

## Practical Foundations for Programming Languages



This text develops a comprehensive theory of programming languages based on type systems and structural operational semantics. Language concepts are precisely defined by their static and dynamic semantics, presenting the essential tools both intuitively and rigorously while relying on only elementary mathematics. These tools are used to analyze and prove properties of languages and provide the framework for combining and comparing language features. The broad range of concepts includes fundamental data types such as sums and products, polymorphic and abstract types, dynamic typing, dynamic dispatch, subtyping and refinement types, symbols and dynamic classification, parallelism and cost semantics, and concurrency and distribution. The methods are directly applicable to language implementation, to the development of logics for reasoning about programs, and to the formal verification language properties such as type safety.

This thoroughly revised second edition includes exercises at the end of nearly every chapter and a new chapter on type refinements.

**Robert Harper** is a professor in the Computer Science Department at Carnegie Mellon University. His main research interest is in the application of type theory to the design and implementation of programming languages and to the mechanization of their meta-theory. Harper is a recipient of the Allen Newell Medal for Research Excellence and the Herbert A. Simon Award for Teaching Excellence, and is an Association for Computing Machinery Fellow.

# Practical Foundations for Programming Languages

*Second Edition*

Robert Harper  
Carnegie Mellon University



CAMBRIDGE  
UNIVERSITY PRESS

32 Avenue of the Americas, New York, NY 10013

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning, and research at the highest international levels of excellence.

[www.cambridge.org](http://www.cambridge.org)

Information on this title: [www.cambridge.org/9781107150300](http://www.cambridge.org/9781107150300)

© Robert Harper 2016

This publication is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published 2016

Printed in the United States of America

*A catalog record for this publication is available from the British Library.*

*Library of Congress Cataloging in Publication Data*

Names: Harper, Robert, 1957–

Title: Practical foundations for programming languages / Robert Harper,  
Carnegie Mellon University.

Description: Second edition. | New York NY : Cambridge University Press,  
2016. | Includes bibliographical references and index.

Identifiers: LCCN 2015045380 | ISBN 9781107150300 (alk. paper)

Subjects: LCSH: Programming languages (Electronic computers)

Classification: LCC QA76.7 .H377 2016 | DDC 005.13–dc23

LC record available at <http://lcn.loc.gov/2015045380>

ISBN 978-1-107-15030-0 Hardback

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party Internet Web sites referred to in this publication and does not guarantee that any content on such Web sites is, or will remain, accurate or appropriate.

Contents

<i>Preface to the Second Edition</i>	<i>page xv</i>
<i>Preface to the First Edition</i>	<i>xvii</i>

Part I Judgments and Rules

1 Abstract Syntax	3
1.1 Abstract Syntax Trees	3
1.2 Abstract Binding Trees	6
1.3 Notes	10
2 Inductive Definitions	12
2.1 Judgments	12
2.2 Inference Rules	12
2.3 Derivations	14
2.4 Rule Induction	15
2.5 Iterated and Simultaneous Inductive Definitions	17
2.6 Defining Functions by Rules	18
2.7 Notes	19
3 Hypothetical and General Judgments	21
3.1 Hypothetical Judgments	21
3.2 Hypothetical Inductive Definitions	24
3.3 General Judgments	26
3.4 Generic Inductive Definitions	27
3.5 Notes	28

Part II Statics and Dynamics

4 Statics	33
4.1 Syntax	33
4.2 Type System	34
4.3 Structural Properties	35
4.4 Notes	37

vi	Contents
<hr/>	
5	Dynamics 39
5.1	Transition Systems 39
5.2	Structural Dynamics 40
5.3	Contextual Dynamics 42
5.4	Equational Dynamics 44
5.5	Notes 46
6	Type Safety 48
6.1	Preservation 48
6.2	Progress 49
6.3	Run-Time Errors 50
6.4	Notes 52
7	Evaluation Dynamics 53
7.1	Evaluation Dynamics 53
7.2	Relating Structural and Evaluation Dynamics 54
7.3	Type Safety, Revisited 55
7.4	Cost Dynamics 56
7.5	Notes 57
 <b>Part III Total Functions</b>	
8	Function Definitions and Values 61
8.1	First-Order Functions 61
8.2	Higher-Order Functions 62
8.3	Evaluation Dynamics and Definitional Equality 65
8.4	Dynamic Scope 66
8.5	Notes 67
9	System <b>T</b> of Higher-Order Recursion 69
9.1	Statics 69
9.2	Dynamics 70
9.3	Definability 71
9.4	Undefinability 73
9.5	Notes 75
 <b>Part IV Finite Data Types</b>	
10	Product Types 79
10.1	Nullary and Binary Products 79
10.2	Finite Products 81
10.3	Primitive Mutual Recursion 82
10.4	Notes 83

vii	Contents
<hr/>	
11	Sum Types 85
11.1	Nullary and Binary Sums 85
11.2	Finite Sums 86
11.3	Applications of Sum Types 88
11.4	Notes 91
 <b>Part V Types and Propositions</b>	
12	Constructive Logic 95
12.1	Constructive Semantics 95
12.2	Constructive Logic 96
12.3	Proof Dynamics 100
12.4	Propositions as Types 101
12.5	Notes 101
13	Classical Logic 104
13.1	Classical Logic 105
13.2	Deriving Elimination Forms 109
13.3	Proof Dynamics 110
13.4	Law of the Excluded Middle 111
13.5	The Double-Negation Translation 113
13.6	Notes 114
 <b>Part VI Infinite Data Types</b>	
14	Generic Programming 119
14.1	Introduction 119
14.2	Polynomial Type Operators 119
14.3	Positive Type Operators 122
14.4	Notes 123
15	Inductive and Coinductive Types 125
15.1	Motivating Examples 125
15.2	Statics 128
15.3	Dynamics 130
15.4	Solving Type Equations 131
15.5	Notes 132
 <b>Part VII Variable Types</b>	
16	System <b>F</b> of Polymorphic Types 137
16.1	Polymorphic Abstraction 137
16.2	Polymorphic Definability 140
16.3	Parametricity Overview 142
16.4	Notes 144

viii	Contents
17	Abstract Types 146
17.1	Existential Types 146
17.2	Data Abstraction 149
17.3	Definability of Existential Types 150
17.4	Representation Independence 151
17.5	Notes 153
18	Higher Kinds 154
18.1	Constructors and Kinds 155
18.2	Constructor Equality 156
18.3	Expressions and Types 157
18.4	Notes 158
Part VIII Partiality and Recursive Types	
19	System <b>PCF</b> of Recursive Functions 161
19.1	Statics 162
19.2	Dynamics 163
19.3	Definability 165
19.4	Finite and Infinite Data Structures 167
19.5	Totality and Partiality 167
19.6	Notes 169
20	System <b>FPC</b> of Recursive Types 171
20.1	Solving Type Equations 171
20.2	Inductive and Coinductive Types 172
20.3	Self-Reference 174
20.4	The Origin of State 176
20.5	Notes 177
Part IX Dynamic Types	
21	The Untyped $\lambda$ -Calculus 181
21.1	The $\lambda$ -Calculus 181
21.2	Definability 182
21.3	Scott's Theorem 184
21.4	Untyped Means Uni-Typed 186
21.5	Notes 187
22	Dynamic Typing 189
22.1	Dynamically Typed PCF 189
22.2	Variations and Extensions 192
22.3	Critique of Dynamic Typing 194
22.4	Notes 195

ix	Contents
<hr/>	
23	Hybrid Typing 198
23.1	A Hybrid Language 198
23.2	Dynamic as Static Typing 200
23.3	Optimization of Dynamic Typing 201
23.4	Static versus Dynamic Typing 203
23.5	Notes 204
 <b>Part X Subtyping</b>	
24	Structural Subtyping 207
24.1	Subsumption 207
24.2	Varieties of Subtyping 208
24.3	Variance 211
24.4	Dynamics and Safety 215
24.5	Notes 216
25	Behavioral Typing 219
25.1	Statics 220
25.2	Boolean Blindness 226
25.3	Refinement Safety 228
25.4	Notes 229
 <b>Part XI Dynamic Dispatch</b>	
26	Classes and Methods 235
26.1	The Dispatch Matrix 235
26.2	Class-Based Organization 238
26.3	Method-Based Organization 239
26.4	Self-Reference 240
26.5	Notes 242
27	Inheritance 245
27.1	Class and Method Extension 245
27.2	Class-Based Inheritance 246
27.3	Method-Based Inheritance 248
27.4	Notes 249
 <b>Part XII Control Flow</b>	
28	Control Stacks 253
28.1	Machine Definition 253
28.2	Safety 255
28.3	Correctness of the <b>K</b> Machine 256
28.4	Notes 259



x	Contents
29	Exceptions 260
29.1	Failures 260
29.2	Exceptions 262
29.3	Exception Values 263
29.4	Notes 264
30	Continuations 266
30.1	Overview 266
30.2	Continuation Dynamics 268
30.3	Coroutines from Continuations 269
30.4	Notes 272
<b>Part XIII Symbolic Data</b>	
31	Symbols 277
31.1	Symbol Declaration 277
31.2	Symbol References 280
31.3	Notes 282
32	Fluid Binding 284
32.1	Statics 284
32.2	Dynamics 285
32.3	Type Safety 286
32.4	Some Subtleties 287
32.5	Fluid References 288
32.6	Notes 289
33	Dynamic Classification 291
33.1	Dynamic Classes 291
33.2	Class References 293
33.3	Definability of Dynamic Classes 294
33.4	Applications of Dynamic Classification 295
33.5	Notes 296
<b>Part XIV Mutable State</b>	
34	Modernized Algol 301
34.1	Basic Commands 301
34.2	Some Programming Idioms 306
34.3	Typed Commands and Typed Assignables 307
34.4	Notes 310
35	Assignable References 313
35.1	Capabilities 313
35.2	Scoped Assignables 314

xi	Contents	
35.3	Free Assignables	316
35.4	Safety	318
35.5	Benign Effects	320
35.6	Notes	321
36	Lazy Evaluation	323
36.1	<b>PCF</b> By-Need	323
36.2	Safety of <b>PCF</b> By-Need	326
36.3	<b>FPC</b> By-Need	328
36.4	Suspension Types	329
36.5	Notes	331
	<b>Part XV Parallelism</b>	
37	Nested Parallelism	335
37.1	Binary Fork-Join	335
37.2	Cost Dynamics	338
37.3	Multiple Fork-Join	341
37.4	Bounded Implementations	342
37.5	Scheduling	346
37.6	Notes	348
38	Futures and Speculations	350
38.1	Futures	350
38.2	Speculations	351
38.3	Parallel Dynamics	352
38.4	Pipelining with Futures	354
38.5	Notes	356
	<b>Part XVI Concurrency and Distribution</b>	
39	Process Calculus	359
39.1	Actions and Events	359
39.2	Interaction	361
39.3	Replication	363
39.4	Allocating Channels	364
39.5	Communication	366
39.6	Channel Passing	369
39.7	Universality	371
39.8	Notes	372
40	Concurrent Algol	375
40.1	Concurrent Algol	375
40.2	Broadcast Communication	378
40.3	Selective Communication	380

xii	Contents	
40.4	Free Assignables as Processes	382
40.5	Notes	383
41	Distributed Algol	385
41.1	Statics	385
41.2	Dynamics	388
41.3	Safety	390
41.4	Notes	391
<b>Part XVII   Modularity</b>		
42	Modularity and Linking	395
42.1	Simple Units and Linking	395
42.2	Initialization and Effects	396
42.3	Notes	398
43	Singleton Kinds and Subkinding	399
43.1	Overview	399
43.2	Singletons	400
43.3	Dependent Kinds	402
43.4	Higher Singletons	405
43.5	Notes	407
44	Type Abstractions and Type Classes	409
44.1	Type Abstraction	410
44.2	Type Classes	412
44.3	A Module Language	414
44.4	First- and Second-Class	418
44.5	Notes	419
45	Hierarchy and Parameterization	422
45.1	Hierarchy	422
45.2	Abstraction	425
45.3	Hierarchy and Abstraction	427
45.4	Applicative Functors	429
45.5	Notes	431
<b>Part XVIII   Equational Reasoning</b>		
46	Equality for System <b>T</b>	435
46.1	Observational Equivalence	435
46.2	Logical Equivalence	439
46.3	Logical and Observational Equivalence Coincide	440
46.4	Some Laws of Equality	443
46.5	Notes	444

xiii	Contents
47	Equality for System <b>PCF</b> 445
47.1	Observational Equivalence 445
47.2	Logical Equivalence 446
47.3	Logical and Observational Equivalence Coincide 446
47.4	Compactness 449
47.5	Lazy Natural Numbers 452
47.6	Notes 453
48	Parametricity 454
48.1	Overview 454
48.2	Observational Equivalence 455
48.3	Logical Equivalence 456
48.4	Parametricity Properties 461
48.5	Representation Independence, Revisited 464
48.6	Notes 465
49	Process Equivalence 467
49.1	Process Calculus 467
49.2	Strong Equivalence 469
49.3	Weak Equivalence 472
49.4	Notes 473
<b>Part XIX Appendices</b>	
A	Background on Finite Sets 477
<i>Bibliography</i> 479	
<i>Index</i> 487	

## Preface to the Second Edition

Writing the second edition to a textbook incurs the same risk as building the second version of a software system. It is difficult to make substantive improvements, while avoiding the temptation to overburden and undermine the foundation on which one is building. With the hope of avoiding the second system effect, I have sought to make corrections, revisions, expansions, and deletions that improve the coherence of the development, remove some topics that distract from the main themes, add new topics that were omitted from the first edition, and include exercises for almost every chapter.

The revision removes a number of typographical errors, corrects a few material errors (especially the formulation of the parallel abstract machine and of concurrency in Algol), and improves the writing throughout. Some chapters have been deleted (general pattern matching and polarization, restricted forms of polymorphism), some have been completely rewritten (the chapter on higher kinds), some have been substantially revised (general and parametric inductive definitions, concurrent and distributed Algol), several have been reorganized (to better distinguish partial from total type theories), and a new chapter has been added (on type refinements). Titular attributions on several chapters have been removed, not to diminish credit, but to avoid confusion between the present and the original formulations of several topics. A new system of (pronounceable!) language names has been introduced throughout. The exercises generally seek to expand on the ideas in the main text, and their solutions often involve significant technical ideas that merit study. Routine exercises of the kind one might include in a homework assignment are deliberately few.

My purpose in writing this book is to establish a comprehensive framework for formulating and analyzing a broad range of ideas in programming languages. If language design and programming methodology are to advance from a trade-craft to a rigorous discipline, it is essential that we first get the definitions right. Then, and only then, can there be meaningful analysis and consolidation of ideas. My hope is that I have helped to build such a foundation.

I am grateful to Stephen Brookes, Evan Cavallo, Karl Crary, Jon Sterling, James R. Wilcox and Todd Wilson for their help in critiquing drafts of this edition and for their suggestions for modification and revision. I thank my department head, Frank Pfenning, for his support of my work on the completion of this edition. Thanks also to my editors, Ada Brunstein and Lauren Cowles, for their guidance and assistance. And thanks to Andrew Shulaev for corrections to the draft.

Neither the author nor the publisher make any warranty, express or implied, that the definitions, theorems, and proofs contained in this volume are free of error, or are consistent with any particular standard of merchantability, or that they will meet requirements for any particular application. They should not be relied on for solving a problem whose incorrect

solution could result in injury to a person or loss of property. If you do use this material in such a manner, it is at your own risk. The author and publisher disclaim all liability for direct or consequential damage resulting from its use.

Pittsburgh  
July 2015

## Preface to the First Edition

Types are the central organizing principle of the theory of programming languages. Language features are manifestations of type structure. The syntax of a language is governed by the constructs that define its types, and its semantics is determined by the interactions among those constructs. The soundness of a language design—the absence of ill-defined programs—follows naturally.

The purpose of this book is to explain this remark. A variety of programming language features are analyzed in the unifying framework of type theory. A language feature is defined by its *statics*, the rules governing the use of the feature in a program, and its *dynamics*, the rules defining how programs using this feature are to be executed. The concept of *safety* emerges as the coherence of the statics and the dynamics of a language.

In this way, we establish a foundation for the study of programming languages. But why these particular methods? The main justification is provided by the book itself. The methods we use are both *precise* and *intuitive*, providing a uniform framework for explaining programming language concepts. Importantly, these methods *scale* to a wide range of programming language concepts, supporting rigorous analysis of their properties. Although it would require another book in itself to justify this assertion, these methods are also *practical* in that they are *directly applicable* to implementation and *uniquely effective* as a basis for mechanized reasoning. No other framework offers as much.

Being a consolidation and distillation of decades of research, this book does not provide an exhaustive account of the history of the ideas that inform it. Suffice it to say that much of the development is not original but rather is largely a reformulation of what has gone before. The notes at the end of each chapter signpost the major developments but are not intended as a complete guide to the literature. For further information and alternative perspectives, the reader is referred to such excellent sources as Constable (1986, 1998), Girard (1989), Martin-Löf (1984), Mitchell (1996), Pierce (2002, 2004), and Reynolds (1998).

The book is divided into parts that are, in the main, independent of one another. Parts I and II, however, provide the foundation for the rest of the book and must therefore be considered prior to all other parts. On first reading, it may be best to skim Part I, and begin in earnest with Part II, returning to Part I for clarification of the logical framework in which the rest of the book is cast.

Numerous people have read and commented on earlier editions of this book and have suggested corrections and improvements to it. I am particularly grateful to Umut Acar, Jesper Louis Andersen, Carlo Angiuli, Andrew Appel, Stephanie Balzer, Eric Bergstrom, Guy E. Blelloch, Iliano Cervesato, Lin Chase, Karl Crary, Rowan Davies, Derek Dreyer, Dan Licata, Zhong Shao, Rob Simmons, and Todd Wilson for their extensive efforts in

reading and criticizing the book. I also thank the following people for their suggestions: Joseph Abrahamson, Arbob Ahmad, Zena Ariola, Eric Bergstrom, William Byrd, Alejandro Cabrera, Luis Caires, Luca Cardelli, Manuel Chakravarty, Richard C. Cobbe, James Cooper, Yi Dai, Daniel Dantas, Anupam Datta, Jake Donham, Bill Duff, Matthias Felleisen, Kathleen Fisher, Dan Friedman, Peter Gammie, Maia Ginsburg, Byron Hawkins, Kevin Hely, Kuen-Bang Hou (Favonia), Justin Hsu, Wojciech Jedynek, Cao Jing, Salil Joshi, Gabriele Keller, Scott Kilpatrick, Danielle Kramer, Dan Kreysa, Akiva Leffert, Ruy Ley-Wild, Karen Liu, Dave MacQueen, Chris Martens, Greg Morrisett, Stefan Muller, Tom Murphy, Aleksandar Nanevski, Georg Neis, David Neville, Adrian Trejo Nuñez, Cyrus Omar, Doug Perkins, Frank Pfenning, Jean Pichon, Benjamin Pierce, Andrew M. Pitts, Gordon Plotkin, David Renshaw, John Reynolds, Andreas Rossberg, Carter Schonwald, Dale Schumacher, Dana Scott, Shayak Sen, Pawel Sobocinski, Kristina Sojakova, Daniel Spoonhower, Paulo Tanimoto, Joe Tassarotti, Peter Thiemann, Bernardo Toninho, Michael Tschantz, Kami Vaniea, Carsten Varming, David Walker, Dan Wang, Jack Wileden, Sergei Winitzki, Roger Wolff, Omer Zach, Luke Zarko, and Yu Zhang. I am very grateful to the students of 15-312 and 15-814 at Carnegie Mellon who have provided the impetus for the preparation of this book and who have endured the many revisions to it over the last ten years.

I thank the Max Planck Institute for Software Systems for its hospitality and support. I also thank Espresso a Mano in Pittsburgh, CB2 Cafe in Cambridge, and Thonet Cafe in Saarbrücken for providing a steady supply of coffee and a conducive atmosphere for writing.

This material is, in part, based on work supported by the National Science Foundation under Grant Nos. 0702381 and 0716469. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Robert Harper  
Pittsburgh  
March 2012