu\sigma$.

Let $R = S\eta \cup \{[Y_1\eta, \ldots, Y_j\eta, W_1\eta, \ldots, W_p\eta]\}$. Since η is a most general unifier of $\{A_1, \ldots, A_i, B_1, \ldots, B_p\}$, and both of $[A_1, \ldots, A_i, Y_1, \ldots Y_j]$ and $[\neg B_1, \ldots, \neg B_k, W_1, \ldots, W_p]$ are in S, we can turn S into R by a single application of the General Literal Resolution Rule. So, to show $S \longrightarrow_{n+1} [\,]$ it is enough to show $R \longrightarrow_n [\,]$.

$$\begin{aligned}
R\mu &= (S\eta \cup \{[Y_1\eta, \ldots, Y_j\eta, W_1\eta, \ldots, W_p\eta]\})\mu \\
&= (S \cup \{[Y_1, \ldots, Y_j, W_1, \ldots, W_p]\})\eta\mu \\
&= (S \cup \{[Y_1, \ldots, Y_j, W_1, \ldots, W_p]\})\nu\sigma \\
&= (S\nu)\sigma \cup \{[Y_1\nu, \ldots, Y_j\nu, W_1\nu, \ldots, W_p\nu]\sigma\} \\
&= S'
\end{aligned}$$

Since $S' \longrightarrow_n [\,]$, $R\mu \longrightarrow_n [\,]$, so by the induction hypothesis, $R \longrightarrow_n [\,]$, and we are done. □

Definition 7.9.3 We write $S \longrightarrow [\,]$ if $S \longrightarrow_n [\,]$ for some n.

Suppose S is a set of clauses that does not contain any free variables. Then the internal structure of the atomic formulas present is not really relevant, and we can think of them as if they were propositional letters. This means all notions of propositional logic can be applied. In particular we need *Boolean valuations*, which now become mappings from closed atomic formulas to $\mathbf{Tr}$. Also, propositional satisfiability notions make sense, in addition to first-order ones. (See Exercise 7.9.1.)

Definition 7.9.4 We say a substitution τ is *grounding* if, for each variable x, $x\tau$ is a closed term of $L^{\mathbf{sko}}$.

Corollary 7.9.5 *Suppose* $\mathbf{C}$ *is a (possibly infinite) set of clauses,* σ *is a grounding substitution, and* $\mathbf{C}\sigma$ *is propositionally unsatisfiable. Then there is some finite subset* S *of* $\mathbf{C}$ *such that* $S \longrightarrow [\,]$.

Proof Extend $\mathbf{C}\sigma$ to a resolution saturated set as in Definition 3.8.3; call the result $\mathbf{D}$. Since $\mathbf{D}$ has a propositionally unsatisfiable subset $\mathbf{C}\sigma$, $\mathbf{D}$ itself is propositionally unsatisfiable and so must contain the empty clause by Proposition 3.8.4. Since $[\,]$ is in $\mathbf{D}$, $[\,]$ must be obtainable from some of the members of $\mathbf{C}\sigma$ using the Propositional Resolution Rule. Since no free variables are present, Propositional Resolution and General Literal Resolution Rule are essentially identical. Also, a derivation of $[\,]$ will use only a finite number of members of $\mathbf{C}\sigma$. It follows that $S\sigma \longrightarrow [\,]$, where S is some finite subset of $\mathbf{C}$. Then $S \longrightarrow [\,]$ by the Lifting Lemma. □

Corollary 7.9.5 contains the essence of resolution completeness for pure clauses. Now we extend it to take more complicated formulas into account. The following is an obvious transfer from our tableau discussion:

Definition 7.9.6 A *resolution construction rule* is a rule $\mathcal{R}$ that, when supplied with a resolution expansion $\mathbf{R}$ and some side information, either (1) says no continuation is possible or (2) produces a new resolution expansion $\mathbf{R}'$ and new side information, where $\mathbf{R}'$ results from $\mathbf{R}$ by the application of a single Resolution Expansion Rule to $\mathbf{R}$. We also assume that $\mathcal{R}$ specifies outright the appropriate side information to accompany the initial resolution expansion of an attempt to prove the sentence X.

The free-variable γ-rule for resolution, like that for tableaux, allows us to introduce any free variable. Let x_1, x_2,... be a list of all the variables of L that do not occur in the sentence we are attempting to prove. From now on we assume that when we work with the formula γ we simply add $\gamma(x_i)$ where x_i is the *first* free variable in the list that has not yet been used with this γ formula occurrence.

Definition 7.9.7 Let Φ be a sentence of L. We say a sequence of resolution expansions (and associated side information) $\mathbf{R}_1$, $\mathbf{R}_2$, $\mathbf{R}_3$,... (finite or infinite) is a *sequence for Φ constructed according to rule $\mathcal{R}$* provided $\mathbf{R}_1$ is the initial resolution expansion for Φ with the side information associated with it by $\mathcal{R}$, and each $\mathbf{R}_{i+1}$ and its side information results from $\mathbf{R}_i$ and its side information by an application of rule $\mathcal{R}$.

Definition 7.9.8 A resolution construction rule $\mathcal{R}$ is *fair* provided, for any sentence Φ, the sequence $\mathbf{R}_1$, $\mathbf{R}_2$,..., of resolution expansions for Φ constructed according to $\mathcal{R}$ has the following properties, for each n:

1. Every disjunction in $\mathbf{R}_n$ containing a non-literal eventually has some resolution expansion rule applied to it.
2. If the γ-rule is applied to a disjunction in $\mathbf{R}_n$, then it is applied to that disjunction arbitrarily often.

Theorem 7.9.9 **(Completeness)** *Let $\mathcal{R}$ be any fair resolution construction rule. If X is a valid sentence of L, X has a proof in which*

1. *All Resolution Expansion Rule applications come first and are according to rule $\mathcal{R}$.*
2. *These are followed by applications of the General Literal Resolution Rule, to clauses only.*

Proof Let $\mathbf{R}_1, \mathbf{R}_2, \ldots$ be the sequence of resolution expansions for $\{\neg X\}$ constructed according to rule $\mathcal{R}$ and suppose that for each n we do not have $S \longrightarrow [\,]$ where S is the set of clauses from $\mathbf{R}_n$. We show X is not valid.

Recall, $x_1, x_2, \ldots$ was our fixed listing of free variables. Let $t_1, t_2, \ldots$ be a listing of all closed terms of the language L, and let σ be the substitution given by $x_i\sigma = t_i$.

Let $\mathbf{C}$ be the set consisting of those clauses that appear in any $\mathbf{R}_n$. If $\mathbf{C}\sigma$ were propositionally unsatisfiable, by Corollary 7.9.5 we would have $S \longrightarrow [\,]$ where S is some finite subset of $\mathbf{C}$. But since S is finite, it would be a subset of some $\mathbf{R}_n$ as well, which contradicts our assumption. Consequently, $\mathbf{C}\sigma$ is propositionally satisfiable, say using the Boolean valuation v.

Now we construct a Herbrand model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ as follows: The domain $\mathbf{D}$ must be the set of closed terms of $L^{\mathbf{par}}$, and the interpretation $\mathbf{I}$ is standard on constant and function symbols. For a relation symbol, we set $R^{\mathbf{I}}(t_1, \ldots, t_n) = \mathbf{t}$ just in case $v(R(t_1, \ldots, t_n)) = \mathbf{t}$. This completes the definition of our model.

In the model $\mathbf{M}$, every member of $\mathbf{C}\sigma$ is true. This observation can be restated in a suggestive way. In Definition 2.6.5 we defined a notion of rank for propositional formulas. That is easily extended to include quantifiers: The rank of a γ- or a δ-formula is one more than the rank of any (equivalently all) of its instances. Rank extends to a generalized disjunction by taking it to be the sum of the formulas making up the disjunction. Clauses have rank 0, so what we have established is the following: If D is a disjunction in some $\mathbf{R}_n$ of rank 0, $D\sigma$ is true in $\mathbf{M}$.

We leave it to you to prove, by induction on rank, that for every generalized disjunction D in any $\mathbf{R}_n$, $D\sigma$ will be true in $\mathbf{M}$. Since $\mathbf{R}_1$ contains only $[\neg X]$, $\neg X\sigma$ must be true in $\mathbf{M}$. But X is a *sentence* of L, hence unaffected by σ. Consequently, $\neg X$ is true in $\mathbf{M}$, X is not, and so X is not valid. $\square$

This completeness proof applies to a number of different implementations, since any fair resolution construction rule will do. Clearly, some will be more efficient than others. For instance, it is not hard to see that it is best to postpone γ-rule applications as long as possible, but it is best to reduce double negations early. If you wrote a Prolog implementation, you might use it to experiment with different priorities for rule applications.

Exercises

7.9.1. Let $\mathbf{C}$ be a set of clauses containing no free variables. If we identify closed atomic formulas with the propositional letters of Chapter 2, then the notion of Boolean valuation can be applied, as well as the notion of first-order model. Show $\mathbf{C}$ is satisfiable in the propositional sense (some Boolean valuation maps all its members to $\mathbf{t}$) if and only if $\mathbf{C}$ is satisfiable in the first-order sense (there is a model in which its members are true).

7.9.2. Complete the proof of Theorem 7.9.9 by doing the induction.

8 Further First-Order Features

8.1 Introduction

Over the years many fundamental and important results have been established for classical first-order logic. This chapter is devoted to what are probably the most basic of these, ranging from Herbrand's theorem to Beth's definability theorem. The Model Existence Theorem plays a role in establishing many of the results of this chapter, thus proving its versatility. It has one big drawback, though—proofs that use it are non-constructive in nature. For some of the things we are interested in, such as Herbrand's theorem, it is important to have a constructive proof as well, and this leads us to the beginnings of *proof theory*, supplementing the semantic methods that we initially rely on.

8.2 The Replacement Theorem

It is often useful to rewrite a sentence before trying to prove it, to avoid time-consuming work during the course of the proof. We do this in the next section. The machinery that allows this is a first-order version of the Replacement Theorem 2.5.1 and 2.5.2. Fortunately, the propositional version extends to first-order logic with no serious complications. To state the theorem more easily, we use the following notation: Suppose A is an atomic formula. We write $\Phi(A)$ to denote a first-order formula in which occurrences of A play a significant role. Then if F is a first-order formula, we write $\Phi(F)$ to denote the formula that is like $\Phi(A)$ except that all occurrences of A have been replaced by occurrences of the formula F.

Theorem 8.2.1 **(Replacement Theorem)** *Let $\Phi(A)$, X, and Y be first-order formulas of the language L. Let $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ be a model for L. If $X \equiv Y$ is true in $\mathbf{M}$, then $\Phi(X) \equiv \Phi(Y)$ is also true in $\mathbf{M}$.*

Proof Suppose $X \equiv Y$ is true in $\mathbf{M}$. Then for every assignment $\mathbf{A}$, $X^{\mathbf{I},\mathbf{A}} = Y^{\mathbf{I},\mathbf{A}}$, by Exercise 5.3.4. We must show that for every assignment $\mathbf{A}$, $[\Phi(X)]^{\mathbf{I},\mathbf{A}} = [\Phi(Y)]^{\mathbf{I},\mathbf{A}}$. This is done by structural induction on $\Phi(A)$. The atomic and propositional cases are treated essentially as they were in the proof of Theorem 2.5.1; we do not repeat the arguments. There are two quantifier cases; we consider only the universal one, where $\Phi(A) = (\forall y)\Psi(A)$.

Induction hypothesis: $[\Psi(X)]^{\mathbf{I},\mathbf{A}} = [\Psi(Y)]^{\mathbf{I},\mathbf{A}}$ for all assignments $\mathbf{A}$.

To be shown: $[(\forall y)\Psi(X)]^{\mathbf{I},\mathbf{B}} = [(\forall y)\Psi(Y)]^{\mathbf{I},\mathbf{B}}$ for all assignments $\mathbf{B}$.

Well, let $\mathbf{B}$ be an assignment. $[(\forall y)\Psi(X)]^{\mathbf{I},\mathbf{B}} = \mathbf{t} \Leftrightarrow$ for every y-variant $\mathbf{A}$ of $\mathbf{B}$, $[\Psi(X)]^{\mathbf{I},\mathbf{A}} = \mathbf{t} \Leftrightarrow$ for every y-variant $\mathbf{A}$ of $\mathbf{B}$, $[\Psi(Y)]^{\mathbf{I},\mathbf{A}} = \mathbf{t}$ (this used the induction hypothesis) $\Leftrightarrow [(\forall y)\Psi(Y)]^{\mathbf{I},\mathbf{B}} = \mathbf{t}$. □

Corollary 8.2.2 *If $X \equiv Y$ is valid, so is $\Phi(X) \equiv \Phi(Y)$.*

The Replacement Theorem concerns logical equivalence. Sometimes we will need something stronger, in which implication plays a role. In this version we will not be able to replace all occurrences of a formula but only those that occur *positively*. Loosely speaking, a subformula occurrence in Φ is positive if it would be within the scope of an even number of negation signs when Φ is rewritten using only $\wedge$ and $\neg$ as propositional connectives. Our formal definition is along different lines, however.

Definition 8.2.3 As usual we write $\Phi(A)$ to indicate a formula in which occurrences of the atomic formula A will play a role. We say that all occurrences of A in $\Phi(A)$ are *positive* provided:

1. $\Phi(A) = A$.
2. $\Phi(A) = \neg\neg\Psi(A)$ and the occurrences of A in $\Psi(A)$ are positive.
3. $\Phi(A)$ is an α-formula and the occurrences of A in α_1 and in α_2 are positive.
4. $\Phi(A)$ is a β-formula and the occurrences of A in β_1 and in β_2 are positive.
5. $\Phi(A)$ is a γ-formula, with x being the variable quantified, and the occurrences of A in $\gamma(x)$ are positive.

6. $\Phi(A)$ is a δ-formula, with x being the variable quantified, and the occurrences of A in $\delta(x)$ are positive.

Example Consider the formula $(\forall x)[P(x,y) \supset \neg(\exists y)\neg R(x,y)]$. This is a γ-formula, so the occurrence of $R(x,y)$ will be positive here, provided it is positive in $P(x,y) \supset \neg(\exists y)\neg R(x,y)$. This is a β-formula; the occurrence of $R(x,y)$ will be positive, provided it is positive in $\beta_2 = \neg(\exists y)\neg R(x,y)$. This is also a γ-formula; $R(x,y)$ will be positive, provided it is positive in $\gamma(y) = \neg\neg R(x,y)$. $R(x,y)$ is positive here, provided it is positive in $R(x,y)$, and it is.

We leave it to you to check that the occurrence of $P(x,y)$ is not positive.

We will often be somewhat informal when we apply this definition. For instance, we might say that the second occurrence of $P(x) \wedge Q(x)$ in $(\forall x)(P(x) \wedge Q(x)) \supset (\exists x)(P(x) \wedge Q(x))$ is positive. What is meant is pretty clear; if we let $\Phi(A)$ be the sentence $(\forall x)(P(x) \wedge Q(x)) \supset (\exists x)A$, the only occurrence of A here is positive, and the sentence we began with is $\Phi(P(x) \wedge Q(x))$.

Theorem 8.2.4 **(Implicational Replacement Theorem)** *Let $\Phi(A)$ be a formula in which the atomic formula A has only positive occurrences. Let X and Y be first-order formulas of the language L, and let $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ be a model for L. If $X \supset Y$ is true in $\mathbf{M}$ then $\Phi(X) \supset \Phi(Y)$ is also true in $\mathbf{M}$.*

Definition 8.2.5 A has only *negative* occurrences in $\Phi(A)$, provided A has only positive occurrences in $\neg\Phi(A)$.

Corollary 8.2.6 *Suppose all occurrences of A in $\Phi(A)$ are negative. If $Y \supset X$ is true in the model $\mathbf{M}$, so is $\Phi(X) \supset \Phi(Y)$.*

Exercises

8.2.1. Prove Theorem 8.2.4 by structural induction.

8.2.2$^{\mathbf{P}}$. Write a Prolog program to determine whether all occurrences of A in $\Phi(A)$ are positive.

8.2.3. The notion of all occurrences of A in $\Phi(A)$ being negative can be also defined directly, by paralleling Definition 8.2.3. Item 1 is replaced by the following: all occurrences of A in $\Phi(A)$ are negative, provided $\Phi(A) = \neg A$, and in the other items the word *positive* is replaced with the word *negative*. Show this definition is equivalent to the one in Definition 8.2.5.

8.3 Skolemization

In the course of a free-variable tableau or resolution proof, function parameters are introduced whenever the δ-rule is applied. It is possible to preprocess a sentence so that these function symbols are all introduced ahead of time. If we do this, the δ-rule will never be needed in a proof, and the theorem-proving machinery can be simplified and perhaps made more efficient. The introduction of these function symbols is called *Skolemization.* We describe the process and prove it does not affect the satisfiability of sentences. It is convenient to continue the notational conventions of the previous section: $\Phi(A)$ is a formula with occurrences of the atomic subformula A playing a special role, and $\Phi(X)$ is the result of replacing all occurrences of A with occurrences of the formula X.

Lemma 8.3.1 *Let Ψ be a formula with free variables among $x, y_1, \ldots, y_n$, and let f be an n-place function symbol that does not occur in Ψ. If $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ is any model, there are models $\mathbf{N}_1 = \langle \mathbf{D}, \mathbf{J}_1 \rangle$, and $\mathbf{N}_2 = \langle \mathbf{D}, \mathbf{J}_2 \rangle$ where $\mathbf{I}$, $\mathbf{J}_1$, and $\mathbf{J}_2$ differ only on the interpretation of f, and*

1. *In $\mathbf{N}_1$, $(\exists x)\Psi \supset \Psi\{x/f(y_1, \ldots, y_n)\}$ is true.*
2. *In $\mathbf{N}_2$, $\Psi\{x/f(y_1, \ldots, y_n)\} \supset (\forall x)\Psi$ is true.*

Proof We only show item 1; the other is similar. We are given the model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$; $\mathbf{J}_1$ is the same as $\mathbf{I}$ on every symbol except f, on which it is defined as follows:

Let $d_1, \ldots, d_n \in \mathbf{D}$; we specify a value for $f^{\mathbf{J}_1}(d_1, \ldots, d_n)$. Let $\mathbf{A}$ be any assignment such that $y_1^{\mathbf{A}} = d_1, \ldots, y_n^{\mathbf{A}} = d_n$. (These are the only variables that occur free in $(\exists x)\Psi$, so the behavior of $\mathbf{A}$ elsewhere is not significant.) If $(\exists x)\Psi^{\mathbf{I},\mathbf{A}} = \mathbf{f}$, choose any arbitrary member $d \in \mathbf{D}$ and set $f^{\mathbf{J}_1}(d_1, \ldots, d_n) = d$. If $(\exists x)\Psi^{\mathbf{I},\mathbf{A}} = \mathbf{t}$, then $\Psi^{\mathbf{I},\mathbf{B}} = \mathbf{t}$ for some x-variant $\mathbf{B}$ of $\mathbf{A}$. Choose one such $\mathbf{B}$ and set $f^{\mathbf{J}_1}(d_1, \ldots, d_n) = x^{\mathbf{B}}$.

We have now specified the model $\mathbf{N}_1$, and $(\exists x)\Psi \supset \Psi\{x/f(y_1, \ldots, y_n)\}$ is true in it, essentially by construction. □

In the following theorem and proof, to make the notation a little easier to read, we write $\Psi(x)$ in place of Ψ, and $\Psi(f(y_1, \ldots, y_n))$ in place of $\Psi\{x/f(y_1, \ldots, y_n)\}$.

Theorem 8.3.2 **(Skolemization)** *Let $\Psi(x)$ be a formula with free variables $x, y_1, \ldots, y_n$, and let $\Phi(A)$ be a formula such that $\Phi((\exists x)\Psi(x))$ is a sentence. Finally, suppose f is a function symbol that does not occur in $\Phi((\exists x)\Psi(x))$.*

1. *If all occurrences of A are positive in $\Phi(A)$, then $\{\Phi((\exists x)\Psi(x))\}$ is satisfiable if and only if $\{\Phi(\Psi(f(y_1, \ldots, y_n)))\}$ is satisfiable.*

2. *If all occurrences of A are negative in $\Phi(A)$ then $\{\Phi((\forall x)\Psi(x))\}$ is satisfiable if and only if $\{\Phi(\Psi(f(y_1,\ldots,y_n)))\}$ is satisfiable.*

Proof Once again we only show item 1; the other has a similar proof.

One direction is easy. The formula $\Psi(f(y_1,\ldots,y_n)) \supset (\exists x)\Psi(x)$ is easily seen to be valid. Then $\Phi(\Psi(f(y_1,\ldots,y_n))) \supset \Phi((\exists x)\Psi(x))$ is true in every model by Theorem 8.2.4, hence if $\Phi(\Psi(f(y_1,\ldots,y_n)))$ is true in some model $\mathbf{M}$, so is $\Phi((\exists x)\Psi(x))$.

For the other direction, suppose the sentence $\Phi((\exists x)\Psi(x))$ is true in the model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I}\rangle$. By Lemma 8.3.1 there is a model $\mathbf{N} = \langle \mathbf{D}, \mathbf{J}\rangle$, where $\mathbf{J}$ is exactly like $\mathbf{I}$ except on the function symbol f and in which $(\exists x)\Psi(x) \supset \Psi(f(y_1,\ldots,y_n))$ is true. By Theorem 8.2.4 again the formula $\Phi((\exists x)\Psi(x)) \supset \Phi(\Psi(f(y_1,\ldots,y_n)))$ is also true in $\mathbf{N}$. Further, $\mathbf{M}$ and $\mathbf{N}$ differ only on f, which does not occur in $\Phi((\exists x)\Psi(x))$. Since this sentence is true in $\mathbf{M}$, it will be true in $\mathbf{N}$. Then $\Phi(\Psi(f(y_1,\ldots,y_n)))$ is true in $\mathbf{N}$. □

Now the idea of Skolemization is simple: keep replacing positively occurring existentially quantified subformulas, and negatively occurring universally quantified subformulas, introducing function symbols as in the Skolemization Theorem, until no more positive existential or negative universal quantifier occurrences are left.

Example Consider sentence $X_1 = (\forall x)(\exists y)[(\exists z)(\forall w)R(x,y,z,w) \supset (\exists w)P(w)]$. In this the subformula beginning with the existential quantifier $(\exists y)$ occurs positively. Now set $X_2 = (\forall x)[(\exists z)(\forall w)R(x,f(x),z,w) \supset (\exists w)P(w)]$, and by Theorem 8.3.2, X_1 will be satisfiable if and only if X_2 is.

Now the subformula beginning with the universal quantifier $(\forall w)$ occurs negatively, so we can introduce yet another function symbol. Let $X_3 = (\forall x)[(\exists z)R(x,f(x),z,g(x,z)) \supset (\exists w)P(w)]$. Then X_3 is satisfiable if and only if X_2 is.

Continuing, the subformula beginning with the existential quantifier $(\exists w)$ occurs positively. Let

$$X_4 = (\forall x)[(\exists z)R(x,f(x),z,g(x,z)) \supset P(h(x))].$$

X_4 is satisfiable if and only if X_3 is, and hence if and only if X_1 is. The process stops here, because there are no further positive existentials or negative universals to replace.

The process of Skolemization is nondeterministic. We can freely choose which subformulas to work with next. This means many significantly different Skolemized sentences can be produced. Some can be less complicated than others, as the following example shows:

Example Consider the sentence $(\forall x)(\exists y)(\exists z)R(x, y, z)$. In Skolemizing we can work with the quantifier $(\exists y)$ first, producing $(\forall x)(\exists z)R(x, f(x), z)$, and then with the quantifier $(\exists z)$, yielding $(\forall x)R(x, f(x), g(x))$. On the other hand, starting over, we could choose to work with $(\exists z)$ first, producing $(\forall x)(\exists y)R(x, y, h(x, y))$. Then if we eliminate $(\exists y)$ next, we get $(\forall x)R(x, k(x), h(x, k(x)))$. This is clearly a more complicated sentence than the first Skolemized version.

The moral of this example is that in Skolemizing we should work from the outside in. That is, whenever we have a choice, we should pick a positive existential or negative universal quantifier that is not itself within the scope of another positive existential or negative universal.

Proposition 8.3.3 *Suppose $\neg X'$ is a Skolemized version of the sentence $\neg X$. Then X is valid if and only if X' is valid.*

Proof X is valid if and only if $\{\neg X\}$ is not satisfiable if and only if $\{\neg X'\}$ is not satisfiable if and only if X' is valid. □

The utility of Skolemization for automated theorem proving is easy to see. Suppose we are trying to prove the sentence X. Instead we can apply our proof techniques to X', where $\neg X'$ is a Skolemized version of $\neg X$. In a free-variable tableau or resolution construction for $\neg X'$, the δ-rule will never come up, thus considerably simplifying the proof attempt.

Exercises

8.3.1. Skolemize the following sentences:

1. $(\forall x)(\exists y)(\forall z)(\exists w)R(x, y, z, w)$.
2. $\neg(\exists x)[P(x) \supset (\forall x)P(x)]$.
3. $\neg(\exists x)(\exists y)[P(x, y) \supset (\forall x)(\forall y)P(x, y)]$.
4. $(\forall x)\{(\forall y)[((\forall z)P(x, y, z) \supset (\exists w)Q(x, y, w)) \supset R(x)] \supset S(x)\}$.

8.3.2$^{\mathbf{P}}$. Write a Prolog program that Skolemizes sentences.

8.3.3$^{\mathbf{P}}$. Revise the free-variable tableau implementation given earlier in Chapter 7, or your resolution theorem-proving implementation, to incorporate a Skolemizing step (previous exercise), eliminate the δ-rule portion, and remove the machinery necessary to keep track of which variables are free.

8.4 Prenex Form

Given any sentence, it is always possible to find another equivalent sentence in which all quantifiers come first. Such a sentence is said to be in *prenex* form. Sentences that are in prenex form and are also Skolemized have a particularly simple structure. We begin this section with a brief discussion of how to convert a sentence to prenex form, then we consider applications to automated theorem proving.

First of all, the same variable may be bound by two different quantifiers in different parts of a formula or may occur both free and bound. Though there is nothing wrong with this in principle, it is generally simpler to avoid it, which we can do by renaming variables. Suppose Φ is a formula in which there is a subformula $(\forall x)\Psi$, say. Let y be a variable that does not occur in Φ. The formula $(\forall x)\Psi \equiv (\forall y)\Psi\{x/y\}$ is valid. Then if we let Φ^* be the result of replacing $(\forall x)\Psi$ in Φ with $(\forall y)\Psi\{x/y\}$, $\Phi \equiv \Phi^*$ will also be valid, by Corollary 8.2.2. In this way we can always rename a bound variable, replacing it by one that is completely new.

Definition 8.4.1 We say a formula Φ has its variables *named apart* if no two quantifiers in Φ bind the same variable and no bound variable is also free.

By the discussion just given, every formula is equivalent to one in which the variables are named apart. For the rest of this section, we assume all formulas have their variables named apart.

It is easy to formulate a list of valid equivalences, which we can think of as rewriting rules, each having the effect of moving a quantifier further out. We want such an equivalence for each quantifier and each type of propositional connective. We only give a few and leave it to you to formulate the rest. Note that since we are assuming formulas have variables named apart, in a formula like $(\forall x)A \wedge B$, say, the variable x can have no occurrences in B.

Quantifier Rewrite Rules

$$\begin{array}{rcl}
\neg(\exists x)A & \equiv & (\forall x)\neg A \\
\neg(\forall x)A & \equiv & (\exists x)\neg A \\
[(\forall x)A \wedge B] & \equiv & (\forall x)\,[A \wedge B] \\
[A \wedge (\forall x)B] & \equiv & (\forall x)\,[A \wedge B] \\
[(\exists x)A \wedge B] & \equiv & (\exists x)\,[A \wedge B] \\
[A \wedge (\exists x)B] & \equiv & (\exists x)\,[A \wedge B] \\
[(\forall x)A \supset B] & \equiv & (\exists x)\,[A \supset B] \\
[A \supset (\forall x)B] & \equiv & (\forall x)\,[A \supset B] \\
[(\exists x)A \supset B] & \equiv & (\forall x)\,[A \supset B] \\
[A \supset (\exists x)B] & \equiv & (\exists x)\,[A \supset B]
\end{array}$$

Using these rules, and those from Exercise 8.4.1, quantifiers can be moved to the front, by replacing left-hand sides of equivalences above by right-hand sides, eventually producing a prenex equivalent for any formula.

Example Start with the sentence $(\exists x)(\forall y)R(x,y) \supset (\forall y)(\exists x)R(x,y)$. Rename variables apart, getting $(\exists x)(\forall y)R(x,y) \supset (\forall z)(\exists w)R(w,z)$. By applying the equivalences, we can first move the x and y quantifiers to the front, then the z and w ones, producing the equivalent sentence $(\forall x)(\exists y)(\forall z)(\exists w)[R(x,y) \supset R(w,z)]$. On the other hand, if we move the z and w quantifiers first, we get the sentence $(\forall z)(\exists w)(\forall x)(\exists y)[R(x,y) \supset R(w,z)]$. Both of these are prenex equivalents of the original sentence.

In applying these rewriting rules, quantifiers that are essentially existential remain so. That is, a positive occurrence of an existential quantifier or a negative occurrence of a universal quantifier both are converted to an existential quantifier in front, and similarly for universals. This means that if we convert into prenex form a sentence that has been Skolemized, the result will have only universal quantifiers in front. Equivalently, we could first convert a sentence to prenex form, then Skolemize away the existential quantifiers. Either way we wind up with a sentence containing only universal quantifiers, all of which are in front. We summarize things so far.

Proposition 8.4.2 *There is an algorithm for converting a sentence Φ into a sentence Φ^* in prenex form, with only universal quantifiers, such that $\{\Phi\}$ is satisfiable if and only if $\{\Phi^*\}$ is satisfiable.*

In the most general version of resolution theorem proving that we presented, Resolution Expansion Rules can be applied anytime during the course of the proof. But by using Proposition 8.4.2 and some additional rewriting, all applications can be done ahead of time. Suppose we have a sentence Φ that we wish to prove. As usual, we begin by negating it, $\neg\Phi$. Next, convert this to a prenex, Skolemized form, $(\forall x_1)\cdots(\forall x_n)\Psi$, where Ψ contains no quantifiers. Ψ Is often called the *matrix* of the sentence. Now we can further convert this matrix into conjunctive normal form—clause form. We get an equivalent sentence, $(\forall x_1)\cdots(\forall x_n)\langle C_1,\ldots,C_k\rangle$, where each of $C_1,\ldots, C_k$ is a clause. Next,

$$(\forall x)(A \wedge B) \equiv ((\forall x)A \wedge (\forall x)B)$$

is valid, and so our sentence can be rewritten in the equivalent form

$$\langle (\forall x_1) \cdots (\forall x_n) C_1, \ldots, (\forall x_1) \cdots (\forall x_n) C_n \rangle.$$

If we start a resolution proof construction from this point, clearly, the only rules that can ever apply are the γ-rule, and the Binary Literal Resolution Rule with Factoring, or the General Literal Resolution Rule.

There is one more, essentially cosmetic, simplification we can make. We can drop the quantifiers altogether, and work with the list $C_1, \ldots, C_k$ of clauses containing free variables. But we must remember the quantifiers are implicitly present. This means that we never work with one of these clauses directly, but rather with the result of renaming its variables, replacing them with new free variables—in effect an application of the free-variable γ-rule.

Most implemented resolution theorem provers take the route we have just outlined. The δ-rule is thus avoided altogether, and the propositional reduction rules are applied ahead of time. The explicit γ-rule that we have been using is generally dropped. In its place the application of the resolution rule to two clauses C_1 and C_2, containing free variables, is defined to be the application of the resolution rule in our sense to the result of renaming the variables in these two clauses so that they do not share variables. In effect this incorporates an implicit application of the γ-rule. Thus, in the discussions of resolution theorem proving to be found in the literature, the *only* rule generally allowed is the resolution rule (and maybe factoring) but combined with variable renaming. It is assumed sentences have been preprocessed into appropriate forms, and the real theorem proving starts after that step. We have explicitly incorporated the propositional and quantifier rules into our version of the system. This makes it possible to experiment with various versions of non-clausal theorem proving. And also, it leads naturally to the topic of theorem proving in non-classical logics, where a simple analog of clause form may not be available [17, 18]. Following up on these topics is beyond our range here, however.

Exercises

8.4.1. Complete the list of Quantifier Rewrite Rules by giving the cases for the other propositional connectives.

8.4.2. Convert the following sentences to prenex form:

1. $\{\neg(\exists x)(\forall y)P(x,y) \vee (\forall x)Q(x)\} \wedge \{(\forall x)A(x) \supset (\forall x)B(x)\}$.
2. $\neg(\exists x)(\exists y)[P(x,y) \supset (\forall x)(\forall y)P(x,y)]$.
3. $(\forall x)\{(\forall y)[((\forall z)P(x,y,z) \supset (\exists w)Q(x,y,w)) \supset R(x)] \supset S(x)\}$.

8.4.3$^{\mathbf{P}}$. Write a Prolog program to rename the variables of sentences apart.

8.4.4$^{\mathbf{P}}$. Write a Prolog program to convert formulas to prenex form.

8.4.5$^{\mathbf{P}}$. Write an implementation of resolution theorem proving following the outline sketched in this section.

8.5 The AE-Calculus

In 1958 Hao Wang designed and implemented one of the first automated theorem provers [55]. It was based on the Gentzen sequent calculus and predated the introduction of resolution. When applied to sections *9 through *13 of Whitehead and Russell's *Principia Mathematica* [56], it was able to prove all of the pure logic theorems found therein, in about 4 minutes! This was a remarkable achievement so early in the history of automated theorem proving. But because unification was not discovered until later (or rediscovered, because it appeared in Herbrand's work of 1930 [25], but this was not realized until much later), how was such success possible? In fact, the success says something remarkable about *Principia Mathematica* itself.

Definition 8.5.1 We say a quantified subformula of a formula Φ is *essentially universal* if it is a positive subformula of the form $(\forall x)\varphi$ or a negative subformula of the form $(\exists x)\varphi$. Likewise, a quantified subformula is *essentially existential* if it is a positive subformula of the form $(\exists x)\varphi$ or a negative subformula of the form $(\forall x)\varphi$.

This definition uses the notion of positive and negative subformula—see Definition 8.2.3 and the discussion following it for terminology.

Definition 8.5.2 An **AE**-*formula* is a formula without function symbols, in which no essentially universal quantifier is within the scope of an essentially existential quantifier.

Example The formula $(\exists x)(\forall y)R(x, y) \supset (\forall z)(\exists w)R(w, z)$ is an **AE** formula. In it the subformulas beginning with $(\exists x)$ and $(\forall z)$ are essentially universal, while the subformulas beginning with $(\forall y)$ and $(\exists w)$ are essentially existential. On the other hand, $(\forall x)(\exists y)R(x, y) \supset (\exists z)(\forall w)R(w, z)$ is not an **AE** formula (for two reasons).

An equivalent way of characterizing **AE** formulas is that they are formulas without function symbols that can be put in prenex form (Section 8.4), so that the initial string of quantifiers consists of a sequence of universal ones, followed by a sequence of existential ones. This version explains where the name for the class of formulas came from: We have a string of Alls followed by a string of Exists. More significantly for us, in tableau terms, if X is a closed **AE** formula and we construct a tableau for $\neg X$, we can carry out all δ-rule applications before any γ-rule applications.

We can now say more precisely what Hao Wang's work actually consisted of. First, there is a decision procedure for **AE** sentences. Second, Wang discovered that the theorems of pure first-order logic in *Principia Mathematica* are all **AE** sentences—quite a surprising fact! In his paper Wang described a full first-order theorem prover but did not implement it, essentially because unification was not yet known. He did implement a theorem prover for **AE** sentences and obtained the success just noted. We now sketch a tableau version of his Gentzen system idea.

We use the first-order tableau system of Chapter 6, with parameters, but without free variables. And we impose the following extra restrictions.

1. No δ-rule application can follow a γ-rule application.
2. The tableau is *strict* in the sense that, except for γ-formulas, no formula on a branch can have a tableau rule applied to it more than once.
3. If γ occurs on a branch, $\gamma(p_1), \ldots, \gamma(p_n)$ can be added where $p_1, \ldots, p_n$ are the *parameters previously introduced* to the branch by a δ-rule. (If there are no δ-formulas on the branch, we allow $\gamma(p_0)$ to be added, for a single, fixed parameter p_0.)

Call a tableau meeting these conditions an **AE** *tableau.* Call an **AE** tableau *complete* if every formula occurrence on each branch has had the appropriate tableau rule applied to it. Now the key facts are these.

Proposition 8.5.3 *Let X be an* **AE** *sentence.*

1. *X is valid if and only if there is a closed, complete* **AE** *tableau for $\neg X$.*
2. *If any complete* **AE** *tableau for $\neg X$ is closed, every complete* **AE** *tableau for $\neg X$ is closed.*
3. *There is a systematic way of constructing a complete* **AE** *tableau for $\neg X$ that terminates, whether X is valid or not.*

According to this proposition, we have a decision procedure for **AE** sentences. To check if the **AE** sentence X is valid, apply the systematic tableau construction procedure of part 3 and produce a complete **AE** tableau for $\neg X$. If the resulting tableau is closed, X is valid by part 1. If the resulting tableau is not closed, by part 2 no complete **AE** tableau for $\neg X$ will be, so X is not valid by part 1 again.

We leave the proof of Proposition 8.5.3 to you as exercises. But here is an example that illustrates the ideas behind the proof. Consider the **AE** sentence $(\exists x)(\forall y)R(x,y) \supset (\forall z)R(z,z)$. Here is a complete **AE** tableau for its negation.

1. $\neg[(\exists x)(\forall y)R(x,y) \supset (\forall z)R(z,z)]$
2. $(\exists x)(\forall y)R(x,y)$
3. $\neg(\forall z)R(z,z)$
4. $(\forall y)R(p_1,y)$
5. $\neg R(p_2,p_2)$
6. $R(p_1,p_1)$
7. $R(p_1,p_2)$

In this, 4 is from 2 and 5 is from 3 by the δ-rule, then 6 and 7 are from 4 by the γ-rule (using all the parameters previously introduced). The tableau is complete and not closed. We now use it to construct a model showing the formula in question is not valid.

Take for the domain $\mathbf{D}$ the set $\{p_1,p_2\}$ of parameters. Interpret the parameters to name themselves, that is, $p_1^{\mathbf{I}} = p_1$ and $p_2^{\mathbf{I}} = p_2$. Finally, interpret R the way 5 through 7 say: $R^{\mathbf{I}}$ is true of $\langle p_1,p_1\rangle$ and $\langle p_1,p_2\rangle$ and false of $\langle p_2,p_2\rangle$. (What happens with $\langle p_2,p_1\rangle$ won't matter—make an arbitrary choice.) We thus have a model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I}\rangle$. Now it is easy to see that, in this model, tableau lines 4, 3, 2, and finally 1 are all true. Since 1 is true in the model, $(\exists x)(\forall y)R(x,y) \supset (\forall z)R(z,z)$ is not valid.

Exercises

8.5.1. Describe a systematic construction procedure for complete **AE** tableaus. Show that when applied to $\neg X$, where X is an **AE** sentence, it must terminate.

8.5.2$^{\mathbf{P}}$. Implement the tableau procedure of the Exercise 8.5.1.

8.5.3. Show that if the tableau procedure of Exercise 8.5.1 is applied to $\neg X$, where X is an **AE** sentence, and the procedure does not produce a closed tableau, any open branch generates a countermodel for X.

8.6 Herbrand's Theorem

In 1930 Herbrand proved a fundamental theorem [25] that is a kind of constructive version of Gödel's completeness theorem, and has since come to be seen as the theoretical foundation of automated theorem proving. The theorem involves the notion of a *Herbrand expansion*—a formula that can be thought of as a finite approximation to a Herbrand model. Herbrand's result is that X is a theorem of first-order logic if and only if some Herbrand expansion of X is a propositional tautology. Thus, it can be seen as reducing a first-order provability problem to an infinite set of propositional problems, one for each possible Herbrand expansion. Since there is a decision procedure for being a tautology, first-order theoremhood has thus been reduced to an open-ended search through an infinite set of finite calculations. In a broad sense the history of first-order automated theorem proving can be split into two parts: what is the best strategy for conducting the search for a Herbrand expansion that is a tautology and what is the most efficient technique for testing tautologyhood.

Herbrand himself was a constructivist and thus could not accept the kinds of semantic arguments we have used throughout, many of which involve proving the existence of infinite models nonconstructively. From a constructive point of view, Herbrand's Theorem provides a proof-theoretic substitute for semantic methods, and indeed, its use has led to proofs of decidability for several natural subclasses of first-order logic (see Exercise 8.6.5 for a simple example). But since many people find semantic arguments easier to follow, in this section we give a nonconstructive proof of Herbrand's Theorem, using the Model Existence Theorem. In the next section we consider a constructive version.

Definition 8.6.1 The sentence X' is a *validity functional form* of X if $\neg X'$ is a Skolemized version of $\neg X$.

Proposition 8.3.3 makes the choice of terminology clear—if X' is a validity functional form of a sentence X, then X is valid if and only if X' is. Also, Skolemization of $\neg X$ removes all its essentially existential quantifiers, which correspond to the essentially universal quantifiers of X. Consequently, we have the following simple, yet basic, observation: *The validity functional form of a sentence contains only essentially existential quantifiers.* Now, the whole idea of Herbrand expansion is to replace these by appropriate disjunctions and conjunctions.

Definition 8.6.2 Let X be a sentence. Its *Herbrand universe* is the set of all closed terms constructed from the constant and function symbols of X. (If X contains no constant symbols, allow c_0, where c_0 is an arbitrary new constant symbol.)

Example The Herbrand universe of the sentence $(\forall x)[(\exists y)R(x,y) \supset R(b,f(x))]$ is the set $\{b, f(b), f(f(b)), \ldots\}$. The Herbrand universe of $(\forall x)(\exists y)R(x,y)$ is just $\{c_0\}$. (The reason for introducing the constant symbol c_0 is simply so that Herbrand universes are never empty.)

Definition 8.6.3 A *Herbrand domain* for a sentence X is a finite, non-empty subset of the Herbrand universe of X.

Now we present the key notion—that of a Herbrand expansion. Often in the literature, Herbrand expansions are defined only for formulas in prenex form, but this is an unnecessary restriction, one that Herbrand himself did not impose, and it is not a restriction imposed here either. The essential idea of a Herbrand expansion is simple: Existential quantifiers become disjunctions of instances over the Herbrand universe, and universal quantifiers become conjunctions. We formally characterize Herbrand expansions using uniform notation because it is most convenient for an application of the Model Existence Theorem, but our definition is easily seen to be equivalent to more conventional ones (see Exercise 8.6.4). For Herbrand's Theorem itself, we are interested only in formulas whose quantifiers are essentially existential, but even so we define the notion of an expansion for every formula, since it will make it easier to present the proof of the theorem—thus, there is not only the expected δ-case, but a γ-case as well.

Definition 8.6.4 Let $D = \{t_1, \ldots, t_n\}$ be a non-empty set of closed terms. For a sentence X, the *Herbrand expansion of X over D*, denoted $\mathcal{E}(X, D)$, is defined recursively as follows:

1. If L is a literal, $\mathcal{E}(L, D) = L$.
2. $\mathcal{E}(\neg\neg Z, D) = \mathcal{E}(Z, D)$.
3. $\mathcal{E}(\alpha, D) = \mathcal{E}(\alpha_1, D) \wedge \mathcal{E}(\alpha_2, D)$.
4. $\mathcal{E}(\beta, D) = \mathcal{E}(\beta_1, D) \vee \mathcal{E}(\beta_2, D)$.
5. $\mathcal{E}(\gamma, D) = \mathcal{E}(\gamma(t_1), D) \wedge \cdots \wedge \mathcal{E}(\gamma(t_n), D)$.
6. $\mathcal{E}(\delta, D) = \mathcal{E}(\delta(t_1), D) \vee \cdots \vee \mathcal{E}(\delta(t_n), D)$.

A *Herbrand expansion* of a sentence X is a Herbrand expansion of Y over D, where Y is a validity functional form of X and D is any Herbrand domain for Y.

Theorem 8.6.5 **(Herbrand's Theorem)** *A sentence X is valid if and only if some Herbrand expansion of X is a tautology.*

Example Consider the valid sentence:

$$(\forall z)(\exists w)(\forall x)[(\forall y)R(x, y) \supset R(w, z)].$$

Converting it to validity functional form produces:

$$(\exists w)[(\forall y)R(f(w), y) \supset R(w, c)],$$

where c is a new constant symbol and f is a new one-place function symbol. Now, $D = \{c, f(c)\}$ is a Herbrand domain for this sentence, and the following computes the Herbrand expansion over D:

$$\begin{aligned}&\mathcal{E}((\exists w)[(\forall y)R(f(w), y) \supset R(w, c)], D)\\ &\quad = \neg R(f(c), c) \vee \neg R(f(c), f(c)) \vee R(c, c) \vee\\ &\qquad \neg R(f(f(c)), c) \vee \neg R(f(f(c)), f(c)) \vee R(f(c), c)\end{aligned}$$

It is easy to see that this is a tautology.

There are a few simple, but very important properties of Herbrand expansions—we will use them over and over in what follows. We state them here and leave the proofs to you as exercises.

Proposition 8.6.6 *Let D be a non-empty set of closed terms.*

1. *For an arbitrary sentence X, $\neg\mathcal{E}(X, D) \equiv \mathcal{E}(\neg X, D)$ is a tautology.*
2. *Let D' be a non-empty set of closed terms such that $D \subseteq D'$.*
   (a) *If X is a sentence all of whose quantifiers are essentially existential, then $\mathcal{E}(X, D) \supset \mathcal{E}(X, D')$ is a tautology.*
   (b) *If X is a sentence all of whose quantifiers are essentially universal, then $\mathcal{E}(X, D') \supset \mathcal{E}(X, D)$ is a tautology.*

According to Proposition 8.3.3, a sentence is valid if and only if its validity functional form is valid, so in demonstrating Herbrand's Theorem, it is enough to prove it for sentences already in validity functional form—that is, we can assume X has only essentially existential quantifiers. Now, Herbrand's Theorem has an "if and only if" form, and the arguments in each direction are different. We begin with the easier half and show that if some Herbrand expansion of X is a tautology, then X itself is valid. But this is an immediate consequence of the following:

Lemma 8.6.7 *Suppose all quantifiers in the sentence X are essentially existential and D is a finite set of closed terms. Then $\mathcal{E}(X, D) \supset X$ is valid.*

Proof This is a simple argument by structural induction on X (or equivalently, we could use Theorem 8.2.4 and its Corollary 8.2.6). We give only the δ-case. (The γ-case does not arise, since only essentially existential quantifiers are present. The other cases to be considered are all propositional.) Suppose, for each $t_1 \in D$, that $\mathcal{E}(\delta(t_i), D) \supset \delta(t_i)$ is valid. Then

$$\begin{array}{rcl}\mathcal{E}(\delta, D) & = & \mathcal{E}(\delta(t_1), D) \vee \cdots \vee \mathcal{E}(\delta(t_n), D)\\ & \supset & \delta(t_1) \vee \cdots \vee \delta(t_n)\\ & \supset & \delta\end{array}$$

where the last step makes use of an obvious validity of first-order logic. □

In the other direction, we make use of the Model Existence Theorem (5.8.2). We use the following notation: For a finite set S of formulas, $\bigwedge S$ is the conjunction of members of S, arranged in some arbitrary order, and parenthesized in some arbitrary way.

Lemma 8.6.8 *Let L be a first-order language, and let L^{par} be its extension having an infinite set of new constant symbols---parameters. Let H be the collection of all closed terms built from constant and function symbols of L^{par}. Call a set S of sentences of L^{par}* Herbrand consistent *if:*

1. *S is finite.*
2. *All members of S are essentially universal.*
3. *$\neg\mathcal{E}(\bigwedge S, D)$ is not a tautology, for any finite $D \subseteq H$.*

Let $\mathcal{C}$ be the collection of all Herbrand-consistent sets. Then $\mathcal{C}$ is a first-order consistency property with respect to L.

Proof Note that by Exercise 8.6.3, details of the formation of $\bigwedge S$ can be safely ignored. Proving the lemma amounts to checking the various conditions for being a first-order consistency property. We begin with the β-condition. Suppose $S \in \mathcal{C}$, $\beta \in S$, but $S \cup \{\beta_1\} \notin \mathcal{C}$ and $S \cup \{\beta_2\} \notin \mathcal{C}$. We derive a contradiction.

Since $S \cup \{\beta_1\} \notin \mathcal{C}$, for some finite $D_1 \subseteq H$, $\neg\mathcal{E}(\bigwedge S \wedge \beta_1, D_1)$ is a tautology. Likewise, since $S \cup \{\beta_2\} \notin \mathcal{C}$, $\neg\mathcal{E}(\bigwedge S \wedge \beta_2, D_2)$ is a tautology for some finite $D_2 \subseteq H$. Let $D = D_1 \cup D_2$. Then both $\neg\mathcal{E}(\bigwedge S \wedge \beta_1, D)$ and $\neg\mathcal{E}(\bigwedge S \wedge \beta_2, D)$ are tautologies, using part 2 of Proposition 8.6.6. But now,

$$\begin{array}{rcl}\mathcal{E}(\bigwedge S \wedge \beta, D) & = & \mathcal{E}(\bigwedge S, D) \wedge \mathcal{E}(\beta, D)\\ & = & \mathcal{E}(\bigwedge S, D) \wedge [\mathcal{E}(\beta_1, D) \vee \mathcal{E}(\beta_2, D)]\\ & \equiv & [\mathcal{E}(\bigwedge S, D) \wedge \mathcal{E}(\beta_1, D)] \vee [\mathcal{E}(\bigwedge S, D) \wedge \mathcal{E}(\beta_2, D)]\\ & = & \mathcal{E}(\bigwedge S \wedge \beta_1, D) \vee \mathcal{E}(\bigwedge S \wedge \beta_2, D)\end{array}$$

Consequently, $\neg\mathcal{E}(\bigwedge S \wedge \beta, D) \equiv \neg\mathcal{E}(\bigwedge S \wedge \beta_1, D) \wedge \neg\mathcal{E}(\bigwedge S \wedge \beta_2, D)$, and so must also be a tautology, which contradicts the facts that $\beta \in S$ and $S \in \mathcal{C}$.

The other propositional cases are similar. We now turn to the γ-case—there is no δ-case since $\mathcal{C}$ involves only essentially universal formulas. So, suppose $S \in \mathcal{C}$, $\gamma \in S$, but for some closed term t, $S \cup \{\gamma(t)\} \notin \mathcal{C}$. We derive a contradiction.

By our assumptions, for some finite $D \subseteq H$, $\neg\mathcal{E}(\bigwedge S \wedge \gamma(t), D)$ must be a tautology. Now, $\bigwedge S \wedge \gamma(t)$ is essentially universal, and $D \subseteq D \cup \{t\}$, so it follows from part 2 of Proposition 8.6.6 that $\neg\mathcal{E}(\bigwedge S \wedge \gamma(t), D \cup \{t\})$ is a tautology. Note also that $\mathcal{E}(\gamma, D \cup \{t\})$ is a conjunction, and one of its conjuncts is $\mathcal{E}(\gamma(t), D \cup \{t\})$. Now,

$$\begin{array}{rcl} \mathcal{E}(\bigwedge S \wedge \gamma, D \cup \{t\}) & = & \mathcal{E}(\bigwedge S, D \cup \{t\}) \wedge \mathcal{E}(\gamma, D \cup \{t\}) \\ & \supset & \mathcal{E}(\bigwedge S, D \cup \{t\}) \wedge \mathcal{E}(\gamma(t), D \cup \{t\}) \\ & = & \mathcal{E}(\bigwedge S \wedge \gamma(t), D \cup \{t\}) \end{array}$$

Since $\neg\mathcal{E}(\bigwedge S \wedge \gamma(t), D \cup \{t\})$ is a tautology, so is $\neg\mathcal{E}(\bigwedge S \wedge \gamma, D \cup \{t\})$, and this contradicts the facts that $\gamma \in S$ and $S \in \mathcal{C}$.

This concludes the proof of the lemma. □

Now, finally, we can put into place the final pieces of the proof of Herbrand's Theorem.

Lemma 8.6.9 *Let X be a sentence whose quantifiers are all essentially existential. If X is valid, then $\mathcal{E}(X, D)$ is a tautology, for some Herbrand domain D for X.*

Proof We first prove a slightly weaker version, from which the lemma follows quickly. Let L be the first-order language whose constant, function, and relation symbols are exactly those of X—that is, L is the smallest language in which X is a sentence. (If X has no constant symbols, then c_0 is allowed as the only constant symbol of L.) As usual, $L^{\mathbf{par}}$ is the extension of L with an infinite set of new constant symbols. We first show that if X is valid, then $\mathcal{E}(X, D)$ is a tautology, for some finite set D of *closed terms of* $L^{\mathbf{par}}$.

Suppose $\mathcal{E}(X, D)$ is not a tautology, for every finite set D of closed terms of $L^{\mathbf{par}}$. Then $\{\neg X\}$ is a member of the first-order consistency property $\mathcal{C}$ constructed in the previous lemma. This follows because (1) $\{\neg X\}$ is obviously finite; (2) all quantifiers in $\neg X$ are essentially universal, since those of X are essentially existential; and (3) $\neg\mathcal{E}(\bigwedge\{\neg X\}, D) = \neg\mathcal{E}(\neg X, D) \equiv \neg\neg\mathcal{E}(X, D)$ (by part 1 of Proposition 8.6.6) $\equiv \mathcal{E}(X, D)$, and this is not a tautology for any D, by assumption. Then by the Model Existence Theorem 5.8.2, $\{\neg X\}$ is satisfiable in a first-order model, and so X is not valid.

We now know that if X is valid, $\mathcal{E}(X, D)$ is a tautology for some finite set D of closed terms of L^{par}. Now we eliminate parameters from D. Simply map each parameter p to some closed term $\tau(p)$ of L—the particular choice won't matter. This mapping on parameters induces a corresponding mapping on terms, sets of terms, formulas, and sets of formulas, in the obvious way—we use τ for the induced mapping as well. Then, for instance, for a term t, $\tau(t)$ is the result of replacing, throughout t, each parameter p by the corresponding closed term $\tau(p)$. It is easy to see that $\tau(\mathcal{E}(X, D))$ is simply $\mathcal{E}(X, \tau(D))$ (recall that X contained no parameters). But also, $\tau(\mathcal{E}(X, D))$ can be thought of as resulting from $\mathcal{E}(X, D)$ by replacing each atomic subformula A by $\tau(A)$, and in Exercise 2.4.7 you showed that such a replacement of atomic formulas in a tautology produced another tautology. Since $\mathcal{E}(X, D)$ is a tautology, so is $\mathcal{E}(X, \tau(D))$, and we thus have a tautologous Herbrand expansion of X. □

Exercises

8.6.1. Prove part 1 of Proposition 8.6.6.

8.6.2. Prove part 2 of Proposition 8.6.6.

8.6.3. Let S be a finite set of sentences, and let C_1 be a conjunction of all the members of S, with an arbitrarily chosen order and parenthesization; likewise, let C_2 be another such conjunction of all the members of S. Finally, let D be a non-empty set of closed terms. Show that $\mathcal{E}(C_1, D) \equiv \mathcal{E}(C_2, D)$ is a tautology.

8.6.4. A more conventional definition of a Herbrand expansion is given by the following, where $D = \{t_1, \ldots, t_n\}$:

1. If A is atomic, $\mathcal{E}'(A, D) = A$.
2. $\mathcal{E}'(\neg Z, D) = \neg\mathcal{E}'(Z, D)$.
3. $\mathcal{E}'(Z \circ W, D) = \mathcal{E}'(Z, D) \circ \mathcal{E}'(W, D)$, for a binary connective $\circ$.
4. $\mathcal{E}'((\forall x)\varphi(x), D) = \mathcal{E}'(\varphi(t_1), D) \wedge \cdots \wedge \mathcal{E}'(\varphi(t_n), D)$.
5. $\mathcal{E}'((\exists x)\varphi(x), D) = \mathcal{E}'(\varphi(t_1), D) \vee \cdots \vee \mathcal{E}'(\varphi(t_n), D)$.

Prove that for every sentence X, $\mathcal{E}(X, D) \equiv \mathcal{E}'(X, D)$ is a tautology.

8.6.5. Use Herbrand's Theorem and give an alternative proof that there is a decision procedure for the validity of **AE** sentences (Definition 8.5.2).

8.7 Herbrand's Theorem, Constructively

The harder half of Herbrand's Theorem says that if a sentence X is valid, then some Herbrand expansion of X must be a tautology. Our proof of this was non-constructive, in the sense that if we "know" X to be valid, we still do not know which Herbrand expansion will be a tautology. Of course, we could systematically search through the infinite set of Herbrand expansions of X for one that is a tautology, and our knowledge that X is valid guarantees our search must succeed. But beyond this, we don't even know how long we must search before success.

How could we "know" X to be valid? We could be given a proof of X using one of the formal proof procedures considered in Chapter 6. Such a proof contains more information than the simple fact of validity of X—it presents *evidence* for this validity. Ideally, we should be able to use that evidence to improve on the naive search procedure for a tautologous Herbrand expansion of X.

Herbrand himself had a constructive philosophy of mathematics: Any talk of validity is essentially meaningless, but talk of proofs is allowed. Proofs, after all, are finite objects, but to check validity of a sentence, infinitely many models must be examined, and the models themselves can be infinite. As a constructivist, Herbrand actually proved something stronger than Theorem 8.6.5—something algorithmic in nature. He showed that from a first-order proof of X one can extract a Herbrand expansion of X that is a tautology. We will prove this stronger, constructive version, though we do not follow Herbrand's proof methods—see Herbrand [26] for these.

We have given several different first-order proof procedures. Herbrand's theorem is quite easy to establish constructively when a tableau, resolution, or Gentzen system is involved, and much harder using a Hilbert or a natural deduction system. In this section we consider tableau as a representative example of the easy class of proof procedures. In the next section we turn to the harder variety. So, the rest of this section is devoted to a proof of the following:

Theorem 8.7.1 **(Herbrand's Theorem, Constructively)** *There is an algorithm that extracts from a tableau proof of a first-order sentence X a Herbrand expansion of X that is a tautology.*

Every sentence can be put into validity functional form—we have an algorithm for doing so. Consequently, it is enough to prove Theorem 8.7.1 for sentences in this form.

Let X be a sentence in validity functional form, and let L be the first-order language whose constant, function, and relation symbols are exactly those of X. (As usual, if X has no constant symbols, let L have c_0 as its only one.) If X is provable, there is a closed tableau $\mathbf{T}$ for $\neg X$.

Since X is in validity functional form, all its quantifiers are essentially existential, so all quantifiers of $\neg X$ are essentially universal. Then in $\mathbf{T}$ the δ-rule cannot be used. This means no rule application "forces" us to introduce parameters. Nonetheless, a tableau for $\neg X$ is constructed using the language L^{par}, so in a γ-rule application, we could have added $\gamma(t)$ to a branch, where t is a closed term containing parameters. But even though this is allowed, it is unnecessary. Let c be some fixed constant symbol of L, and throughout $\mathbf{T}$ replace every occurrence of any parameter by an occurrence of c. It is easy to see that the resulting tree is still a correctly constructed tableau and is still closed. Further, since $\neg X$ contains no parameters, the new tableau is still one for $\neg X$. We can sum up these simple observations as follows:

Definition 8.7.2 Call a tableau *parameter free* if it contains no parameter occurrences.

Lemma 8.7.3 *If all quantifiers of X are essentially existential, a tableau proof of X can be converted into a parameter-free tableau proof.*

Now we come to the central fact, one showing how close Herbrand's theorem is to tableau methods. The similarity of its proof to that of Lemma 8.6.8 is no coincidence.

Proposition 8.7.4 *Let X be a sentence whose quantifiers are essentially existential, let $\mathbf{T}$ be a closed, parameter-free tableau for $\neg X$, and let D be the set of closed terms that are used in applications of the γ-rule in $\mathbf{T}$. Then $\mathcal{E}(X, D)$ is a tautology.*

Proof To show this Proposition we must prove a somewhat stronger statement, from which the result we want follows easily as a special case, using part 1 of Proposition 8.6.6. We must show the following:

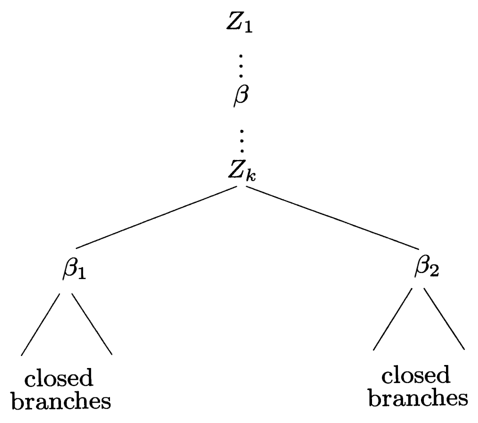
- Let S be a *finite set* of sentences, all of whose quantifiers are essentially existential, let $\mathbf{T}$ be a closed, parameter-free tableau for S, and let D be the set of closed terms introduced in $\mathbf{T}$ by the γ-rule. Then $\neg\mathcal{E}(\bigwedge S, D)$ is a tautology.

The proof of this is, roughly, by induction on the number of branch extension rule applications in the tableau $\mathbf{T}$ for the set S. More precisely, let us denote by $\mathcal{B}(\mathbf{T}, S)$ the number of nodes in the tree $\mathbf{T}$ *below the initial nodes labeled with S*. The induction, properly, is on $\mathcal{B}(\mathbf{T}, S)$.

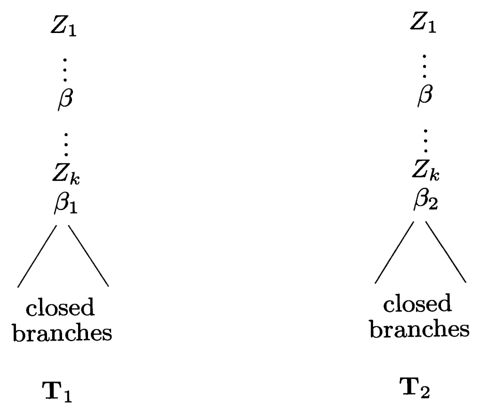
The ground case is when $\mathcal{B}(\mathbf{T}, S) = 0$. In this case, S itself must contain a formula Z and its negation, $\neg Z$. But then $\neg \bigwedge S$ itself is a tautology—a degenerate case, if you will.

Now suppose $\mathcal{B}(\mathbf{T}, S) > 0$, and the result is known for all "simpler" tableaux. There are several cases, depending on which branch extension rule was first applied in $\mathbf{T}$. We consider two of the cases and leave the rest to you.

Suppose the first rule applied in $\mathbf{T}$ is the β-rule. Thus, $S = \{Z_1, \ldots, \beta, \ldots, Z_k\}$, and $\mathbf{T}$ has the following form:



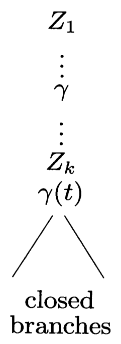
Let $\mathbf{T}_1$ be the subtableau consisting of the left half of $\mathbf{T}$—it is a correctly constructed, closed tableau for $S \cup \{\beta_1\}$. And let D_1 be the set of closed terms introduced by the γ-rule in $\mathbf{T}_1$. Likewise let $\mathbf{T}_2$ be the right half of $\mathbf{T}$—a closed tableau for $S \cup \{\beta_2\}$, and let D_2 be the set of closed terms introduced in it by the γ-rule. We now have two "simpler" tableaux—specifically, we have the following:



$\mathbf{T}_1$ $\mathbf{T}_2$

Clearly $\mathcal{B}(\mathbf{T}, S) = \mathcal{B}(\mathbf{T}_1, S \cup \{\beta_1\}) + \mathcal{B}(\mathbf{T}_2, S \cup \{\beta_2\}) + 2$ (the "+2" corresponds to the initial β-rule application in $\mathbf{T}$). Thus, $\mathcal{B}(\mathbf{T}_1, S \cup \{\beta_1\}) < \mathcal{B}(\mathbf{T}, S)$ and $\mathcal{B}(\mathbf{T}_2, S \cup \{\beta_2\}) < \mathcal{B}(\mathbf{T}, S)$, so the induction hypothesis applies, and both $\neg\mathcal{E}(\bigwedge S \wedge \beta_1, D_1)$ and $\neg\mathcal{E}(\bigwedge S \wedge \beta_2, D_2)$ are tautologies. Since all formulas involved are essentially existential, it follows by part 2 of Proposition 8.6.6 that both $\neg\mathcal{E}(\bigwedge S \wedge \beta_1, D)$ and $\neg\mathcal{E}(\bigwedge S \wedge \beta_2, D)$ are tautologies. It further follows that $\neg\mathcal{E}(\bigwedge S \wedge \beta, D)$ is a tautology, exactly as in the proof of Lemma 8.6.8, and since $\beta \in S$, $\neg\mathcal{E}(\bigwedge S, D)$ is a tautology.

Next we turn to the case where the first branch extension rule applied in $\mathbf{T}$ is the γ-rule. In this case $\mathbf{T}$ looks like the following:



But instead of considering this to be a closed tableau for S, we can also consider it to be a closed tableau for $S \cup \{\gamma(t)\}$. If we do, one fewer branch extension rule application is involved, since $\gamma(t)$ is one of the sentences the tableau construction begins with. That is, $\mathcal{B}(\mathbf{T}, S) = \mathcal{B}(\mathbf{T}, S \cup \{\gamma(t)\}) + 1$, so $\mathcal{B}(\mathbf{T}, S \cup \{\gamma(t)\}) < \mathcal{B}(\mathbf{T}, S)$, and the induction hypothesis applies. Suppose we let D be the set of closed terms introduced in $\mathbf{T}$ by the γ-rule, when $\mathbf{T}$ is thought of as a tableau for S, and we let D_0 be the set of closed terms introduced in $\mathbf{T}$ by the γ-rule, when it is thought of as a tableau for $S \cup \{\gamma(t)\}$. (Then $D = D_0 \cup \{t\}$, though it is possible that $D = D_0$, since t may have been involved in an earlier γ-rule application in $\mathbf{T}$.) By the induction hypothesis, $\neg\mathcal{E}(\bigwedge S \wedge \gamma(t), D_0)$ is a tautology, and since $D_0 \subseteq D$ and all sentences are essentially existential, part 2 of Proposition 8.6.6 again tells us that $\neg\mathcal{E}(\bigwedge S \wedge \gamma(t), D)$ is a tautology. Once again, just as in the proof of Lemma 8.6.8, it follows that $\neg\mathcal{E}(\bigwedge S \wedge \gamma, D)$ is a tautology, and since $\gamma \in S$, $\neg\mathcal{E}(\bigwedge S, D)$ is a tautology, completing this case, and the proof. □

Exercises

8.7.1. Give tautologous Herbrand expansions for the following:

1. $(\exists x)[P(x) \supset (\forall y)P(y)]$.
2. $(\exists x)(\forall y)R(x,y) \supset (\forall y)(\exists x)R(x,y)$.
3. $[\mathit{trans} \wedge \mathit{sym} \wedge \mathit{nontriv}] \supset \mathit{ref}$, where terminology is as in Exercise 6.1.2.

8.8 Gentzen's Theorem

We have proved Herbrand's Theorem constructively, using a tableau proof procedure. What about using a Hilbert system, or a natural deduction system? We know that if a sentence X is provable in the Hilbert system of Section 6.5, then X is valid (since the Hilbert system is sound). If X is valid, then X is provable in the tableau system (since the tableau system is complete). And from a tableau proof, we can construct a tautologous Herbrand expansion. But this is quite nonconstructive, making use of both soundness and completeness theorems. Even though we may start with an explicit proof of X in the Hilbert system, all we know from this argument is that there must exist a tableau proof of X—we don't actually have one in front of us. Of course, we could start constructing a tableau for $\neg X$ systematically, and we are guaranteed success, but we don't know how long it will take. Briefly, we are using the fact that X had a Hilbert system proof, but we are making no use of the proof itself.

What we need is a translation procedure: an algorithm to convert a Hilbert system proof into a tableau proof. Combined with the work of the previous section, this would allow us to extract a tautologous Herbrand expansion from a Hilbert system proof. (This was not the route Herbrand took—see Herbrand [25] for a presentation of his approach.) The general idea of such a translation procedure is straightforward. We must show that each of the Hilbert system axioms has a tableau proof, and we must show that, if we have tableau proofs of the premises of a Hilbert system rule, then we can produce a tableau proof of its conclusion. Showing that Hilbert system axioms have tableau proofs is no problem—it is a routine exercise. This leaves the rules of inference, and of these, the Universal Generalization Rule is simple. It reads as follows:

$$\frac{\Phi \supset \gamma(p)}{\Phi \supset \gamma}$$

where p is a parameter that does not occur in $\Phi \supset \gamma$. Because of uniform notation, this is really two rules—we consider the case $\gamma = (\forall x)\varphi(x)$, the other is similar. Suppose we have a tableau proof of $\Phi \supset \varphi(p)$, where p meets the conditions. Such a proof begins as follows:

$$\begin{array}{l}\neg(\Phi \supset \varphi(p))\\ \Phi\\ \neg\varphi(p)\end{array}$$

and continues to closure. Clearly if the α-rule is applied to the first line more than once, the effect is to add Φ and $\neg\varphi(p)$ to branches already containing them, so we may assume the initial rule application to the first line is the only such application. It follows that if we throw away the first line, we have a closed tableau, call it **T**, beginning with Φ and $\neg\varphi(p)$. Now to convert this shortened tableau into one for $\Phi \supset (\forall x)\varphi(x)$, simply begin as follows:

$$\begin{array}{l}\neg(\Phi \supset (\forall x)\varphi(x))\\ \Phi\\ \neg(\forall x)\varphi(x)\\ \neg\varphi(p)\end{array}$$

The last line is added using the δ-rule—the conditions on the Universal Generalization Rule serve to tell us that p is a new parameter at this point. Now we have both Φ and $\neg\varphi(p)$ present, so simply copy underneath the steps of **T**, producing a closed tableau.

This leaves us with one last case, the Rule of Modus Ponens,

$$\frac{X \quad X \supset Y}{Y}$$

and here things are no longer simple. We must convert tableau proofs of X and $X \supset Y$ into a tableau proof of Y. Now a tableau proof of Y begins with $\neg Y$, and so each formula appearing will be a subformula of $\neg Y$ or the negation of one. But X need not be such a formula—indeed, it can be very much longer and more complicated than Y. Consequently, a tableau proof of $X \supset Y$, which begins with $\neg(X \supset Y)$ followed by X and $\neg Y$, can contain many lines that are subformulas of X but not of $\neg Y$, and such lines simply cannot appear in a tableau proof of Y. Similarly for a tableau proof of X itself, which begins with $\neg X$. Then extracting a tableau proof of Y from proofs of X and $X \supset Y$ is no longer just a matter of discarding a few lines and copying others—the whole proof structure must be modified. How to do this is a major insight, discovered by Gentzen at about the same time as Herbrand's work.

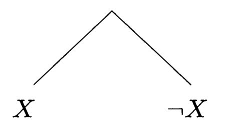
Gentzen added a new rule, which he called "cut," to the sequent calculus of Section 6.6. Several variations are possible in its formulation; here is a simple version.

Sequent Cut Rule

$$\frac{\Gamma \to \Delta, X \quad \Gamma, X \to \Delta}{\Gamma \to \Delta}$$

In this, X is any sentence. Informally, if we can prove a sequent with X on the left and also with X on the right, then X can be "cut" out. Smullyan formulated a version for tableau systems that is particularly simple.

Tableau Cut Rule



In words, at any point in a tableau construction, we can split a branch and introduce the sentences X and $\neg X$.

If the Cut Rule is added to the tableau (or sequent) system, we still have a sound proof procedure. The argument for this is quite elementary. Suppose, for instance, that θ is a satisfiable branch of a tableau, and θ is extended using the Cut Rule, involving the sentences X and $\neg X$. All the sentences on θ are true in some model, and in that model either X or $\neg X$ must be true, so one of the two branches extending θ is satisfiable, in the same model. This is enough to guarantee the system is sound. Since the tableau system was complete without this additional rule, it is still complete. We have soundness and completeness whether the Cut Rule is allowed or not, and thus the same sentences are provable whether Cut is used or not. But, of course, this argument is not constructive—it makes essential use of models.

Gentzen realized two things. First, if the Cut Rule is available, it is easy to convert proofs of X and $X \supset Y$ into a proof of Y, thus completing the constructive argument that Hilbert system proofs can be turned into tableau proofs (but proofs in an extended tableau system). Second, there is an *algorithm* for removing all Cut Rule applications from a proof.

Proposition 8.8.1 *If the Cut Rule is allowed, a tableau proof of X and a tableau proof of $X \supset Y$ can be converted into a tableau proof of Y.*

Proof Assume X and $X \supset Y$ have tableau proofs—say $\mathbf{T}_1$ is a closed tableau for $\neg X$ and $\mathbf{T}_2$ is a closed tableau for $\neg(X \supset Y)$. Now we produce a closed tableau for $\neg Y$ as follows:

$$\begin{tikzcd}
& & \neg Y \arrow[dl, no head] \arrow[dr, no head] & \\
& X \supset Y \arrow[dl, no head] \arrow[dr, no head] & & \neg(X \supset Y) \\
\neg X & & Y & 3 \\
1 & & 2 &
\end{tikzcd}$$

We begin with $\neg Y$, then use the Cut Rule and branch to $X \supset Y$ and $\neg(X \supset Y)$. On the left branch, $X \supset Y$ is a β-sentence, so we branch again to $\neg X$ and Y. Now branch 1 can be continued to closure by simply copying the steps of tableau $\mathbf{T}_1$, which was a closed tableau for $\neg X$. Branch 2 is already closed, because of Y and $\neg Y$. And branch 3 can be continued to closure by copying the steps of tableau $\mathbf{T}_2$, which was a closed tableau for $\neg(X \supset Y)$. □

This was easy. But proving there is a constructive way of eliminating applications of the Cut Rule from proofs is hard. We devote a separate section to it.

8.9 Cut Elimination

Gentzen's algorithm for eliminating Cut Rule applications, together with the proof that the algorithm works, is one of the major foundations of proof theory. The fact that cuts can be eliminated constructively also has consequences for automated theorem proving. Our presentation of his work uses a tableau system, though all the ideas come directly from Gentzen's sequent calculus argument (with some minor terminology changes that are discussed later). His presentation is clear, readable, and recommended [22].

A proof of cut elimination involves consideration of many different cases. In order to keep things as simple as possible, we use uniform notation. There are a few observations about uniform notation that we can make now, rather than presenting them in the middle of the argument.

Suppose X is a formula of the form $A \circ B$, where $\circ$ is one of the primary connectives. If X is $A \wedge B$, X is an α-formula, and $\neg X$ is a β-formula. Likewise, if X is $A \supset B$, X is a β-formula, and $\neg X$ is an α-formula. It is easy to see that no matter which primary connective $\circ$ might be, one of $\{X, \neg X\}$ is an α and one is a β. But this pattern can be carried still further. Suppose again that X is $A \wedge B$, an α-formula, and so $\neg X$

is $\neg(A \wedge B)$, a β-formula. Then $\alpha_1 = X_1$ is A, and $\beta_1 = (\neg X)_1$ is $\neg A$, which is its negation. Likewise, if X is $A \supset B$, a β-formula, and $\neg X$ is $\neg(A \supset B)$, an α-formula, then $\beta_1 = X_1 = \neg A$ and $\alpha_1 = (\neg X)_1 = A$. Again, one is the negation of the other, though this time it is the other way around—the first is the negation of the second. This kind of thing always happens, as can easily be verified by a glance at Table 2.2. We summarize as follows.

Fact 1 Suppose $X = (A \circ B)$, where $\circ$ is a primary connective. Then one of $\{X, \neg X\}$ is an α- and one is a β-formula. Further, of the two first components, α_1 and β_1, one is the negation of the other. Similarly for the two second components, α_2 and β_2, one is the negation of the other.

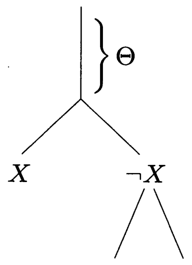
There are similar patterns involving quantified formulas. We state them, and leave verification to you.

Fact 2 Suppose $X = (Qx)\varphi(x)$, where Q is a quantifier, one of $\forall$ or $\exists$. Then one of $\{X, \neg X\}$ is a γ-formula and one is a δ-formula. Further, of the instances $\gamma(t)$ and $\delta(t)$, one is the negation of the other.

Now, the rest of this section is devoted to a proof of the following, which is sometimes referred to as *Gentzen's Hauptsatz*, which simply means Gentzen's Main Theorem.

Theorem 8.9.1 **(Cut Elimination)** *Any closed tableau in which the Cut Rule has been used can be converted into a closed tableau in which there are no Cut Rule applications.*

The idea of the proof in broad outline is simple. We execute a sequence of moves, each of which either reduces the complexity of the sentence involved in some cut, or pushes a cut further down in the tableau, or generally, both. Eventually, all cuts are pushed to the bottom of the tableau, and these simply disappear. Let us begin with this last point. Suppose a closed tableau contains a cut involving the sentence X, and it is at the bottom of at least one of the branches through it. Say part of the tableau looks like this.



That is, we have a cut introducing X and $\neg X$, X comes at the end of its branch, and both forks are closed. We show we have closure even with the cut removed. Incidentally, the situation is similar if the roles of X and $\neg X$ are reversed, using essentially the same argument.

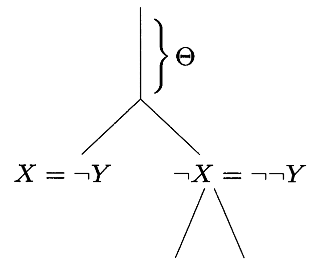
The left fork is closed, so it contains a syntactic contradiction. If X plays no role as part of this contradiction, the contradiction must involve sentences in Θ, and so we will still have closure if the cut is eliminated. Now suppose X does play a role in the closure of the left fork.

There are two ways branches can be closed: because $\perp$ is present or because a sentence and its negation are present. Since we are now assuming X plays a role in the closure of the left fork, X must play one of these roles.

If $X = \perp$, then $\neg X = \neg\perp$, and this can have no function in the closing of the right fork. Consequently, the right fork is closed even if $\neg X$ is removed, so the cut can be eliminated.

Next, suppose X is of the form $(A \circ B)$ where $\circ$ is a binary connective, or X is $(\forall x)\varphi$ or $(\exists x)\varphi$, or X is atomic. Since X plays a role in the closure of its branch, it must be that $\neg X$ occurs as part of Θ. But in this case the cut adds a redundant sentence to the right fork. Consequently if the cut is eliminated, we still have closure.

Finally suppose X is of the form $\neg Y$ for some sentence Y. This case requires somewhat more consideration.



Since X plays a role in branch closure, either Y or $\neg\neg Y$ must occur as part of Θ. If $\neg\neg Y$ is part of Θ, the cut adds a redundant sentence to the right fork and so can be eliminated. Now suppose it is Y that occurs as part of Θ.

If $\neg\neg Y$ is not used in the right fork at all, obviously the cut can be eliminated. If it is used, either the double negation rule is applied to it at one or more places in the subtableau below $\neg\neg Y$, or it is involved directly in the closure of a branch, or both. Since Y is part of Θ, applying the double negation rule to $\neg\neg Y$ adds a redundant sentence, so any such double negation rule application can be dropped. If $\neg\neg Y$ is involved directly in a branch closure, the branch must contain either $\neg Y$ or $\neg\neg\neg Y$. In the first case the branch is closed without using $\neg\neg Y$, since it contains both $\neg Y$ and Y (which is part of Θ). In the second case we can insert an application of the double negation rule to $\neg\neg\neg Y$, adjoining $\neg Y$, and we are reduced to the first case. Thus the tableau can be turned into one in which $\neg\neg Y$ is not used in the right fork, and so the cut can be eliminated.

We are left with the problem of showing that all cuts in a tableau can be "pushed" to branch ends, for which we introduce some special terminology.

Definition 8.9.2 Suppose that in tableau $\mathbf{T}$ there is a cut to sentences X and $\neg X$.

1. We say the cut is *at a branch end* if either there are no sentences below X, or there are no sentences below $\neg X$, or both.
2. The *rank* of the cut is the rank of X (Definitions 2.6.5, 5.1.5).
3. The *weight* of the cut is the number of sentences in $\mathbf{T}$ *strictly below* the cut; that is, the weight is the number of sentences below X plus the number of sentences below $\neg X$.

4. We say a cut is *minimal* if there are no cuts below it in the tableau—that is, if there are no cuts in the subtableaux below X and below $\neg X$.

Incidentally, Gentzen used the notion of "degree" (Exercise 2.2.2) instead of "rank," but the idea is the same no matter what—both measure sentence complexity. (Gentzen used the word *rank* for what we are calling *weight*.) Now, the heart of the proof is the following:

Lemma 8.9.3 *Let* $\mathbf{T}$ *be a closed tableau in which there is a minimal cut of rank* n *and weight* k*, not at the end of a branch. Then* $\mathbf{T}$ *can be transformed into another closed tableau in which the cut has been replaced by cuts of lower rank, by cuts of the same rank but of lower weight, or both.*

Before proving this Lemma, we show that it does yield a proof of Theorem 8.9.1. Intuitively, the idea is simple. To eliminate cuts from a tableau $\mathbf{T}$, keep applying the transformation of Lemma 8.9.3 to minimal cuts, lowering rank or, if not, lowering weight. Eventually, cuts must be transformed into cuts at branch ends (because once cut rank has been reduced to 0, transformations must reduce weight). And cuts at branch ends can be deleted. The formal proof (which amounts to a termination proof for this process) is by a double induction.

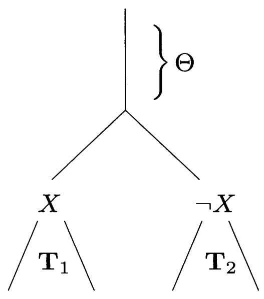
Gentzen's transformation techniques, to be given shortly, are intended to eliminate cuts from tableau proofs. Suppose they are not capable of eliminating *all* cuts. Let $\mathcal{C}$ be the collection consisting of those cuts that the techniques do not allow us to transform away. Then, members of $\mathcal{C}$ are uneliminable cuts, and $\mathcal{C}$ is not empty. Now, among the cuts in $\mathcal{C}$ that are minimal in their tableaux, there must be one or more of smallest rank (say of rank n). Further, among these minimal cuts of rank n in $\mathcal{C}$, there must be one or more of smallest weight (say of weight k). Choose one such cut: in $\mathcal{C}$, minimal in its tableau, of rank n, and of weight k. This cut can not be at a branch end or else it could be eliminated, as we have seen, and this is impossible for cuts in $\mathcal{C}$. But by Lemma 8.9.3, a closed tableau containing a minimal cut of rank n and weight k, not at a branch end, can be transformed into another in which the cut has been replaced by cuts of rank $< n$, by cuts of rank n and weight $< k$, or both. But if we replace the original cut in this way, among the new cuts that result, any that are minimal can be eliminated (since they are either of lower rank than n, which was the smallest rank of cuts in $\mathcal{C}$, or of rank n but of lower weight than k, which was the smallest weight among cuts of rank n in $\mathcal{C}$). Once these minimal cuts have been eliminated, new ones become minimal and likewise can be eliminated, and so on, until all the cuts that replaced the original one have been eliminated. The final outcome is that the original cut itself has been eliminated. This contradicts our assumptions about C, and consequently all cuts must be eliminable.

Before proving the Lemma, there are a few more elementary points about tableaux that will be needed, and it will be simplest to present them now.

Fact 3 Suppose $\mathbf{T}$ is a closed tableau for a finite set S of sentences and $S \subseteq S'$, where S' is also finite. Then there is a closed tableau for S' as well, with essentially the same, and certainly the same number, of steps. Except for one point, this is almost trivial. If the δ-rule is not used in $\mathbf{T}$, then to get a closed tableau for S', just repeat the steps of $\mathbf{T}$, making no use of those sentences of S' that are not in S. But if the δ-rule was used in $\mathbf{T}$, there is a small problem, since a parameter that was new when it was introduced in $\mathbf{T}$ might appear in S' (though not, of course, in S), and so can no longer be considered new. But the solution is simple. First, modify tableau $\mathbf{T}$ by substituting different parameters for those introduced by the δ-rule, making sure to choose parameters that do not appear in S'. This gives an altered tableau, $\mathbf{T}'$, and it is easy to see that $\mathbf{T}'$ is also a closed tableau for S. Now the steps of $\mathbf{T}'$ serve to produce a closed tableau for S', as before.

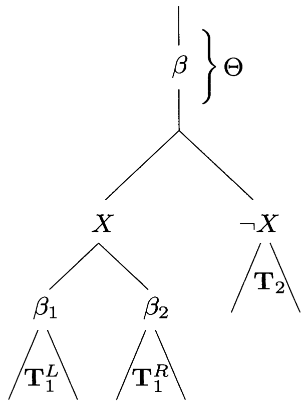
Fact 4 Suppose $\mathbf{T}$ is a closed tableau for the finite set $S \cup \{\delta(c)\}$ of sentences, where c is a parameter that does not occur in S or δ. Then for each closed term t, there is a closed tableau for $S \cup \{\delta(t)\}$ with the same number of steps. This point is a little harder to verify. First, modify $\mathbf{T}$ as we did in the discussion of Fact 3, so that any parameters introduced by δ-rule applications do not occur in t and are distinct from c. This gives us a tableau $\mathbf{T}'$ of the same size and "shape" as $\mathbf{T}$. Now, further modify $\mathbf{T}'$ by replacing every occurrence of c with an occurrence of t, yielding yet another tableau $\mathbf{T}''$. Since c did not occur in S or δ, these do not change under the replacement, so $S \cup \{\delta(c)\}$ simply becomes $S \cup \{\delta(t)\}$, and it is not hard to check that $\mathbf{T}''$ will be a correctly constructed closed tableau for it.

Proof of Lemma 8.9.3 Suppose $\mathbf{T}$ is a tableau containing a minimal cut to X and $\neg X$, not at a branch end, of rank n and weight k. Then part of the tableau has the following form:

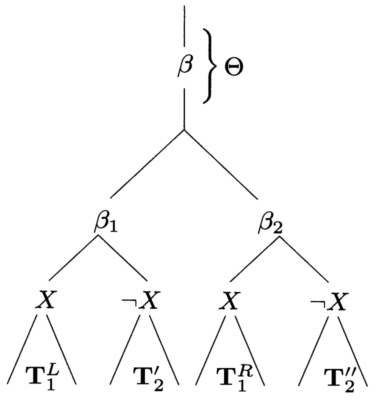


where $\mathbf{T}_1$ and $\mathbf{T}_2$ are the subtableaux below X and $\neg X$, respectively. There are two constructions, depending on whether the uppermost sentences in $\mathbf{T}_1$ or $\mathbf{T}_2$ were obtained by applying a tableau rule to a sentence from Θ, or to X and $\neg X$ themselves. We begin with the case involving a sentence from Θ; this in turn divides into subcases, depending on what kind of a sentence it was. We present the discussion for the left fork, through X—the fork through $\neg X$ is treated similarly.

β-Case Suppose that β is a disjunctive sentence on Θ, and a branch extension rule was applied to it, adding β_1 and β_2 underneath X, thus:

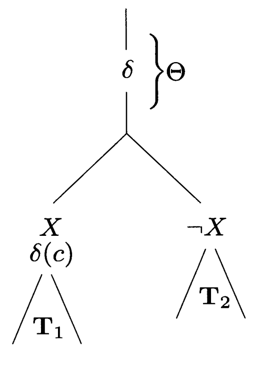


The weight of this cut is $|\mathbf{T}_1^L| + |\mathbf{T}_1^R| + |\mathbf{T}_2| + 2$ (where $|\mathbf{T}_1^L|$ is the number of formulas in $\mathbf{T}_1^L$, and so on). Now replace this part of $\mathbf{T}$ by the following:

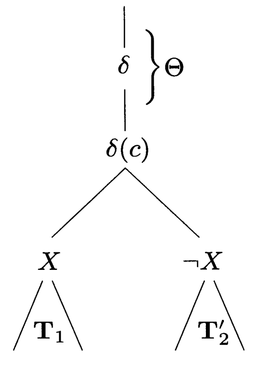


Here the leftmost displayed part (through β_1 and X) is the same as the original leftmost part, except that the occurrences of X and β_1 have been switched around. The next subtableau to the right (through β_1 and $\neg X$) needs some further explanation. We know $\mathbf{T}_2$ provides the steps to close a tableau beginning with $\Theta \cup \{\neg X\}$. Since $\Theta \cup \{\beta_1, \neg X\}$ extends this set, as we noted in Fact 3, $\mathbf{T}_2$ can be converted to $\mathbf{T}_2'$, providing a closed tableau for this larger set—assume that we make this modification. Similarly for the other two branches (leave $\mathbf{T}_1^R$ as is and modify the other occurrence of $\mathbf{T}_2$, as appropriate, into $\mathbf{T}_2''$). The result is a correctly constructed tableau. Now, although the original cut on X has been replaced by two, each involving the same sentence, the weight of each is less. That of the leftmost cut is $|\mathbf{T}_1^L| + |\mathbf{T}_2'| = |\mathbf{T}_1^L| + |\mathbf{T}_2|$ and that of the rightmost one is $|\mathbf{T}_1^R| + |\mathbf{T}_2''| = |\mathbf{T}_1^R| + |\mathbf{T}_2|$.

δ-Case This time suppose that on Θ is an existential sentence δ, and a branch extension rule was applied to it, adding $\delta(c)$ underneath X, where c is new, thus:



Here the weight of the cut is $|\mathbf{T}_1|+|\mathbf{T}_2|+1$. This is simply replaced by the following:

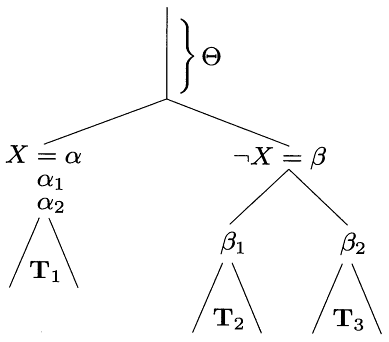


Because c was a new parameter when originally introduced, it is still new in the modified tableau (since it is introduced earlier). The left subtableau is still correctly constructed, since the only change here was to reverse the positions of X and $\delta(c)$. In the original tableau, $\mathbf{T}_2$ provided closure for the set $\Theta \cup \{\neg X\}$, and we now have the larger set $\Theta \cup \{\delta(c), \neg X\}$, but $\mathbf{T}_2$ can be modified, as described in Fact 3, into $\mathbf{T}'_2$, to provide closure for this enlarged set. Now the weight of the cut in this new tableau is $|\mathbf{T}_1| + |\mathbf{T}'_2| = |\mathbf{T}_1| + |\mathbf{T}_2|$, and thus is smaller than the weight of the original cut.

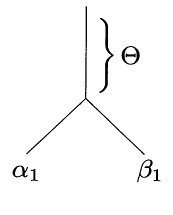
α-, γ-, and Negation Cases These are similar to the δ-case just discussed, and are left to you.

This ends the discussion of what modifications to make if the first sentences in $\mathbf{T}_1$ (or in $\mathbf{T}_2$) arise from the application of a rule to a sentence in Θ. Now we turn to the case where they arise by applying rules to X and $\neg X$ themselves, and again we have several subcases, depending on the form of X.

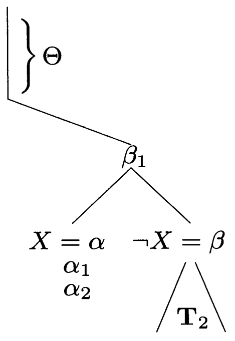
Primary Connective Case Suppose the cut sentence X is of the form $A \circ B$, where $\circ$ is a Primary Connective, and the first rule applied on each fork of the cut is to the sentences introduced by the cut. By Fact 1, one of $\{X, \neg X\}$ is an α-sentence, the other is a β-sentence. Say X is the α and $\neg X$ is the β—the argument is symmetrical. Thus, we suppose the cut looks like this:



The weight of this cut is $|\mathbf{T}_1| + |\mathbf{T}_2| + |\mathbf{T}_3| + 4$. Recall, from Fact 1 again, that one of $\{\alpha_1, \beta_1\}$ is the negation of the other. Now the first step in modifying the tableau is to replace the cut to X and $\neg X$ by one to α_1 and β_1.

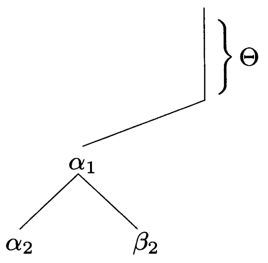


This is a cut of lower rank. Next we say how to continue each branch, and we begin with the right one, since it is simpler. To keep the pictures uncluttered, *we display only the right branch for now.* And on it we begin with another cut, to X and $\neg X$ again!

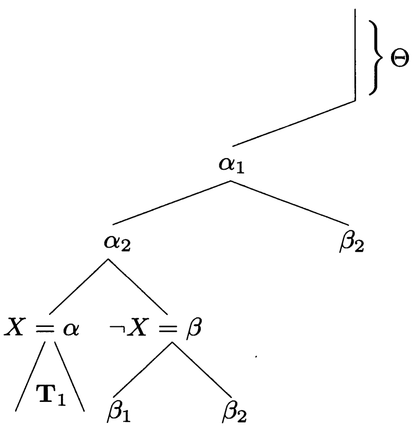


We have applied a Branch Extension Rule to the α-sentence, adding α_1 and α_2, closing the branch because of the presence of α_1 and β_1 (one of which is the negation of the other). Underneath $\neg X = \beta$ we have reproduced $\mathbf{T}_2$ from the original tableau—this is still correct, since the same sentences are present above it, though in a different order. The weight of this new cut to X and $\neg X$ is $|\mathbf{T}_2| + 2$, which is smaller than the weight of the original cut.

Now we turn to the left-hand fork of the cut to α_1 and β_1, and here we continue with a cut to α_2 and β_2 (again by Fact 1, one of these is the negation of the other). This too is a cut of lower rank.

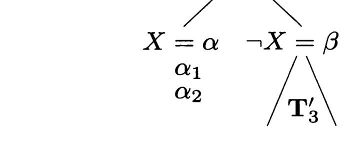


Finally, we say how to continue each of the forks of this cut to α_2 and β_2, and again, each of them begins with a cut to X and $\neg X$! We begin by displaying the fork through α_2.



We have applied to $\neg X$ the β-Branch Extension Rule. Each branch below it is closed because of the presence of α_1 and β_1, or α_2 and β_2, respectively (in each case, one is the negation of the other). Below $X = \alpha$ we copy $\mathbf{T}_1$, still appropriate because the branch above it contains the same sentences that were above $\mathbf{T}_1$ in the original tableau. The weight of this cut to X and $\neg X$ is $|\mathbf{T}_1| + 2$, again a lower weight than that of the original.

Finally, we display the fork through β_2.

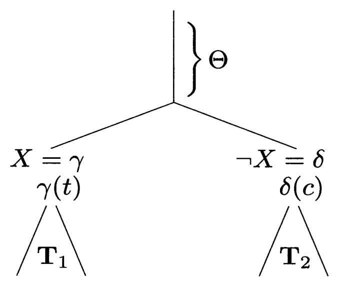


The branch through X is closed because of α_2 and β_2. The branch through $\neg X$ continues with $\mathbf{T}'_3$, which is a modification of $\mathbf{T}_3$ ($\mathbf{T}_3$

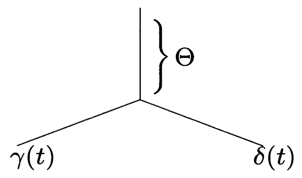
was for $\Theta \cup \{\beta, \beta_2\}$ and we now have the larger set $\Theta \cup \{\alpha_1, \beta_2, \beta\}$, so Fact 3 comes in.) The weight of the new cut to X and $\neg X$ is $|\mathbf{T}_3| + 2$, so it too is less than the weight of the original cut. This concludes the Primary Connective case.

Other Propositional Cases If X is $\neg\neg Z$, $\top$, $\neg\top$, $\bot$, or $\neg\bot$, the construction is simpler than that above, and we leave the details to you.

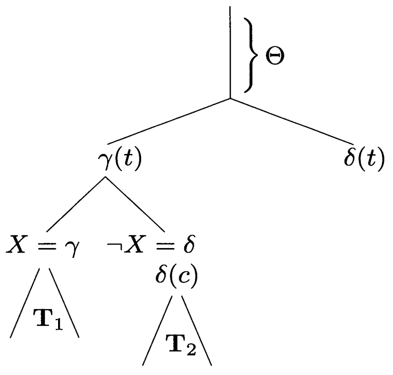
Quantifier Case Suppose the cut sentence X is a quantified sentence, and the first rule applied on each fork of the cut is to the sentences introduced by the cut. By Fact 2, one of $\{X, \neg X\}$ is a γ-sentence and the other is a δ-sentence. Say X is the γ and $\neg X$ is the δ—as in the propositional case, the argument is symmetric. So we suppose the cut looks like this:



where t is some closed term, and c is a parameter that is new at the point it is introduced. The weight of this cut is $|\mathbf{T}_1| + |\mathbf{T}_2| + 2$. Recall by Fact 2 that one of $\{\gamma(t), \delta(t)\}$ is the negation of the other. The tableau modifications begin by replacing the original cut by one to $\gamma(t)$ and $\delta(t)$, which is a cut of lower rank. Thus we start as follows:



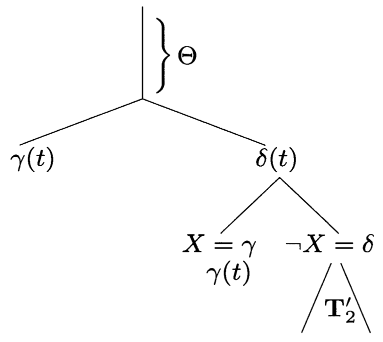
Now we continue each branch, and we start with the construction for the left one, beginning with a cut, to X and $\neg X$ again.



Under $X = \gamma$, the subtableau $\mathbf{T}_1$ has been copied over—justified because the branch above it contains the same sentences that were in the original tableau above $\mathbf{T}_1$. Under $\neg X = \delta$, we display $\mathbf{T}_2$ copied over, but this may not be quite right since the parameter c, which was new when introduced into the original tableau, might not be new when introduced in the modified tableau, since it could appear in t. The solution is to further modify the subtableau from $\delta(c)$ down, in accordance with Fact 3. This yields a correct closed tableau construction of the same size and shape. The weight of this cut to X and $\neg X$ is $|\mathbf{T}_1| + |\mathbf{T}_2| + 1$, which is lower than the weight of the original cut.

Now we turn to the branch through $\delta(t)$, which requires the most work. The right-hand fork of the original cut, through $\neg X = \delta$, contained the sentences of Θ, δ, $\delta(c)$, and finally the subtableau $\mathbf{T}_2$. Note that the parameter c was new when introduced in $\delta(c)$. According to Fact 4, $\mathbf{T}_2$ can be modified to yield a correctly constructed closed subtableau in which the closed term t takes the place of c. Let $\mathbf{T}_2'$ be such a modification. Then $\mathbf{T}_2'$ provides closure for a branch that begins with Θ, δ, and $\delta(t)$.

The fork of the cut through $\delta(t)$ is continued as follows:

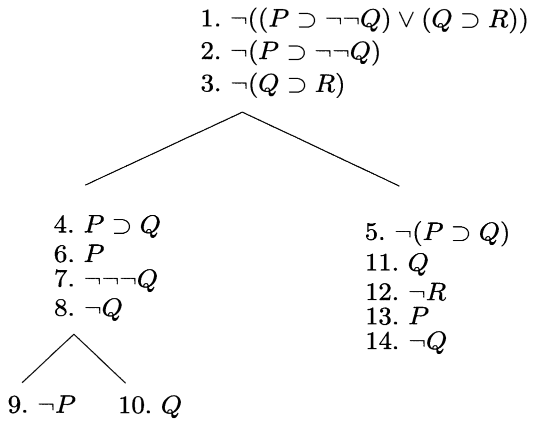


A Branch Extension Rule is applied on the fork through $X = \gamma$, and the branch closes because it contains $\gamma(t)$ and $\delta(t)$ (one is the negation of the other, by Fact 2). The right fork closes because of what was said earlier. The weight of this cut is $|\mathbf{T}'_2| + 1 = |\mathbf{T}_2| + 1$, which is less than the weight of the original cut to X and $\neg X$. This concludes the quantifier case.

And this concludes the proof. □

Exercises

8.9.1. Following is a propositional tableau containing a cut. In it, 2 and 3 are from 1 by α; 4 and 5 constitute the cut; 6 and 7 are from 2 by α; 8 is from 7 by double negation; 9 and 10 are from 4 by β; 11 and 12 are from 3 by α; and 13 and 14 are from 5 by α.



Follow the procedure given in this section and transform the tableau into a cut-free proof.

8.10 Do Cuts Shorten Proofs?

Allowing cuts in tableau proofs is important from the point of view of automated theorem proving. Without cuts, each tableau proof starts from a state of zero knowledge—previously constructed proofs play no role. But with cuts, known theorems can serve as lemmas for further work, in the way mathematicians normally expect. Suppose that we have already proved a sentence X, and now we are trying to prove a different sentence, Y. In a tableau for $\neg Y$, if cut is an allowed rule, we can split a branch at any point, adding $\neg X$ to one fork and X to the other. Underneath $\neg X$, we can simply copy the closed tableau for $\neg X$ that we already have, thus closing that fork. (In practice, we would just refer to where a proof of X can be found, rather than reproducing it.) This leaves us the other fork to work with, and we have X available on it, which may help us in producing a closed branch. The net effect is that we have added the previously proved sentence X to an open branch of the tableau we are constructing for $\neg Y$, thus making X available as a lemma. Cut elimination tells us that if we discover a proof using lemmas in this way, it can be converted into one that does not use them. This leads to a natural question: Can such use of lemmas, or cuts, shorten proofs?

Injudicious use of cuts can, of course, sometimes lengthen proofs, not shorten them. In Section 4.1 we gave a Hilbert system proof of the propositional formula $P \supset P$. That can be turned into a tableau proof with cuts, as outlined in Section 8.8, yielding a lengthy and complex tableau. But if cuts are eliminated, following the procedure in Section 8.9, the result must be the tableau that begins with $\neg(P \supset P)$, applies the α-rule, and closes immediately—since this is the *only* closed cut-free tableau for $\neg(P \supset P)$. In this case, eliminating cuts is a great simplification.

The example is misleading, however. The short tableau proof of $P \supset P$ is also a proof in a tableau system allowing cuts—we simply didn't happen to apply the cut rule. The real question is not whether (mis)use of cuts can make things worse, but whether wise use of them can ever make things better. And to this question the answer is most emphatically yes—cuts can make the difference between linear and exponential! Boolos [6] gives an intuitively clear example showing this, but it involves not only quantification but also equality. Here we give a somewhat more complex example that is entirely propositional, due to Statman [50] and simplified by Takeuti and Buss.

Theorem 8.10.1 *For each $n = 1, 2, \ldots$ there is a finite unsatisfiable set S_n of propositional formulas such that:*

1. *There is a closed tableau for S_n, allowing cuts, of weight $12n - 4$, hence of weight linear in n.*

2. *The shortest cut-free closed tableau for S_n has weight $\geq 2^n$.*

We devote the rest of the section to a proof of this theorem. From now on, let a_1, a_2,... and b_1, b_2,... be two disjoint lists of distinct propositional letters. Using them, we define three families of more complex formulas.

First, for each $n = 1, 2, \ldots$ let

$$F_n = (\cdots(((a_1 \vee b_1) \wedge (a_2 \vee b_2)) \wedge (a_3 \vee b_3)) \wedge \cdots \wedge (a_n \vee b_n)).$$

F_n intuitively says that for each i from 1 to n we must have one of a_i or b_i. The definition can also be given recursively, as follows:

$$\begin{aligned} F_1 &= a_1 \vee b_1 \\ F_{n+1} &= F_n \wedge (a_{n+1} \vee b_{n+1}) \end{aligned}$$

Semantically, F_n is equivalent to $\langle [a_1, b_1], \ldots, [a_n, b_n] \rangle$, using notation from Chapter 2. This may help with motivation, though it can play no role in our proof because there are no tableau rules for clauses and dual clauses. It also suggests a natural definition for F_0, namely the empty conjunction $\langle\ \rangle$, semantically equivalent to $\top$. We do not use F_0, though the proof could be revised in minor ways to do so. We start with $n = 1$.

Next, we set $A_1 = a_1$, $B_1 = b_1$, and for each $n = 1, 2, \ldots$:

$$\begin{aligned} A_{n+1} &= F_n \supset a_{n+1} \\ B_{n+1} &= F_n \supset b_{n+1} \end{aligned}$$

If we allow the use of F_0, A_1 turns out to be equivalent to $F_0 \supset a_1$, and similarly for B_1. Both of these follow the general pattern.

Notice that each A_n intuitively says that, if we have one of a_i or b_i for each i from 1 to $n - 1$, then we have a_n, and B_n says a similar thing concerning b_n. Consequently, $A_n \vee B_n$ intuitively says that, if we have one of a_i or b_i for all i up to n, then the pattern continues one more step, and we have one of a_n or b_n as well.

Finally, for $n = 1, 2, \ldots$, let S_n be the set

$$\{A_1 \vee B_1, A_2 \vee B_2, \ldots, A_n \vee B_n, A_{n+1} \vee B_{n+1}, \neg a_{n+1}, \neg b_{n+1}\}.$$

Informally, each S_n is unsatisfiable by the following argument. Since $A_1 \vee B_1$ is present, we have one of a_1 or b_1. Then by the informal meaning of $A_2 \vee B_2$, we have one of a_2 or b_2 as well. Since we have one of a_1 or b_1 and also one of a_2 or b_2, and $A_3 \vee B_3$ is present, we have one of a_3 or b_3, too. This pattern continues until finally we must have one of a_{n+1} or b_{n+1}, but S_n explicitly says we don't have either. This informal reasoning can be turned into a mathematically correct semantical argument, using

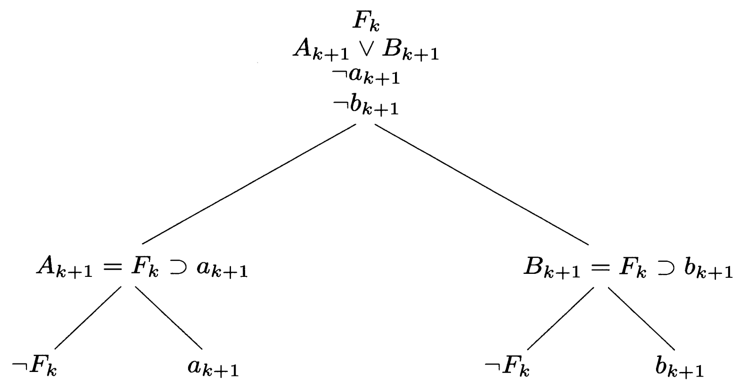


FIGURE 8.1. Tableau for Lemma 8.10.2

induction, to show the unsatisfiability of S_n. Then, by the completeness of the propositional tableau rules, there must be a closed tableau for each S_n whether cuts are allowed or not.

The rest of this section is devoted to our promised proof, using induction, that:

1. There is a closed tableau of weight $12n - 4$ for S_n, allowing cuts;
2. Every closed tableau for S_n that does not allow cuts has weight at least 2^n.

This, of course, establishes that cut elimination can introduce an exponential blowup. Now for the details.

Lemma 8.10.2 *Independently of k there is a closed, cut-free tableau for $\{F_k, A_{k+1} \vee B_{k+1}, \neg a_{k+1}, \neg b_{k+1}\}$ of weight 6.*

Proof The tableau is shown in Figure 8.1. In it, there are three β-rule applications. □

Lemma 8.10.3 *Again independently of k, there is a closed, cut-free tableau for $\{F_k, A_{k+1} \vee B_{k+1}, \neg F_{k+1}\}$ of weight 10.*

Proof The tableau is given in Figure 8.2. □

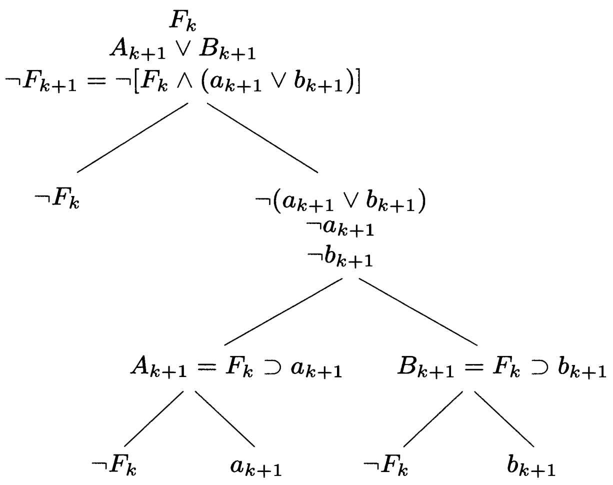

F_k
$A_{k+1} \vee B_{k+1}$
$\neg F_{k+1} = \neg[F_k \wedge (a_{k+1} \vee b_{k+1})]$
$\neg F_k$
$\neg(a_{k+1} \vee b_{k+1})$
$\neg a_{k+1}$
$\neg b_{k+1}$
$A_{k+1} = F_k \supset a_{k+1}$
$B_{k+1} = F_k \supset b_{k+1}$
$\neg F_k$
a_{k+1}
$\neg F_k$
b_{k+1}

FIGURE 8.2. Tableau for Lemma 8.10.3

Proposition 8.10.4 *There is a closed tableau for S_n, with cuts, of weight* $12n - 4$.

Proof The tableau, schematically, is given in Figure 8.3. In it there are n cuts on F_n, $F_{n-1}, \ldots, F_1$, contributing $2n$ formulas to the tableau weight. Underneath F_n appears a subtableau denoted $\mathbf{U}$, which is the closed tableau of weight 6 for $\{F_n, A_{n+1} \vee B_{n+1}, \neg a_{n+1}, \neg b_{n+1}\}$ from Lemma 8.10.2. Underneath F_{n-1} is a subtableau denoted $\mathbf{T}_{n-1}$, which is the closed tableau of weight 10 for $\{F_{n-1}, A_n \vee B_n, \neg F_n\}$ whose existence is given by Lemma 8.10.3, and similarly for $\mathbf{T}_{n-2}$ below F_{n-2}, and so on. The subtableaux $\mathbf{T}_1, \ldots, \mathbf{T}_{n-1}$ contribute a total of $10(n-1)$ formulas. Note that the rightmost branch is directly closed, since $\neg F_1 = \neg(a_1 \vee b_1)$ and $A_1 \vee B_1 = a_1 \vee b_1$ both appear on it. Thus, we have a closed tableau of weight $2n + 6 + 10(n-1) = 12n - 4$. □

We have now established the first half of Theorem 8.10.1 and turn to the second. Recall the term *strict* (Definition 3.1.4) which designates a tableau in which no propositional rule is applied to a formula more than once on any branch. A careful look at the completeness proof for the tableau system shows that strict tableaux are enough. The following makes this more concrete—we leave its proof as an exercise:

Lemma 8.10.5 *If there is a closed non-strict tableau of weight n for a set S, there is also a closed strict tableau for S whose weight is $< n$.*

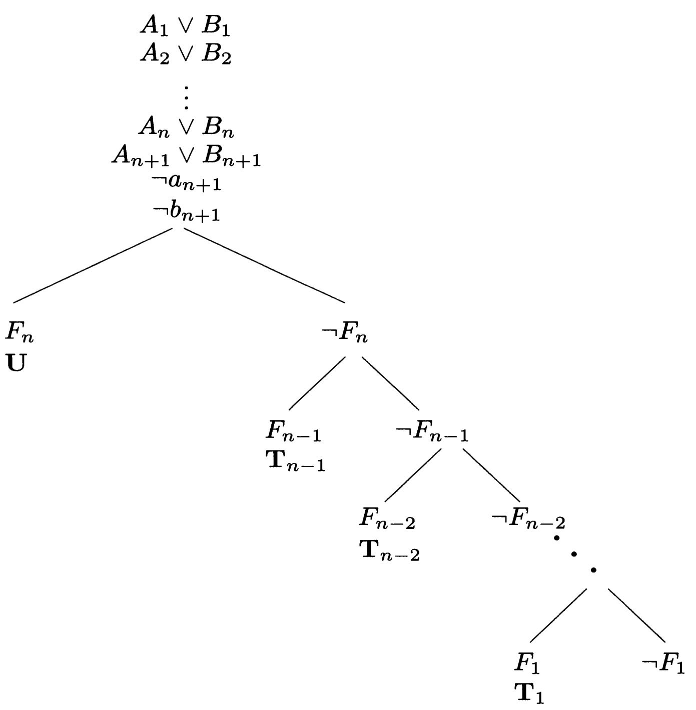


FIGURE 8.3. Tableau for Proposition 8.10.4

In investigating which cut-free tableau proofs are shortest, we can confine our attention to the strict ones, because of this lemma.

We say a formula X is *directly involved in a branch closure* in tableau $\mathbf{T}$ if some branch of $\mathbf{T}$ closes because it contains a formula and its negation, and X is one of these two formulas. Now we state and prove the chief tool for what follows:

Lemma 8.10.6

1. *If there is a closed strict tableau* $\mathbf{T}$ *of weight* n *for the set* $S \cup \{\beta\}$*, and* β *is not directly involved in a branch closure in* $\mathbf{T}$*, then there is a closed strict tableau for the set* $S \cup \{\beta_1\}$ *of weight* $\leq n$*, and similarly for* $S \cup \{\beta_2\}$*.*
2. *If there is a closed strict tableau* $\mathbf{T}$ *of weight* n *for* $S \cup \{\alpha\}$*, and* α *is not directly involved in a branch closure in* $\mathbf{T}$*, then there is a closed strict tableau for the set* $S \cup \{\alpha_1, \alpha_2\}$ *of weight* $\leq n$*.*

Proof Suppose $\mathbf{T}$ is a closed strict tableau for $S \cup \{\beta\}$. If no rule is applied to β in $\mathbf{T}$, then just replace β by β_1 throughout $\mathbf{T}$, and make no other changes. This gives a closed strict tableau for $S \cup \{\beta_1\}$ of the same weight. Otherwise, say a rule is applied to β on at least one branch of $\mathbf{T}$, so there are parts of $\mathbf{T}$ of the following form:

$$\begin{tikzcd}[column sep=small] & {} \arrow[d, dash] & \\ & S \cup \{\beta\} \arrow[dl, dash] \arrow[dr, dash] & \\ \beta_1 & & \beta_2 \\ \mathbf{T}_1 & & \mathbf{T}_2 \end{tikzcd}$$

Here, $\mathbf{T}_1$ represents the subtableau beneath β_1; and $\mathbf{T}_2$, the subtableau beneath β_2. Now, replace this by the following:

$$\begin{tikzcd} {} \arrow[d, dash] \\ S \cup \{\beta_1\} \arrow[d, dash] \\ \mathbf{T}_1 \end{tikzcd}$$

The result contains at least one fewer formula (β_2). Do a similar thing throughout $\mathbf{T}$ for each branch on which a rule was applied to β, producing a strict tableau for $S \cup \{\beta_1\}$ of smaller weight, a tableau that is still closed because β itself played no direct role in the closure of any branch.

Part 2 has a similar, and simpler, proof. □

We now come to the central result.

Proposition 8.10.7 *Suppose a shortest closed cut-free tableau for S_n has weight $\geq k$. Then a shortest closed cut-free tableau for S_{n+1} has weight $\geq 2k$.*

Proof Let $\mathbf{T}$ be a shortest closed cut-free tableau for $S_{n+1} = \{A_1 \vee B_1, A_2 \vee B_2, \ldots, A_{n+1} \vee B_{n+1}, A_{n+2} \vee B_{n+2}, \neg a_{n+2}, \neg b_{n+2}\}$. By Lemma 8.10.5 we can assume $\mathbf{T}$ is strict. Since a_{n+2} and b_{n+2} are propositional letters, $\mathbf{T}$ must begin with a β-rule application, so $\mathbf{T}$ looks like the following:

$$\begin{tikzcd}
& A_1 \lor B_1 & \\
& A_2 \lor B_2 & \\
& \vdots & \\
& A_{n+1} \lor B_{n+1} & \\
& A_{n+2} \lor B_{n+2} & \\
& \neg a_{n+2} & \\
& \neg b_{n+2} \arrow[dl, no head] \arrow[dr, no head] & \\
A_i & & B_i \\
\mathbf{T}_1 & & \mathbf{T}_2
\end{tikzcd}$$

Here $\mathbf{T}_1$ and $\mathbf{T}_2$ denote the subtableaux below A_i and B_i, respectively. Now we show $|\mathbf{T}_1| \geq k$ and $|\mathbf{T}_2| \geq k$ (recall, $|\mathbf{U}|$ is the number of formulas in $\mathbf{U}$). From this it follows that the weight of the tableau $\mathbf{T}$ for S is $\geq 2k$. The argument for $\mathbf{T}_2$ is similar to that for $\mathbf{T}_1$, which we present. The idea is to show that the left fork can be modified into a shorter strict closed tableau for S_n. Since we know the weight of such a tableau must be $\geq k$, the original left fork must also have weight $\geq k$.

Much of the argument involves the use of Lemma 8.10.6, and to be able to apply it we must know that certain formulas are not directly involved in branch closures. We begin by verifying this is so for all the cases we need, so that the heart of the argument will not be cluttered with such details. Fortunately, the idea is always the same. If a formula X has only positive occurrences in a tableau, X obviously can not be directly involved in a branch closure. But if X occurs only in positive positions in formulas of a set S, Exercise 8.10.3 says X can occur only in positive positions in the formulas of a tableau for S. Combining these facts: if X occurs only in positive positions in S, X cannot be directly involved in a branch closure in any tableau for S. Of course, the same result holds if X occurs only negatively in S. Now for the cases we need.

First we argue that for $h = 1, 2, \ldots, n{+}2$, $A_h \lor B_h$ is not directly involved in a branch closure in $\mathbf{T}$. $A_h \lor B_h$ can't be a subformula of $\neg a_{n+2}$ or $\neg b_{n+2}$ because these are literals. If it were a subformula of $A_k \lor B_k$ for $k \neq h$, it would be a subformula of one of A_k or B_k, but both of these are implications if $k > 1$, or atomic if $k = 1$, so this is impossible. It follows that the only occurrence of $A_h \lor B_h$ in any formula of S_{n+1} is its occurrence explicitly as a member of S_{n+1}. Conseqently, $A_h \lor B_h$ occurs only positively, and so cannot be directly involved in a branch closure in $\mathbf{T}$.

Next, for $h = 2, 3, \ldots, n + 2$, neither A_h nor B_h is directly involved in a branch closure in $\mathbf{T}$. We argue for A_h; the argument for B_h is similar. Since $h \geq 2$, A_h is an implication, so it cannot be a subformula of $\neg a_{n+2}$

or $\neg b_{n+2}$. Now suppose A_h is a subformula of $A_j \vee B_j$. Since this is a disjunction and A_h is an implication, A_h must be a proper subformula, and hence is a subformula of A_j or of B_j. If it is a subformula of A_j, it cannot be a *proper* subformula—there are two cases to consider. If $j = 1$, A_j is atomic, but A_h is not. If $j > 1$, $A_j = F_{j-1} \supset a_j$, so A_h would be a subformula of F_{j-1}, or of a_j. The first of these is impossible since A_h is an implication, but F_{j-1} contains no implication symbols. The second is impossible because a_j is atomic. We leave it to you to argue that A_h cannot be a subformula of B_j at all. The only possibility that is left is that A_h must be a subformula of A_j, but not proper—in other words, $A_h = A_j$, and so $h = j$. Thus the only occurrence of A_h in S_{n+1} is its occurrence in $A_h \vee B_h$, and this is a positive one. Then A_h cannot be directly involved in a branch closure in $\mathbf{T}$.

Next we claim that for $h \geq 2$, $a_h \vee b_h$ occurs only negatively in S_{n+1} and so it (and likewise its negation) cannot be directly involved in a branch closure in $\mathbf{T}$. Certainly, $a_h \vee b_h$ cannot be a subformula of $\neg a_{n+2}$ or of $\neg b_{n+2}$, since these are literals. Because $h \geq 2$, $a_h \vee b_h$ cannot be a subformula of $A_1 \vee B_1$. If $a_h \vee b_h$ is a subformula of $A_j \vee B_j$ for some $j \geq 2$, it must be a proper subformula, and so is a subformula of A_j or of B_j—say of A_j, since the argument is similar either way. Since $A_j = F_{j-1} \supset a_j$ is an implication, the disjunction $a_h \vee b_h$ must be a proper subformula of it. It cannot be a subformula of a_j, and if it is a subformula of F_{j-1}, it must be one of the conjuncts of it, and consequently it must be a positive subformula of F_{j-1}. But then that occurrence of $a_h \vee b_h$ is in a negative position in $A_j = F_{j-1} \supset a_j$, and so also in $A_j \vee B_j$.

Finally, F_{n+1} is obviously a subformula of both $A_{n+2} = F_{n+1} \supset a_{n+2}$ and $B_{n+2} = F_{n+1} \supset b_{n+2}$, but it is easy to see it is not a subformula of any member of S_{n+1} besides $A_{n+2} \vee B_{n+2}$. Since F_{n+1} appears only negatively in $A_{n+2} \vee B_{n+2}$, neither it nor its negation can be directly involved in a branch closure in $\mathbf{T}$.

With all this out of the way, we now turn to the main argument—showing the left branch of $\mathbf{T}$ can be shortened into a strict closed tableau for S_n. Since $\mathbf{T}$ is strict and the β rule has been applied to $A_i \vee B_i$ at the start, this formula never again has a rule applied to it—in particular, not in $\mathbf{T}_1$. Also, as shown above, $A_i \vee B_i$ cannot play a direct role in the closure of a branch. Consequently, if $A_i \vee B_i$ is removed, $\mathbf{T}_1$ constitutes a correctly constructed, strict, closed tableau for the set $\{A_1 \vee B_1, \ldots, A_{i-1} \vee B_{i-1}, A_{i+1} \vee B_{i+1}, \ldots, A_{n+2} \vee B_{n+2}, \neg a_{n+2}, \neg b_{n+2}, A_i\}$.

$$\begin{array}{c} A_1 \vee B_1 \\ A_2 \vee B_2 \\ \vdots \\ A_{i-1} \vee B_{i-1} \\ A_{i+1} \vee B_{i+1} \\ \vdots \\ A_{n+2} \vee B_{n+2} \\ \neg a_{n+2} \\ \neg b_{n+2} \\ A_i \\ \mathbf{T}_1 \end{array}$$

Now there are two cases, depending on whether $i = n + 2$ or not.

Case 1 ($i = n + 2$) In this case $\mathbf{T}_1$ is a closed strict tableau for the set $\{A_1 \vee B_1, A_2 \vee B_2, \ldots, A_{n+1} \vee B_{n+1}, \neg a_{n+2}, \neg b_{n+2}, A_{n+2}\}$.

$A_{n+2} = F_{n+1} \supset a_{n+2}$, and A_{n+2} plays no direct role in closing any branch, so by Lemma 8.10.6 there is a closed strict tableau, of no greater weight, for $\{A_1 \vee B_1, A_2 \vee B_2, \ldots, A_{n+1} \vee B_{n+1}, \neg a_{n+2}, \neg b_{n+2}, \neg F_{n+1}\}$; that is, of weight $\leq |\mathbf{T}_1|$. But $\neg F_{n+1} = \neg[F_n \wedge (a_{n+1} \vee b_{n+1})]$ also plays no direct role in closing any branch, so by Lemma 8.10.6 once again, there is a closed strict tableau for $\{A_1 \vee B_1, A_2 \vee B_2, \ldots, A_{n+1} \vee B_{n+1}, \neg a_{n+2}, \neg b_{n+2}, \neg(a_{n+1} \vee b_{n+1})\}$ of no greater weight, that is, of weight $\leq |\mathbf{T}_1|$. The propositional letter a_{n+2} does not occur in any formula of this set except for $\neg a_{n+2}$, and similarly for b_{n+2}; consequently neither can play a role in the tableau construction or closure. It follows that if we remove them from the tableau, we will have a closed strict tableau for $\{A_1 \vee B_1, A_2 \vee B_2, \ldots, A_{n+1} \vee B_{n+1}, \neg(a_{n+1} \vee b_{n+1})\}$ of weight $\leq |\mathbf{T}_1|$. Finally, it follows, from Lemma 8.10.6 again, that there is a closed strict tableau for $\{A_1 \vee B_1, A_2 \vee B_2, \ldots, A_{n+1} \vee B_{n+1}, \neg a_{n+1}, \neg b_{n+1}\}$ of weight $\leq |\mathbf{T}_1|$. But this is the set S_n, and the smallest closed strict tableau for it has weight $\geq k$. Consequently, $|\mathbf{T}_1| \geq k$.

Case 2 ($i \leq n + 1$) In this case $\mathbf{T}_1$ is a closed tableau for the set $\{A_1 \vee B_1, \ldots, A_{i-1} \vee B_{i-1}, A_{i+1} \vee B_{i+1}, \ldots, A_{n+2} \vee B_{n+2}, \neg a_{n+2}, \neg b_{n+2}, A_i\}$. If $i \geq 2$, $A_i = F_{i-1} \supset a_i$ plays no direct role in branch closure, so by Lemma 8.10.6, there is a closed strict tableau of no greater weight for this set with A_i removed and a_i added. If $i = 1$, A_i itself is a_1. Either way there is a closed strict tableau, call it $\mathbf{T}'_1$, for the set $\{A_1 \vee B_1, \ldots, A_{i-1} \vee B_{i-1}, A_{i+1} \vee B_{i+1}, \ldots, A_{n+2} \vee B_{n+2}, \neg a_{n+2}, \neg b_{n+2}, a_i\}$, with $|\mathbf{T}'_1| \leq |\mathbf{T}_1|$.

We modify this tableau, telling it to forget all about $a_i \vee b_i$. The details are slightly more complicated if $i = 1$, so first suppose $i \geq 2$. We defined F_k to be the conjunction of the formulas $a_1 \vee b_1$, $a_2 \vee b_2, \ldots,$ $a_k \vee b_k$

(associated to the right). Let F_k^* be the same conjunction, but with $a_i \vee b_i$ removed. Then of course $F_i^* = F_{i-1}$, and if $k < i$ then $F_k^* = F_k$.

Let m be any integer from 1 to $n+2$ except i. We say how to *m-modify* the tableau, which intuitively means removing $a_i \vee b_i$ from $A_m \vee B_m$ and all formulas that derive from it in the tableau. (We don't consider $m = i$ because $A_i \vee B_i$ is missing.) A bit of notation first. Recall, $A_m = F_{m-1} \supset a_m$, except for $m = 1$, and $A_1 = a_1$; and similarly for B_m. We define $A_m^* = F_{m-1}^* \supset a_m$, except for $m = 1$, and $A_1^* = A_1 = a_1$; and similarly for B_m^*.

Here is the m-modification process. In the tableau, where $A_m \vee B_m$ appears, replace it by $A_m^* \vee B_m^*$. Next, in the resulting tableau, wherever the β-rule was applied to $A_m \vee B_m$, branching to A_m and B_m, replace it by an application of β to $A_m^* \vee B_m^*$, branching to A_m^* and B_m^*. Then, where a β-rule was applied to $A_m = F_{m-1} \supset a_m$, branching to $\neg F_{m-1}$ and a_m, replace it by an application of β to $A_m^* = F_{m-1}^* \supset a_m$, branching to $\neg F_{m-1}^*$ and a_m. Again similarly for B_m. Where a β-rule was applied to $\neg F_k = \neg(F_{k-1} \wedge (a_k \vee b_k))$, branching to $\neg F_{k-1}$ and $\neg(a_k \vee b_k)$, *if* $k \neq i$, replace this by a β-rule application to $\neg F_k^* = \neg(F_{k-1}^* \wedge (a_k \vee b_k))$, branching to $\neg F_{k-1}^*$ and $\neg(a_k \vee b_k)$. But if a β-rule was applied to $\neg F_i = \neg[F_{i-1} \wedge (a_i \vee b_i)]$, branching to $\neg F_{i-1}$ and $\neg(a_i \vee b_i)$, in the revised tableau $\neg F_i$ will have been replaced by $\neg F_i^* = \neg F_{i-1}$. Simply remove the right branch, beginning with $\neg(a_i \vee b_i)$, and keep the left branch that begins with $\neg F_{i-1}$.

We have described the m-modification process under the assumption that $i \neq 1$. In case $i = 1$, small changes are necessary. $A_1 \vee B_1$ will not be present, since the formula $A_i \vee B_i$ has been removed, but A_2 and B_2 are slightly problematic. Since $F_1 = (a_1 \vee b_1)$ we can not simply delete $a_1 \vee b_1$ from F_1 to produce F_1^*, since this would eliminate everything. But $A_2 = F_1 \supset a_2$, so we can take A_2^* to be a_2 (recall, the empty conjunction is semantically true); similarly we take B_2^* to be b_2. Now, in making an m-modification, where $m = 2$, A_2 is replaced by A_2^*, and any β-rule application to A_2, branching to $\neg F_1$ and a_2 has its left branch through $\neg F_1$ discarded, and just the branch through a_2 retained. Similarly for B_2. Everything else is as it was above.

Now carry the m-modification process out for $m = 1$, starting with $\mathbf{T}_1'$, then for $m = 2$, starting with the result, and so on (skipping $m = i$), up to $m = n + 2$. It is not hard to see that each stage leaves us with a correctly constructed, strict, closed tableau. Further, at the end we have a tableau in which $a_i \vee b_i$ no longer appears. Call this tableau $\mathbf{T}_1^*$.

$\mathbf{T}_1^*$ is a closed tableau for $\{A_1^* \vee B_1^*, \ldots, A_{i-1}^* \vee B_{i-1}^*, A_{i+1}^* \vee B_{i+1}^*, \ldots, A_{n+2}^* \vee B_{n+2}^*, \neg a_{n+2}, \neg b_{n+2}, a_i\}$. The propositional letter a_i occurs in this set, but because of the modifications made above, its only occurrence is the explicitly displayed one. Consequently, $\mathbf{T}_1^*$ also provides us with a closed strict tableau for $\{A_1^* \vee B_1^*, \ldots, A_{i-1}^* \vee B_{i-1}^*, A_{i+1}^* \vee B_{i+1}^*, \ldots, A_{n+2}^* \vee B_{n+2}^*, \neg a_{n+2}, \neg b_{n+2}\}$, and $|\mathbf{T}_1^*| \leq |\mathbf{T}_1|$. No occurrences of a_i or b_i remain.

Finally, we rename propositional letters, pushing everything down one level from i onwards. Specifically: for each $m > i$ we replace a_m by a_{m-1}, and b_m by b_{m-1}. If $k < i$, we already observed that $F_k^* = F_k$. But also $A_k^* = A_k$ and $B_k^* = B_k$, and this replacement has no effect on any of these formulas. But now, if $k > i$, this replacement turns A_k^* into A_{k-1} and similarly for B_k^*. Consequently, it converts the tableau $\mathbf{T}_1^*$ into a closed strict tableau (with the same number of formulas) for $\{A_1 \vee B_1, \ldots, A_{i-1} \vee B_{i-1}, A_i \vee B_i, \ldots, A_{n+1} \vee B_{n+1}, \neg a_{n+1}, \neg b_{n+1}\}$, that is, for S_n. Since the smallest closed strict tableau for S_n has weight $\geq k$, once again $|\mathbf{T}_1| \geq k$.

□

By Lemma 8.10.5, strict tableaus are all we need to consider if we are interested in short proofs. It is easy to check, by trying all the possibilities, that the smallest closed cut-free tableau proof for S_1 is of weight 6. It follows from Proposition 8.10.7 that the smallest cut-free tableau proof of S_n must be of weight $\geq 6 \cdot 2^{n-1} \geq 2^n$.

Analytic Cut We have just seen that if cuts are allowed, proofs can be dramatically shortened. But even so, cuts are a problem for automated theorem proving. If Cut is an allowed rule, at any point of a tableau construction, we can cut to X and $\neg X$ for *any* formula X—how do we make a useful choice of X? Recall, the use of Cut is tantamount to allowing Lemmas in proofs, and finding good Lemmas is a mathematical art, not a science.

One approach to this problem that has had some success is to restrict cuts to *subformulas of formulas that already appear on the tableau branch.* Such cuts are called *analytic.* Since a formula has only a finite number of subformulas, this gives us a limited search space to try. Notice that the cuts in Proposition 8.10.4 are all analytic.

Exercises

8.10.1. Give a direct mathematical argument that S_n is unsatisfiable.

8.10.2. Prove Lemma 8.10.5.

8.10.3. Show that, if every occurrence of a formula X in a set S of propositional formulas is in a positive position, the same is true for every occurrence of X in any tableau for S; and similarly for negative occurrences.

8.10.4. Show there is a closed tableau, allowing cuts, of weight $12n$ for $\{A_1 \vee B_1, A_2 \vee B_2, \ldots, A_n \vee B_n, A_{n+1} \vee B_{n+1}, \neg F_{n+1}\}$.

8.10.5$^{\mathbf{P}}$. Modify the propositional tableau implementation of Chapter 3 to add analytic cut.

8.11 Craig's Interpolation Theorem

We sketched a proof of Craig's interpolation theorem for propositional logic in Chapter 3 (Theorem 3.6.5). The propositional theorem does not really have any interesting applications, but the first-order version most certainly does. In this section we extend the earlier proof, which is nonconstructive, to the first-order setting, and in the next section we give a constructive argument as well. Applications then follow. The proof in this section has superficial differences with that of Theorem 3.6.5, but the essential features are the same.

Definition 8.11.1 Let S_1 and S_2 be sets of sentences. An *interpolant* for the pair S_1, S_2 is a sentence Z such that all constant, function, and relation symbols of Z occur in formulas of both S_1 and S_2 and such that $S_1 \cup \{Z\}$ and $S_2 \cup \{\neg Z\}$ are not satisfiable.

Definition 8.11.2 A finite set S of sentences is *Craig consistent* if there is a partition S_1, S_2 of S that lacks an interpolant. (S_1, S_2 is a partition of S if $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \emptyset$.)

The following lemma is in a form that is particularly suited for an application of the Model Existence Theorem. Craig's Theorem itself is an easy consequence.

Lemma 8.11.3 *Let C be the collection of Craig-consistent sets. C is a first-order consistency property.*

Proof Several items must be checked; we only do a few.

γ-case Suppose $\gamma \in S$ but for some closed term t, $S \cup \{\gamma(t)\}$ is not Craig consistent. We show S is not Craig-consistent either. Let S_1, S_2 be a partition of S; we show it has an interpolant. And we assume $\gamma \in S_1$; the argument if $\gamma \in S_2$ is similar.

$S_1 \cup \{\gamma(t)\}$, S_2 is a partition of $S \cup \{\gamma(t)\}$, so it has an interpolant, say Z, since $S \cup \{\gamma(t)\}$ is not Craig-consistent. Now $S_2 \cup \{\neg Z\}$ is not satisfiable. Also, $S_1 \cup \{\gamma(t)\} \cup \{Z\}$ is not satisfiable, and it follows easily that $S_1 \cup \{Z\}$ is not satisfiable either, since $\gamma \in S_1$.

We know that all constant, function, and relation symbols of Z are common to $S_1 \cup \{\gamma(t)\}$ and S_2. If they all occur in S_1, we are done;

Z is an interpolant for S_1, S_2. So now suppose Z contains some symbol occurring in $S_1 \cup \{\gamma(t)\}$ but not in S_1. Since $\gamma \in S_1$, any such symbol must occur in t and so must be a constant or a function symbol. There may be several; for simplicity let us say Z contains just one subterm, $f(u_1, \ldots, u_n)$, where f occurs in t but not in S_1. The more general situation is treated similarly.

Let x be a new free variable, and let Z^* be like Z but with the occurrence of $f(u_1, \ldots, u_n)$ replaced by x, so $Z = Z^*\{x/f(u_1, \ldots, u_n)\}$. We claim $(\exists x)Z^*$ is an interpolant for S_1, S_2.

First, all constant, function, and relation symbols of $(\exists x)Z^*$ are common to both S_1 and S_2, because we have removed the only one that was a problem. Next, $S_2 \cup \{\neg Z\}$ is not satisfiable, hence, neither is $S_2 \cup \{\neg(\exists x)Z^*\}$. This follows from the validity of

$$Z^*\{x/f(u_1, \ldots, u_n)\} \supset (\exists x)Z^*.$$

Finally, $S_1 \cup \{Z\}$ is not satisfiable, and it follows that $S_1 \cup \{(\exists x)Z^*\}$ is also not satisfiable. This argument needs a little more discussion than the others. (Its similarity to the proof of Lemma 8.3.1 is no coincidence.)

Suppose $S_1 \cup \{(\exists x)Z^*\}$ is satisfiable, we show $S_1 \cup \{Z\}$ also is. Suppose the members of $S_1 \cup \{(\exists x)Z^*\}$ are true in the model $\langle \mathbf{D}, \mathbf{I} \rangle$. Then in particular, $Z^{*\mathbf{I},\mathbf{A}}$ is true for some assignment $\mathbf{A}$. Now define a new interpretation $\mathbf{J}$ to be like $\mathbf{I}$ on all symbols except f, and set $f^{\mathbf{J}}$ to be the same as $f^{\mathbf{I}}$ on all members of $\mathbf{D}$ except $u_1^{\mathbf{I},\mathbf{A}}, \ldots, u_n^{\mathbf{I},\mathbf{A}}$. Finally, set $f^{\mathbf{J}}(u_1^{\mathbf{I},\mathbf{A}}, \ldots, u_n^{\mathbf{I},\mathbf{A}}) = x^{\mathbf{I}}$. Since $\mathbf{I}$ and $\mathbf{J}$ differ only on f, and that does not occur in S_1, the members of this set will have the same truth values using either interpretation. Consequently, the members of S_1 are true in $\langle \mathbf{D}, \mathbf{J} \rangle$. Using Proposition 5.3.7, $[Z^*\{x/f(u_1, \ldots, u_n)\}]^{\mathbf{J},\mathbf{A}} = Z^{*\mathbf{J},\mathbf{A}} = Z^{*\mathbf{I},\mathbf{A}} = \mathbf{t}$.

δ-case This time suppose $\delta \in S$ but for each parameter p, $S \cup \{\delta(p)\}$ is not Craig-consistent. We show S is also not Craig-consistent. Let S_1, S_2 be a partition of S; we show it has an interpolant. And once again, we suppose $\delta \in S_1$; if it is in S_2, the argument is similar.

Let p be a parameter that does not occur in S (there must be one, since S is finite). $S \cup \{\delta(p)\}$ is not Craig-consistent, so its partition $S_1 \cup \{\delta(p)\}$, S_2 has an interpolant, say, Z. We claim Z is also an interpolant for S_1, S_2.

All constant, function, and relation symbols of Z are common to both $S_1 \cup \{\delta(p)\}$ and S_2. Since p was new to S, it does not occur in S_2, and hence not in Z either. It follows that all constant, function, and relation symbols of Z are common to S_1 and S_2.

$S_2 \cup \{\neg Z\}$ is not satisfiable. Also $S_1 \cup \{\delta(p)\} \cup \{Z\}$ is not satisfiable, and it follows that $S_1 \cup \{Z\}$ is not satisfiable either. This is shown by a 'redefining interpretations' argument much like in the γ-case. We omit it. But the conclusion is that Z is an interpolant for S_1, S_2.

□

Now we come to Craig's Theorem itself [12].

Definition 8.11.4 The sentence Z is an *interpolant* for the sentence $X \supset Y$ if every relation symbol, function symbol, and constant symbol of Z is common to X and Y, and both $X \supset Z$ and $Z \supset Y$ are valid.

Theorem 8.11.5 **(First-Order Craig Interpolation)** *If $X \supset Y$ is a valid sentence, then it has an interpolant.*

Proof We show the contrapositive. Suppose $X \supset Y$ lacks an interpolant. Let $S = \{X, \neg Y\}$, and consider the partition $S_1 = \{\neg Y\}$ and $S_2 = \{X\}$. If Z were an interpolant for S_1, S_2, it follows directly that Z would also be an interpolant for $X \supset Y$. Thus, S_1, S_2 has no interpolant, so S is Craig-consistent. Then by the Model Existence Theorem, S is satisfiable, and so $X \supset Y$ is not valid. □

Exercises

8.11.1. We said in the proof of Lemma 8.11.3, in doing the γ-case, that the subcase where $\gamma \in S_2$ was similar to that where $\gamma \in S_1$. Nonetheless, there are some notable differences. Do this subcase in detail.

8.12 Craig's Interpolation Theorem—Constructively

The proof of Craig's Theorem in the previous section used the Model Existence Theorem and was nonconstructive. Now we give a constructive proof, showing how to extract an interpolant from a closed tableau. (The previous sentence mentions an important point—one that is easy to miss. An interpolant for a valid sentence $X \supset Y$ is not constructed just from the sentences X and Y. It is constructed from a *proof* of $X \supset Y$—different proofs can give different interpolants.) We use a modified version of the symmetric Gentzen system method introduced by Smullyan [48].

In proving $X \supset Y$, we start a tableau with $\neg(X \supset Y)$, then apply the α-rule, adding X and $\neg Y$. After this, any sentence added to the tableau must be either a descendant of X or of $\neg Y$. We begin by enhancing the usual tableau machinery to keep track of the ancestor of each sentence. Think of X as "left" and $\neg Y$ as "right," which are respective positions of X and Y in $X \supset Y$. We symbolize this by writing $L(X)$ and $R(\neg Y)$ and systematically use the "L" and "R" notation throughout. In effect, "L" and "R" are bookkeeping devices that record a sentence's ancestry; they play no other role.

Definition 8.12.1 A *biased sentence* is an expression of one of the forms $L(Z)$ or $R(Z)$, where Z is a sentence.

Next, the usual tableau rules are extended to biased sentences in a straightforward way. For instance, the standard α-rule yields the following two biased rules:

$$\frac{L(\alpha)}{\begin{array}{c}L(\alpha_1)\\ L(\alpha_2)\end{array}} \qquad \frac{R(\alpha)}{\begin{array}{c}R(\alpha_1)\\ R(\alpha_2)\end{array}}$$

The other tableau rules are treated in a similar way. We call a tableau that is constructed using these rules a *biased tableau.* A branch of a biased tableau is closed if it contains a syntactic contradiction, ignoring the L and R symbols. Thus, a branch is closed if it contains $L(Z)$ and $R(\neg Z)$ or if it contains $L(Z)$ and $L(\neg Z)$, and so on.

If the sentence $X \supset Y$ has a tableau proof, it can be converted into a closed biased tableau for $\{L(X), R(\neg Y)\}$. Simply take the closed tableau beginning with $\neg(X \supset Y)$, drop the first line, thus getting a tableau beginning with X and $\neg Y$; replace X by $L(X)$ and $\neg Y$ with $R(\neg Y)$, then continue the insertion of L and R symbols downward through the tableau, in the obvious way. We extract an interpolant from this closed biased tableau.

Essentially, the idea is this. We begin with each closed branch, assign an interpolant (to be defined shortly) to it, then, one by one, we undo each tableau rule application, calculating interpolants for the resulting shortened branches from those for the original longer ones. Thus, for instance, suppose the last rule applied on a branch is one of the biased α-rules—say the set of biased sentences on the branch is $S \cup \{L(\alpha), L(\alpha_1), L(\alpha_2)\}$, and we have an interpolant for this set. Using it, we say how to calculate an interpolant for the smaller set $S \cup \{L(\alpha)\}$, corresponding to the branch before the α-rule was applied. Continuing in this way, we work our way back to the beginning, thus producing an interpolant for the set $\{L(X), R(\neg Y)\}$, and we will see this is also an interpolant for the sentence $X \supset Y$. Now to define the terminology precisely.

Definition 8.12.2 We say the sentence Z is an interpolant for the finite set $\{L(A_1), \ldots, L(A_n), R(B_1), \ldots, R(B_k)\}$, provided Z is an interpolant, in the sense of Definition 8.11.4, for the sentence $(A_1 \wedge \ldots \wedge A_n) \supset (\neg B_1 \vee \ldots \vee \neg B_k)$. (Take the empty conjunction to be $\top$ and the empty disjunction to be $\bot$.)

We use the notation $S \xrightarrow{int} Z$ to symbolize that Z is an interpolant for the finite set S of biased sentences.

Note that by this definition, an interpolant for the set $\{L(X), R(\neg Y)\}$ will be an interpolant for the sentence $X \supset \neg\neg Y$, and hence for $X \supset Y$.

Now we give calculation rules, starting with those for closed branches.

$$\begin{aligned}
S \cup \{L(A), L(\neg A)\} &\xrightarrow{int} \bot \\
S \cup \{R(A), R(\neg A)\} &\xrightarrow{int} \top \\
S \cup \{L(A), R(\neg A)\} &\xrightarrow{int} A \\
S \cup \{R(A), L(\neg A)\} &\xrightarrow{int} \neg A \\
S \cup \{L(\bot)\} &\xrightarrow{int} \bot \\
S \cup \{R(\bot)\} &\xrightarrow{int} \top
\end{aligned}$$

Next the easy propositional cases.

$$\frac{S \cup \{L(\top)\} \xrightarrow{int} A}{S \cup \{L(\neg\bot)\} \xrightarrow{int} A} \qquad \frac{S \cup \{L(\bot)\} \xrightarrow{int} A}{S \cup \{L(\neg\top)\} \xrightarrow{int} A}$$

$$\frac{S \cup \{R(\top)\} \xrightarrow{int} A}{S \cup \{R(\neg\bot)\} \xrightarrow{int} A} \qquad \frac{S \cup \{R(\bot)\} \xrightarrow{int} A}{S \cup \{R(\neg\top)\} \xrightarrow{int} A}$$

$$\frac{S \cup \{L(Z)\} \xrightarrow{int} A}{S \cup \{L(\neg\neg Z)\} \xrightarrow{int} A} \qquad \frac{S \cup \{R(Z)\} \xrightarrow{int} A}{S \cup \{R(\neg\neg Z)\} \xrightarrow{int} A}$$

The α-cases are also straightforward.

$$\frac{S \cup \{L(\alpha_1), L(\alpha_2)\} \xrightarrow{int} A}{S \cup \{L(\alpha)\} \xrightarrow{int} A} \qquad \frac{S \cup \{R(\alpha_1), R(\alpha_2)\} \xrightarrow{int} A}{S \cup \{R(\alpha)\} \xrightarrow{int} A}$$

Finally the β-cases, which are the most interesting of the propositional rules, follow:

$$\frac{S \cup \{L(\beta_1)\} \xrightarrow{int} A \quad S \cup \{L(\beta_2)\} \xrightarrow{int} B}{S \cup \{L(\beta)\} \xrightarrow{int} A \vee B}$$

$$\frac{S \cup \{R(\beta_1)\} \xrightarrow{int} A \quad S \cup \{R(\beta_2)\} \xrightarrow{int} B}{S \cup \{R(\beta)\} \xrightarrow{int} A \wedge B}$$

This completes the set of propositional rules. Before moving to those for quantifiers, we verify one of the rules and give an example. The rule we verify is the one for $R(\beta)$.

Verification Suppose $S = \{L(X_1), \ldots, L(X_n), R(Y_1), \ldots, R(Y_k)\}$, and we have both

$$S \cup \{R(\beta_1)\} \xrightarrow{int} A \quad \text{and} \quad S \cup \{R(\beta_2)\} \xrightarrow{int} B.$$

Then A is an interpolant for the sentence $(X_1 \wedge \ldots \wedge X_n) \supset (\neg Y_1 \vee \ldots \vee \neg Y_k \vee \neg\beta_1)$, so all the relation, function, and constant symbols of A appear in both $X_1 \wedge \ldots \wedge X_n$ and $\neg Y_1 \vee \ldots \vee \neg Y_k \vee \neg\beta_1$. It follows that they also appear in $\neg Y_1 \vee \ldots \vee \neg Y_k \vee \neg\beta$, since every relation, constant, and function symbol of β_1 appears in β. A similar observation applies to B. It follows that all relation, constant, and function symbols of $A \wedge B$ are common to both $X_1 \wedge \ldots \wedge X_n$ and $\neg Y_1 \vee \ldots \vee \neg Y_k \vee \neg\beta$.

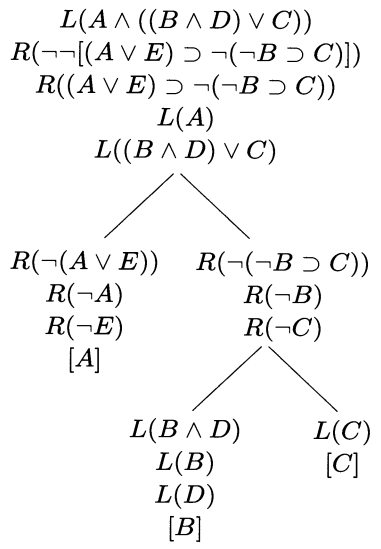
Next, $A \supset (\neg Y_1 \vee \ldots \vee \neg Y_k \vee \neg\beta_1)$ and $B \supset (\neg Y_1 \vee \ldots \vee \neg Y_k \vee \neg\beta_2)$ are both valid. Then we have the following:

$$\begin{aligned} A \wedge B &\supset (\neg Y_1 \vee \ldots \vee \neg Y_k \vee \neg\beta_1) \wedge (\neg Y_1 \vee \ldots \vee \neg Y_k \vee \neg\beta_2) \\ &\equiv (\neg Y_1 \vee \ldots \vee \neg Y_k \vee (\neg\beta_1 \wedge \neg\beta_2)) \\ &\equiv (\neg Y_1 \vee \ldots \vee \neg Y_k \vee \neg(\beta_1 \vee \beta_2)) \\ &\equiv (\neg Y_1 \vee \ldots \vee \neg Y_k \vee \neg\beta) \end{aligned}$$

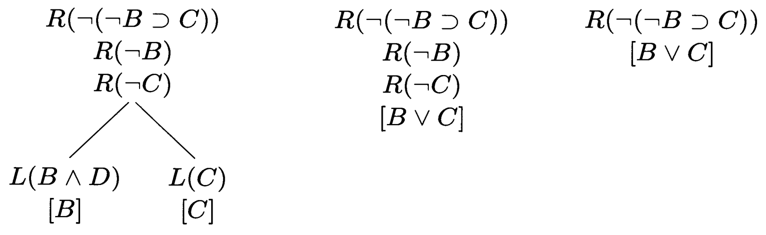
In a similar (and simpler) way, we have the validity of $(X_1 \wedge \ldots \wedge X_n) \supset (A \wedge B)$. Consequently,

$$S \cup \{R(\beta)\} \xrightarrow{int} A \wedge B.$$

Example We compute an interpolant for the tautology $[A \wedge ((B \wedge D) \vee C)] \supset \neg[(A \vee E) \supset \neg(\neg B \supset C)]$. First, here is a closed biased tableau for $\{L(A \wedge ((B \wedge D) \vee C)), R(\neg\neg[(A \vee E) \supset \neg(\neg B \supset C)])\}$. At the end of each branch, we give in square brackets an interpolant for the set of biased sentences on that branch, computed using the rules just given.


$L(A \wedge ((B \wedge D) \vee C))$
$R(\neg\neg[(A \vee E) \supset \neg(\neg B \supset C)])$
$R((A \vee E) \supset \neg(\neg B \supset C))$
$L(A)$
$L((B \wedge D) \vee C)$
$R(\neg(A \vee E))$
$R(\neg A)$
$R(\neg E)$
$[A]$
$R(\neg(\neg B \supset C))$
$R(\neg B)$
$R(\neg C)$
$L(B \wedge D)$
$L(B)$
$L(D)$
$[B]$
$L(C)$
$[C]$

Now we progressively undo tableau rule applications. For reasons of space, we concentrate on the subtree beginning with the biased sentence $R(\neg(\neg B \supset C))$, displaying only it. Progressively, it becomes:



We leave it to you to continue the process fully. When complete, $A \wedge (B \vee C)$ is the computed interpolant.

Next we give the first-order rules, which are the most complicated. We assume the language has no function symbols—a treatment of these can be added, but it obscures the essential ideas. Once again, we suppose $S = \{L(X_1), \ldots, L(X_n), R(Y_1), \ldots, R(Y_k)\}$. The first two rules are simple. Let p be a parameter that does not occur in S or in δ.

$$\frac{S \cup \{L(\delta(p))\} \xrightarrow{int} A}{S \cup \{L(\delta)\} \xrightarrow{int} A} \qquad \frac{S \cup \{R(\delta(p))\} \xrightarrow{int} A}{S \cup \{R(\delta)\} \xrightarrow{int} A}$$

This leaves the γ-cases, each of which splits in two, giving four rules. In the following, c is some constant symbol (the only kind of closed term we have now), and $A\{c/x\}$ is the result of replacing all occurrences of c in A with occurrences of the variable x. We assume x is a "new" variable, one that does not appear in S or in γ.

$$\frac{S \cup \{L(\gamma(c))\} \xrightarrow{int} A}{S \cup \{L(\gamma)\} \xrightarrow{int} A} \qquad \frac{S \cup \{R(\gamma(c))\} \xrightarrow{int} A}{S \cup \{R(\gamma)\} \xrightarrow{int} A}$$

if c occurs in $\{X_1, \ldots, X_n\}$ if c occurs in $\{Y_1, \ldots, Y_k\}$

$$\frac{S \cup \{L(\gamma(c))\} \xrightarrow{int} A}{S \cup \{L(\gamma)\} \xrightarrow{int} (\forall x)A\{c/x\}} \qquad \frac{S \cup \{R(\gamma(c))\} \xrightarrow{int} A}{S \cup \{R(\gamma)\} \xrightarrow{int} (\exists x)A\{c/x\}}$$

otherwise otherwise

Once again we verify the correctness of one of these rules, that for $R(\gamma)$, leaving the other to you.

Verification Suppose $S \cup \{R(\gamma(c))\} \xrightarrow{int} A$. Then both $(X_1 \wedge \ldots \wedge X_n) \supset A$ and $A \supset (\neg Y_1 \vee \ldots \vee \neg Y_k \vee \neg\gamma(c))$ are valid, and all constant and relation symbols of A are common to $\{X_1, \ldots, X_n\}$ and $\{Y_1, \ldots, Y_k, \gamma(c)\}$.

First, assume we are in the case where c occurs in one of the sentences of $\{Y_1, \ldots, Y_k\}$. Since $\gamma \supset \gamma(c)$ is valid, so is $\neg\gamma(c) \supset \neg\gamma$, and it follows that $A \supset (\neg Y_1 \vee \ldots \vee \neg Y_k \vee \neg\gamma)$ is valid. Also, all relation and constant symbols of A occur in both $\{X_1, \ldots, X_n\}$ and $\{Y_1, \ldots, Y_k, \gamma\}$, since the only one that might have been a problem was c, which appeared in $\gamma(c)$ and does not appear in γ. This causes no difficulties, however, since we are assuming that c is in one of $\{Y_1, \ldots, Y_k\}$. Thus, A is still an interpolant.

Next, assume c does not occur in $\{Y_1, \ldots, Y_k\}$. We have that $(X_1 \wedge \ldots \wedge X_n) \supset A$ is valid. Also $A \supset (\exists x)A\{c/x\}$ is valid, consequently,

$(X_1 \wedge \ldots \wedge X_n) \supset (\exists x)A\{c/x\}$ is valid. Further, all constant and relation symbols of $(\exists x)A\{c/x\}$ also appear in A and hence in $\{X_1, \ldots, X_n\}$.

On the right hand side, $A \supset (\neg Y_1 \vee \ldots \vee Y_k \vee \neg\gamma(c))$ is valid, hence so is $(Y_1 \wedge \ldots \wedge Y_k \wedge \gamma(c)) \supset \neg A$. For a new variable x, the validity of $(\forall x)[Y_1 \wedge \ldots \wedge Y_k \wedge \gamma(c)]\{c/x\} \supset (\forall x)\neg A\{c/x\}$ follows. But c does not occur in $\{Y_1, \ldots, Y_k\}$, so $(\forall x)[Y_1 \wedge \ldots \wedge Y_k \wedge \gamma(c)]\{c/x\} \equiv [Y_1 \wedge \ldots \wedge Y_k \wedge (\forall x)\gamma(c)(c/x)] \equiv [Y_1 \wedge \ldots \wedge Y_k \wedge \gamma]$. Thus, we have the validity of $\neg(\forall x)\neg A\{c/x\} \supset (\neg Y_1 \vee \ldots \vee \neg Y_k \vee \gamma)$, or $(\exists x)A\{c/x\} \supset (\neg Y_1 \vee \ldots \vee \neg Y_k \vee \gamma)$.

Finally, the constant and relation symbols of $(\exists x)A\{c/x\}$ are those of A except for c. (In fact, if c does not actually appear in A, the quantification is vacuous, and $(\exists x)A\{c/x\}$ and A have the same constant and relation symbols.) The constant and relation symbols of $\{X_1, \ldots, X_n\}$ include those of A, hence trivially those of $(\exists x)A\{c/x\}$. The constant and relation symbols of $\{Y_1, \ldots, Y_k, \gamma(c)\}$ include those of A, so the constant and relation symbols of $\{Y_1, \ldots, Y_k, \gamma\}$ also include those of $(\exists x)A\{c/x\}$.

We have thus verified that $(\exists x)A\{c/x\}$ is an interpolant.

Craig's Theorem was strengthened by Lyndon [32]. Recall from Definition 8.2.3 the notion of *positive* and *negative* formula occurrence. Say a relation symbol R *occurs positively* in a formula X if it appears in a positive atomic subformula of X—similarly for negative occurrences. Note that a relation symbol may appear both positively and negatively in the same formula.

Theorem 8.12.3 **(Lyndon Interpolation)** *If $X \supset Y$ is a valid sentence, then it has an interpolant Z such that every relation symbol occurring positively in Z has a positive occurrence in both X and Y, and every relation symbol occurring negatively in Z has a negative occurrence in both X and Y.*

Lyndon's result cannot be extended to account for positive and negative occurrences of constant or function symbols. Consider the valid sentence $[(\forall x)(P(x) \supset \neg Q(x)) \wedge P(c)] \supset \neg Q(c)$. The constant symbol c will occur in any interpolant for this sentence, but it occurs positively on the left and negatively on the right.

In effect, Lyndon's interpolation theorem has already been proved. That is, the two proofs we gave actually verify the stronger version. We leave it to you to check this, as an exercise.

Here is a different strengthening of Craig's theorem. As usual, we need some terminology first.

Definition 8.12.4 A formula is *universal* if every quantifier occurrence in it is essentially universal, in the sense of Definition 8.5.1. Likewise, a formula is *existential* if every quantifier occurrence in it is essentially existential.

Theorem 8.12.5 *Suppose $X \supset Y$ is a valid sentence. If Y is universal, there is an interpolant that is also universal. Similarly, if X is existential, there is an existential interpolant. Finally, if X is existential and Y is universal, there is a quantifier-free interpolant.*

Exercises

8.12.1. Use the procedure of this section to compute an interpolant for the valid sentence $[(\forall x)(P(x) \supset \neg Q(x)) \wedge P(c)] \supset \neg Q(c)$.

8.12.2. Show that the procedure of this section actually verifies Lyndon's strengthening of the Craig Interpolation Theorem.

8.12.3. Prove Theorem 8.12.5 using the procedures of this section.

8.12.4$^{\mathbf{P}}$. Implement the propositional part of the procedure of this section in Prolog, producing a propositional theorem prover for implications that computes interpolants.

8.12.5$^{\mathbf{P}}$. Extend the Prolog implementation of the previous exercise to a full first-order version.

8.13 Beth's Definability Theorem

Sometimes information is explicit: "The murderer is John Smith." Often we find an implicit characterization instead, as in "The murderer is the only person in town who wears glasses, has red hair, and owns a dog and a canary." Puzzles often involve turning implicit characterizations into explicit ones. Beth's Definability Theorem [4] essentially says that in classical logic such puzzles can always be solved. This is a fundamental result that says that classical logic has a kind of completeness where definability is concerned. We begin this section with some terminology, then we state and prove Beth's Theorem. The proof we give is not Beth's original one but is based on Craig's Theorem and comes from Craig's 1957 paper [12].

Definition 8.13.1 Let R be an n-place relation symbol, and $\Phi(x_1, \ldots, x_n)$ be a formula with free variables among $x_1, \ldots, x_n$, and with no occurrences of R. We say Φ is an *explicit definition* of R, with respect to a set S of sentences, provided

$$S \models_f (\forall x_1) \cdots (\forall x_n)[R(x_1, \ldots, x_n) \equiv \Phi(x_1, \ldots, x_n)].$$

Example In a group, y is the *conjugate* of x under conjugation by a if $y = a^{-1}xa$. Conjugation is a three-place relation, between a, x, and y that has an explicit definition (we just gave it informally) with respect to the set S of axioms for a group. Of course, to properly present this as an example, we need a first-order language *with equality*. Equality will be investigated in the next chapter, and it can be shown that the results of this section do carry over.

Definition 8.13.2 Again, let R be an n-place relation symbol. We say R is *implicitly defined* by a set S of sentences, provided S determines R uniquely, in the following sense. Let R^* be an n-place relation symbol different from R, that does not occur in S, and let S^* be like S except that every occurrence of R has been replaced by an occurrence of R^*. Then S determines R uniquely if

$$S \cup S^* \models_f (\forall x_1) \cdots (\forall x_n)[R(x_1, \ldots, x_n) \equiv R^*(x_1, \ldots, x_n)].$$

Example Let $S = \{(\forall x)(R(x) \supset A(x)), (\forall x)(R(x) \supset B(x)), (\forall x)(A(x) \supset (B(x) \supset R(x)))\}$. It is easy to check that S determines R uniquely, and so R is implicitly defined by S. In fact, R also has the explicit definition $(\forall x)[R(x) \equiv (A(x) \wedge B(x))]$.

Theorem 8.13.3 **(Beth Definability)** *If R is implicitly defined by a set S, then R has an explicit definition with respect to S.*

Proof Suppose R is implicitly defined by S. Let R^* be a new relation symbol, and let S^* be like S but with occurrences of R replaced by occurrences of R^*. Then

$$S \cup S^* \models_f (\forall x_1) \cdots (\forall x_n)[R(x_1, \ldots, x_n) \equiv R^*(x_1, \ldots, x_n)],$$

so by Theorem 5.10.2,

$$S_0 \cup S_0^* \models_f (\forall x_1) \cdots (\forall x_n)[R(x_1, \ldots, x_n) \equiv R^*(x_1, \ldots, x_n)],$$

where S_0 and S_0^* are finite subsets of S and S^*, respectively. Let $\bigwedge S_0$ be the conjunction of the members of S_0, and let $\bigwedge S_0^*$ be the conjunction of the members of S_0^*, so that both $\bigwedge S_0$ and $\bigwedge S_0^*$ are sentences. Then (Exercise 5.10.3, part 7),

$$(\bigwedge S_0 \wedge \bigwedge S_0^*) \supset (\forall x_1) \cdots (\forall x_n)[R(x_1, \ldots, x_n) \equiv R^*(x_1, \ldots, x_n)]$$

is valid. Choose n distinct parameters, $p_1, \ldots, p_n$. Then the following is also valid:

$$(\bigwedge S_0 \wedge \bigwedge S_0^*) \supset [R(p_1, \ldots, p_n) \equiv R^*(p_1, \ldots, p_n)],$$

and from this follows the validity of

$$[\bigwedge S_0 \wedge R(p_1, \ldots, p_n)] \supset [\bigwedge S_0^* \supset R^*(p_1, \ldots, p_n)].$$

Now by Craig's Theorem 8.11.5, there is an interpolant for this. The interpolant may contain some or all of the parameters we introduced, so we denote it by $\Phi(p_1, \ldots, p_n)$. Since it is an interpolant, all constant, function, and relation symbols of $\Phi(p_1, \ldots, p_n)$ are common to $[\bigwedge S_0 \wedge R(p_1, \ldots, p_n)]$ and $[\bigwedge S_0^* \supset R^*(p_1, \ldots, p_n)]$, and both

$$[\bigwedge S_0 \wedge R(p_1, \ldots, p_n)] \supset \Phi(p_1, \ldots, p_n)$$

and

$$\Phi(p_1, \ldots, p_n) \supset [\bigwedge S_0^* \supset R^*(p_1, \ldots, p_n)]$$

are valid.

The relation symbol R^* does not occur in $[\bigwedge S_0 \wedge R(p_1, \ldots, p_n)]$, and the relation symbol R does not occur in $[\bigwedge S_0^* \supset R^*(p_1, \ldots, p_n)]$, and so neither R nor R^* can occur in $\Phi(p_1, \ldots, p_n)$. Also, $\Phi(p_1, \ldots, p_n) \supset [\bigwedge S_0^* \supset R^*(p_1, \ldots, p_n)]$ is valid, hence so is the result of replacing all occurrences of R^* by a relation symbol that does not appear in this sentence. Since R^* does not occur in $\Phi(p_1, \ldots, p_n)$, replacing occurrences of R^* with occurrences of R yields the validity of $\Phi(p_1, \ldots, p_n) \supset [\bigwedge S_0 \supset R(p_1, \ldots, p_n)]$, from which the validity of $\bigwedge S_0 \supset [\Phi(p_1, \ldots, p_n) \supset R(p_1, \ldots, p_n)]$ follows. We also have the validity of $[\bigwedge S_0 \wedge R(p_1, \ldots, p_n)] \supset \Phi(p_1, \ldots, p_n)$ from which the validity of $\bigwedge S_0 \supset [R(p_1, \ldots, p_n) \supset \Phi(p_1, \ldots, p_n)]$ follows. Combining these two, we have the validity of

$$\bigwedge S_0 \supset [R(p_1, \ldots, p_n) \equiv \Phi(p_1, \ldots, p_n)],$$

from which we immediately obtain

$$S_0 \models_f [R(p_1, \ldots, p_n) \equiv \Phi(p_1, \ldots, p_n)]$$

and hence

$$S \models_f [R(p_1, \ldots, p_n) \equiv \Phi(p_1, \ldots, p_n)].$$

Finally, from this (Exercise 8.13.1) we have the validity of

$$S \models_f (\forall x_1) \cdots (\forall x_n)[R(x_1, \ldots, x_n) \equiv \Phi(x_1, \ldots, x_n)].$$

Since $\Phi(x_1, \ldots, x_n)$ contains no occurrences of R, it is an explicit definition of R. □

The proof of Beth's Theorem given above is entirely constructive, provided we have a constructive method of finding interpolants. Our first proof of Craig's Theorem was not constructive, but our second was. Also Craig's original proof [12] was constructive by design. It follows that the conversion from implicit to explicit definition is a constructive one, suitable for mechanization, at least in principle.

Exercises

8.13.1. Suppose Φ is a formula with only x free, and p is a parameter that does not occur in Φ, or in any member of the set S of sentences. Show that if $S \models_f \Phi\{x/p\}$ then $S \models_f (\forall x)\Phi$. Hint: see Exercises 5.3.2 and 5.10.2.

8.14 Lyndon's Homomorphism Theorem

Let L be a first-order language with no constant or function symbols, and let $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ and $\mathbf{N} = \langle \mathbf{E}, \mathbf{J} \rangle$ be two models for the language L. A mapping $h : \mathbf{D} \to \mathbf{E}$ is a *homomorphism*, provided the following: For each n-place relation symbol R of L and for each $b_1, \ldots, b_n \in \mathbf{D}$, if $R^{\mathbf{I}}(b_1, \ldots, b_n)$ is true in $\mathbf{M}$ then $R^{\mathbf{J}}(h(b_1), \ldots, h(b_n))$ is true in $\mathbf{N}$.

Example Suppose L has a single three-place relation symbol, R. Let $\mathbf{M}$ be the model whose domain is the integers and in which R is interpreted by the addition relation; that is, $R^{\mathbf{I}}(a, b, c)$ is true in $\mathbf{M}$ just in case $a + b = c$. Let $\mathbf{N}$ have domain $\{0, 1\}$, and interpret R by the addition modulo 2 relation; $R^{\mathbf{J}}(a, b, c)$ is true in $\mathbf{N}$ if $a+b = c \pmod 2$. If h is the function mapping the evens to 0 and the odds to 1, h is a homomorphism from $\mathbf{M}$ to $\mathbf{N}$.

Definition 8.14.1 A sentence X of L is *preserved under homomorphisms*, provided, whenever X is true in a model $\mathbf{M}$, and there is a homomorphism from $\mathbf{M}$ to $\mathbf{N}$, then X is also true in $\mathbf{N}$.

Being preserved under homomorphisms is a semantic notion—the definition talks about models and truth. The question for this section is, Does this semantic notion have a syntactic counterpart? We will see that it does—being a *positive* sentence.

Definition 8.14.2 Let L be a first-order language (with or without constant and function symbols). The *positive* formulas of L are the members of the smallest set S such that:

1. If A is atomic, $A \in S$.
2. $Z \in S \Longrightarrow \neg\neg Z \in S$.
3. $\alpha_1 \in S$ and $\alpha_2 \in S \Longrightarrow \alpha \in S$.
4. $\beta_1 \in S$ and $\beta_2 \in S \Longrightarrow \beta \in S$.
5. $\gamma(t) \in S \Longrightarrow \gamma \in S$.
6. $\delta(t) \in S \Longrightarrow \delta \in S$.

Informally, the positive formulas are those that can be rewritten using only conjunctions and disjunctions as propositional connectives (in particular without negation symbols). You are asked to show this as Exercise 8.14.1. Equivalently, the positive formulas are those in which every relation symbol appears only positively.

Now, here is the easy half of the semantic/syntactic connection we are after; we leave its proof to you. (It generalizes considerably—see Exercise 9.2.3.)

Proposition 8.14.3 *Let L be a language with no constant or function symbols. Every positive sentence of L is preserved under homomorphisms.*

The hard half is Lyndon's Homomorphism Theorem. It asserts the converse.

Theorem 8.14.4 **(Lyndon Homomorphism)** *Let L be a language with no constant or function symbols. If the sentence X is preserved under all homomorphisms, then X is equivalent to some positive sentence.*

Proof Without loss of generality, we assume L has a finite number of relation symbols, since it is only necessary to consider those that actually appear in X. For each (n-place) relation symbol R of L, let R' be another relation symbol (also n-place) that is new to the language. Let L' be the result of enlarging L with these additional relation symbols. By the *homomorphism sentence* for R, we mean the sentence

$$(\forall x_1)\cdots(\forall x_n)[R(x_1,\ldots,x_n) \supset R'(x_1,\ldots,x_n)].$$

Now, let H be the conjunction of all the homomorphism sentences for the relation symbols of L. Also, let X' be the sentence that is like X, except that each relation symbol R has been replaced by its counterpart R'. (More generally, for any sentence Z, let Z' be the result of replacing each R by the corresponding R'.)

Now assume that X is preserved under homomorphisms. We claim the following sentence is valid:

$$X \supset (H \supset X').$$

Suppose this is not so. Let $\mathbf{M}_0 = \langle \mathbf{D}_0, \mathbf{I}_0 \rangle$ be a model for L' in which it is not true, hence in which X and H are true, but X' is false. We construct two new models $\mathbf{M}_1 = \langle \mathbf{D}_1, \mathbf{I}_1 \rangle$ and $\mathbf{M}_2 = \langle \mathbf{D}_2, \mathbf{I}_2 \rangle$, for the original language L, by "pulling apart" this one. First, set $\mathbf{D}_1 = \mathbf{D}_2 = \mathbf{D}_0$, so all models have the same domain. Next, for each relation symbol R of L, set $R^{\mathbf{I}_1} = R^{\mathbf{I}_0}$ and $R^{\mathbf{I}_2} = R'^{\mathbf{I}_0}$.

In effect, $\mathbf{M}_1$ acts like the "unprimed" part of $\mathbf{M}$, and $\mathbf{M}_2$ acts like the "primed" part. This can be made more precise, as follows (we leave the proof as an exercise):

Pulling Apart Assertion For any sentence Z of L:

1. Z is true in $\mathbf{M}_0$ if and only if Z is true in $\mathbf{M}_1$.
2. Z is true in $\mathbf{M}_0$ if and only if Z' is true in $\mathbf{M}_2$.

Now, let $a_1, \ldots, a_n$ be in $\mathbf{D}_0$, and suppose $R^{\mathbf{I}_1}(a_1, \ldots, a_n)$ is true in $\mathbf{M}_1$. By definition, $R^{\mathbf{I}_0}(a_1, \ldots, a_n)$ is true in $\mathbf{M}_0$. But the sentence H is true in $\mathbf{M}_0$, so in particular $(\forall x_1) \cdots (\forall x_n)[R(x_1, \ldots, x_n) \supset R'(x_1, \ldots, x_n)]$ is true. It follows that $R'^{\mathbf{I}_0}(a_1, \ldots, a_n)$ is true in $\mathbf{M}_0$, and hence $R^{\mathbf{I}_2}(a_1, \ldots, a_n)$ is true in $\mathbf{M}_2$. Then the identity map is a homomorphism from $\mathbf{M}_1$ to $\mathbf{M}_2$!

Finally, X is true in $\mathbf{M}_0$, hence X is true in $\mathbf{M}_1$ by the Pulling Apart Assertion. But we are assuming that X is preserved under homomorphisms, hence X is true in $\mathbf{M}_2$. Then by the Pulling Apart Assertion again, X' must be true in $\mathbf{M}_0$, but it is not. This contradiction establishes the validity of $X \supset (H \supset X')$.

Now, by Lyndon's Interpolation Theorem 8.12.3, there is a sentence Z such that

1. $X \supset Z$ is valid.
2. $Z \supset (H \supset X')$ is valid.
3. Every relation symbol that occurs positively (negatively) in Z occurs positively (negatively) in both X and $H \supset X'$.

No primed relation symbol R' occurs in X, hence it cannot occur in Z. Also, R does not occur in X', and the only occurrence of R in H is in the homomorphism sentence for R, $(\forall x_1) \cdots (\forall x_n)[R(x_1, \ldots, x_n) \supset R'(x_1, \ldots, x_n)]$, where it occurs negatively. Consequently, the only occurrence of R in $H \supset X'$ is positive, and hence any occurrence of R in Z must be positive. It follows that Z itself is a positive sentence of L.

We have established that $Z \supset (H \supset X')$ is valid. For each relation symbol R of L, if every occurrence of R' in this sentence is replaced by an occurrence of R, the sentence remains valid. This does not affect Z, since no R' occurs in Z. Making the replacement turns X' into X. And finally, replacing R' by R converts H into a tautology, since the homomorphism sentence for R becomes $(\forall x_1) \cdots (\forall x_n)[R(x_1, \ldots, x_n) \supset R(x_1, \ldots, x_n)]$. Consequently, $Z \supset X$ must be valid. We also know that $X \supset Z$ is valid, hence we have the validity of $X \equiv Z$ for a positive sentence Z of L. □

We note that the Interpolation Theorem 8.12.5 can also be used to prove a "preservation" result. It can be used to show that a sentence is preserved under passage to submodels if and only if it is equivalent to a universal sentence. We do not give the argument here.

Exercises

8.14.1. Show that if X is a positive formula then there is a formula X^* in which there are no negation symbols and in which the only binary propositional connectives are $\wedge$ and $\vee$, such that $X \equiv X^*$ is valid.

8.14.2. Prove Proposition 8.14.3. Hint: You need to prove a more general result, about *formulas*, not just sentences. Let h be a homomorphism from the model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ to the model $\mathbf{N} = \langle \mathbf{E}, \mathbf{J} \rangle$. For each assignment $\mathbf{A}$ in $\mathbf{M}$, let $h\mathbf{A}$ be the assignment in $\mathbf{N}$ given by $x^{h\mathbf{A}} = h(x^{\mathbf{A}})$. Now show by structural induction that, for each positive formula X, and any assignment $\mathbf{A}$ in $\mathbf{M}$, if $X^{\mathbf{I},\mathbf{A}}$ is true in $\mathbf{M}$, then $X^{\mathbf{J},h\mathbf{A}}$ is true in $\mathbf{N}$.

8.14.3. Verify the Pulling Apart Assertion in the proof of Lyndon's Theorem. Hint: As in the previous exercise, you need to establish a stronger result about formulas.

8.14.4. Let L be the language with R as a binary relation symbol and no function or constant symbols. Being a reflexive relation can be characterized by a positive sentence: $(\forall x)R(x,x)$. The obvious characterization of being a symmetric relation is $(\forall x)(\forall y)[R(x,y) \supset R(y,x)]$, but this is not a positive sentence.

1. Show there is no positive sentence that characterizes the symmetric relations.
2. Show the same for transitivity.

9 Equality

9.1 Introduction

Most relations are meaningful only in restricted circumstances. *Bigger_than* makes no sense when applied to colors; *to_the_left_of* makes no sense when applied to smells. But the relation of equality has the universality that is shared by logical connectives and quantifiers. It is meaningful no matter what the domain. Consequently, the study of equality is generally considered to be part of logic, and formal rules for it have been developed. (For a wonderful discussion of what constitutes the subject matter of logic, see Tarski [52]). Treating equality as a part of formal logic makes sense from a practical point of view as well. Virtually every mathematical theory of interest uses the equality relation in its formulation, so efficient rules for it should be incorporated into theorem provers at a deep level.

When we moved from propositional to first-order logic, we gained power of expression, but our theorem provers became more complex and running them became more time costly. Adding equality rules has a similar effect, much intensified. With equality rules present, complete theorem provers have so many paths to explore in attempting to produce proofs that they can generally only succeed with simple, not very interesting theorems. It is here that good heuristics, and even machine/human interaction, becomes critical. We will provide only the theoretical foundation. Going beyond that is a matter for experimentation and psychology, not just mathematics.

Before we start on a formal treatment, we should consider, informally, what properties of equality are fundamental. It is, of course, an equivalence relation: reflexive, symmetric, and transitive. It obeys a substitutivity principle, one can 'substitute equals for equals.' That is, if we know that $a = b$, then the truth value of a statement is preserved if we replace some occurrences of a by occurrences of b. We will refer to this as the *replacement property* of equality. Further, in domains where it makes sense, one can add the same thing to both sides of an equality, and get another equality, and similarly for other operations. It turns out that some of these principles follow easily from others.

Transitivity. Suppose we know that $a = b$ and $b = c$. Since $b = c$, using the replacement principle we could replace occurrences of b with occurrences of c without affecting truth values. We are given that $a = b$ is true; replacing the occurrence of b in this with c we get that $a = c$ is true. Thus, the transitivity of equality follows from the replacement property of equality.

Symmetry. Suppose we know that $a = b$. Again, by the replacement property of equality, we can replace occurrences of a with occurrences of b. Also by the reflexive property of equality, $a = a$. If we replace the first occurrence of a here by an occurrence of b, we get $b = a$. Thus, the symmetry of equality follows from the replacement property, together with reflexivity.

Operation application. Say we are talking about numbers, we are told $a = b$, and we would like to conclude that $a + c = b + c$; that is, we can add c to both sides and preserve equality. Well, by the reflexive principle, $a + c = a + c$. If we use the replacement property with $a = b$, and replace the second occurrence of a by an occurrence of b we get $a + c = b + c$. Thus, again we need only replacement and reflexivity.

In fact, all the basic properties of equality can be deduced from replacement and reflexivity alone. In what follows we will simply build these notions into formal logic systems in direct ways. Of course, we must establish that when we do so we really have captured equality, that the informal arguments did not mislead us.

9.2 Syntax and Semantics

The syntax of first-order logic with equality is the same as that of first-order logic in the previous chapters, except that now we assume there is a designated two-place relation symbol that we think of as standing for the equality relation. We will not use the symbol $=$ for this purpose; we prefer to keep that for its standard mathematical role of representing the equality relation in informal arguments. The formal symbol we use is $\approx$. Thus, for the rest of this chapter all languages will include the two place relation symbol $\approx$. Also, to make reading easier, we will generally write $t \approx u$ instead of the official $\approx(t, u)$ for atomic formulas involving this relation symbol.

Semantics is straightforward. We have already defined the basic notion of first-order logic model. Now we simply restrict ourselves to those models in which the symbol we have chosen to represent equality really does so.

Definition 9.2.1 A model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ is called *normal* provided $\approx^{\mathbf{I}}$ is the equality relation on $\mathbf{D}$.

The goal of this chapter is now easily stated. We want to create proof procedures that prove exactly those sentences that are true in all normal models.

The notion of logical consequence adapts in a straightforward manner. Recall we have been writing $S \models_f X$ to denote that X is a first-order consequence of the set S, meaning that X is true in every model in which the members of S are true.

Definition 9.2.2 Let X be a sentence, and S be a set of sentences. We write $S \models_{\approx} X$, provided X is true in every *normal* model in which the members of S are true.

Then $S \models_{\approx} X$ is like $S \models_f X$, except that it takes equality into account. Trivially, if $S \models_f X$, then $S \models_{\approx} X$. The converse is not true, however. Suppose $S = \{a \approx b, P(a)\}$. Then $S \models_{\approx} P(b)$, but it is not the case that $S \models_f P(b)$.

Many mathematical structures can be easily characterized now, by just writing down a few defining axioms. For example, let G be the following set of sentences:

1. $(\forall x)(\forall y)(\forall z)[x \circ (y \circ z) \approx (x \circ y) \circ z]$
2. $(\forall x)[(x \circ e \approx x) \wedge (e \circ x \approx x)]$
3. $(\forall x)[(x \circ i(x) \approx e) \wedge (i(x) \circ x \approx e)]$

A *group* is simply a normal model in which the members of G are true. Then a sentence X is true in all groups just in case $G \models_{\approx} X$. In a similar way notions of *ring*, *field*, *integral domain*, *vector space*, etc. can be captured.

Sometimes working with an infinite set of sentences is useful. Consider the following list:

1. $E_{>1} = (\exists x_1)(\exists x_2)\neg(x_1 \approx x_2)$
2. $E_{>2} = (\exists x_1)(\exists x_2)(\exists x_3)[\neg(x_1 \approx x_2) \wedge \neg(x_1 \approx x_3) \wedge \neg(x_2 \approx x_3)]$
3. etc.

The pattern being followed is straightforward. $E_{>3}$ would say there are things, x_1, x_2, x_3, and x_4, no two of which are equal. Then $E_{>n}$ is true in a normal model if and only if the domain of the model has more than n members. Let *Inf* be the set $\{E_{>1}, E_{>2}, \ldots\}$. All the members of *Inf* are true in a normal model if and only if the model has an infinite domain. Now, $G \cup \mathit{Inf} \models_{\approx} X$ if and only if X is true in all infinite groups. Likewise $G \cup \{E_{>4}, \neg E_{>5}\} \models_{\approx} X$ if and only if X is true in all five element groups, and so on.

On the other hand, the notion of *finite* group, or finite structure of any kind, can *not* be captured this way. Corollary 5.9.2 implied it could not be done using the machinery of first-order logic. This corollary was proved using the Compactness Theorem, and we will see that the Compactness Theorem carries over to first-order logic with equality, the same argument applies, and so a characterization of finiteness is still not possible.

Exercises

9.2.1. Which of the following sentences are true in all normal models? Which are true in all models?

1. $(\exists x)(\forall y)(x \approx y) \supset (\forall x)(\forall y)(x \approx y)$.
2. $(\exists x)(\exists y)(x \approx y) \supset (\forall x)(\forall y)(x \approx y)$.
3. $(\forall x)(\forall y)(x \approx y) \supset (\exists x)(\forall y)(x \approx y)$.
4. $(\forall x)(\exists y)(x \approx y)$.

9.2.2. Let L be a first-order language with equality, and let $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ and $\mathbf{N} = \langle \mathbf{E}, \mathbf{J} \rangle$ be two models for the language L. A mapping $h : \mathbf{D} \to \mathbf{E}$ is a *homomorphism* if:

1. For each constant symbol c of L, $h(c^{\mathbf{I}}) = c^{\mathbf{J}}$.

2. For each n-place function symbol f of L and for each $d_1,\dots,d_n \in \mathbf{D}$, $h(f^{\mathbf{I}}(d_1,\dots,d_n)) = f^{\mathbf{J}}(h(d_1),\dots,h(d_n))$.
3. For each n-place relation symbol R of L and for each $d_1,\dots,d_n \in \mathbf{D}$, $R^{\mathbf{I}}(d_1,\dots,d_n)$ true $\Longrightarrow R^{\mathbf{J}}(h(d_1),\dots,h(d_n))$ true.

If h is a homomorphism from the model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ to the model $\mathbf{N} = \langle \mathbf{E}, \mathbf{J} \rangle$, its action is extended to assignments as follows: For an assignment $\mathbf{A}$ in $\mathbf{M}$, let $h\mathbf{A}$ be the assignment in $\mathbf{N}$ such that, for each variable x, $x^{h\mathbf{A}} = h(x^{\mathbf{A}})$.

Now, let h be a homomorphism from $\mathbf{M}$ to $\mathbf{N}$. Show the following:

1. For each term t of the language L, and for each assignment $\mathbf{A}$ in $\mathbf{M}$, $h(t^{\mathbf{I},\mathbf{A}}) = t^{\mathbf{J},h\mathbf{A}}$.
2. If h is *onto*, then for each assignment $\mathbf{A}^*$ in the model $\mathbf{N}$, there is an assignment $\mathbf{A}$ in $\mathbf{M}$ such that $\mathbf{A}^* = h\mathbf{A}$.

9.2.3. As in the previous exercise, suppose L is a first-order language with equality, and h is a homomorphism from the model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ *onto* the model $\mathbf{N} = \langle \mathbf{E}, \mathbf{J} \rangle$. Show that for each *positive* sentence X (Definition 8.14.2), if X is true in $\mathbf{M}$, then X is true in $\mathbf{N}$. Hint: You will have to prove the following more general result: For any positive formula X and any assignment $\mathbf{A}$ in $\mathbf{M}$, if $X^{\mathbf{I},\mathbf{A}}$ is true in $\mathbf{M}$, then $X^{\mathbf{J},h\mathbf{A}}$ is true in $\mathbf{N}$. Show it follows that the homomorphic image of a group is a group.

9.2.4. Suppose L is a first-order language with equality. Also suppose $\mathbf{M} = \langle \mathbf{D}_M, \mathbf{I}_M \rangle$ and $\mathbf{N} = \langle \mathbf{D}_N, \mathbf{I}_N \rangle$ are two normal models for this language. We use the following shorthand notation: For a constant symbol c of L, c_M is the member assigned to c by $\mathbf{I}_M$. For a function symbol f, f_M is the function assigned to f by $\mathbf{I}_M$. For a relation symbol R, R_M is the relation assigned to R by $\mathbf{I}_M$. Similarly for $\mathbf{N}$. Call $\mathbf{M}$ a *substructure* of $\mathbf{N}$ if

1. The domain $\mathbf{D}_M$ is a subset of the domain $\mathbf{D}_N$.
2. For every constant symbol c of L, $c_M = c_N$.
3. For every n-place function symbol f of L, f_M is f_N restricted to $(\mathbf{D}_M)^n$.
4. For every n-place relation symbol R of L, R_M is the restriction of R_N to $(\mathbf{D}_M)^n$.

Note that if $\mathbf{M}$ is a substructure of $\mathbf{N}$, then an assignment $\mathbf{A}$ in $\mathbf{M}$ is also an assignment in $\mathbf{N}$. Now, assume $\mathbf{M}$ is a substructure of $\mathbf{N}$ and prove the following:

1. For each term t of L, $t^{\mathbf{I}_M,\mathbf{A}} = t^{\mathbf{I}_N,\mathbf{A}}$.
2. For each quantifier-free formula Φ of L, $\Phi^{\mathbf{I}_M,\mathbf{A}} = \Phi^{\mathbf{I}_N,\mathbf{A}}$.
3. An existential formula is one of the form $(\exists x_1)\ldots(\exists x_n)\Phi$, where Φ is quantifier free. If X is existential and $X^{\mathbf{I}_M,\mathbf{A}} = \mathbf{t}$, then $X^{\mathbf{I}_N,\mathbf{A}} = \mathbf{t}$.
4. A universal formula is one of the form $(\forall x_1)\ldots(\forall x_n)\Phi$, where Φ is quantifier free. If X is universal and $X^{\mathbf{I}_N,\mathbf{A}} = \mathbf{t}$, then $X^{\mathbf{I}_M,\mathbf{A}} = \mathbf{t}$.
5. Show it follows from the previous item that if S is a subset of a group and S contains the identity, is closed under the group operation, and is closed under inverses, then S is a itself a group.

9.3 The Equality Axioms

We mentioned earlier that the reflexivity and replacement properties of equality were basic; other important features of equality followed from them. It is possible to express these properties by sentences of first-order logic; such sentences are often called *equality axioms*. In a sense that we will make precise later on, these equality axioms tell us everything we need to know about the equality relation.

Definition 9.3.1 *ref* is the sentence $(\forall x)(x \approx x)$.

The sentence *ref* captures the reflexive property of equality in a single sentence. The replacement property takes more effort.

Definition 9.3.2 Let f be an n-place function symbol. The following sentence is a *replacement axiom* for f: $(\forall v_1)\cdots(\forall v_n)(\forall w_1)\cdots(\forall w_n)\{[(v_1 \approx w_1)\wedge\cdots\wedge(v_n \approx w_n)] \supset [f(v_1,\ldots,v_n) \approx f(w_1,\ldots,w_n)]\}$.

For example, if $+$ is a two-place function symbol (which we write in infix position), $(\forall x)(\forall y)(\forall z)(\forall w)\{[(x \approx z) \wedge (y \approx w)] \supset [x+y \approx z+w]\}$. Suppose we call this replacement axiom A for the moment. If c is a constant symbol, it is easy to see that $\{A\} \models_f (\forall x)(\forall z)\{[(x \approx z) \wedge (c \approx c)] \supset [x+c \approx z+c]\}$, and so $\{A, \mathit{ref}\} \models_f (\forall x)(\forall z)\{(x \approx z) \supset [x+c \approx z+c]\}$.

Definition 9.3.3 For a language L, *fun*(L) is the set of replacement axioms for all the function symbols of L. Members of *fun*(L) are called *function replacement axioms*.

Function replacement axioms are language dependent; there is one for each function symbol of the language. If the language has infinitely many function symbols, the set of function replacement axioms will also be infinite. In effect, these equality axioms allow replacements in the simplest kind of terms; replacement in more complicated terms follows, however. That is, if these axioms are true in a model, whether or not the model is normal, then replacing equals by equals can be done in arbitrary terms.

Proposition 9.3.4 *Suppose t is a term of the language L, with free variables $v_1, \dots, v_n$. Let u be the term $t\{v_1/w_1, \dots, v_n/w_n\}$ where $w_1, \dots, w_n$ are distinct new variables. Then*

$$\{ref\} \cup fun(L) \models_f$$
$$(\forall v_1)\cdots(\forall v_n)(\forall w_1)\cdots(\forall w_n)\{[(v_1 \approx w_1) \wedge \ldots \wedge (v_n \approx w_n)] \supset (t \approx u)\}.$$

Having dealt with function symbols and terms, we turn to the replacement property for relation symbols and formulas.

Definition 9.3.5 Let R be an n-place relation symbol. The following sentence is a *replacement axiom* for R:

$$(\forall v_1)\cdots(\forall v_n)(\forall w_1)\cdots(\forall w_n)\{[(v_1 \approx w_1) \wedge \cdots \wedge (v_n \approx w_n)]$$
$$\supset [R(v_1, \dots, v_n) \supset R(w_1, \dots, w_n)]\}.$$

For example, suppose $>$ is a two-place relation symbol of the language. The replacement axiom for it is $(\forall v_1)(\forall v_2)(\forall w_1)(\forall w_2)\{[(v_1 \approx w_1) \wedge (v_2 \approx w_2)] \supset [(v_1 > v_2) \supset (w_1 > w_2)]\}$. Further, $\approx$ itself is a two-place relation symbol. Its replacement axiom is $(\forall v_1)(\forall v_2)(\forall w_1)(\forall w_2)\{[(v_1 \approx w_1) \wedge (v_2 \approx w_2)] \supset [(v_1 \approx v_2) \supset (w_1 \approx w_2)]\}$. Suppose we denote this sentence by B for now. It is easy to see that $\{B\} \models_f (\forall x)(\forall y)\{[(x \approx y) \wedge (x \approx x)] \supset [(x \approx x) \supset (y \approx x)]\}$, and so $\{B, ref\} \models_f (\forall x)(\forall y)[(x \approx y) \supset (y \approx x)]$. That is, symmetry of equality is a logical consequence of the replacement axiom for $\approx$ together with *ref*. Transitivity is also a consequence. And once again, more general replacement also follows.

Definition 9.3.6 For a language L, *rel*(L) is the set of replacement axioms for all the relation symbols of L. Members of *rel*(L) are called *relation replacement axioms.*

Proposition 9.3.7 *Suppose X is a sentence of the language L, with free variables $v_1, \dots, v_n$. Let Y be the sentence $X\{v_1/w_1, \dots, v_n/w_n\}$ where $w_1, \dots, w_n$ are distinct new variables. Then*

$$\{ref\} \cup fun(L) \cup rel(L) \models_f (\forall v_1)\cdots(\forall v_n)(\forall w_1)\cdots(\forall w_n)$$
$$\{[(v_1 \approx w_1) \wedge \cdots \wedge (v_n \approx w_n)] \supset (X \supset Y)\}.$$

Definition 9.3.8 For a language L, by *eq*(L) we mean the set $\{\mathit{ref}\} \cup \mathit{fun}(L) \cup \mathit{rel}(L)$. Members of this set are called *equality axioms* for L.

We have now described all the equality axioms for a given language. Propositions 9.3.4 and 9.3.7 say they seem to have the power we need. In fact, they are *exactly* what we need; they suffice to reduce problems about logic with equality to more general questions about first-order logic. The following makes this precise:

Theorem 9.3.9 *Let L be a first-order language. Then*

$$S \models_{\approx} X \text{ if and only if } S \cup \mathit{eq}(L) \models_f X.$$

This theorem can be given a direct proof, but with little more work we can prove a version of the Model Existence Theorem that will have this as an easy consequence and that will be of use for other purposes as well. Consequently, we postpone the proof until Section 9.6. Note, however, the importance of this result. It implies that our earlier proof procedures can be used directly for logic with equality, provided only that we add the equality axioms. Whether this is always the most efficient way to proceed is another matter, however. It also implies that many theoretical results carry over rather directly to take equality into account.

Exercises

9.3.1. Give a proof by structural induction of Proposition 9.3.4.

9.3.2. Let B be the sentence $(\forall v_1)(\forall v_2)(\forall w_1)(\forall w_2)\{[(v_1 \approx w_1) \wedge (v_2 \approx w_2)] \supset [(v_1 \approx v_2) \supset (w_1 \approx w_2)]\}$, the replacement axiom for $\approx$. Show $\{B, \mathit{ref}\} \models_f (\forall x)(\forall y)(\forall z)[x \approx y \supset (y \approx z \supset x \approx z)]$.

9.3.3. Give a proof by structural induction of Proposition 9.3.7.

9.3.4. Use Theorem 9.3.9 and show the Compactness Theorem carries over to first-order logic with equality. Show it in the following form: $S \models_{\approx} X$ if and only if $S_0 \models_{\approx} X$ for some finite subset S_0 of S.

9.4 Hintikka's Lemma

As usual in our proofs of various versions of the Model Existence Theorem, we will prove Hintikka's Lemma first to get details of the model construction out of the way. The version we prove now is a direct extension of the one we proved earlier for first-order logic without equality. Later we will prove an alternative, strengthened version. The version proved here will not be superseded, however. We will use it in proving the stronger one.

In dealing with first-order logic without equality, we found that Herbrand models played an important role. We can not expect this now. After all, if the language L has infinitely many closed terms, any Herbrand model will have an infinite domain, but the sentence $(\forall x)(\forall y)(x \approx y)$, which is satisfiable, is never satisfiable in such a domain, assuming $\approx$ is interpreted as equality. Still, a relatively simple class of models plays the role that Herbrand models did when we were not considering equality. These are the *canonical models*. The idea is that in a canonical model every member of the domain has a name in the language. Herbrand models are also canonical, since members of the domain are closed terms of the language and name themselves. But there are canonical models that are not Herbrand.

Definition 9.4.1 A model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ for the language L is a *canonical model*, provided, for each member $d \in \mathbf{D}$, there is some closed term t of L such that $t^{\mathbf{I}} = d$.

In Section 5.3 we proved some useful results concerning Herbrand models. In fact, they extend readily to canonical models. The statements become a little more complicated because in canonical models assignments are not generally substitutions as well. But assignments give to variables values in the domain of the model, and those members of the domain have names in the language, names that can be used by a substitution.

Definition 9.4.2 Let $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ be a model for L; let σ be a grounding substitution, replacing variables by *closed terms* of L; and let $\mathbf{A}$ be an assignment in $\mathbf{M}$. We say σ and $\mathbf{A}$ *correspond*, provided, for each variable x, $x^{\mathbf{A}} = (x\sigma)^{\mathbf{I}}$. (Informally, for each variable x, $x\sigma$ is a name for the member $x^{\mathbf{A}}$ of $\mathbf{D}$.)

Proposition 9.4.3 *Suppose* $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ *is a canonical model for the language* L*, and assignment* $\mathbf{A}$ *and substitution* σ *correspond. Then for every term* t*, we have* $t^{\mathbf{I},\mathbf{A}} = (t\sigma)^{\mathbf{I}}$.

Proof By structural induction on t.

Basis cases If t is a variable, say x, then $t^{\mathbf{I},\mathbf{A}} = x^{\mathbf{I},\mathbf{A}} = x^{\mathbf{A}} = (x\sigma)^{\mathbf{I}} = (t\sigma)^{\mathbf{I}}$. Likewise, if t is a constant symbol, say c, then $t^{\mathbf{I},\mathbf{A}} = c^{\mathbf{I},\mathbf{A}} = c^{\mathbf{I}} = (c\sigma)^{\mathbf{I}} = (t\sigma)^{\mathbf{I}}$.

Induction step Suppose the result known for each term $t_1,\ldots,t_n$, and $t = f(t_1,\ldots,t_n)$. Then

$$\begin{aligned}
t^{\mathbf{I},\mathbf{A}} &= [f(t_1,\ldots,t_n)]^{\mathbf{I},\mathbf{A}} \\
&= f^{\mathbf{I}}(t_1^{\mathbf{I},\mathbf{A}},\ldots,t_n^{\mathbf{I},\mathbf{A}}) \\
&= f^{\mathbf{I}}((t_1\sigma)^{\mathbf{I}},\ldots,(t_n\sigma)^{\mathbf{I}}) \\
&= [f(t_1\sigma,\ldots,t_n\sigma)]^{\mathbf{I}} \\
&= [f(t_1,\ldots,t_n)\sigma]^{\mathbf{I}} \\
&= (t\sigma)^{\mathbf{I}}.
\end{aligned}$$

□

Proposition 9.4.4 *Suppose* $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ *is a canonical model for the language* L, *and assignment* $\mathbf{A}$ *and substitution* σ *correspond. For a formula* Φ *of* L, $\Phi^{\mathbf{I},\mathbf{A}} = (\Phi\sigma)^{\mathbf{I}}$.

Now we have established the basic properties of canonical models, and we turn to Hintikka's Lemma itself. An example should help clarify the basic ideas of the proof.

It is entirely possible to have a model in which all members of $eq(L)$ are true, but which is not normal. The following is a simple, and probably familiar, example. Let L be the first-order language with $\approx$ as the only relation symbol, and with the two-place function symbol $+$. Let $\mathbf{M} = \langle \mathbf{Z}, \mathbf{I} \rangle$ be the model in which $\mathbf{Z}$ is the set of signed integers, $+^{\mathbf{I}}$ is addition, and $\approx^{\mathbf{I}}$ is congruence modulo 2. When interpreted in this model, the equality axiom for $+$ says that the addition of congruences produces another congruence, which is true. Likewise, the equality axiom for $\approx$ is true, because $\approx^{\mathbf{I}}$ is an equivalence relation. Still, the model is not normal.

Even though we do not have a normal model, there is a standard procedure for constructing a normal model from it. Partition $\mathbf{Z}$ into congruence classes, and use these as the domain of a new model. In this new model, interpret $+$ to be addition modulo 2. The resulting model is usually designated $\mathbf{Z}_2$. Note that its domain is $\mathbf{Z}/\approx^{\mathbf{I}}$. This construction generalizes and forms the basis for the proof of Hintikka's Lemma.

Definition 9.4.5 Let L be a first-order language. A set $\mathbf{H}$ of sentences of L is a *first-order Hintikka set with equality*, provided $\mathbf{H}$ is a first-order Hintikka set (meeting the conditions of Definitions 3.5.1 and 5.6.1), and in addition:

8. $t \approx t \in \mathbf{H}$ for every closed term t of L.

9. $t_1 \approx u_1, \ldots, t_n \approx u_n \in \mathbf{H} \Rightarrow f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n) \in \mathbf{H}$ for every n-place function symbol f of L.

10. $t_1 \approx u_1, \ldots, t_n \approx u_n, R(t_1, \ldots, t_n) \in \mathbf{H} \Rightarrow R(u_1, \ldots, u_n) \in \mathbf{H}$ for every n-place relation symbol R of L.

Now we state and prove a version of Hintikka's Lemma that takes equality into account. In the proof we first construct a model that may not be normal, but in which $\approx$ is interpreted by some *equivalence* relation. Then this model is used to produce a 'factor' model, where the members of the domain are *equivalence classes* from the old domain. This construction is common throughout mathematics. You may have seen it used to produce factor groups in a course on group theory. The principal items about equivalence classes that we will need are contained in Exercise 9.4.3.

Proposition 9.4.6 **(Hintikka's Lemma)** *Suppose L is a language with a nonempty set of closed terms. If* $\mathbf{H}$ *is a first-order Hintikka set with equality with respect to L, then* $\mathbf{H}$ *is satisfiable in a normal model. More precisely,* $\mathbf{H}$ *is satisfiable in a normal model that is canonical with respect to L.*

Proof We can think of the equality requirements on a Hintikka set $\mathbf{H}$ as saying that, as far as $\mathbf{H}$ is concerned, $\approx$ is reflexive and has the replacement property. In fact, a Hintikka set $\mathbf{H}$ also 'thinks' $\approx$ is symmetric and transitive, by the following argument. First, suppose $t \approx u \in \mathbf{H}$. We also have $t \approx t \in \mathbf{H}$ (one of the Hintikka set conditions), so by replacement with respect to relation symbols (another of the Hintikka set conditions), we can derive that $u \approx t \in \mathbf{H}$ by replacing the first occurrence of t in $t \approx t$ by u. Similarly, if $t \approx u$ and $u \approx v$ are both in $\mathbf{H}$, we will also have $t \approx v \in \mathbf{H}$. Incidentally, the similarity of this reasoning to the informal arguments presented in the introduction to this chapter is not a coincidence.

Now, let $\mathbf{H}$ be a Hintikka set with equality, with respect to the language L. We will produce a canonical, normal model in which the members of $\mathbf{H}$ are true.

Ignoring the role of equality for the moment, $\mathbf{H}$ is a first-order Hintikka set according to Definition 5.6.1 so by the first-order version of Hintikka's Lemma 5.6.2, $\mathbf{H}$ is satisfiable in a Herbrand model. The problem is that this model may not be normal. But we can use it as a basis for the construction of one that is normal.

For the rest of this proof, $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ is the Herbrand model that was constructed in the earlier proof of Hintikka's Lemma 5.6.2: $\mathbf{D}$ is the set of closed terms of L; $\mathbf{I}$ is such that, for each closed term t, $t^{\mathbf{I}} = t$; and for an atomic sentence A, A is true in $\mathbf{M}$ if and only if $A \in \mathbf{H}$. (The first two of these conditions just say the model is Herbrand. The third condition is more special.)

$\approx$ is a two-place relation symbol, so $\approx^{\mathbf{I}}$ is a two-place relation on $\mathbf{D}$. $t \approx^{\mathbf{I}} u$ is true if and only if $t \approx u \in \mathbf{H}$. Since $\mathbf{H}$ 'thinks' $\approx$ is reflexive, symmetric, and transitive, $\approx^{\mathbf{I}}$ actually is an equivalence relation on the domain $\mathbf{D}$. An equivalence relation partitions its domain into disjoint *equivalence classes*. We denote the equivalence class containing t by $\langle\langle t\rangle\rangle$. Formally, $\langle\langle t\rangle\rangle = \{u \in \mathbf{D} \mid t \approx^{\mathbf{I}} u\}$. As usual, $\langle\langle t\rangle\rangle = \langle\langle u\rangle\rangle$ if and only if $t \approx^{\mathbf{I}} u$ (see Exercise 9.4.3). Finally, we let $\mathbf{D}^*$ be the set of equivalence classes, $\{\langle\langle d\rangle\rangle \mid d \in \mathbf{D}\}$. We take $\mathbf{D}^*$ to be the domain of a new model.

A new interpretation $\mathbf{I}^*$ must also be defined. For a constant symbol c we simply set $c^{\mathbf{I}^*} = \langle\langle c\rangle\rangle$. For a function symbol f, we set

$$f^{\mathbf{I}^*}(\langle\langle t_1\rangle\rangle, \ldots, \langle\langle t_n\rangle\rangle) = \langle\langle f(t_1, \ldots, t_n)\rangle\rangle.$$

It must be shown that this is well-defined, since the behavior of $f^{\mathbf{I}^*}$ on the *class* $\langle\langle t_i\rangle\rangle$ depends on t_i, a *member* of the class, and the class generally will have many members. We leave it to you to verify that our definition of $\mathbf{I}^*$ on function symbols is well-defined, in Exercise 9.4.4.

As a consequence of the definitions so far, for any closed term t, $t^{\mathbf{I}^*} = \langle\langle t\rangle\rangle$. This can be proved by structural induction and is left to you as Exercise 9.4.5. This implies the model we are constructing is canonical; the member $\langle\langle t\rangle\rangle$ of the domain $\mathbf{D}$ will have the closed term t as a name.

Finally, for a relation symbol R, we set $R^{\mathbf{I}^*}(\langle\langle t_1\rangle\rangle, \ldots, \langle\langle t_n\rangle\rangle)$ to have the same truth value as $R^{\mathbf{I}}(t_1, \ldots, t_n)$. That is, $\langle\langle\langle t_1\rangle\rangle, \ldots, \langle\langle t_n\rangle\rangle\rangle \in R^{\mathbf{I}^*}$ if and only if $\langle t_1, \ldots, t_n\rangle \in R^{\mathbf{I}}$. In particular, $\langle\langle t\rangle\rangle \approx^{\mathbf{I}^*} \langle\langle u\rangle\rangle$ if and only if $t \approx^{\mathbf{I}} u$. Once again it must be established that our definition is proper, since whether or not $R^{\mathbf{I}^*}$ holds of *equivalence classes* depends on whether or not $R^{\mathbf{I}}$ holds of particular *members*. We leave it to you, in Exercise 9.4.6, to show that the choice of members makes no difference.

Now the interpretation $\mathbf{I}^*$ has been completely defined. Let $\mathbf{M}^*$ be the model $\langle \mathbf{D}^*, \mathbf{I}^*\rangle$. We claim $\mathbf{M}^*$ is the desired model.

First, $\mathbf{M}^*$ is normal because $\langle\langle t\rangle\rangle \approx^{\mathbf{I}^*} \langle\langle u\rangle\rangle$ if and only if $t \approx^{\mathbf{I}} u$ if and only if $\langle\langle t\rangle\rangle = \langle\langle u\rangle\rangle$. Also, as we observed earlier, $\mathbf{M}^*$ is canonical. Finally, we will show that the same sentences of L are true in $\mathbf{M}$ and $\mathbf{M}^*$, which will complete the proof, since all members of $\mathbf{H}$ are true in $\mathbf{M}$. To show this we must show something a bit stronger, allowing formulas with free variables. And for this we need some notation.

Let $\mathbf{A}$ be an assignment in the model $\mathbf{M}$. We denote by $\mathbf{A}^*$ the assignment in $\mathbf{M}^*$ such that $x^{\mathbf{A}^*} = \langle\langle x^{\mathbf{A}}\rangle\rangle$. A proof by structural induction will show that for any term t, not necessarily closed, $t^{\mathbf{I}^*,\mathbf{A}^*} = \langle\!\langle t^{\mathbf{I},\mathbf{A}} \rangle\!\rangle$. We leave this to you, as Exercise 9.4.7.

We show the following: For any formula Φ and any assignment $\mathbf{A}$ in $\mathbf{M}$, $\Phi^{\mathbf{I}^*,\mathbf{A}^*} = \Phi^{\mathbf{I},\mathbf{A}}$. Then in particular, each sentence has the same truth values in $\mathbf{M}$ and in $\mathbf{M}^*$.

Basis Case Say $\Phi = R(t_1, \ldots, t_n)$. Then

$$\begin{aligned}
[R(t_1, \ldots, t_n)]^{\mathbf{I}^*,\mathbf{A}^*} &= R^{\mathbf{I}^*}(t_1^{\mathbf{I}^*,\mathbf{A}^*}, \ldots, t_n^{\mathbf{I}^*,\mathbf{A}^*}) \\
&= R^{\mathbf{I}^*}(\langle\!\langle t_1^{\mathbf{I},\mathbf{A}} \rangle\!\rangle, \ldots, \langle\!\langle t_n^{\mathbf{I},\mathbf{A}} \rangle\!\rangle) \\
&= R^{\mathbf{I}}(t_1^{\mathbf{I},\mathbf{A}}, \ldots, t_n^{\mathbf{I},\mathbf{A}}) \\
&= [R(t_1, \ldots, t_n)]^{\mathbf{I},\mathbf{A}}.
\end{aligned}$$

Induction Cases The propositional cases are straightforward and are left to you. We consider one of the quantifier cases; the other is similar. Suppose Φ is $(\exists x)\Psi$, the result is known for formulas simpler than Φ, and $\mathbf{A}$ is an assignment in $\mathbf{M}$.

Suppose $[(\exists x)\Psi]^{\mathbf{I},\mathbf{A}} = \mathbf{t}$. Then for some x-variant $\mathbf{B}$ of $\mathbf{A}$, $\Psi^{\mathbf{I},\mathbf{B}} = \mathbf{t}$. By the induction hypothesis, $\Psi^{\mathbf{I}^*,\mathbf{B}^*} = \mathbf{t}$. But $\mathbf{B}^*$ must be an x-variant of $\mathbf{A}^*$, and so $[(\exists x)\Psi]^{\mathbf{I}^*,\mathbf{A}^*} = \mathbf{t}$.

Suppose $[(\exists x)\Psi]^{\mathbf{I}^*,\mathbf{A}^*} = \mathbf{t}$. Then for some x-variant $\mathbf{V}$ of $\mathbf{A}^*$, $\Psi^{\mathbf{I}^*,\mathbf{V}} = \mathbf{t}$. Define an assignment $\mathbf{B}$ in $\mathbf{M}$ as follows: On variables other than x, $\mathbf{B}$ agrees with $\mathbf{A}$. And on x, $x^{\mathbf{A}}$ is some arbitrary member of $x^{\mathbf{V}}$. ($x^{\mathbf{V}}$ is a member of $\mathbf{D}^*$ and hence is an equivalence class; choose any member.) Then $\mathbf{B}$ is an x-variant of $\mathbf{A}$, and it is easy to see that $\mathbf{B}^* = \mathbf{V}$. Then $\Psi^{\mathbf{I}^*,\mathbf{B}^*} = \mathbf{t}$, so by the induction hypothesis $\Psi^{\mathbf{I},\mathbf{B}} = \mathbf{t}$, and hence $[(\exists x)\Psi]^{\mathbf{I},\mathbf{A}} = \mathbf{t}$.

This concludes the proof of Hintikka's Lemma. □

Exercises

9.4.1. Show that if $\mathbf{M}$ is any model, then for every substitution, there is a corresponding assignment, and if $\mathbf{M}$ is canonical, then for every assignment, there is also a corresponding substitution.

9.4.2. Show that condition 8 of Definition 9.4.5 can be replaced by $c \approx c \in \mathbf{H}$ for every constant symbol c of L. (Note that this is actually a special case of condition 9.)

9.4.3. (Standard facts about equivalence relations.) Suppose $\sim$ is an equivalence relation on a set $\mathbf{D}$. For each $d \in \mathbf{D}$, let $\langle\!\langle d \rangle\!\rangle = \{x \in \mathbf{D} \mid x \sim d\}$. These subsets are called *equivalence classes*. Show the following:

1. Every member of $\mathbf{D}$ belongs to some equivalence class.

2. No member of $\mathbf{D}$ belongs to more than one equivalence class. More precisely, show that if $c \in \langle\!\langle d_1 \rangle\!\rangle$ and $c \in \langle\!\langle d_2 \rangle\!\rangle$, then $\langle\!\langle d_1 \rangle\!\rangle = \langle\!\langle d_2 \rangle\!\rangle$.
3. $d_1 \sim d_2$ if and only if $\langle\!\langle d_1 \rangle\!\rangle = \langle\!\langle d_2 \rangle\!\rangle$.

9.4.4. Show that if $\langle\!\langle t_1 \rangle\!\rangle = \langle\!\langle u_1 \rangle\!\rangle, \ldots, \langle\!\langle t_n \rangle\!\rangle = \langle\!\langle u_n \rangle\!\rangle$ then

$$\langle\!\langle f(t_1, \ldots, t_n) \rangle\!\rangle = \langle\!\langle f(u_1, \ldots, u_n) \rangle\!\rangle.$$

9.4.5. Show that for any closed term t of L, $t^{\mathbf{I}^*} = \langle\!\langle t \rangle\!\rangle$.

9.4.6. Show that if $\langle\!\langle t_1 \rangle\!\rangle = \langle\!\langle u_1 \rangle\!\rangle, \ldots, \langle\!\langle t_n \rangle\!\rangle = \langle\!\langle u_n \rangle\!\rangle$ then $R^{\mathbf{I}}(t_1, \ldots, t_n)$ and $R^{\mathbf{I}}(u_1, \ldots, u_n)$ have the same truth values.

9.4.7. Show that for a term t, not necessarily closed, $t^{\mathbf{I}^*, \mathbf{A}^*} = \langle\!\langle t^{\mathbf{I}, \mathbf{A}} \rangle\!\rangle$.

9.5 The Model Existence Theorem

The Model Existence Theorem for first-order logic easily extends to encompass equality, now that we have done the work of proving a suitable version of Hintikka's Lemma.

Definition 9.5.1 Let L be a first-order language, and let $\mathcal{C}$ be a collection of sets of sentences of $L^{\mathbf{par}}$. We call $\mathcal{C}$ a *first-order consistency property with equality* (with respect to L) if it is a first-order consistency property (Definitions 3.6.1 and 5.8.1) and, in addition, for each $S \in \mathcal{C}$:

8. $S \cup \{t \approx t\} \in \mathcal{C}$ for every closed term t of $L^{\mathbf{par}}$.
9. $t_1 \approx u_1, \ldots, t_n \approx u_n \in S \Rightarrow S \cup \{f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n)\} \in \mathcal{C}$ for every n-place function symbol f of L.
10. $t_1 \approx u_1, \ldots, t_n \approx u_n, R(t_1, \ldots, t_n) \in S \Rightarrow S \cup \{R(u_1, \ldots, u_n)\} \in \mathcal{C}$ for every n-place relation symbol R of L.

Theorem 9.5.2 **(Model Existence Theorem With Equality)**
If $\mathcal{C}$ is a first-order consistency property with equality with respect to L and $S \in \mathcal{C}$, then S is satisfiable in a normal model; in fact S is satisfiable in a normal model that is canonical with respect to $L^{\mathbf{par}}$.

Proof In Chapter 5 we proved a version of the Model Existence Theorem 5.8.2 that did not take equality into account. We showed that if a set S was a member of a first-order consistency property, then S could be extended to a first-order Hintikka set. That proof, with no essential changes, also shows that if S is a member of a first-order consistency property *with equality*, then S can be extended to a Hintikka set *with equality*. We leave it to you to verify this. You should show that if one starts with a first-order consistency property $\mathcal{C}$ meeting the conditions for equality, at each stage of the construction used in the proof of Theorem 5.8.2, one produces a consistency property, or an alternate consistency property, that also meets these equality conditions, and hence the resulting Hintikka set will also be one with equality.

Now if $S \in \mathcal{C}$, where $\mathcal{C}$ is a first-order consistency property with equality, S extends to a Hintikka set with equality, and this is satisfiable in a normal, canonical model by Hintikka's Lemma 9.4.6, completing the argument. □

Exercises

9.5.1. Supply the missing steps of the proof for Theorem 9.5.2.

9.6 Consequences

Earlier we introduced the equality axioms and stated a result that said they did what they were supposed to do, Theorem 9.3.9. Now we have the machinery available to prove that result. For convenience we begin by restating it.

Theorem 9.3.9 *Let L be a first-order language. Then $S \models_{\approx} X$ if and only if $S \cup eq(L) \models_f X$.*

Proof One direction is trivial. Suppose $S \cup eq(L) \models_f X$. Let $\mathbf{M}$ be any normal model in which the members of S are true. The members of $eq(L)$ are trivially true in every normal model, $\mathbf{M}$ in particular. Then X is true in $\mathbf{M}$ as well. Since $\mathbf{M}$ was an arbitrary normal model, $S \models_{\approx} X$.

For the converse direction, we use the Model Existence Theorem. Form a collection $\mathcal{C}$ of sets of sentences of L^{par} as follows: Put W in $\mathcal{C}$ if infinitely many parameters are new to W, and $W \cup eq(L)$ is satisfiable in some, not necessarily normal, model. We leave it to you to check that $\mathcal{C}$ is a first-order consistency property with equality. Now, suppose we do not have that $S \cup eq(L) \models_f X$. Since S and X are from the language L, they have no parameters. It follows that $S \cup \{\neg X\} \in \mathcal{C}$. Then by Theorem 9.5.2, this set is satisfiable in a *normal* model, and hence we do not have $S \models_{\approx} X$. □

We conclude this section by rounding up the usual consequences of the Model Existence Theorem. These can also be derived from Theorem 9.3.9.

Theorem 9.6.1 **(Compactness With Equality)** *Let S be a set of sentences of L. If every finite subset of S is satisfiable in a* normal *model, so is S.*

Proof Exactly as we proved Theorem 5.9.1. □

Theorem 9.6.2 **(Löwenheim-Skolem)** *Let S be a set of sentences of L.*

1. *If S is satisfiable in a normal model, S is satisfiable in a normal model that is finite or countable.*
2. *If S is satisfiable in an infinite normal model, S is satisfiable in a normal model that is countable.*

Proof For part 1 we follow the proof of the version of the Löwenheim-Skolem Theorem that did not take equality into account, Theorem 5.9.3. Form a collection $\mathcal{C}$ of sets of sentences of L^{par} as follows: Put a set W in $\mathcal{C}$, provided infinitely many parameters are new to W and W is satisfiable in a normal model. It is easy to verify that $\mathcal{C}$ is a consistency property with equality. If S is satisfiable in a normal model, $S \in \mathcal{C}$, so by the Model Existence Theorem, S will be satisfiable in a canonical normal model. Since every member of the domain of a canonical model is named by some closed term, and the collection of closed terms is countable, the domain must be countable or finite.

Part 2 follows from part 1 with a little extra work. Suppose S is satisfiable in an infinite normal model. In Section 9.2 we defined sentences $E_{>n}$ for each n, so that $E_{>n}$ is true in a normal model if and only if the model has more than n members in its domain. Since S is satisfiable in an infinite normal model, so is $S \cup \{E_{>1}, E_{>2}, \ldots\}$. Then by part 1, this set is satisfiable in a normal model that is finite or countable. Since each $E_{>n}$ must be true in it, the model can not be finite, so S must be satisfiable in a countable normal model. □

Theorem 9.6.3 **(Canonical Model)** *Let S be a set of sentences of L. If S is satisfiable in a normal model, then S is satisfiable in a normal model that is canonical with respect to L^{par}.*

The Compactness Theorem makes it possible to construct *nonstandard* versions of several standard mathematical structures. We illustrate this with arithmetic. Suppose we take for L a language suitable for discussing the natural numbers. Say it has constant symbols c_0, c_1, c_2,..., which we can take as names for the natural numbers. It should also have function symbols $+$, $\times$, and whatever others you may find useful. It should have relation symbols $\approx$, $>$, and so on. The *standard model* for this language is the normal model $\langle \mathbf{N}, \mathbf{I} \rangle$, where $\mathbf{N}$ is the set $\{0, 1, 2, \ldots\}$, $c_i^{\mathbf{I}} = i$, $+^{\mathbf{I}}$ is the addition operation, $>^{\mathbf{I}}$ is the greater-than relation, and so on. Let T be the set of all sentences of L that are true in this standard model. T is the *truth set* of arithmetic.

Now, let L' be the language that is like L but with d added as a new constant symbol. Consider the set $S = T \cup \{d > c_0, d > c_1, \ldots\}$. Every finite subset of S is satisfiable. Indeed, any finite subset of S can be made true in the standard model once a suitable interpretation for d is given, and that is easy. A finite subset of S will contain only finitely many sentences of the form $d > c_i$; simply interpret d to be any integer that is bigger than each of the finitely many $c_i^{\mathbf{I}}$. Since every finite subset of S is satisfiable, by the Compactness Theorem, the entire of S is satisfiable, say in the normal model $\langle \mathbf{M}, \mathbf{J} \rangle$.

The model $\langle \mathbf{M}, \mathbf{J} \rangle$ is a nonstandard model for arithmetic. Suppose X is a sentence of L. If X is true in the standard model $\langle \mathbf{N}, \mathbf{I} \rangle$, X will be a member of T, hence a member of S, and so true in the nonstandard model $\langle \mathbf{M}, \mathbf{J} \rangle$. On the other hand, if X is not true in the standard model $\neg X$ will be, so by the same argument, $\neg X$ will be true in the nonstandard model, and X will be false. It follows that the standard model and the nonstandard model agree on the truth values of every sentence that does not involve the constant symbol d; they agree on all of arithmetic. Indeed, it is even possible to show the nonstandard model has a natural submodel that is isomorphic to the standard one.

On the other hand, in the nonstandard model, there must be a member $d^{\mathbf{J}}$ in the domain. Since every sentence of S is true in this model, this member must be $>^{\mathbf{J}}$ any standard integer, anything of the form $c_i^{\mathbf{J}}$. In effect, there must be an 'infinite integer'. But all the usual laws of arithmetic apply to this infinite integer. For example, $(\forall x)(x + c_0 \approx x)$ is true in the standard model, hence in the nonstandard model, and hence $d^{\mathbf{J}} +^{\mathbf{J}} c_0^{\mathbf{J}} = d^{\mathbf{J}}$, and so on.

Nonstandard models for arithmetic were first created by Skolem [46, 47], using different machinery. In 1961 Abraham Robinson showed that nonstandard models of the *real numbers* had remarkable properties [39, 40]. Since the early days of the calculus, infinitesimals have played an important role, though their use could not be justified. Robinson showed there are nonstandard models of the reals in which infinitesimals exist and in which they can be used to help establish the truth of assertions. Then, if the sentence whose truth has been established does not mention infinitesimals, it will be true in the standard model as well, for much the same reasons that a sentence that does not involve d will have the same truth value in the two models for arithmetic we considered earlier. Since then, *nonstandard analysis* has become a subject of considerable general mathematical interest. This is as far as we can take the discussion here, however.

Exercises

9.6.1. Prove Theorem 9.6.3.

9.6.2. Use the results of this section and show that the Craig Interpolation Theorem for first-order logic, Theorem 8.11.5, extends to first-order logic with equality, in the following form.

If the sentence $X \supset Y$ is valid in all normal models, then there is a sentence Z such that

1. $X \supset Z$ and $Z \supset Y$ are both valid in all normal models.
2. Every relation, constant, and function symbol of Z *except possibly for* $\approx$ occurs in both X and Y.

9.6.3. Using the previous exercise, show that Beth's Definability Theorem 8.13.3 extends to first-order logic with equality.

9.7 Tableau and Resolution Systems

Theorem 9.3.9 tells us that adding the equality axioms to a proof procedure for first-order logic is sufficient to fully capture the notion of equality. For Hilbert systems this is the best way to proceed. For tableau and resolution, however, it is not as appropriate. It is better to formulate additional Tableau or Resolution Expansion Rules to capture the essential properties of equality. We do so in this section, producing proof procedures that use parameters and are best suited for hand calculation. Later we will modify these systems, producing free-variable versions better adapted to implementation.

The simplest essential property of equality is reflexivity, but there are different ways we might build this into a proof procedure. We could take reflexivity to be the assertion that we always have $t \approx t$, or the assertion that we never have $\neg(t \approx t)$. The first version sounds like it could be a tableau expansion rule; the second sounds more like a branch closure condition. In a formal proof procedure, these two might behave differently, and so care must be exercised in our formulations.

The replacement principle of equality could say that, given $t \approx u$ we can replace t occurrences by u occurrences, or it could also say we can replace u occurrences by t occurrences, thus building in symmetry. It turns out that whether we must build in symmetry this way is not independent of how we treat reflexivity. In this section we take one route; in the next we explore another.

For the rest of this section, we assume L is a first-order language (with $\approx$) and L^{par} is the usual extension with a countable set of new constant symbols, parameters.

To start, we need convenient notation with which to state a Replacement Rule. Suppose $\Phi(x)$ is a formula containing free occurrences of x. Then we will write $\Phi(t)$ for $\Phi(x)\{x/t\}$, where t is a term. The notation $\Phi(t)$ by itself is really incomplete, because it does not specify the variable for which t has been substituted, but generally we will use such notation, leaving it to context to supply the missing information. Now, suppose $\Phi(x)$ is a formula and t and u are two distinct terms. Then $\Phi(u)$ differs from $\Phi(t)$ in that $\Phi(u)$ has occurrences of u at some of the places that $\Phi(t)$ has occurrences of t. It need not be the case that *all* occurrences of t have been replaced by u. For instance, say $\Phi(x) = R(x, a)$; then $\Phi(a) = R(a, a)$ but $\Phi(b) = R(b, a)$. In effect, $R(b, a)$ is $R(a, a)$ with only the first occurrence of a replaced by b. Generally, we will leave $\Phi(x)$ unspecified, write $\Phi(t)$, and later write $\Phi(u)$; the intention is that it is like $\Phi(t)$ except that some occurrences of t, not necessarily all, have been replaced by occurrences of u.

Tableau Equality Rules We build on the first-order tableau system with quantifier rules given in Chapter 6. Free-variable occurrences are not allowed; only sentences appear in proofs; strictness is not required.

Tableau Reflexivity Rule The sentence $t \approx t$ can be added to the end of any branch of a tableau for S, producing another tableau for S. Here t is any closed term of L^{par}. Schematically:

$$\frac{}{t \approx t}$$

Tableau Replacement Rule If $t \approx u$ and $\Phi(t)$ occur on a branch of a tableau for S (in either order), $\Phi(u)$ can be added to the end, producing another tableau for S. Here t and u are any closed terms of L^{par}. Schematically:

$$\frac{\begin{array}{c} t \approx u \\ \Phi(t) \end{array}}{\Phi(u)}$$

Notice that the Replacement Rule is *directional.* It allows us to replace terms on the left-hand side of an equality by terms on the right-hand side, but not conversely. Also Reflexivity has been treated as an expansion, rather than as a closure rule.

Here are some examples of tableau proofs using these rules (and of course the propositional and first-order rules from earlier).

Transitivity: $(\forall x)(\forall y)(\forall z)((x \approx y \land y \approx z) \supset x \approx z)$

1. $\neg(\forall x)(\forall y)(\forall z)((x \approx y \wedge y \approx z) \supset x \approx z)$
2. $\neg(\forall y)(\forall z)((a \approx y \wedge y \approx z) \supset a \approx z)$
3. $\neg(\forall z)((a \approx b \wedge b \approx z) \supset a \approx z)$
4. $\neg((a \approx b \wedge b \approx c) \supset a \approx c)$
5. $a \approx b \wedge b \approx c$
6. $\neg(a \approx c)$
7. $a \approx b$
8. $b \approx c$
9. $\neg(b \approx c)$

In this example, 2 through 4 are by the δ-rule, 5 through 8 are by the α-rule. Finally, 9 is by Replacement, using 7 to replace a by b in 6. Closure is by 8 and 9. This is not the only way of giving a proof. We could have used 8 to replace b by c in 7 instead.

Symmetry: $(\forall x)(\forall y)(x \approx y \supset y \approx x)$

1. $\neg(\forall x)(\forall y)(x \approx y \supset y \approx x)$
2. $\neg(\forall y)(a \approx y \supset y \approx a)$
3. $\neg(a \approx b \supset b \approx a)$
4. $a \approx b$
5. $\neg(b \approx a)$
6. $\neg(b \approx b)$
7. $b \approx b$

Here 6 is by Replacement, using 4 to replace a by b in 5; 7 is by the Reflexive Rule. Closure is by 6 and 7.

Resolution Equality Rules Here too, we build on the first-order resolution system of Chapter 6. Free variables are not allowed; strictness is not required.

Resolution Reflexivity Rule Adding the clause $[t \approx t]$ to a resolution expansion for S produces another resolution expansion for S, for any closed term t of L^{par}. Schematically:

$$\frac{}{[t \approx t]}$$

Resolution Replacement Rule Suppose we have a resolution expansion for S containing the generalized disjunctions D_1 and D_2, where D_1 contains $t \approx u$ and D_2 contains $\Phi(t)$. Then we produce a new resolution expansion for S if we add a new generalized disjunction D to the resolution expansion, where D consists of the members of D_1 except for $t \approx u$, the members of D_2 except for $\Phi(t)$, and also $\Phi(u)$. Schematically:

$$\frac{\begin{array}{c}[A_1, \ldots, A_n, t \approx u] \\ [B_1, \ldots, B_k, \Phi(t)]\end{array}}{[A_1, \ldots, A_n, B_1, \ldots, B_k, \Phi(u)]}$$

Here is an example, giving a resolution proof of transitivity. Symmetry has a similar proof.

Transitivity: $(\forall x)(\forall y)(\forall z)((x \approx y \wedge y \approx z) \supset x \approx z)$.

1. $[\neg(\forall x)(\forall y)(\forall z)((x \approx y \wedge y \approx z) \supset x \approx z)]$
2. $[\neg(\forall y)(\forall z)((a \approx y \wedge y \approx z) \supset a \approx z)]$
3. $[\neg(\forall z)((a \approx b \wedge b \approx z) \supset a \approx z)]$
4. $[\neg((a \approx b \wedge b \approx c) \supset a \approx c)]$
5. $[a \approx b \wedge b \approx c]$
6. $[\neg(a \approx c)]$
7. $[a \approx b]$
8. $[b \approx c]$
9. $[\neg(b \approx c)]$
10. $[]$

Here 2 through 4 are by the δ-rule; 5 and 6 are from 4 by the α-rule; 7 and 8 are from 5, again by α. Now 9 is by the Replacement Rule, applied to 6 and 7. Then 10 is from 8 and 9.

Theorem 9.7.1 **(Soundness)** *If the sentence X of L has a tableau or resolution proof, using the systems of this section, X is true in all normal models.*

Proof When soundness of the first-order tableau and resolution systems without equality was shown, Lemma 6.3.2 was used. It said the application of a Tableau or Resolution Rule will not affect satisfiability. This time we need a slightly different result: The application of a Tableau Expansion Rule, or an Equality Rule, to a tableau that is satisfiable in a *normal* model will produce another tableau that is satisfiable in a normal model, and similarly for resolution. It is easy to check that this is so. With this modified Lemma used in place of Lemma 6.3.2, the soundness proof itself is exactly the same as the proof of Theorem 6.3.3. □

Theorem 9.7.2 **(Completeness)** *If the sentence X of L is true in all normal models, X has a resolution and a tableau proof, using the systems of this section.*

Proof Call a finite set of sentences of L^{par} *tableau consistent* if no tableau for it closes, even allowing the equality rules. It must be shown that the collection of tableau-consistent sets is a first-order consistency property with equality. Much of this is exactly as in the proof of Lemma 6.4.2. What is new is the equality conditions. We check one of these. Suppose S is a finite set of sentences, $t_1 \approx u_1, \ldots, t_n \approx u_n \in S$, but $S \cup \{f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n)\}$ is not tableau consistent; we show S itself is not tableau consistent. By assumption, there is a closed tableau $\mathbf{T}$ for $S \cup \{f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n)\}$; we convert that into a closed tableau for S as follows:

First, begin a tableau with the members of S, giving a one-branch tableau. Next, add the sentence $f(t_1, \ldots, t_n) \approx f(t_1, \ldots, t_n)$ to the branch end, which is allowed by the reflexive rule. Each sentence $t_1 \approx u_1, \ldots, t_n \approx u_n$ is in S and hence is on the tableau branch, so n applications of the Tableau Replacement Rule starting with $f(t_1, \ldots, t_n) \approx f(t_1, \ldots, t_n)$ will allow us to add $f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n)$ to the branch. Finally, complete the construction of a closed tableau by copying the steps of tableau $\mathbf{T}$.

Once it is known that tableau consistency is a first-order consistency property with equality, completeness of the tableau system follows from the Model Existence Theorem 9.5.2 in the usual way.

Completeness of the resolution system has a similar proof. □

Although certain tableau rules are taken as official, other rules may be allowed without harm. These are the *derived rules.* A derived rule is one such that any application of it can be translated into a sequence of applications of the official rules. For instance, the following is a derived tableau rule: If X and $X \supset Y$ occur on a branch, Y may be added. Any application of this rule can be translated away as follows: Suppose X and $X \supset Y$ both occur on θ, and we want to add Y using only official rules. Well, since $X \supset Y$ occurs, using the β-rule we can split θ into two branches, call them θ_1 and θ_2, each like θ except that θ_1 ends with $\neg X$ and θ_2 ends with Y. Since X occurs on θ, hence on θ_1, the branch θ_1 is closed, and we are left with only θ_2 open, which is, indeed, θ with Y added at the end.

The way the Replacement Rule was stated for tableaux, it allowed only left-right replacements. In fact right-left replacements can be allowed, as a derived rule. We leave it to you to verify this, as an exercise. Similar remarks apply to the resolution system as well, of course.

Exercises

9.7.1. Give tableau proofs of the following:

1. $(\forall x)(\exists y)(x \approx y)$
2. $(\forall x)(\forall y)(\forall z)(\forall w)((x \approx y \wedge z \approx w) \supset (R(x,z) \supset R(y,w)))$
3. $(\forall x)(\forall y)((x \approx y \wedge f(y) \approx g(y)) \supset h(f(x)) \approx h(g(y)))$

9.7.2. Suppose the language L also includes the two-place function symbol $\cap$ and the two-place relation symbols $\in$ and $\subseteq$. For reading ease we will write all of these in infix form, $t \cap u$ instead of $\cap(t,u)$, and so on. Let $A = (\forall x)(\forall y)[(\forall z)(z \in x \supset z \in y) \supset x \subseteq y]$. Let $B = (\forall x)(\forall y)(\forall z)[(x \in y \cap z) \supset (x \in y \wedge x \in z)]$. Finally, let $C = (\forall x)(\forall y)[(x \approx x \cap y) \supset x \subseteq y]$. Give a tableau proof of $(A \wedge B) \supset C$.

9.7.3. A cancellation law for groups is usually the first thing one proves when beginning the study of group theory. To formalize this we need the following machinery: Suppose L is a language with constant symbol e (for the group identity), the one-place function symbol i (for the group inverse), and the two-place function symbol $\circ$ (for the group operation). For reading ease we will write $\circ$ using infix notation. Let

$assoc = (\forall x)(\forall y)(\forall z)(x \circ (y \circ z) \approx (x \circ y) \circ z)$

$id = (\forall x)(x \circ e \approx x \wedge e \circ x \approx x)$

$inv = (\forall x)(x \circ i(x) \approx e \wedge i(x) \circ x \approx e)$

$canc = (\forall x)(\forall y)(\forall z)(x \circ z \approx y \circ z \supset x \approx y)$

Give a tableau proof of $((assoc \wedge id) \wedge inv) \supset canc$.

9.7.4. Use the resolution system to redo Exercises 9.7.1, 9.7.2, and 9.7.3.

9.7.5. Formulate a version of tableaux or resolution with equality that includes an S-introduction Rule. Then prove a Strong Completeness Theorem.

9.7.6. Show the tableau system is complete if the Tableau Replacement Rule is restricted by requiring that $\Phi(t)$ be *atomic*, and similarly for the resolution system.

9.7.7. Show the following are derived Tableau Rules:

1. If $t \approx u$ occurs on a branch then $u \approx t$ can be added to the end.
2. If $u \approx t$ and $\Phi(t)$ occur on a branch (in either order), $\Phi(u)$ can be added to the end, where t and u are any closed terms of L^{par}.

9.7.8. Show that the following is a derived resolution rule: A resolution expansion containing $[A_1, \ldots, A_n, t \approx u]$ can have added the disjunction $[A_1, \ldots, A_n, u \approx t]$.

9.8 Alternate Tableau and Resolution Systems

The Tableau and Resolution Rules for equality are rather delicate and become especially so when free-variable versions are considered. To get a feeling for their sensitivity to modification before free variables complicate matters, in this section we consider alternate versions of the systems presented in Section 9.7. We will see how changes can affect one another and how apparently small changes can greatly increase the effort of proving completeness. We begin with tableaux and consider resolution afterward.

In Section 9.7 we chose to treat reflexivity via a branch extension rule: $t \approx t$ could be added to branch ends. We remarked that a branch closure version was also reasonable: A branch containing $\neg(t \approx t)$ could be closed. Suppose we try adding this in place of the original reflexivity rule.

Alternate Tableau Reflexivity Rule A branch can be closed if it contains $\neg(t \approx t)$, for any closed term t.

If you go through the tableau examples from Section 9.7, you will find that they are all easily converted into tableaux using the Alternate Tableau Reflexivity Rule. Still, the Alternate Rule is not as powerful as the one it replaces. Using either version, the sentence expressing the symmetric property of equality is provable: $(\forall x)(\forall y)(x \approx y \supset y \approx x)$. But in Exercise 9.7.7, we asked you to show that a *rule* incorporating the symmetry principle was derivable for the tableau system. This rule is no longer derivable if we modify the reflexive rule as proposed. This has the consequence of making completeness proofs break down.

We can compensate for the problem with symmetry by building it into the Replacement Rule. As formulated in Section 9.7, it only allowed left-right replacements. We could allow right-left replacements as well. This is a simple solution, although it can increase the difficulty of finding a proof.

The tableau and resolution systems we gave earlier remain complete even if the Replacement Rules are restricted to allow replacements only in *atomic* sentences. You were asked to prove this in Exercise 9.7.6. If we make the modifications we have discussed, such a severe restriction no longer gives us a complete proof procedure. As it happens, though, a slightly weaker restriction can still be imposed.

Definition 9.8.1 A formula is *simple* if it is atomic or if it is the negation of an atomic formula whose relation symbol is $\approx$.

Alternate Tableau Replacement Rules If $t \approx u$ or $u \approx t$, and also $\Phi(t)$ occur on a branch of a tableau for S, where $\Phi(t)$ is simple, then

$\Phi(u)$ can be added to the end, producing another tableau for S. Here t and u are any terms of $L^{\mathbf{par}}$. Schematically:

$$\frac{\begin{array}{c} t \approx u \\ \Phi(t) \end{array}}{\Phi(u)} \qquad \frac{\begin{array}{c} u \approx t \\ \Phi(t) \end{array}}{\Phi(u)}$$

We have now proposed many changes in the tableau equality rules. Each of the changes is comparatively small, but they are interrelated. Rules for equality are not robust, and care must be exercised when making modifications.

Soundness of the tableau system with the alternate equality rules is easy to establish, by the usual methods. We skip over this and move directly to completeness, which is quite another matter. In Section 9.4 we gave a version of Hintikka's Lemma with equality that allowed us to prove a Model Existence Theorem. To prove completeness of the tableau system with the alternate equality rules, we need something a bit stronger. We achieve this by weakening the notion of Hintikka set.

Definition 9.8.2 Let L be a first-order language. A set $\mathbf{H}$ of sentences of L is an *alternate first-order Hintikka set with equality,* provided $\mathbf{H}$ is a first-order Hintikka set, and in addition

8. $t \approx u \in \mathbf{H}$, $\Phi(t) \in \mathbf{H}$, Φ simple $\Rightarrow$ $\Phi(u) \in \mathbf{H}$.
9. $u \approx t \in \mathbf{H}$, $\Phi(t) \in \mathbf{H}$, Φ simple $\Rightarrow$ $\Phi(u) \in \mathbf{H}$.
10. $\neg(t \approx t) \notin \mathbf{H}$ for every closed term t of L.

Proposition 9.8.3 **(Hintikka's Lemma)** *Suppose L is a language with a non-empty set of closed terms. If $\mathbf{H}$ is an alternate first-order Hintikka set with equality with respect to L, then $\mathbf{H}$ is satisfiable in a canonical normal model.*

Proof Let $\mathbf{H}$ be an alternate Hintikka set with equality. We show that $\mathbf{H}$ can be extended to a Hintikka set with equality, and then appeal to the earlier version of Hintikka's Lemma 9.4.6.

Suppose the closed term t' results from the closed term t by replacing a subterm u by a subterm u', where either $u \approx u' \in \mathbf{H}$ or $u' \approx u \in \mathbf{H}$. Then we say t' results from t by an $\mathbf{H}$-*rewriting.* We say the closed terms u and v are $\mathbf{H}$-*equivalent* provided, u can be turned into v via a sequence of $\mathbf{H}$-rewritings (possibly 0). By repeated applications of conditions 8 and 9, if u and v are $\mathbf{H}$-equivalent, $\Phi(u)$ is simple, and $\Phi(u) \in \mathbf{H}$, then $\Phi(v) \in \mathbf{H}$. A similar result holds for terms as well as formulas. Suppose

$a(u)$ is a closed term containing occurrences of u as a subterm, and $a(v)$ is like $a(u)$ except that some occurrences of u have been replaced by v. If u and v are **H**-equivalent, so are $a(u)$ and $a(v)$, because we can transform occurrences of u in $a(u)$ into occurrences of v by **H**-rewritings. Finally, it is easy to see that **H**-equivalence is an equivalence relation, that is, transitive, reflexive, and symmetric.

Let $\mathbf{H}^*$ be the result of adding to **H** all atomic sentences of the form $u \approx v$ where u and v are **H**-equivalent. It follows that u and v are **H**-equivalent if and only if $u \approx v \in \mathbf{H}^*$. In one direction, this is by the definition of $\mathbf{H}^*$. For the other direction, suppose $u \approx v \in \mathbf{H}^*$. If $u \approx v$ was added to **H** in forming $\mathbf{H}^*$, then trivially u and v are **H**-equivalent. But if $u \approx v$ was in **H** to begin with, we are allowed to change u to v with a single **H**-rewriting, so u is still **H**-equivalent with v.

We claim $\mathbf{H}^*$ is a Hintikka set with equality according to Definition 9.4.5. Once we verify this claim, the proof is finished.

We begin by checking that $\mathbf{H}^*$ is a first-order Hintikka set according to Definition 5.6.1. Since all the sentences added to **H** are atomic, the various closure conditions such as the α- or the β-condition trivially continue to hold for $\mathbf{H}^*$. The only condition that is problematic is that of consistency. If consistency is violated, it must be because both $u \approx v$ and $\neg(u \approx v)$ belong to $\mathbf{H}^*$, where $\neg(u \approx v)$ belonged to **H**, and u and v are **H**-equivalent. But $\neg(u \approx v)$ is simple, so if u and v were **H**-equivalent and $\neg(u \approx v)$ were in **H**, $\neg(v \approx v)$ would also be in **H**, contradicting condition 8. Consequently, $\mathbf{H}^*$ meets the consistency condition and is a first-order Hintikka set.

Next we show that $\mathbf{H}^*$ meets the same replacement condition as **H**: If one of $u \approx v$ or $v \approx u$ is in $\mathbf{H}^*$, and if $\Phi(u) \in \mathbf{H}^*$, then $\Phi(v) \in \mathbf{H}^*$, for simple Φ. So, suppose $\Phi(u) \in \mathbf{H}^*$, where $\Phi(u)$ is simple, and $u \approx v \in \mathbf{H}^*$; we must show $\Phi(v) \in \mathbf{H}^*$. We consider two cases.

First, suppose $\Phi(u) \in \mathbf{H}$. Since $u \approx v \in \mathbf{H}^*$, u and v are **H**-equivalent, so if $\Phi(u)$ is in **H**, $\Phi(v)$ is also, and hence $\Phi(v)$ is in $\mathbf{H}^*$.

Next, suppose $\Phi(u)$ is in $\mathbf{H}^*$ but not **H**. Then $\Phi(u)$ must be of the form $a \approx b$ where a and b are **H**-equivalent, and the occurrences of u being replaced are subterms of a, or of b, or both. To symbolize this we write $\Phi(u)$ as $a(u) \approx b(u)$. Then $\Phi(v)$ is $a(v) \approx b(v)$. As we observed earlier, since u and v are **H**-equivalent, so are $a(u)$ and $a(v)$, and so are $b(u)$ and $b(v)$. Also, $a(u)$ and $b(u)$ are **H**-equivalent, since $a(u) \approx b(u)$ is in $\mathbf{H}^*$. Then, since **H**-equivalence is an equivalence relation, $a(v)$ and $b(v)$ are **H**-equivalent, and hence $a(v) \approx b(v) \in \mathbf{H}^*$, that is, $\Phi(v) \in \mathbf{H}^*$.

Now we can easily finish the verification that $\mathbf{H}^*$ is a first-order Hintikka set *with equality*. First, $t \approx t \in \mathbf{H}^*$ for every closed term t, because **H**-

equivalence is reflexive. The remaining conditions require slightly more work.

Suppose $t_1 \approx u_1, \ldots, t_n \approx u_n \in \mathbf{H}^*$ and $R(t_1, \ldots, t_n) \in \mathbf{H}^*$. Then since $R(t_1, \ldots, t_n)$ is atomic, it is simple, and we showed that $\mathbf{H}^*$ meets a replacement condition for simple sentences, so by repeated application of it, we have $R(u_1, \ldots, u_n) \in \mathbf{H}^*$ as well.

Finally, suppose $t_1 \approx u_1, \ldots, t_n \approx u_n \in \mathbf{H}^*$. We know $f(t_1, \ldots, t_n) \approx f(t_1, \ldots, t_n) \in \mathbf{H}^*$, since $\mathbf{H}$-equivalence is reflexive. Then using the replacement condition again, $f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n) \in \mathbf{H}^*$.

We have verified that $\mathbf{H}^*$ is a first-order Hintikka set with equality, extending $\mathbf{H}$. Now by Proposition 9.4.6, we are done. □

Now that we have this version of Hintikka's Lemma, a similar modified version of the Model Existence Theorem is straightforward. We leave the proof to you.

Definition 9.8.4 Let L be a first-order language, and let $\mathcal{C}$ be a collection of sets of sentences of $L^{\mathbf{par}}$. We call $\mathcal{C}$ an *alternate first-order consistency property with equality* (with respect to L) if it is a first-order consistency property (Definitions 3.6.1 and 5.8.1) and, in addition, for each $S \in \mathcal{C}$, and for all closed terms t and u of $L^{\mathbf{par}}$,

8. $t \approx u \in S$ and $\Phi(t) \in S$, where $\Phi(t)$ is simple $\Rightarrow S \cup \{\Phi(u)\} \in \mathcal{C}$.
9. $u \approx t \in S$ and $\Phi(t) \in S$, where $\Phi(t)$ is simple $\Rightarrow S \cup \{\Phi(u)\} \in \mathcal{C}$.
10. $\neg(t \approx t) \notin S$.

Theorem 9.8.5 **(Alternate Model Existence Theorem)**
If $\mathcal{C}$ is an alternate first-order consistency property with equality with respect to L and $S \in \mathcal{C}$, then S is satisfiable in a normal model; in fact S is satisfiable in a normal model that is canonical with respect to $L^{\mathbf{par}}$.

Theorem 9.8.6 **(Completeness)** *If the sentence X of L is true in all normal models, X has a tableau proof using the alternate rules for equality.*

Similar changes can be made to the resolution rules, of course.

Alternate Resolution Reflexivity Rule The sentence $\neg(t \approx t)$ can be dropped from a disjunction of a resolution expansion for S to produce another resolution expansion for S. Schematically:

$$\frac{[A_1, \ldots, A_n, \neg(t \approx t)]}{[A_1, \ldots, A_n]}$$

Alternate Resolution Replacement Rule Suppose we have a resolution expansion for S containing the generalized disjunctions D_1 and D_2, where D_1 contains $t \approx u$ or $u \approx t$, and D_2 contains $\Phi(t)$ where $\Phi(t)$ is simple. Then we produce a new resolution expansion for S if we add a new generalized disjunction D to the resolution expansion, where D consists of the members of D_1 except for $t \approx u$ or $u \approx t$, the members of D_2 except for $\Phi(t)$, and also $\Phi(u)$. Schematically:

$$\frac{\begin{array}{c}[A_1, \ldots, A_n, t \approx u] \\ [B_1, \ldots, B_k, \Phi(t)]\end{array}}{[A_1, \ldots, A_n, B_1, \ldots, B_k, \Phi(u)]} \qquad \frac{\begin{array}{c}[A_1, \ldots, A_n, u \approx t] \\ [B_1, \ldots, B_k, \Phi(t)]\end{array}}{[A_1, \ldots, A_n, B_1, \ldots, B_k, \Phi(u)]}$$

Soundness and completeness of resolution, using these rules for equality, are left to you.

Exercises

9.8.1. Use the tableau system of this section to prove the sentences of Exercise 9.7.1.

9.8.2. Prove Theorem 9.8.5, then use it to prove Theorem 9.8.6.

9.8.3. Prove soundness and completeness for the resolution system using the alternate rules for equality.

9.9 A Free-Variable Tableau System With Equality

The tableau systems that we gave earlier, incorporating equality rules, are not well suited to implementation for reasons that are familiar by now. There are several ways of dealing with this problem. See Reeves [38] for one version, together with a discussion of other possibilities. The approach we take is to adopt devices similar to those of Chapter 7: Use free variables in γ-rule applications, introduce Skolem function symbols in δ-rule applications, and allow free substitutions of terms for variables. Then we can add rules for equality to this mechanism. But as we have seen, the exact form of equality rules is important, and the rules are not independent of each other. This sensitivity becomes more pronounced when free variables are present and unification plays a role. We will build on the system of Section 9.7, rather than on the alternate version of Section 9.8. Thus, we will have only left-right replacement, and reflexivity will be treated as a branch extension rule. Even so, reflexivity holds surprises for us, so we begin with the more straightforward replacement rule.

Free-Variable Tableau Rules for Equality We use the first-order free-variable tableau system as presented in Section 7.4, together with the following Replacement and Reflexivity Rules.

Free-Variable Tableau Replacement Rule If $t \approx u$ and also $\Phi(t)$ occur on a branch of a tableau for S, then $\Phi(u)$ can be added to the end, producing another tableau for S, provided $\Phi(x)$ is atomic and t and u are any terms, not necessarily closed, of L^{par}. Schematically:

$$\frac{\begin{array}{c} t \approx u \\ \Phi(t) \end{array}}{\Phi(u)}$$

For reflexivity, the following should certainly be expected:

Free-Variable Tableau Reflexivity Rule Adding the formula $x \approx x$, where x is a free variable, to the end of a branch of a tableau for S produces another tableau for S. Schematically:

$$\frac{}{x \approx x}$$

As it happens, this is not enough to ensure completeness once restrictions are imposed, as we will do shortly. We also adopt the following:

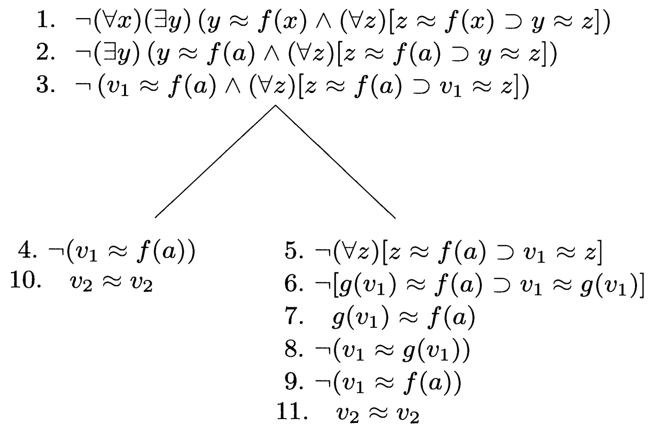
Tableau Function Reflexivity Rule $f(x_1, \ldots, x_n) \approx f(x_1, \ldots, x_n)$ can be added to the end of a branch of a tableau for S to produce another tableau for S, where f is any function symbol of L^{par} and $x_1, \ldots, x_n$ are free variables. Schematically:

$$\frac{}{f(x_1, \ldots, x_n) \approx f(x_1, \ldots, x_n)}$$

Example Figure 9.1 contains a tableau proof of $(\forall x)(\exists y)\{y \approx f(x) \wedge (\forall z)[z \approx f(x) \supset y \approx z]\}$. In it 2 is from 1 by δ, with a as a zero-place Skolem function symbol (a parameter); 3 is from 2 by γ; and 4 and 5 are from 3 by β. Next 6 is from 5 by δ, introducing the one-place Skolem function symbol g; 7 and 8 are from 6 by α. Finally, 9 is from 7 and 8 by a Replacement and 10 and 11 are by Reflexivity. The free substitution $\{v_1/f(a), v_2/f(a)\}$ produces a closed tableau.

Soundness of this system can be established along the same lines as earlier. You should review the proof in Section 7.7. There, we used the notion of *ground satisfiable*. Now we need something a little more restricted.

Definition 9.9.1 A tableau $\mathbf{T}$ allowing free variables is *ground normal satisfiable* if $\mathbf{T}\tau$ is satisfiable *in a normal model* in the sense of Definition 6.3.1, for every grounding substitution τ.



1. $\neg(\forall x)(\exists y)\,(y \approx f(x) \wedge (\forall z)[z \approx f(x) \supset y \approx z])$
2. $\neg(\exists y)\,(y \approx f(a) \wedge (\forall z)[z \approx f(a) \supset y \approx z])$
3. $\neg\,(v_1 \approx f(a) \wedge (\forall z)[z \approx f(a) \supset v_1 \approx z])$
4. $\neg(v_1 \approx f(a))$
10. $v_2 \approx v_2$
5. $\neg(\forall z)[z \approx f(a) \supset v_1 \approx z]$
6. $\neg[g(v_1) \approx f(a) \supset v_1 \approx g(v_1)]$
7. $g(v_1) \approx f(a)$
8. $\neg(v_1 \approx g(v_1))$
9. $\neg(v_1 \approx f(a))$
11. $v_2 \approx v_2$

FIGURE 9.1. Proof of $(\forall x)(\exists y)\{y \approx f(x) \wedge (\forall z)[z \approx f(x) \supset y \approx z]\}$

Lemma 9.9.2 *If any propositional Tableau Expansion Rule, the free-variable γ- or δ-rule, or a free-variable Tableau Equality Rule is applied to a ground normal satisfiable tableau, the result is another ground normal satisfiable tableau.*

Theorem 9.9.3 **(Tableau Soundness)** *If the sentence Φ has a free-variable tableau proof with equality, Φ is true in all normal models.*

The tableau rules we have given are nondeterministic. Many deterministic implementations of them are possible, and no single one is likely to be best under all circumstances. The Free Substitution Rule, as always, is somewhat problematic: What substitutions will be most useful to make? A general principle we followed earlier is to make only those substitutions that enable some other rule to be applied. The following combination of free substitution and equality is particularly useful:

MGU Tableau Replacement Rule Suppose $\mathbf{T}$ is a tableau for S, and θ is a branch of $\mathbf{T}$. Then if $t \approx u$ and $\Phi(t')$ occurs on θ where Φ is atomic, and σ is a most general unifier of t and t', then apply the Free Variable Tableau Replacement Rule to $\Phi(t')\sigma$ and $(t \approx u)\sigma$ on $\theta\sigma$ in $\mathbf{T}\sigma$. The result is another tableau for S.

Example Suppose we have a tableau $\mathbf{T}$ with a branch θ containing the formulas $f(x, a) \approx g(y, b)$ and $k(f(h(z), y)) \approx k(f(h(z), y))$. The term $f(x, a)$ unifies with a subterm $f(h(z), y)$, with $\sigma = \{x/h(z), y/a\}$ as most general unifier. Consequently, we apply σ throughout, to get $\mathbf{T}\sigma$, in which branch $\theta\sigma$ contains $f(h(z),a) \approx g(a,b)$ and $k(f(h(z),a)) \approx k(f(h(z),a))$. Finally, we add $k(g(a,b)) \approx k(f(h(z),a))$ to the end of $\theta\sigma$.

The MGU Tableau Replacement Rule still only allows us to prove sentences that are valid in all normal models, because it is a combination of rules we already know to have this property. We will show we have completeness even if all free substitutions are restricted to those in MGU Tableau Replacement Rules and MGU Atomic Closure substitutions. As a key step in the completeness proof, we need a version of Lemma 7.8.2 that takes equality into account.

Lemma 9.9.4 **(Lifting Lemma)** *Suppose* $\mathbf{T}$ *is a tableau,* τ *is a grounding substitution, and* $\mathbf{T}\tau$ *can be closed using only applications of the Tableau Replacement Rule. Then* $\mathbf{T}$ *can be closed using Free-Variable Reflexivity and Function Reflexivity, MGU Tableau Replacement, and MGU Atomic Closure Rules.*

Proof The argument is by induction on the number of applications of Tableau Replacement needed to close $\mathbf{T}\tau$. If 0 applications are needed, the result reduces to Lemma 7.8.2.

For the induction step assume the result is known whenever n applications of Replacement are needed, and suppose $\mathbf{T}\tau$ can be closed using $n+1$ applications of Replacement. We must show $\mathbf{T}$ can be closed using Free-Variable Reflexivity, Function Reflexivity, MGU Tableau Replacement, and MGU Atomic Closure Rules.

Pick a branch of $\mathbf{T}\tau$ that needs Replacement for closure, say $\theta\tau$ where θ is a branch of $\mathbf{T}$. Say $\theta\tau$ needs $k+1$ applications of Replacement to close. Consider the first application of Replacement made here. To keep the notation manageable, we work with a particular example of Replacement; the ideas are general though. So, let us assume θ is:

$$\begin{array}{l} A_1 \\ \vdots \\ A_n \\ P(f(x)) \\ h(y) \approx b. \end{array}$$

And suppose τ is such that $x\tau = g(h(a))$ and $y\tau = a$. Then $\theta\tau$ is

$$\begin{array}{l} A_1\tau \\ \vdots \\ A_n\tau \\ P(f(g(h(a)))) \\ h(a) \approx b. \end{array}$$

Say Replacement was used to add $P(f(g(b)))$, producing the branch

$$\begin{array}{l} A_1\tau \\ \vdots \\ A_n\tau \\ P(f(g(h(a)))) \\ h(a) \approx b \\ P(f(g(b))). \end{array}$$

Let us designate this branch extending $\theta\tau$ by θ^*. Since one application of Replacement has been made, only k further applications are needed to turn θ^* into a closed branch.

We would like to lift this application of Replacement from $\theta\tau$ to θ itself. The problem is, no application of MGU Tableau Replacement to $P(f(x))$ and $h(y) \approx b$ can introduce an occurrence of the function symbol g. We must drag such an occurrence in by brute force.

In the tableau **T** we can extend branch θ using Function Reflexivity, adding $g(z) \approx g(z)$, where z is a free variable new to the tableau. After this both $h(y) \approx b$ and $g(z) \approx g(z)$ are on the branch, and MGU Tableau Replacement allows us to carry out the substitution $\{z/h(y)\}$ and then add $g(h(y)) \approx g(b)$. (Note that since z was new to the tableau, no formulas are affected by the substitution $\{z/h(y)\}$ except for $g(z) \approx g(z)$.)

At this point we have a tableau that is like **T** except that branch θ has been modified to the following:

$$\begin{array}{l} A_1 \\ \vdots \\ A_n \\ P(f(x)) \\ h(y) \approx b \\ g(h(y)) \approx g(h(y)) \\ g(h(y)) \approx g(b). \end{array}$$

We have both $P(f(x))$ and $g(h(y)) \approx g(b)$ on the branch displayed and once again MGU Tableau Replacement applies. We carry out the substitution $\{x/g(h(y))\}$ throughout $\mathbf{T}$, giving us a branch with both $P(f(g(h(y))))$ and $g(h(y)) \approx g(b)$, to which we add $P(f(g(b)))$.

The situation now is as follows: We have a tableau, call it $\mathbf{T}'$, in which every branch except one arises from a corresponding branch of $\mathbf{T}$ by the substitution $\{x/g(h(y))\}$. And the branch θ of $\mathbf{T}$ has become the following, which we call θ':

$$\begin{array}{l}
A_1\{x/g(h(y))\} \\
\vdots \\
A_n\{x/g(h(y))\} \\
P(f(g(h(y)))) \\
h(y) \approx b \\
g(h(y)) \approx g(h(y)) \\
g(h(y)) \approx g(b) \\
P(f(g(b))).
\end{array}$$

It is easy to check that $\{x/g(h(y))\}\tau = \tau$. Consequently, for every branch B of $\mathbf{T}'$ other than θ', $B\tau$ is a branch of $\mathbf{T}\tau$. And finally, $\theta'\tau$ is the following:

$$\begin{array}{l}
A_1\tau \\
\vdots \\
A_n\tau \\
P(f(g(h(a)))) \\
h(a) \approx b \\
g(h(a)) \approx g(h(a)) \\
g(h(a)) \approx g(b) \\
P(f(g(b))).
\end{array}$$

Now, every formula on $\theta'\tau$ also occurs on θ^*, and θ^* requires only k applications of Replacement to close, so the same is true of $\theta'\tau$. Consequently $\mathbf{T}'\tau$ requires n applications of Replacement, and the induction hypothesis applies: $\mathbf{T}'$ can be closed using Free-Variable Reflexivity, Function Reflexivity, MGU Tableau Replacement, and MGU Atomic Closure Rules. Since we passed from $\mathbf{T}$ to $\mathbf{T}'$ using only these rules, $\mathbf{T}$ itself closes using them. This completes the induction step. □

Now we can prove completeness in a form strong enough to cover the implementation presented in the next section. First, we carry over the notion of a *fair tableau construction rule* from Definition 7.8.5, but we modify it slightly to build in reflexive rule applications. From now on we require that a fair rule also (1) eventually introduce onto each open tableau branch $x \approx x$ for each free variable x; (2) eventually introduce onto each open tableau branch $f(x_1, \ldots, x_n) \approx f(x_1, \ldots, x_n)$ for each function symbol f and all free variables $x_1, \ldots, x_n$.

Theorem 9.9.5 **(Tableau Completeness)** *Let $\mathcal{R}$ be a fair tableau construction rule. If X is a valid sentence of L, X has a tableau proof in which*

1. *All Tableau Expansion Rule, Reflexivity, and Function Reflexivity Rule applications come first and are according to rule $\mathcal{R}$.*
2. *All replacement applications and branch closure substitutions come next and are according to the MGU Tableau Replacement and MGU Atomic Closure Rules.*

We gave a proof of completeness for the free-variable tableau system without equality, Theorem 7.8.6. Essentially, the idea was to show a 'ground' proof existed, then to use the Lifting Lemma 7.8.2 to establish the existence of a proof using free variables and most general unifiers. The same ideas apply here, now that we have a version of the Lifting Lemma taking equality into account. Consequently, we leave the details of the Completeness proof to you in Exercise 9.9.4. Incidentally, notice that no order is specified in the Completeness Theorem for applications of replacement and closure rules. Intermingling of these must be allowed to ensure completeness.

Exercises

9.9.1. Give free-variable tableau proofs of

1. $(\forall x)(\exists y)(\exists z)(z \approx g(y) \wedge y \approx f(x))$.
2. $(\forall x)(\forall y)(\forall z)((y \approx f(x) \wedge z \approx f(x)) \supset y \approx z)$.

9.9.2. Let L be a language with constant symbols 0 and 1, a one-place function symbol s, and a two-place function symbol $+$ (which we write in infix position). Let

id be the formula $(\forall x)(x + 0 \approx x)$

suc be the formula $(\forall x)(\forall y)(x + s(y) \approx s(x + y))$

def be the formula $1 \approx s(0)$

add1 be the formula $(\forall x)(x + 1 \approx s(x))$

Give a free-variable tableau proof of $((\mathit{id} \wedge \mathit{suc}) \wedge \mathit{def}) \supset \mathit{add1}$.

9.9.3. Prove Lemma 9.9.2.

9.9.4. Prove Theorem 9.9.5. Use the proof of Theorem 7.8.6 as a guide.

9.10 A Tableau Implementation With Equality

In Section 7.5 we gave an implementation of a tableau-based first-order theorem prover. Now we modify and extend it to incorporate equality rules. The resulting theorem prover is complete, at least in principle, but it is not very useful. The rules for equality generally allow for so many possibilities that nothing very interesting can be proved before available time and space are exhausted. Heuristics of some kind are essential. In replacing equals by equals, some replacements will be more useful than others, and much work has gone into the development of general principles for deciding which. Such heuristics are beyond our scope here; we are treating the logical foundations only. We leave it to you to try incorporating experimental rules of thumb into the system that follows.

The Prolog clauses that follow generally are *additions* to the first-order program of Section 7.5, occasionally *replacements.* We give them in groups, preceding each with a discussion of its purpose.

In Section 7.5 we gave operator declarations for the propositional connectives. Now we add one more declaration, for eq, allowing it to be used in infix position. The intention, of course, is that it represents the equality relation. (We can not use the usual equality symbol, as it already has a meaning in Prolog.)

```
?-op(120, xfy, eq).
```

We begin with a utility predicate to get it out of the way, a test for being a non-member of a list. This is different from simply negating the `member` relation from Section 7.5, though. `member(X, L)` is true if `X` *unifies* with something in the list `L`. `not_member(X, L)` is true if `X` is not a member of the list `L`, *with no unification allowed.* We will use this later on to ensure we do not add redundant items to a tableau branch.

```
/*    not_member(Item, List) :-
          Item does not occur in List, where an
          occurrence must be via identity, not
          via unification.
*/

not_member(_, [ ]).
not_member(Item, [Head | Tail]) :-
   Item \== Head, not_member(Item, Tail).
```

As we have seen, the basic idea in dealing with the equality relation is to build in rules allowing the replacing of equals by equals. So, we begin with Prolog clauses that do just this. The relation `replace_term(T, Term, U, NewTerm)` is true just in case `NewTerm` is the result of replacing exactly one occurrence of a subterm `T` in `Term` by an occurrence of `U`. It is important to note that `T` itself need not occur as a subterm of `Term`, but rather there should be a subterm that unifies with `T`. This is in accordance with the MGU Tableau Replacement Rule from the previous section. On backtracking, `replace_term` will run through all single replacements. For example, the query

```
replace_term(p(A,B),p(X,p(Y,Z)),p(B,A),W)
```

will, on backtracking, produce the following as values for `W`:

```
p(p(Y,Z),X)
p(p(B,A),p(Y,Z))
p(X,p(Z,Y))
p(X,p(p(B,A),Z))
p(X,p(Y,p(B,A))).
```

Now, the clauses for `replace_term`.

```
/*   replace_term(T, Term, U, NewTerm) :-
         NewTerm is the result of replacing one
         occurrence of T in Term (after a unification)
         with an occurrence of U.
*/

replace_term(T, Term, U, NewTerm) :-
   unify(T, Term),
   unify(U, NewTerm).

replace_term(T, Term, U, NewTerm) :-
   compound(Term),
   Term =.. [F | Args],
   replace_term_list(T, Args, U, NewArgs),
   NewTerm =.. [F | NewArgs].

/*   replace_term_list(T, OldList, U, NewList) :-
         OldList is a list of terms, and NewList
         is the result of applying replace_term
         to one of them, replacing T by U.
```

```
*/

replace_term_list(T, [First | Rest],
   U, [NewFirst | Rest]) :-
      replace_term(T, First, U, NewFirst).

replace_term_list(T, [First | Rest],
   U, [First | NewRest]) :-
      replace_term_list(T, Rest, U, NewRest).
```

Next we have similar predicates that carry out single equality replacements in atomic formulas and in literals.

```
/*   replace_atomic(T, Atomic, U, NewAtomic) :-
         NewAtomic is the result of replacing one
         occurrence of the term T in the atomic formula
         Atomic with an occurrence of U.
*/

replace_atomic(T, Atomic, U, NewAtomic):-
   atomicfmla(Atomic),
   Atomic =.. [F | Args],
   replace_term_list(T, Args, U, NewArgs),
   NewAtomic =.. [F | NewArgs].

/*   replace_literal(T, Literal, U, NewLiteral) :-
         NewLiteral is the result of replacing one
         occurrence of the term T in the literal
         Literal with an occurrence of U.
*/

replace_literal(T, Atomic, U, NewAtomic) :-
   replace_atomic(T, Atomic, U, NewAtomic).
replace_literal(T, neg Atomic, U, neg NewAtomic) :-
   replace_atomic(T, Atomic, U, NewAtomic).
```

The first-order tableau system is complete even if we require all branch closures to be *atomic*. This continues to be the case when we add equality. Also, replacements of equals by equals need be done only in atomic formulas. We are a little more generous and allow replacements in literals, which often makes it easier to find a proof. It will simplify things if we purge branches of everything but literals before we start testing for closure or replacing equals by equals. Thus, there is a certain amount of branch preprocessing to be done.

One serious problem with replacing equals by equals is that the process might never stop. For example, if we have $a = f(a)$ and $P(a, g(a))$ we can replace the first occurrence of a, getting $P(f(a), g(a))$; then we can replace in this, getting $P(f(f(a)), g(a))$; once again, getting $P(f(f(f(a))), g(a))$, and so on forever. We can also run into infinite replacement with the second occurrence of a in $P(a, g(a))$, and endless alternating patterns are also possible. To avoid a similar runaway problem involving the γ-rule, we introduced a *Q-depth* parameter, to limit the number of rule applications. We use a similar idea here. We introduce an *R-Depth* parameter, intended to bound the number of times we are allowed to replace equals by equals. Any given valid formula will have a proof of some R-Depth, but the nonexistence of a proof at a given R-Depth does not ensure nonvalidity.

As we implemented it, Q-Depth was a global bound; it limited the number of γ-rule applications in the entire tableau. This time, for reasons of both variety and convenience, we implement R-Depth locally; each formula has its own counter. The tableau system allowing equality remains complete if all equality replacements are made after all Tableau Expansion Rule applications, and we will make use of this fact. In particular, by the time we start making equality replacements, the list of variables that are free in a formula no longer has any use for us. This list was stored as the *notation* part of a notated formula. We can reuse this notation part to store the R-Depth available for that formula. At the start, the available R-Depth will be the same for each literal. Each time a replacement is carried out on a literal, the R-Depth associated with it will be reduced by 1; this will also be the R-depth of the literal resulting from the replacement.

The predicate `reorganize` modifies a tableau in accordance with the ideas just discussed. Each branch of a tableau is purged of all notated formulas except for notated literals, and for these the notation part is changed to the value of R-depth.

```
/*   reorganize(OldTableau, RDepth, NewTableau) :-
         NewTableau is OldTableau with each branch
         purged of all non-literals, and for each
         notated literal, the notation is changed
         to RDepth.
*/

reorganize([ ], _, [ ]).
reorganize([OldBranch | Rest], RDepth,
     [NewBranch | NewRest]) :-
   modifybranch(OldBranch, RDepth, NewBranch),
   reorganize(Rest, RDepth, NewRest).

/*   modifybranch(OldBranch, RDepth, NewBranch) :-
         NewBranch is OldBranch with each non-literal
         removed, and for each notated literal, the
         notation is changed to RDepth.
*/

modifybranch([ ], _, [ ]).

modifybranch([NotatedLiteral | Rest], RDepth,
     [NewNotatedLiteral | NewRest]) :-
   fmla(NotatedLiteral, Literal),
   literal(Literal),
   fmla(NewNotatedLiteral, Literal),
   notation(NewNotatedLiteral, RDepth),
   modifybranch(Rest, RDepth, NewRest).

modifybranch([NonNotatedLiteral | Rest], RDepth,
     NewRest) :-
   modifybranch(Rest, RDepth, NewRest).
```

The tableau system we are supposed to be implementing treats reflexivity as a branch extension, rather than as a branch closure rule. Still, the system remains sound if we add a rule allowing us to close a branch that says some term does not equal itself. This rule is particularly simple and quite useful. We add such a rule, both for its own sake and as a convenient first approximation to the implementation we really want. (You might guess that we are going to leave some of the work to you.)

In the first-order tableau system, a branch was closed if it contained a falsehood or an obvious contradiction. We now add additional rules for closure. These rules for `closed` are intended to immediately follow the earlier ones for `closed`, from Section 7.5. The first rule embodies the version of Reflexivity just discussed: A branch is closed if it contains a formula asserting some term does not equal itself.

```
closed([Branch | Rest]) :-
   member(Notated, Branch),
   fmla(Notated, neg X eq Y),
   unify(X, Y),
   closed(Rest).
```

The second rule is more complicated and embodies the principle of replacing equals by equals. Roughly, it says a tableau can be considered closed if the result of applying a single equality replacement leads to a closed tableau. Recall that notated formulas now contain R-Depth information. The rule we give applies only to a notated formula that has not exhausted its allowed quota of replacements, and when it is applied, R-Depth is reduced by 1 (also the new notated formula added to the branch has this same reduced R-depth).

```
closed([OldBranch | Rest]) :-
   member(NotatedEquality, OldBranch),
   fmla(NotatedEquality, X eq Y),
   member(NotatedLiteral, OldBranch),
   notation(NotatedLiteral, RDepth),
   RDepth > 0,
   fmla(NotatedLiteral, Literal),
   replace_literal(X, Literal, Y, NewLiteral),
   NewRDepth is RDepth - 1,
   fmla(NewNotatedLiteral, NewLiteral),
   notation(NewNotatedLiteral, NewRDepth),
   fmla(RevisedNotatedLiteral, Literal),
   notation(RevisedNotatedLiteral, NewRDepth),
   remove(NotatedLiteral, OldBranch, TempBranch),
   NewBranch = [NewNotatedLiteral, RevisedNotatedLiteral |
     TempBranch],
   closed([NewBranch | Rest]).
```

Finally, we have the driver, the `test` predicate. It has one more argument than it did in Section 7.5, to take R-Depth into account. The following should replace the clauses for `test`, yes, and no from the earlier program.

```
/*   test(X, Qdepth, RDepth) :- create a complete tableau
         expansion for neg X, allowing Qdepth applications
         of the gamma rule, but no equality rules.  Then
         reorganize the tableau and test for closure,
         allowing the equality replacement rule to be
         used RDepth times on each formula.

      Note that this replaces the earlier test predicate.
*/

test(X, Qdepth, RDepth) :-
   reset,
   notation(NotatedFormula, [ ]),
   fmla(NotatedFormula, neg X),
   expand([[NotatedFormula]], Qdepth, Tree), !,
   reorganize(Tree, RDepth, NewTree), !,
   if_then_else(closed(NewTree),
      yes(Qdepth, RDepth), no(Qdepth, RDepth)),
   !, fail.

yes(Qdepth, RDepth) :-
   write('Theorem at Q-depth '),
   write(Qdepth),
   write(' and R-depth '),
   write(RDepth),
   write('.'),
   nl.

no(Qdepth, RDepth) :-
   write('Not a theorem at Q-depth '),
   write(Qdepth),
   write(' and R-depth '),
   write(RDepth),
   write('.'),
   nl.
```

This completes our implementation. But of course, we have cheated: We used the wrong version of reflexivity. Correcting this takes some care. The Free-Variable Tableau Reflexivity Rule is easy to deal with. It says we can add $x \approx x$ to a branch, and we might want to do this several times, using different free variables. The easiest way is to add a quantified version, $(\forall x)(x \approx x)$ and then let the mechanism for γ-formulas take care of adding instances. We can do a similar thing with the Tableau Function Reflexivity Rules: Add formulas of the form $(\forall x_1) \cdots (\forall x_n)[f(x_1, \ldots, x_n) \approx f(x_1, \ldots, x_n)]$ for every function symbol f. There may be infinitely many function symbols in the language, but fortunately it is easy to see we only need to introduce these formulas for function symbols that have already appeared in the tableau, on the branch in question. Unfortunately, we do not know at the start of a proof what function symbols these will be, because new Skolem function symbols may be introduced from time to time. Basically, there are two ways around this. We might modify the δ-rule so that every time a new Skolem function symbol is introduced on a branch the corresponding Function Reflexivity Rule application is made. But probably a simpler technique is to begin by Skolemizing (see Section 8.3) so that all function symbols are known at the start, and then make all Function Reflexivity Rule applications in one step, at the beginning of the tableau construction. We leave it to you to decide which of these is better and to implement the version you choose. But, remember, the resulting system, while complete, may take years to prove anything interesting. Good heuristics are vital now.

Exercises

9.10.1$^{\mathbf{P}}$. Modify the program of this section to incorporate the Free-Variable Reflexive Rule and some version of the Function Reflexivity Rule.

9.11 Paramodulation

It was necessary to modify the tableau system with equality by introducing free variables to get a version suitable for implementation. We must do a similar thing with the resolution system. The result is a variation of what is known in the literature as *paramodulation* [41]. This is not the only way a treatment of equality can be added to an implementable resolution system; there are also *E-Resolution* [33] and *RUE* (resolution with unification and equality) [15], among others. But paramodulation is the best developed of the approaches, and it is all we consider here.

We use the first-order free-variable resolution system of Section 7.6. Equality rules are added to that. Reflexivity rules are built in directly, as Expansion Rules.

Resolution Reflexivity Rule Adding the clause $[x \approx x]$ to a resolution expansion for S produces another resolution expansion for S, where x is a free variable. Schematically:

$$\frac{}{[x \approx x]}.$$

Resolution Function Reflexivity Rule
$[f(x_1, \ldots, x_n) \approx f(x_1, \ldots, x_n)]$ can be added to a resolution expansion for S to produce another resolution expansion for S, where f is a function symbol and $x_1, \ldots, x_n$ are free variables. Schematically:

$$\frac{}{[f(x_1, \ldots, x_n) \approx f(x_1, \ldots, x_n)]}.$$

Next, a rule allowing the replacement of equals by equals is added. As with tableaux, it is combined with an application of Free Substitution, where the substitution is a most general one that enables a replacement to be carried out.

Paramodulation Rule Suppose $\mathbf{R}$ is a resolution expansion for S containing the generalized disjunctions D_1 and D_2, where D_1 contains $u \approx v$ and D_2 contains the literal $\Phi(t)$. Also suppose σ is a most general unifier for t and u. Let D be the generalized disjunction consisting of $\Phi(v)$, the members of D_1 except for $u \approx v$, and the members of D_2 except for $\Phi(t)$. Then $\mathbf{R}^*\sigma$ is also a resolution expansion for S, where $\mathbf{R}^*$ is $\mathbf{R}$ with D added.

Example Suppose $\mathbf{R}$ is a resolution expansion for S containing only the disjunctions $[P(f(a), x), Q]$ and $[f(x) \approx g(x), R]$. The substitution $\sigma = \{x/a\}$ is a most general unifier for $f(a)$ and $f(x)$. Then $\mathbf{R}^*\sigma$ is a resolution expansion for S, where $\mathbf{R}^*$ is like $\mathbf{R}$ but with $[P(g(x), x), Q, R]$ added. That is, $\mathbf{R}^*\sigma$ contains $[P(f(a), a), Q]$, $[f(a) \approx g(a)]$ and $[P(g(a), a), Q, R]$.

Soundness of this system is straightforward to prove. We leave it as an exercise. Completeness, as usual, is more work. A proof along the lines of that in Section 9.9 is possible. First, prove a generalization of the Lifting Lemma for Resolution 7.9.2, modified like Lemma 9.9.4. Then completeness can be lifted from the completeness of the system of resolution in Section 9.7, much as we did with tableaux. We omit the work; a proof can be found in Loveland [31]. In Section 8.4 we pointed out that most implemented resolution theorem provers begin with a preprocessing step that converts to prenex form, eliminates existential quantifiers by introducing Skolem functions, and puts a matrix into clause form. The completeness proof in Loveland [31] begins from this point.

Exercises

9.11.1. Give a resolution proof, using the system of this section, of the formula in Exercise 9.9.2.

9.11.2. Prove the soundness of the system we gave.

9.11.3$^{\mathrm{P}}$. Write an implementation of the resolution system we gave.

References

[1] Ajtai, M. J. The complexity of the pigeonhole principle. In *29th Annual Symposium on the Foundations of Computer Science (FOCS)* (1988), IEEE Computer Society Press, pp. 346–355.

[2] Andrews, P. B. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, New York, 1986.

[3] Barwise, J., and Etchemendy, J. *The Language of First-order Logic.* CSLI, 1990. (second, revised edition, 1992), distributed by Cambridge University Press.

[4] Beth, E. W. On Padoa's method in the theory of definition. *Indag. Math. 15* (1953), 330–339.

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

[6] Boolos, G. Don't eliminate cut. *Journal of Philosophical Logic 13* (1984), 373–378.

[7] Carnielli, W. A. Systematization of finite many-valued logics through the method of tableaux. *Journal of Symbolic Logic 52* (1987), 473–493.

[8] Church, A. A note on the Entscheidungsproblem. *Journal of Symbolic Logic 1* (1936), 40–41. Reprinted in *The Undecidable*, M. Davis Ed., pp. 110–115, Raven Press, New York 1965.

[9] Chvátal, V., and Szemerédi, E. Many hard examples for resolution. *Journal of the ACM 35* (1988), 759–768.

[10] Clocksin, W. F., and Mellish, C. S. *Programming in Prolog, Third Edition.* Springer-Verlag, Berlin, 1987.

[11] Cook, S. A., and Reckhow, R. A. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic 44* (1979), 36–50.

[12] Craig, W. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic 22* (1957), 250–268.

[13] D'Agostino, M., Gabbay, D., Hähnle, R., and Posegga, J., Eds. *Handbook of Tableau Methods.* Kluwer, Dordrecht, 1996?

[14] Davis, M., and Putnam, H. A computing procedure for quantification theory. *Journal of the ACM 7* (1960), 201–215.

[15] Digricoli, V. J., and Harrison, M. C. Equality-based binary resolution. *Journal of the ACM 33* (1986), 253–289.

[16] Fitting, M. C. *Proof Methods for Modal and Intuitionistic Logics.* D. Reidel Publishing Co., Dordrecht, 1983.

[17] Fitting, M. C. Resolution for intuitionistic logic. In *Proceedings of the ACM SIGART International Symposium on Methodologies for Intelligent Systems 1987* (Amsterdam, 1987), Z. W. Ras and M. Zemankova, Eds., North-Holland, pp. 400–407.

[18] Fitting, M. C. First-order modal tableaux. *Journal of Automated Reasoning 4* (1988), 191–213.

[19] Fitting, M. C. Negation as refutation. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science* (1989), R. Parikh, Ed., IEEE, pp. 63–70.

[20] Gabbay, D. M., Hogger, C. J., and Robinson, J. A., Eds. *Handbook of Logic in AI and Logic Programming.* Oxford University Press, Oxford, 1993.

[21] Gallier, J. H. *Logic for Computer Science, Foundations of Automatic Theorem Proving.* Harper Row, New York, 1986.

[22] Gentzen, G. Investigation into logical deduction. In *The Collected Papers of Gerhard Gentzen*, M. E. Szabo, Ed. North-Holland, 1969, pp. 68–131. Originally published as 'Untersuchungen über das logische Schliessen', in *Mathematische Zeitschrift 39* (1935), 176–210 and 405–431.

[23] Hähnle, R., and Schmitt, P. H. The liberalized δ-rule in free variable semantic tableaux. *Journal of Automated Reasoning 13*, 2 (1994), 211–221.

[24] Haken, A. The intractability of resolution. *Theoretical Computer Science 39* (1985), 297–308.

[25] Herbrand, J. Investigations in proof theory. 1930. English translation in [26] and in [54].

[26] Herbrand, J. *Logical Writings.* Harvard University Press, Cambridge, MA, 1971. Translation of *Écrits logiques*, Jean Van Heijenoort, Ed., Presses Universitaires de France, Paris.

[27] Hintikka, K. J. J. Form and content in quantification theory. *Acta Philosophica Fennica 8* (1955), 7–55.

[28] Hopcroft, J. E., and Ullman, J. D. *Introduction to Automata Theory.* Addison-Wesley, Reading, MA, 1979.

[29] Lassez, J.-L., Maher, M. J., and Marriott, K. Unification revisited. In *Foundations of Deductive Databases and Logic Programming* (Los Altos, CA, 1988), J. Minker, Ed., Morgan Kauffman, pp. 587–625.

[30] Lis, Z. Wynikanie semantyczne a wynikanie formalne (logical consequence, semantic and formal). *Studia Logica 10* (1960), 39–60. Polish, with Russian and English summaries.

[31] Loveland, D. W. *Automated Theorem Proving: A Logical Bases.* North-Holland, Amsterdam, 1978.

[32] Lyndon, R. C. An interpolation theorem in the predicate calculus. *Pacific Journal of Mathematics 9* (1959), 155–164.

[33] MORRIS, J. R. E-resolution: An extension of resolution to include the equality relation. In *Proceedings of the International Joint Conference on Artificial Intelligence (Washington, D.C.)* (1969), D. E. Walker and J. M. Norton, Eds., pp. 287–294.

[34] OPPACHER, F., AND SUEN, E. HARP: A tableau-based theorem prover. *Journal of Automated Reasoning 4* (1988), 69–100.

[35] POPPLESTONE, R. J. Beth-tree methods in automatic theorem-proving. In *Machine Intelligence* (1967), N. L. Collins and D. Michie, Eds., American Elsevier, New York, pp. 31–46.

[36] PRAWITZ, D. *Natural Deduction, A Proof-Theoretical Study.* Acta Universitatis Stockholmiensis, Stockholm Studies in Philosophy 3. Almqvist & Wiksell, Stockholm, 1965.

[37] RECKHOW, R. *On the lengths of proofs in the propositional calculus.* PhD thesis, Univ. of Toronto, Dept. of Computer Science, Toronto, Ontario, Canada, 1975.

[38] REEVES, S. V. Adding equality to semantic tableaux. *Journal of Automated Reasoning 3* (1987), 225–246.

[39] ROBINSON, A. Non-standard analysis. *Proc. Royal Acad Amsterdam Ser. A 64* (1961), 432–440.

[40] ROBINSON, A. *Non-Standard Analysis.* North-Holland, Amsterdam, 1974. Revised edition.

[41] ROBINSON, G. A., AND WOS, L. A. Paramodulation and theorem proving in first-order theories with equality. In *Machine Intelligence* (1969), B. Meltzer and D. Michie, Eds., vol. 4, Edinburgh University Press, pp. 135–150.

[42] ROBINSON, J. A. A machine-oriented logic based on the resolution principle. *Journal of the ACM 12* (1965), 23–41.

[43] ROBINSON, J. A. The generalized resolution principle. In *Machine Intelligence* (1968), Dale and D. Michie, Eds., vol. 3, Oliver and Boyd, pp. 77–93.

[44] ROBINSON, J. A. *Logic: Form and Function.* Elsevier, North-Holland, Amsterdam, 1979.

[45] SIEG, W., AND KAUFFMANN, B. Unification for quantified formulae. Tech. rep., Carnegie Mellon, 1993. Technical Report PHIL-44, Philosophy, Methodology, and Logic.

[46] SKOLEM, T. Über die Nicht-charakterisierbarkeit der Zahlenreihe mittels endlich oder abzählbar unendlich vieler Aussagen mit ausschliesslich Zahlenvariablen. *Fundamenta Mathematica 23* (1934), 150–161.

[47] SKOLEM, T. Peano's axioms and models of arithmetic. In *Mathematical Interpretations of Formal Systems.* North-Holland, Amsterdam, 1955, pp. 1–14.

[48] SMULLYAN, R. M. *First-Order Logic.* Springer-Verlag, Berlin, 1968. Revised Edition, Dover Press, New York, 1994.

[49] SMULLYAN, R. M. Trees and ball games. *Annals of the New York Academy of Sciences 321* (1979), 86–90.

[50] STATMAN, R. Bounds for proof-search and speed-up in the predicate calculus. *Annals of Mathematical Logic 15*, 3 (1978), 225–287.

[51] STERLING, L., AND SHAPIRO, E. *The Art of Prolog.* The MIT Press, Boston, 1986.

[52] Tarski, A. What are logical notions? *History and Philosophy of Logic 7* (1986), 143–154. J. Corcoran, Ed.

[53] Urquhart, A. Hard examples for resolution. *Journal of the ACM 34* (1987), 209–219.

[54] van Heijenoort, J. *From Frege to Gödel.* Harvard University Press, Cambridge, MA, 1967.

[55] Wang, H. Toward mechanical mathematics. *IBM Journal for Research and Development 4* (1960). Reprinted in *A Survey of Mathematical Logic.* Hao Wang, North-Holland, 1963, pp. 224–268.

[56] Whitehead, A. N., and Russell, B. *Principia Mathematica*, 2nd ed. Cambridge University Press, Cambridge, England, 1927.

[57] Wos, L., Overbeek, R., Lusk, E., and Boyle, J. *Automated Reasoning, Introduction and Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

Index