Logic For Computer Science

Foundations of Automatic Theorem Proving

Copyright 2003, Jean H. Gallier

June 2003

Chapter 1

Introduction

Logic is concerned mainly with two concepts: *truth* and *provability*. These concepts have been investigated extensively for centuries, by philosophers, linguists, and mathematicians. The purpose of this book is by no means to give a general account of such studies. Instead, the purpose of this book is to focus on a mathematically well defined logical system known as *first-order logic* (and, to some extent, *many-sorted logic*), and prove some basic properties of this system. In particular, we will focus on algorithmic methods for proving theorems (often referred to as *automatic theorem proving*).

Every logical system consists of a *language* used to write statements also called *propositions* or *formulae*. Normally, when one writes a formula, one has some intended *interpretation* of this formula in mind. For example, a formula may assert a true property about the natural numbers, or some property that must be true in a data base. This implies that a formula has a well-defined *meaning* or *semantics*. But how do we define this meaning precisely? In logic, we usually define the meaning of a formula as its *truth value*. A formula can be either true (or valid) or false.

Defining rigorously the notion of truth is actually not as obvious as it appears. We shall present a concept of truth due to Tarski. Roughly speaking, a formula is true if it is satisfied in all possible interpretations. So far, we have used the intuitive meaning of such words as *truth*, *interpretation*, etc. One of the objectives of this book is to define these terms rigorously, for the language of first-order logic (and many-sorted first-order logic). The branch of logic in which abstract structures and the properties true in these structures are studied is known as *model theory*.

Once the concept of truth has been defined rigorously, the next question

is to investigate whether it is possible to find methods for deciding in a finite number of steps whether a formula is true (or valid). This is a very difficult task. In fact, by a theorem due to Church, there is no such general method for first-order logic.

However, there is another familiar method for testing whether a formula is true: to give a *proof* of this formula.

Of course, to be of any value, a proof system should be *sound*, which means that every provable formula is true.

We will also define rigorously the notion of proof, and proof system for first-order logic (and many-sorted first-order logic). The branch of logic concerned with the study of proof is known as *proof theory*.

Now, if we have a sound proof system, we know that every provable formula is true. Is the proof system strong enough that it is also possible to prove every true formula (of first-order logic)?

A major theorem of Gödel shows that there are logical proof systems in which every true formula is provable. This is referred to as the *completeness* of the proof system.

To summarize the situation, if one is interested in algorithmic methods for testing whether a formula of first-order logic is valid, there are two logical results of central importance: one positive (Gödel's completeness theorem), the other one negative (Church's undecidability of validity). Roughly speaking, Gödel's completeness theorem asserts that there are logical calculi in which every true formula is provable, and Church's theorem asserts that there is no decision procedure (procedure which always terminates) for deciding whether a formula is true (valid). Hence, any algorithmic procedure for testing whether a formula is true (or equivalently, by Gödel's completeness theorem, provable in a complete system) must run forever when given certain non-true formulae as input.

This book focuses on Gödel's positive result and its applications to automatic theorem proving. We have attempted to present a coherent approach to automatic theorem proving, following a main thread: Gentzen-like sequent calculi. The restriction to the positive result was dictated mostly by the lack of space. Indeed, it should be stressed that Church's negative result is also important, as well as other fundamental negative results due to Gödel. However, the omission of such topics should not be a severe inconvenience to the reader, since there are many texts covering such material (see the notes at the end of Chapter 5).

In spite of the theoretical limitation imposed by Church's result, the goal of automatic theorem proving (for short, atp) is to find *efficient* algorithmic methods for finding proofs of those formulae that are true.

A fairly intuitive method for finding such algorithms is the completeness proof for Gentzen-like sequent calculi. This approach yields a complete procedure (the *search* procedure) for proving valid formulae of first-order logic.

However, the *search* procedure usually requires an enormous amount of space and time and it is not practical. Hence, we will try improve it or find more efficient proof procedures.

For this, we will analyze the structure of proofs carefully. Fundamental results of Gentzen and Herbrand show that if a formula is provable, then it has a proof having a certain form, called a *normal form*.

The existence of such normal forms can be exploited to reduce the size of the search space that needs to be explored in trying to find a proof. Indeed, it is sufficient to look for proofs in normal form.

The existence of normal forms is also fundamental because it reduces the problem of finding a proof of a first-order formula to the problem of finding a proof of a simpler type of formula, called a proposition. Propositions are much simpler than first-order formulae. Indeed, there are algorithms for deciding truth. One of the methods based on this reduction technique is the *resolution method*, which will be investigated in Chapters 4 and 8.

Besides looking for general methods applying to the class of all true (first-order) formulae, it is interesting to consider subclasses for which simpler or more efficient proof procedures exist. Indeed, for certain subclasses there may be decision procedures. This is the case for propositions, and for quantifier-free formulae. Such cases are investigated in Chapters 3 and 10 respectively.

Unfortunately, even in cases in which algorithms exist, another difficulty emerges. A decision procedure may take too much time and space to be practical. For example, even testing whether a proposition is true may be very costly. This will be discussed in Chapter 3.

Automatic theorem proving techniques can be used by computer scientists to axiomatize structures and prove properties of programs working on these structures. Another recent and important role that logic plays in computer science, is its use as a *programming language* and as a *model of computation*. For example, in the programming language PROLOG, programs are specified by sets of assertions. In such a programming language, a computation is in fact a proof, and the output of a program is extracted from the proof. Promoters of such languages claim that since such programs are essentially logical formulae, establishing their correctness is trivial. This is not quite so, because the concept of correctness is relative, and the semantics of a PROLOG program needs to be expressed in a language other than PROLOG. However, using logic as a vehicle for programming is a very interesting idea and should be a selling point for skeptics. This use of logic will be investigated in Chapter 9.