Contents

List of Figures

List of Tables

Preface

This book presents a framework for the analysis of syntactic, operational and semantic properties of programming languages. The framework is based on a mathematical system called typed lambda calculus. The main features of lambda calculus are a notation for functions and other computable values, together with an equational logic and rules for evaluating expressions. The book is organized around a sequence of lambda calculi with progressively more complicated type systems. These are used to analyze and discuss relevant programming language concepts. The emphasis is on sequential languages, although many of the techniques and concepts also apply to concurrent programming languages.

The simplest system in the book is an equational system sometimes called universal algebra. This logic without function variables may be used to axiomatize and analyze many of the data types commonly used in programming. The next system is a lambda calculus with function types and, optionally, cartesian products and disjoint unions. When enriched with recursive definitions, this language provides a useful framework for studying operational and semantic properties of functional programs. When combined with algebraic data types, this system is adequate to define many Algol-like languages. In particular, with types for memory locations and stores, we may study traditional axiomatic, operational and denotational semantics of imperative programs. More advanced technical machinery, such as the method of logical relations, category theory, and the semantics of recursively defined types are covered in the middle chapters. The last three chapters of the book study polymorphic types, along with declaration forms for abstract data types and program modules, systems of subtyping, and type inference.

Prerequisites and relation to other topics

The book is written for upper-level undergraduates or beginning graduate students specializing in theoretical computer science, software systems, or mathematics. It is also suitable for advanced study or technical reference. While the only true prerequisite is the proverbial "appropriate level of mathematical maturity," most students will find some prior experience with formal logic, computability or complexity theory, and programming languages helpful. In general, students familiar with these topics at the level of a general introductory course such as [AU92] or above should proceed with confidence and with their sleeves rolled up. To give the prospective reader or instructor more information, the primary connections with related topics are summarized below.

Mathematical logic. The systems of lambda calculus used in this book share many features with traditional mathematical logic. Each has a syntax, a proof system, and a model theory. For this reason, general ideas from logic such as the definition of well-formed formulas, soundness and completeness of proof systems, and interpretation of expressions in mathematical structures are used. These are introduced briefly as needed. First-order logic itself is used only in the sections on proving properties of programs; here an intuitive understanding of the meaning of formulas is assumed.

Computability and complexity theory. The basic distinction between computable and non-computable functions is used in the study of PCF (Chapter 2). The text defines and uses the class of partial recursive functions and refers to Turing machines in the exercises of two sections. A few additional concepts from recursion theory are assumed in constructing semantic models using Gödel numbering of recursive functions (Chapter 4). All of these would be familiar from any course that covers universal Turing machines or undecidable properties of computable functions. A certain amount of basic recursion theory is developed in the text using PCF, including a simple exercise showing that the halting problem for PCF programs is not programmable in PCF.

Programming. Although no specific programming experience is required, students with some exposure to a programming language with higher-order functions, such as Lisp, Scheme or ML, will find it easier to relate this theory to practice. To give a general feel for the expressiveness of typed lambda calculus, Chapter 2 contains a series of programming examples and techniques. This provides a self-contained overview of some relevant programming issues.

Category theory. Category theory appears only in more advanced sections of the book. While all the necessary definitions are presented and illustrated by example, a non-mathematical reader with no prior exposure to category theory may wish to consult additional sources. If a more leisurely or comprehensive introduction is needed, the reader is referred to an elementary introduction tailored to computer scientists, *e.g.*, [BW90, Pie91].

Sample Course Outlines

Three sample course outlines are given in Tables 0.1 through 0.3. The first is an introductory course that has been taught several times as Stanford CS 258. The listed prerequisites for this course, which covers the core topics in Chapters 2–6, are a one-quarter course in automata and computability theory and a one-quarter course that includes mathematical logic but does not cover soundness, completeness or model-theoretic constructions in depth. CS 258 has been completed successfully by undergraduates, M.S. students specializing in systems or theory, and beginning Ph.D. students. While the Stanford course is taught in 10 weeks, it is easy to expand the course to a 15-week semester. Some options for expansion are: (i) cover the topics listed at a more leisurely pace, (ii) include the section on algebraic rewrite systems, (iii) prove soundness, completeness and other properties of typed lambda calculus, or (iv) survey selected topics from Chapters 9–11. It is also possible to drop imperative programs (Chapter 6) in favor of one or more of these options. While the chapter on algebra (Chapter 3) is not strictly required for later topics, universal algebra provides a useful opportunity to introduce or review logical concepts in a relatively simple mathematical setting. This aspect of the chapter may be redundant if students have taken a more rigorous undergraduate course on mathematical logic.

The second course, in Table 0.2, is a more mathematical course on typed lambda calculus and semantic techniques, with more technical detail and less programming motivation. The third course, in Table 0.3, covers type systems, beginning with typed lambda calculus and proceeding with polymorphism, subtyping and type inference. These three overlapping courses cover most of the book.

Acknowledgements and Disclaimers

Many people have read drafts and provided useful comments and suggestions. I would like to thank M. Abadi, S. Abramsky, V. Breazu-Tannen, K. Bruce, L. Cardelli, R. Casley, P.-L. Curien, P. Gardner, D. Gifford, D. Gries, C. Gunter, R. Harper, S. Hayashi, F. Henglein, B. Howard, P. Kanellakis, A. Kfoury, P. Lescanne, H. Mairson, I. Mason, A. Meyer, E. Moggi, N. Marti-Oliet, A.

Pitts, J. Riecke, K. Ross, D. Sanella, P. Scott, D. Tranah, T. Uribe and the students of Stanford CS 258 and CS 358. Special thanks to teaching assistants My Hoang, Brian Howard and Ramesh Viswanathan for their help with homework exercises and sample solutions, a few of which made their way into examples in the text.

Almost all of this book is based on previously published research, some by the author. When specific results are taken from the literature, an effort has been made to cite original sources as well as relevant survey articles and books. However, as with any project of this size, there are likely to be some errors and omissions. In addition, while an effort has been made to circulate and teach any original material or alternate proofs developed for this book, there are undoubtably some remaining errors.

John C. Mitchell
Stanford, CA

Sample Introductory Course

1. *Functional programming and typed lambda calculus* (Chapter 2)
   (a) Boolean, natural number, pairing and function expressions; definition of recursive functions using fixed-point operator (Section 2.2)
   (b) Comparison of axiomatic, operational and denotational semantics (Section 2.3)
   (c) Properties of reduction; deterministic symbolic interpreters (Section 2.4)
   (d) Programming techniques, expressive power, limitations (Section 2.5)
2. *Universal algebra and algebraic data types* (Chapter 3)
   (a) Algebraic terms, equations and algebras (Sections 3.1–3)
   (b) Equational proof system, soundness and completeness (Section 3.4)
   (c) Homomorphisms and initiality (Section 3.5)
   (d) Aspects of algebraic theory of data types (Section 3.6)
3. *Semantics of typed lambda calculus and recursion* (Parts of Chapters 4 and 5)
   (a) Presentation of context-sensitive syntax by typing rules (Sections 4.3.1, 4.3.2, 4.3.5)
   (b) General models, summary of soundness and completeness (Sections 4.5.1–4)
   (c) Domain-theoretic models of typed lambda calculus with fixed-point operators (Sections 5.1 and 5.2; Sections 5.3 and 5.4 time permitting)
4. *Imperative programs* (Chapter 6)
   (a) Syntax of `while` programs; L-values and R-values (Section 6.2)
   (b) Structured operational semantics (Section 6.3)
   (c) Denotational semantics using typed lambda calculus with *location* and *store* types, fixed-point operator (Section 6.4)
   (d) Partial correctness assertions. Soundness, relative completeness and example proofs (Section 6.5)

Table 0.1: Introductory course outline

Course on semantics and typed lambda calculus

1. *Syntax and proof systems of typed lambda calculus*
   (a) Context-sensitive syntax and typing algorithm (Sections 4.1—3)
   (b) Equational proof system and reduction (Section 4.4)
   (c) Recursion using fixed-point operators (Skim 2.2.2–4, cover 2.2.5)
   (d) Recursive types and explicit lifting (Section 2.6)
2. *Model theory of typed lambda calculus*
   (a) General definitions, soundness and completeness (Sections 4.4.1, 4.5.1–6)
   (b) Domains (Sections 5.1, 5.2)
   (c) Modest sets (Sections 5.5, 5.6)
3. *Logical relations*
   (a) Definition and basic lemmas (Sections 8.1, 8.2)
   (b) Proof-theoretic results: completeness, normalization and confluence (Section 8.3)
   (c) Completeness theorems for set-theoretic hierarchy, modest sets and domains (Section 8.4)
4. *Category theory and recursive types*
   (a) Categories, functors and natural transformations (Sections 7.1, 7.2.1–2)
   (b) Cartesian closed categories and typed lambda calculus (Section 7.2.3–6)
   (c) An example of a category that is not well-pointed: Kripke lambda models (Section 7.3)
   (d) Domain models of recursive types (Section 7.4)

Table 0.2: Mathematical course on typed lambda calculus

Course on type theory

1. *Simply-typed lambda calculus*
   (a) Context-sensitive syntax and typing algorithm (Sections 4.1–3)
   (b) Equational proof system and reduction (Section 4.4.1, 4.4.2)
2. *Polymorphism*
   (a) Introduction to polymorphic types (Section 9.1)
   (b) Predicative polymorphism (Section 9.2)
   (c) Properties of impredicative polymorphism (Section 9.3.1–4)
   (d) Data abstraction and existential types (Section 9.4)
   (e) General products, sums and program modules (Section 9.5)
3. *Subtyping*
   (a) Basic syntactic issues, equational reasoning, containment and conversion interpretations of subtyping (Sections 10.1–10.4)
   (b) Records, recursive types, records-as-objects (Section 10.5)
   (c) Polymorphism with subtype constraints (Section 10.6)
4. *Type inference*
   (a) Type inference and erasure functions (Section 11.1)
   (b) Type inference for simply-type lambda calculus using unification (Section 11.2)
   (c) ML-style polymorphic declarations (Section 11.3)

Table 0.3: Course on type theory