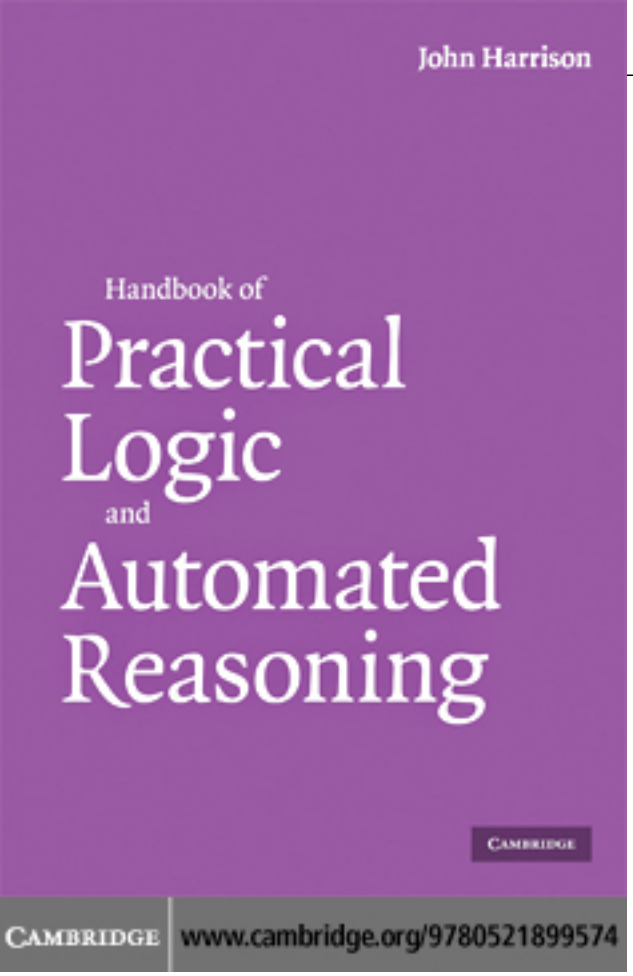
John Harrison
Handbook of
Practical
Logic
and
Automated
Reasoning
CAMBRIDGE
CAMBRIDGE
www.cambridge.org/9780521899574

This page intentionally left blank

HANDBOOK OF PRACTICAL LOGIC AND AUTOMATED REASONING

John Harrison

The sheer complexity of computer systems has meant that automated reasoning, i.e. the use of computers to perform logical inference, has become a vital component of program construction and of programming language design. This book meets the demand for a self-contained and broad-based account of the concepts, the machinery and the use of automated reasoning. The mathematical logic foundations are described in conjunction with their practical application, all with the minimum of prerequisites.

The approach is constructive, concrete and algorithmic: a key feature is that methods are described with reference to actual implementations (for which code is supplied) that readers can use, modify and experiment with.

This book is ideally suited for those seeking a one-stop source for the general area of automated reasoning. It can be used as a reference, or as a place to learn the fundamentals, either in conjunction with advanced courses or for self study.

John Harrison is a Principal Engineer at Intel Corporation in Portland, Oregon. He specialises in formal verification, automated theorem proving, floating-point arithmetic and mathematical algorithms.

HANDBOOK OF PRACTICAL LOGIC AND AUTOMATED REASONING

JOHN HARRISON

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK
Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521899574

First published in print format 2009

ISBN-13 978-0-511-50865-3 eBook (NetLibrary)

ISBN-13 978-0-521-89957-4 hardback

To Porosusha

When a man *Reasoneth*, hee does nothing else but conceive a summe totall, from *Addition* of parcels.

For as Arithmeticians teach to adde and substract in *numbers*; so the Geometricians teach the same in *lines*, *figures* (solid and superficiall,) *angles*, *proportions*, *times*, degrees of *swiftnesse*, *force*, *power*, and the like; The Logicians teach the same in *Consequences of words*; adding together *two Names*, to make an *Affirmation*; and *two Affirmations*, to make a *Syllogisme*; and many *Syllogismes* to make a *Demonstration*; and from the *summe*, or *Conclusion* of a *Syllogisme*, they substract one *Proposition*, to finde the other.

For REASON, in this sense, is nothing but *Reckoning* (that is, Adding and Substracting) of the Consequences of generall names agreed upon, for the *marking* and *signifying* of our thoughts.

And as in Arithmetique, unpractised men must, and Professors themselves may often erre, and cast up false; so also in any other subject of Reasoning, the ablest, most attentive, and most practised men, may deceive themselves and inferre false Conclusions; Not but that Reason it selfe is always Right Reason, as well as Arithmetique is a certain and infallible Art: But no one mans Reason, nor the Reason of any one number of men, makes the certaintie; no more than an account is therefore well cast up, because a great many men have unanimously approved it.

Thomas Hobbes (1588–1697), '*Leviathan, or The Matter, Forme, & Power of a Common-Wealth Ecclesiasticall and Civill*'.
Printed for ANDREW CROOKE, at the Green Dragon in St. *Pauls* Church-yard, 1651.

Contents

Preface

This book is about computer programs that can perform *automated reasoning*. I interpret 'reasoning' quite narrowly: the emphasis is on formal deductive inference rather than, for example, poker playing or medical diagnosis. On the other hand I interpret 'automated' broadly, to include interactive arrangements where a human being and machine reason together, and I'm always conscious of the applications of deductive reasoning to real-world problems. Indeed, as well as being inherently fascinating, the subject is deriving increasing importance from its industrial applications.

This book is intended as a first introduction to the field, and also to logical reasoning itself. No previous knowledge of mathematical logic is assumed, although readers will inevitably find some prior experience of mathematics and of computer programming (especially in a functional language like OCaml, F#, Standard ML, Haskell or LISP) invaluable. In contrast to the many specialist texts on the subject, this book aims at a broad and balanced general introduction, and has two special characteristics.

- Pure logic and automated theorem proving are explained in a closely intertwined manner. Results in logic are developed with an eye to their role in automated theorem proving, and wherever possible are developed in an explicitly computational way.
- Automated theorem proving methods are explained with reference to actual concrete implementations, which readers can experiment with if they have convenient access to a computer. All code is written in the high-level functional language OCaml.

Although this organization is open to question, I adopted it after careful consideration, and extensive experimentation with alternatives. A more detailed self-justification follows, but most readers will want to skip straight to the main content, starting with 'How to read this book' on page xvi.

Ideological orientation

This section explains in more detail the philosophy behind the present text, and attempts to justify it. I also describe the focus of this book and major topics that I do not include. To fully appreciate some points made in the discussion, knowledge of the subject matter is needed. Readers may prefer to skip or skim this material.

My primary aim has been to present a broad and balanced discussion of many of the principal results in automated theorem proving. Moreover, readers mainly interested in pure mathematical logic should find that this book covers most of the traditional results found in mainstream elementary texts on mathematical logic: compactness, Löwenheim–Skolem, completeness of proof systems, interpolation, Gödel's theorems etc. But I consistently strive, even when it is not directly necessary as part of the code of an automated prover, to present results in a concrete, explicit and algorithmic fashion, usually involving real code that can actually be experimented with and used, at least in principle. For example:

- the proof of the interpolation theorem in Section 5.13 contains an algorithm for constructing interpolants, utilizing earlier theorem proving code;
- decidability based on the finite model property is demonstrated in Section 5.5 by explicitly interleaving proving and refuting code rather than a general appeal to Theorem 7.13.

I hope that many readers will share my liking for this concrete hands-on style. Formal logic usually involves a considerable degree of care over tedious syntactic details. This can be quite painful for the beginner, so teachers and authors often have to make the unpalatable choice between (i) spelling everything out in excruciating detail and (ii) waving their hands profusely to cover over sloppy explanations. While teachers rightly tend to recoil from (i), my experience of teaching has shown me that many students nevertheless resent the feeling of never being told the whole story. By implementing things on a computer, I think we get the best of both worlds: the details are there in precise formal detail, but we can mostly let the computer worry about their unpleasant consequences.

It is true that mathematics in the last 150 years has become more abstractly set-theoretic and less constructive. This is particularly so in contemporary model theory, where traditional topics that lie at the historical root of the subject are being de-emphasized. But I'm not alone in swimming against this tide, for the rise of the computer is helping to restore the place of explicit algorithmic methods in several areas of mathematics. This is

particularly notable in algebraic geometry and related areas (Cox, Little and O'Shea 1992; Schenk 2003) where computer algebra and specifically Gröbner bases (see Section 5.11) have made considerable impact. But similar ideas are being explored in other areas, even in category theory (Rydeheard and Burstall 1988), often seen as the quintessence of abstract nonconstructive mathematics. I can do no better than quote Knuth (1974) on the merits of a concretely algorithmic point of view in mathematics generally:

> For three years I taught a sophomore course in abstract algebra for mathematics majors at Caltech, and the most difficult topic was always the study of "Jordan canonical forms" for matrices. The third year I tried a new approach, by looking at the subject algorithmically, and suddenly it became quite clear. The same thing happened with the discussion of finite groups defined by generators and relations, and in another course with the reduction theory of binary quadratic forms. By presenting the subject in terms of algorithms, the purpose and meaning of the mathematical theorems became transparent.
>
> Later, while writing a book on computer arithmetic [Knuth (1969)], I found that virtually every theorem in elementary number theory arises in a natural, motivated way in connection with the problem of making computers do high-speed numerical calculations. Therefore I believe that the traditional courses in number theory might well be changed to adopt this point of view, adding a practical motivation to the already beautiful theory.

In the case of logic, this approach seems especially natural. From the very earliest days, the development of logic was motivated by the desire to reduce reasoning to calculation: the word *logos*, the root of 'logic', can mean not just logical thought but also computation or 'reckoning'. More recently, it was decidability questions in logic that led Turing and others to define precisely the notion of a 'computable function' and set up the abstract models that delimit the range of algorithmic methods. This relationship between logic and computation, which dates from before the Middle Ages, has continued to the present day. For example, problems in the design and verification of computer systems are stimulating more research in logic, while logical principles are playing an increasingly important role in the design of programming languages. Thus, logical reasoning can be seen not only as one of the many beneficiaries of the modern computer age, but as its most important intellectual wellspring.

Another feature of the present text that some readers may find surprising is its systematically model-theoretic emphasis; by contrast many other texts such as Goubault-Larrecq and Mackie (1997) place proof theory at the centre. I introduce traditional proof systems late (Chapter 6), and I hardly mention, and never exploit, structural properties of natural deduction or sequent calculus proofs. While these topics are fascinating, I believe that all the traditional computer-based proof methods for classical logic can be presented

perfectly well without them. Indeed the special refutation-complete calculi for automated theorem proving (binary resolution, hyperresolution, etc.) also provide strong results on canonical forms for proofs. In some situations these are even more convenient for theoretical results than results from Gentzen-style proof theory (Matiyasevich 1975), as with our proof of the Nullstellensatz in Section 5.10 à la Lifschitz (1980). In any case, the details of particular proof systems can be much less significant for automated reasoning than the way in which the corresponding search space is examined. Note, for example, how different tableaux and the inverse method are, even though they can both be understood as search for cut-free sequent proofs.

I wanted to give full, carefully explained code for all the methods described. (In my experience it's easy to underestimate the difficulty in passing from a straightforward-looking algorithm to a concrete implementation.) In order to present real executable code that's almost as readable as the kind of pseudocode often used to describe algorithms, it seemed necessary to use a very high-level language where concrete issues of data representation and memory allocation can be ignored. I selected the functional programming language Objective CAML (OCaml) for this purpose. OCaml is a descendant of Edinburgh ML, a programming language specifically designed for writing theorem provers, and several major systems are written in it.

A drawback of using OCaml (rather than say, C or Java) is that it will be unfamiliar to many readers. However, I only use a simple subset, which is briefly explained in Appendix 2; the code is functional in style with no assignments or sequencing (except for producing diagnostic output). In a few cases (e.g. threading the state through code for binary decision diagrams), imperative code might have been simpler, but it seemed worthwhile to stick to the simplest subset possible. Purely functional programming is particularly convenient for the kind of tinkering that I hope to encourage, since one doesn't have to worry about accidental side-effects of one computation on others.

I will close with a quotation from McCarthy (1963) that nicely encapsulates the philosophy underlying this text, implying as it does the potential new role of logic as a truly *applied* science.

> It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last.

What's not in this book

Although I aim to cover a broad range of topics, selectivity was essential to prevent the book from becoming unmanageably huge. I focus on theories in classical one-sorted first-order logic, since in this coherent setting many of

the central methods of automated reasoning can be displayed. Not without regret, I have therefore excluded from serious discussion major areas such as model checking, inductive theorem proving, many-sorted logic, modal logic, description logics, intuitionistic logic, lambda calculus, higher-order logic and type theory. I believe, however, that this book will prepare the reader quite well to proceed with any of those areas, many of which are best understood precisely in terms of their contrast with classical first-order logic.

Another guiding principle has been to present topics only when I felt competent to do so at a fairly elementary level, without undue technicalities or difficult theory. This has meant the neglect of, for example, ordered paramodulation, cylindrical algebraic decomposition and Gödel's second incompleteness theorem. However, in such cases I have tried to give ample references so that interested readers can go further on their own.

Acknowledgements

This book has taken many years to evolve in haphazard fashion into its current form. During this period, I worked in the University of Cambridge Computer Laboratory, Åbo Akademi University/TUCS and Intel Corporation, as well as spending shorter periods visiting other institutions; I'm grateful above all to Tania and Yestin, for accompanying me on these journeys and tolerating the inordinate time I spent working on this project. It would be impossible to fairly describe here the extent to which my thinking has been shaped by the friends and colleagues that I have encountered over the years. But I owe particular thanks to Mike Gordon, who first gave me the opportunity to get involved in this fascinating field.

I wrote this book partly because I knew of no existing text that presents the range of topics in logic and automated reasoning that I wanted to cover. So the general style and approach is my own, and no existing text can be blamed for its malign influence. But on the purely logical side, I have mostly followed the presentation of basic metatheorems given by Kreisel and Krivine (1971). Their elegant development suits my purposes precisely, being purely model-theoretic and using the workaday tools of automated theorem proving such as Skolemization and the (so-called) Herbrand theorem. For example, the appealingly algorithmic proof of the interpolation theorem given in Section 5.13 is essentially theirs.

Though I have now been a researcher in automated reasoning for almost 20 years, I'm still routinely finding old results in the literature of which I was previously unaware, or learning of them through personal contact with

colleagues. In this connection, I'm grateful to Grigori Mints for pointing me at Lifschitz's proof of the Nullstellensatz (Section 5.10) using resolution proofs, to Loïc Pottier for telling me about Hörmander's algorithm for real quantifier elimination (Section 5.9), and to Lars Hörmander himself for answering my questions on the genesis of this procedure.

I've been very lucky to have numerous friends and colleagues comment on drafts of this book, offer welcome encouragement, take up and modify the associated code, and even teach from it. Their influence has often clarified my thinking and sometimes saved me from serious errors, but needless to say, they are not responsible for any remaining faults in the text. Heartfelt thanks to Rob Arthan, Jeremy Avigad, Clark Barrett, Robert Bauer, Bruno Buchberger, Amine Chaieb, Michael Champigny, Ed Clarke, Byron Cook, Nancy Day, Torkel Franzén (who, alas, did not live to see the finished book), Dan Friedman, Mike Gordon, Alexey Gotsman, Jim Grundy, Tom Hales, Tony Hoare, Peter Homeier, Joe Hurd, Robert Jones, Shuvendu Lahiri, Arthur van Leeuwen, Sean McLaughlin, Wojtek Moczydlowski, Magnus Myreen, Tobias Nipkow, Michael Norrish, John O'Leary, Cagdas Ozgenc, Heath Putnam, Tom Ridge, Konrad Slind, Jørgen Villadsen, Norbert Voelker, Ed Westbrook, Freek Wiedijk, Carl Witty, Burkhart Wolff, and no doubt many other correspondents whose contributions I have thoughtlessly forgotten about over the course of time, for their invaluable help.

Even in the age of the Web, access to good libraries has been vital. I want to thank the staff of the Cambridge University Library, the Computer Laboratory and DPMMS libraries, the mathematics and computer science libraries of Åbo Akademi, and more recently Portland State University Library and Intel Library, who have often helped me track down obscure references. I also want to acknowledge the peerless Powell's Bookstore (`www.powells.com`), which has proved to be a goldmine of classic logic and computer science texts.

Finally, let me thank Frances Nex for her extraordinarily painstaking copyediting, as well as Catherine Appleton, Charlotte Broom, Clare Dennison and David Tranah at Cambridge University Press, who have shepherded this book through to publication despite my delays, and have provided invaluable advice, backed up by the helpful comments of the Press's anonymous reviewers.

How to read this book

The text is designed to be read sequentially from beginning to end. However, after a study of Chapter 1 and a good part of each of Chapters 2 and 3, the reader may be in a position to dip into other parts according to taste.

To support this, I've tried to make some important cross-references explicit, and to avoid over-elaborate or non-standard notation where possible.

Each chapter ends with a number of exercises. These are almost never intended to be routine, and some are very difficult. This reflects my belief that it's more enjoyable and instructive to solve one really challenging problem than to plod through a large number of trivial drill exercises. The reader shouldn't be discouraged if most of them seem too hard. They are all optional, i.e. the text can be understood without doing any of them.

The mathematics used in this book

Mathematics plays a double role in this book: the subject matter itself is treated mathematically, and automated reasoning is also applied to some problems *in* mathematics. But for the most part, the mathematical knowledge needed is not all that advanced: basic algebra, sets and functions, induction, and perhaps most fundamentally, an understanding of the notion of a proof. In a few places, more sophisticated analysis and algebra are used, though I have tried to explain most things as I go along. Appendix 1 is a summary of relevant mathematical background that the reader might refer to as needed, or even skim through at the outset.

The software in this book

An important part of this book is the associated software, which includes simple implementations, in the OCaml programming language, of the various theorem-proving techniques described. Although the book can generally be understood without detailed study of the code, explanations are often organized around it, and code is used as a proxy for what would otherwise be a lengthy and formalistic description of a syntactic process. (For example, the completeness proof for first-order logic in Sections 6.4–6.8 and the proof of Σ_1-completeness of Robinson arithmetic in Section 7.6 are essentially detailed informal arguments that some specific OCaml functions always work.) So without at least a weak impressionistic idea of how the code works, you will probably find some parts of the book heavy going.

Since I expect that many readers will have little or no experience of programming, at least in a functional language like OCaml, I have summarized some of the key ideas in Appendix 2. I don't delude myself into believing that reading this short appendix will turn a novice into an accomplished functional programmer, but I hope it will at least provide some orientation, and it does include references that the reader can pursue if necessary. In fact,

the whole book can be considered an extended case study in functional programming, illustrating many important ideas such as structured data types, recursion, higher-order functions, continuations and abstract data types.

I hope that many readers will not only look at the code, but actually run it, apply it to new problems, and even try modifying or extending it. To do any of these, though, you will need an OCaml interpreter (see Appendix 2 again). The theorem-proving code itself is almost entirely listed in piecemeal fashion within the text. Since the reader will presumably profit little from actually typing it in, all the code can be downloaded from the website for this book (www.cambridge.org/9780521899574) and then just loaded into the OCaml interpreter with a few keystrokes or cut-and-pasted one phrase at a time.

In the future, I hope to make updates to the code and perhaps ports to other languages available at the same URL. More details can be found there about how to run the code, and hence follow along the explanations given in the book while trying out the code in parallel, but I'll just mention a couple of important points here. Probably the easiest way to proceed is to load the entire code associated with this book, e.g. by starting the OCaml interpreter `ocaml` in the directory (folder) containing the code and typing:

```
#use "init.ml";;
```

The default environment is set up to automatically parse anything in French-style ≪quotations≫ as a first-order formula. To use some code in Chapter 1 you will need to change this to parse arithmetic expressions:

```
let default_parser = make_parser parse_expression;;
```

and to use some code in Chapter 2 on propositional logic, you will need to change it to parse propositional formulas:

```
let default_parser = parse_prop_formula;;
```

Otherwise, you can more or less dip into any parts of the code that interest you. In a very few cases, a basic version of a function is defined first as part of the expository flow but later replaced by a more elaborate or efficient version with the same name. The default environment in such cases will always give you the latest one, and if you want to follow the exposition conscientiously you may want to cut-and-paste the earlier version from its source file.

The code is mainly intended to serve a pedagogical purpose, and I have always given clarity and/or brevity priority over efficiency. Still, it sometimes

might be genuinely useful for applications. In any case, before using it, please pay careful attention to the (minimal) legal restrictions listed on the website. Note also that Stålmarck's algorithm (Section 2.10) is patented, so the code in the file `stal.ml` should not be used for commercial applications.

1

Introduction

In this chapter we introduce logical reasoning and the idea of mechanizing it, touching briefly on important historical developments. We lay the groundwork for what follows by discussing some of the most fundamental ideas in logic as well as illustrating how symbolic methods can be implemented on a computer.

1.1 What is logical reasoning?

There are many reasons for believing that something is true. It may seem obvious or at least immediately plausible, we may have been told it by our parents, or it may be strikingly consistent with the outcome of relevant scientific experiments. Though often reliable, such methods of judgement are not infallible, having been used, respectively, to persuade people that the Earth is flat, that Santa Claus exists, and that atoms cannot be subdivided into smaller particles.

What distinguishes *logical* reasoning is that it attempts to avoid any unjustified assumptions and confine itself to inferences that are infallible and beyond reasonable dispute. To avoid making any unwarranted assumptions, logical reasoning cannot rely on any special properties of the objects or concepts being reasoned about. This means that logical reasoning must abstract away from all such special features and be equally valid when applied in other domains. Arguments are accepted as logical based on their conformance to a general *form* rather than because of the specific *content* they treat. For instance, compare this traditional example:

All men are mortal
Socrates is a man
Therefore Socrates is mortal

with the following reasoning drawn from mathematics:

All positive integers are the sum of four integer squares
15 is a positive integer
Therefore 15 is the sum of four integer squares

These two arguments are both correct, and both share a common pattern:

All X are Y
a is X
Therefore a is Y

This pattern of inference is logically valid, since its validity does not depend on the content: the meanings of 'positive integer', 'mortal' etc. are irrelevant. We can substitute anything we like for these X, Y and a, provided we respect grammatical categories, and the statement is still valid. By contrast, consider the following reasoning:

All Athenians are Greek
Socrates is an Athenian
Therefore Socrates is mortal

Even though the conclusion is perfectly true, this is not logically valid, because it does depend on the content of the terms involved. Other arguments with the same superficial form may well be false, e.g.

All Athenians are Greek
Socrates is an Athenian
Therefore Socrates is beardless

The first argument can, however, be turned into a logically valid one by making explicit a hidden assumption 'all Greeks are mortal'. Now the argument is an instance of the general logically valid form:

All G are M
All A are G
s is A
Therefore s is M

At first sight, this forensic analysis of reasoning may not seem very impressive. Logically valid reasoning never tells us anything fundamentally new about the world – as Wittgenstein (1922) says, 'I know nothing about the weather when I know that it is either raining or not raining'. In other words, if we *do* learn something new about the world from a chain of reasoning, it must contain a step that is *not* purely logical. Russell, quoted in Schilpp (1944) says: