

Foundations for Programming Languages

John C. Mitchell
Department of Computer Science
Stanford University
`jcm@cs.stanford.edu`

This book will be published by MIT Press.

Please do not circulate.

© John C. Mitchell 1995.

July 24, 1995

Contents

1	Introduction	16
1.1	Model Programming Languages	16
1.2	Lambda Notation	17
1.3	Equations, Reduction, and Semantics	20
1.3.1	Axiomatic semantics	20
1.3.2	Operational semantics	21
1.3.3	Denotational semantics	22
1.4	Types and Type Systems	22
1.5	Notation and Mathematical Conventions	25
1.6	Set-theoretic Background	26
1.6.1	Fundamentals	26
1.6.2	Relations and Functions	29
1.7	Syntax and Semantics	32
1.7.1	Object language and meta-language	32
1.7.2	Grammars	32
1.7.3	Lexical analysis and parsing	34
1.7.4	Example mathematical interpretation	35
1.8	Induction	37
1.8.1	Induction on the natural numbers	37
1.8.2	Induction on expressions and proofs	41
1.8.3	Well-founded induction	46
2	The Language PCF	50
2.1	Introduction	50
2.2	Syntax of PCF	51
2.2.1	Overview	51
2.2.2	Booleans and natural numbers	52
2.2.3	Pairing and functions	55
2.2.4	Declarations and syntactic sugar	59
2.2.5	Recursion and fixed-point operators	62
2.2.6	PCF syntax summary and collected examples	66
2.3	PCF Programs and Their Semantics	69
2.3.1	Programs and results	69
2.3.2	Axiomatic semantics	70
2.3.3	Denotational semantics	73
2.3.4	Operational semantics	74
2.3.5	Equivalence relations defined by each form of semantics	76

2.4	PCF Reduction and Symbolic Interpreters	78
2.4.1	Nondeterministic reduction	78
2.4.2	Reduction strategies	82
2.4.3	The left-most and lazy reduction strategies	84
2.4.4	Parallel reduction	88
2.4.5	Eager PCF	89
2.5	PCF Programming Examples, Expressive Power and Limitations	93
2.5.1	Records and n -tuples.	93
2.5.2	Searching the natural numbers.	95
2.5.3	Iteration and tail recursion.	97
2.5.4	Total recursive functions	100
2.5.5	Partial recursive functions	103
2.5.6	Non-definability of parallel operations	107
2.6	Variations and Extensions of PCF	114
2.6.1	Summary of extensions	114
2.6.2	Unit and sum types	114
2.6.3	Recursive types	117
2.6.4	Lifted types	123
3	Universal Algebra and Algebraic Data Types	133
3.1	Introduction	133
3.2	Preview of algebraic specification	134
3.3	Algebras, Signatures and Terms	135
3.3.1	Algebras	135
3.3.2	Syntax of algebraic terms	136
3.3.3	Algebras and the interpretation of terms	138
3.3.4	The substitution lemma	141
3.4	Equations, Soundness and Completeness	143
3.4.1	Equations	143
3.4.2	Term algebras and substitution	144
3.4.3	Semantic implication and an equational proof system	146
3.4.4	Forms of completeness	156
3.4.5	Congruence, quotients and deductive completeness	157
3.4.6	Nonempty sorts and the least model property	160
3.5	Homomorphisms and Initiality	161
3.5.1	Homomorphisms and isomorphisms	161
3.5.2	Initial algebras	163
3.6	Algebraic Data Types	169
3.6.1	Specification and data abstraction	169
3.6.2	Initial algebra semantics and datatype induction	171
3.6.3	Examples and error values	176
3.6.4	Alternative approaches to error values	181
3.7	Rewrite Systems	183
3.7.1	Basic definitions	183
3.7.2	Confluence and provable equality	186
3.7.3	Termination	188
3.7.4	Critical pairs	192

3.7.5	Left-linear non-overlapping rewrite systems	198
3.7.6	Local confluence, termination and completion	202
3.7.7	Applications to algebraic datatypes	204
4	Simply-Typed Lambda Calculus	207
4.1	Introduction	207
4.2	Types	208
4.2.1	Syntax	208
4.2.2	Interpretation of types	209
4.3	Terms	210
4.3.1	Context-sensitive syntax	210
4.3.2	Syntax of λ^{\rightarrow} terms	212
4.3.3	Terms with product, sum and related types	217
4.3.4	Formulas-as-types correspondence	219
4.3.5	Typing algorithm	222
4.4	Proof Systems	225
4.4.1	Equations and theories	225
4.4.2	Reduction rules	233
4.4.3	Reduction with additional rules	235
4.4.4	Proof-theoretic methods for consistency and conservativity	237
4.5	Henkin Models, Soundness and Completeness	242
4.5.1	General models and the meanings of terms	242
4.5.2	Applicative structures, extensionality and frames	243
4.5.3	Environment model condition	245
4.5.4	Type and equational soundness	249
4.5.5	Completeness for Henkin models without empty types	252
4.5.6	Completeness with empty types	254
4.5.7	Combinators and the combinatory model condition	256
4.5.8	Combinatory and lambda algebras	258
4.5.9	Henkin models for other types	259
5	Models of Typed Lambda Calculus	262
5.1	Introduction	262
5.2	Domain-Theoretic Models and Fixed Points	263
5.2.1	Recursive definitions and fixed point operators	263
5.2.2	Complete partial orders, lifting and cartesian products	265
5.2.3	Continuous functions	268
5.2.4	Fixed points and the full continuous hierarchy	272
5.2.5	CPO model for PCF	278
5.3	Fixed-point Induction	285
5.4	Computational Adequacy and Full Abstraction	289
5.4.1	Approximation theorem and computational adequacy	289
5.4.2	Full abstraction for PCF with parallel operations	294
5.5	Recursion-theoretic Models	301
5.5.1	Introduction	301
5.5.2	Modest sets	302
5.5.3	Full recursive hierarchy	305

5.6	Partial Equivalence Relations and Recursion	308
5.6.1	Partial equivalence relation interpretation of types	308
5.6.2	Generalization to partial combinatory algebras	311
5.6.3	Lifting, partial functions and recursion	315
5.6.4	Recursion and the intrinsic order	317
5.6.5	Lifting, products and function spaces of effective cpos	321
6	Imperative programs	325
6.1	Introduction	325
6.2	While programs	327
6.2.1	L-values and R-values	327
6.2.2	Syntax of while programs	328
6.3	Operational Semantics	329
6.3.1	Basic symbols in expressions	329
6.3.2	Locations and stores	329
6.3.3	Evaluation of expressions	330
6.3.4	Execution of commands	331
6.4	Denotational Semantics	336
6.4.1	Typed lambda calculus with stores	336
6.4.2	Semantic functions	339
6.4.3	Equivalence of operational and denotational semantics	343
6.5	Before-after Assertions About While Programs	346
6.5.1	First-order and partial correctness assertions	346
6.5.2	Proof rules	348
6.5.3	Soundness	353
6.5.4	Relative completeness	354
6.6	Semantics of Additional Program Constructs	359
6.6.1	Overview	359
6.6.2	Blocks with local variables	359
6.6.3	Procedures	365
6.6.4	Combining blocks and procedure declarations	367
7	Categories and Recursive Types	371
7.1	Introduction	371
7.2	Cartesian Closed Categories	372
7.2.1	Category theory and typed languages	372
7.2.2	Categories, functors and natural transformations	372
7.2.3	Definition of cartesian closed category	381
7.2.4	Soundness and the interpretation of terms	390
7.2.5	Henkin models as ccc's	402
7.2.6	Categorical characterization of meaning function	404
7.3	Kripke Lambda Models and Functor Categories	407
7.3.1	Overview	407
7.3.2	Possible worlds	407
7.3.3	Applicative structures	407
7.3.4	Extensionality, combinators and functor categories	409
7.3.5	Environments and meanings of terms	412

7.3.6	Soundness and completeness	414
7.3.7	Kripke lambda models as cartesian closed categories	416
7.4	Domain models of recursive types	419
7.4.1	A motivating example	419
7.4.2	Diagrams, cones and limits	422
7.4.3	F -algebras	424
7.4.4	ω -Chains and initial F -algebras	426
7.4.5	O-categories and embeddings	430
7.4.6	Colimits and O-colimits	432
7.4.7	Locally continuous functors	436
7.4.8	Examples of the general method	437
8	Logical Relations	442
8.1	Introduction to Logical Relations	442
8.2	Logical Relations Over Applicative Structures	443
8.2.1	Definition of Logical Relation	443
8.2.2	The Basic Lemma	444
8.2.3	Partial functions and theories of models	449
8.2.4	Logical partial equivalence relations	450
8.2.5	Quotients and extensionality	451
8.3	Proof-Theoretic Results	455
8.3.1	Completeness for Henkin models	455
8.3.2	Normalization	457
8.3.3	Confluence and other reduction properties	459
8.3.4	Reduction with fix and additional operations	464
8.4	Partial Surjections and Specific Models	473
8.4.1	Partial surjections and the full classical hierarchy	473
8.4.2	Full recursive hierarchy	474
8.4.3	Full continuous hierarchy	476
8.5	Representation Independence	478
8.5.1	Motivation	478
8.5.2	Example language	478
8.5.3	General representation independence	481
8.6	Generalizations of logical relations	483
8.6.1	Introduction	483
8.6.2	Motivating examples: complete partial orders and Kripke models	484
8.6.3	Sconing and relations	489
8.6.4	Comparison with logical relations	494
8.6.5	General case and applications to specific categories	496
9	Polymorphism and Modularity	499
9.1	Introduction	499
9.1.1	Overview	499
9.1.2	Types as function arguments	500
9.1.3	General products and sums	504
9.1.4	Types as specifications	505
9.2	Predicative Polymorphic Calculus	508

9.2.1	Syntax of types and terms	508
9.2.2	Comparison with other forms of polymorphism	513
9.2.3	Equational proof system and reduction	516
9.2.4	Models of predicative polymorphism	518
9.2.5	ML-style polymorphic declarations	521
9.3	Impredicative Polymorphism	524
9.3.1	Introduction	524
9.3.2	Expressiveness and properties of theories	525
9.3.3	Termination of reduction	538
9.3.4	Summary of semantic models	543
9.3.5	Models based on universal domains	545
9.3.6	Partial equivalence relation models	548
9.4	Data Abstraction and Existential Types	555
9.5	General Products, Sums and Program Modules	560
9.5.1	The ML Module Language	560
9.5.2	Predicative calculus with products and sums	564
9.5.3	Representing Modules With Products and Sums	568
9.5.4	Predicativity and the relationship between universes	570
10	Subtyping and related concepts	573
10.1	Introduction	573
10.2	Simply Typed Lambda Calculus with Subtyping	575
10.3	Records	581
10.3.1	General properties of record subtyping	581
10.3.2	Typed calculus with records and subtyping	582
10.4	Semantic Models of Subtyping	586
10.4.1	Overview	586
10.4.2	Conversion interpretation of subtyping	586
10.4.3	Subset interpretation of types	593
10.4.4	Partial equivalence relations as types	598
10.5	Recursive Types and a Record Model of Objects	603
10.6	Polymorphism with Subtype Constraints	611
11	Type Inference	621
11.1	Introduction to Type Inference	621
11.2	Type Inference for λ^{\rightarrow} with Type Variables	624
11.2.1	The language λ_t^{\rightarrow}	624
11.2.2	Substitution, instances and unification	625
11.2.3	An algorithm for principal Curry typings	630
11.2.4	Implicit typing	635
11.2.5	Equivalence of typing and unification	636
11.3	Type Inference with Polymorphic Declarations	641
11.3.1	ML type inference and polymorphic variables	641
11.3.2	Two sets of implicit typing rules	642
11.3.3	Type inference algorithms	645
11.3.4	Equivalence of ML_1 and ML_2	650
11.3.5	Complexity of ML type inference	653

Bibliography	661
---------------------	------------

Index	676
--------------	------------

List of Figures

1.1	Binary trees.	40
3.1	A locally confluent but non-confluent reduction.	193
3.2	Disjoint reductions	194
3.3	Trivial overlap	195
3.4	Critical pair	196
5.1	Ordering of continuous functions $\mathcal{B}_\perp \rightarrow \mathcal{B}_\perp$	270
7.1	Morphism of cones.	423
7.2	F -algebras and unique morphisms of cocones.	428
7.3	Unique morphism from $\nu^{(i)}$ into limit cone μ over Δ^{prj}	434
11.1	Unification on expression graphs	627

List of Tables

0.1	Introductory course outline	13
0.2	Mathematical course on typed lambda calculus	14
0.3	Course on type theory	15
1.1	Well-founded relations for common forms of induction.	47
2.1	Equational proof system for PCF	71
2.2	Reduction axioms for PCF	75
2.3	Left-most reduction for PCF.	85
2.4	Lazy reduction for PCF.	87
2.5	Eager PCF reduction.	90
2.6	Evaluation contexts for lazy PCF reduction.	108
3.1	Algebraic specification of stacks.	152
3.2	Algebraic specification of multi-sets, <i>nat</i> and <i>bool</i>	153
3.3	Algebraic specification of trees.	154
3.4	A specification for <i>set</i> , <i>nat</i> and <i>bool</i>	170
3.5	A specification for <i>list</i> , <i>atom</i> and <i>bool</i>	177
3.6	Naive treatment of error values for <i>list</i> , <i>atom</i> and <i>bool</i>	178
3.7	A specification for <i>list</i> , <i>atom</i> and <i>bool</i> with error values.	180
4.1	Type-checking algorithm.	223
7.1	Smyth-Plotkin method for finding fixed-points of functors.	438
11.1	Recursive algorithm <i>Unify</i>	628
11.2	Algorithm <i>PT</i> for principal λ_t^{\rightarrow} (Curry) typing.	631
11.3	Algorithm reducing λ_t^{\rightarrow} (Curry) typing to unification.	637
11.4	Algorithm <i>PTL</i> for principal typing with let	648

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 undoubtedly 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 theory1. *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