# Constructive Logic

Frank Pfenning
Carnegie Mellon University

$\boxed{\text{Draft of January 9, 2003}}$

Material for the course *Constructive Logic* at Carnegie Mellon University, Fall 2000. Material for this course is available at
`http://www.cs.cmu.edu/~fp/courses/logic/`.

Please send comments to `fp@cs.cmu.edu`

This material is in rough draft form and is likely to contain errors. Furthermore, citations are in no way adequate or complete. Please do not cite or distribute this document.

This work was supported in part by the University Education Council at Carnegie Mellon University and by NSF Grant CCR-9619684.

# Contents

# Chapter 1

# Introduction

According to the Encyclopædia Britannica, logic is the study of propositions and their use in argumentation. From the breadth of this definition it is immediately clear that logic constitutes an important area in the disciplines of philosophy and mathematics. Logical tools and methods also play an essential role in the design, specification, and verification of computer hardware and software. It is these applications of logic in computer science which will be the focus of this course. In order to gain a proper understanding of logic and its relevance to computer science, we will need to draw heavily on the much older logical traditions in philosophy and mathematics. We will discuss some of the relevant history of logic and pointers to further reading throughout these notes. In this introduction, we give only a brief overview of the contents and approach of this class.

The course is divided into four parts:

I. Basic Concepts

II. Constructive Reasoning and Programming

III. Automatic Verification

IV. Properties of Logical Systems

In Part I we establish the basic vocabulary and systematically study propositions and proofs, mostly from a philosophical perspective. The treatment will be rather formal in order to permit an easy transition into computational applications. We will also discuss some properties of the logical systems we develop and strategies for proof search. We aim at a systematic account for the usual forms of logical expression, providing us with a flexible and thorough foundation for the remainder of the course. Exercises in this section will test basic understanding of logical connectives and how to reason with them.

In Part II we focus on constructive reasoning. This means we consider only proofs that describe algorithms. This turns out to be quite natural in the framework we have established in Part I. In fact, it may be somewhat surprising that many proofs in mathematics today are *not* constructive in this sense. Concretely, we find that for a certain fragment of logic, constructive proofs correspond to functional programs and vice versa. More generally, we can extract functional programs from constructive proofs of their specifications. We often refer to constructive reasoning as *intuitionistic*, while non-constructive reasoning is *classical*. Exercises in this part explore the connections between proofs and programs, and between theorem proving and programming.

In Part III we study fragments of logic for which the question whether a proposition is true of false can be effectively decided by an algorithm. Such fragments can be used to specify some aspects of the behavior of software or hardware and then automatically verify them. A key technique here is model-checking that exhaustively explores the truth of a proposition over a finite state space. Model-checking and related methods are routinely used in industry, for example, to support hardware design by detecting design flaws at an early stage in the development cycle.

In Part IV we look more deeply at properties of logical system of the kind we developed and applied in Parts I–III. Among the questions we consider is the relation between intuitionistic and classical reasoning, and the soundness and completeness of various algorithms for proof search.

There are several related goals for this course. The first is simply that we would like students to gain a good working knowledge of constructive logic and its relation to computation. This includes the translation of informally specified problems to logical language, the ability to recognize correct proofs and construct them. The skills further include writing and inductively proving the correctness of recursive programs.

The second goals concerns the transfer of this knowledge to other kinds of reasoning. We will try to illuminate logic and the underlying philosophical and mathematical principles from various points of view. This is important, since there are many different kinds of logics for reasoning in different domains or about different phenomena[1], but there are relatively few underlying philosophical and mathematical principles. Our second goal is to teach these principles so that students can apply them in different domains where rigorous reasoning is required.

A third goal relates to specific, important applications of logic in the practice of computer science. Examples are the design of type systems for programming languages, specification languages, or verification tools for finite-state systems. While we do not aim at teaching the use of particular systems or languages, students should have the basic knowledge to quickly learn them, based on the materials presented in this class.

These learning goals present different challenges for students from different disciplines. Lectures, recitations, exercises, and the study of these notes are all necessary components for reaching them. These notes do not cover all aspects of the material discussed in lecture, but provide a point of reference for defini-

[1]for example: classical, intuitionistic, modal, second-order, temporal, belief, non-monotonic, linear, relevance, authentication, ...

tions, theorems, and motivating examples. Recitations are intended to answer students' questions and practice problem solving skills that are critical for the homework assignments. Exercises are a combination of written homework to be handed at lecture and theorem proving or programming problems to be submitted electronically using the software written in support of the course. An introduction to this software is included in these notes, a separate manual is available with the on-line course material.

# Bibliography

[CGP99] E.M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, Cambridge, Massachusetts, 1999.

[CR36] Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

[Dav96] Rowan Davies. A temporal logic approach to binding-time analysis. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[Har95] John Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38:162–170, 1995.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard.

[HR00] Michael R.A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press, 2000.

[ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[Oka99] Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, July 1999.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257, Montreal, Canada, June 1998. ACM Press.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.