Types and Programming Languages

Benjamin C. Pierce

Types and Programming Languages
Benjamin C. Pierce

A type system is a syntactic method for automatically checking the absence of certain erroneous behaviors by classifying program phrases according to the kinds of values they compute. The study of type systems—and of programming languages from a type-theoretic perspective—has important applications in software engineering, language design, high-performance compilers, and security.

This text provides a comprehensive introduction both to type systems in computer science and to the basic theory of programming languages. The approach is pragmatic and operational; each new concept is motivated by programming examples and the more theoretical sections are driven by the needs of implementations. Each chapter is accompanied by numerous exercises and solutions, as well as a running implementation, available via the Web. Dependencies between chapters are explicitly identified, allowing readers to choose a variety of paths through the material.

Benjamin C. Pierce is Associate Professor of Computer and Information Science at the University of Pennsylvania.

"Types are the leaven of computer programming; they make it digestible. This excellent book uses types to navigate the rich variety of programming languages, bringing a new kind of unity to their usage, theory, and implementation. Its author writes with the authority of experience in all three of these aspects."
—Robin Milner, Computer Laboratory, University of Cambridge

"Written by an outstanding researcher, this book is well organized and very clear, spanning both theory and implementation techniques, and reflecting considerable experience in teaching and expertise in the subject."
—John Reynolds, School of Computer Science, Carnegie Mellon University

"Pierce's book not only provides a comprehensive account of types for programming languages, but it does so in an engagingly elegant and concrete style that places equal emphasis on theoretical foundations and the practical problems of programming. This book will be the definitive reference for many years to come."
—Robert Harper, Professor, Computer Science Department, Carnegie Mellon University

"*Types and Programming Languages* is carefully written with a well-balanced choice of topics. It focuses on pragmatics, with the right level of necessary theory. The exercises range from easy to challenging and provide stimulating material for beginning and advanced readers, both programmers and the more theoretically minded."
—Henk Barendregt, Faculty of Science, Mathematics, and Computer Science, University of Nijmegen, The Netherlands

Cover photo by Benjamin C. Pierce

The MIT Press
Massachusetts Institute of Technology
Cambridge, Massachusetts 02142
http://mitpress.mit.edu

0-262-16209-1

Types and Programming Languages Pierce

Types and Programming Languages

Benjamin C. Pierce

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Lucida Bright by the author using the LaTeX document preparation system.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Pierce, Benjamin C.
Types and programming languages / Benjamin C. Pierce
p. cm.
Includes bibliographical references and index.
ISBN 0-262-16209-1 (hc. : alk. paper)
1. Programming languages (Electronic computers). I. Title.

QA76.7 .P54 2002
005.13—dc21

2001044428

Types and Programming Languages

Preface

The study of type systems—and of programming languages from a type-theoretic perspective—has become an energetic field with major applications in software engineering, language design, high-performance compiler implementation, and security. This text offers a comprehensive introduction to the fundamental definitions, results, and techniques in the area.

Audience

The book addresses two main audiences: graduate students and researchers specializing in programming languages and type theory, and graduate students and mature undergraduates from all areas of computer science who want an introduction to key concepts in the theory of programming languages. For the former group, the book supplies a thorough tour of the field, with sufficient depth to proceed directly to the research literature. For the latter, it provides extensive introductory material and a wealth of examples, exercises, and case studies. It can serve as the main text for both introductory graduate-level courses and advanced seminars in programming languages.

Goals

A primary aim is **coverage** of core topics, including basic operational semantics and associated proof techniques, the untyped lambda-calculus, simple type systems, universal and existential polymorphism, type reconstruction, subtyping, bounded quantification, recursive types, and type operators, with shorter discussions of numerous other topics.

A second main goal is **pragmatism**. The book concentrates on the use of type systems in programming languages, at the expense of some topics (such as denotational semantics) that probably would be included in a more mathematical text on typed lambda-calculi. The underlying computational substrate

is a call-by-value lambda-calculus, which matches most present-day programming languages and extends easily to imperative constructs such as references and exceptions. For each language feature, the main concerns are the practical *motivations* for considering this feature, the techniques needed to prove *safety* of languages that include it, and the *implementation issues* that it raises—in particular, the design and analysis of typechecking algorithms.

A further goal is respect for the **diversity** of the field; the book covers numerous individual topics and several well-understood combinations but does not attempt to bring everything together into a single unified system. Unified presentations have been given for some subsets of the topics—for example, many varieties of "arrow types" can be elegantly and compactly treated in the uniform notation of *pure type systems*—but the field as a whole is still growing too rapidly to be fully systematized.

The book is designed for **ease of use**, both in courses and for self-study. Full solutions are provided for most of the exercises. Core definitions are organized into self-contained figures for easy reference. Dependencies between concepts and systems are made as explicit as possible. The text is supplemented with an extensive bibliography and index.

A final organizing principle is **honesty.** All the systems discussed in the book (except a few that are only mentioned in passing) are implemented. Each chapter is accompanied by a typechecker and interpreter that are used to check the examples mechanically. These implementations are available from the book's web site and can be used for programming exercises, experimenting with extensions, and larger class projects.

To achieve these goals, some other desirable properties have necessarily been sacrificed. The most important of these is **completeness** of coverage. Surveying the whole area of programming languages and type systems is probably impossible in one book—certainly in a textbook. The focus here is on careful development of core concepts; numerous pointers to the research literature are supplied as starting points for further study. A second non-goal is the practical **efficiency** of the typechecking algorithms: this is not a book on industrial-strength compiler or typechecker implementation.

Structure

Part I of the book discusses untyped systems. Basic concepts of abstract syntax, inductive definitions and proofs, inference rules, and operational semantics are introduced first in the setting of a very simple language of numbers and booleans, then repeated for the untyped lambda-calculus. Part II covers the simply typed lambda-calculus and a variety of basic language features such as products, sums, records, variants, references, and exceptions. A pre-

liminary chapter on typed arithmetic expressions provides a gentle introduction to the key idea of type safety. An optional chapter develops a proof of normalization for the simply typed lambda-calculus using Tait's method. Part III addresses the fundamental mechanism of subtyping; it includes a detailed discussion of metatheory and two extended case studies. Part IV covers recursive types, in both the simple *iso-recursive* and the trickier *equi-recursive* formulations. The second of the two chapters in this part develops the metatheory of a system with equi-recursive types and subtyping in the mathematical framework of coinduction. Part V takes up polymorphism, with chapters on ML-style type reconstruction, the more powerful impredicative polymorphism of System F, existential quantification and its connections with abstract data types, and the combination of polymorphism and subtyping in systems with bounded quantification. Part VI deals with type operators. One chapter covers basic concepts; the next develops System F_ω and its metatheory; the next combines type operators and bounded quantification to yield System $F^\omega_{<:}$; the final chapter is a closing case study.

The major dependencies between chapters are outlined in Figure P-1. Gray arrows indicate that only part of a later chapter depends on an earlier one.

The treatment of each language feature discussed in the book follows a common pattern. Motivating examples are first; then formal definitions; then proofs of basic properties such as type safety; then (usually in a separate chapter) a deeper investigation of metatheory, leading to typechecking algorithms and their proofs of soundness, completeness, and termination; and finally (again in a separate chapter) the concrete realization of these algorithms as an OCaml (Objective Caml) program.

An important source of examples throughout the book is the analysis and design of features for object-oriented programming. Four case-study chapters develop different approaches in detail—a simple model of conventional imperative objects and classes (Chapter 18), a core calculus based on Java (Chapter 19), a more refined account of imperative objects using bounded quantification (Chapter 27), and a treatment of objects and classes in the purely functional setting of System $F^\omega_{<:}$, using existential types (Chapter 32).

To keep the book small enough to be covered in a one-semester advanced course—and light enough to be lifted by the average graduate student—it was necessary to exclude many interesting and important topics. Denotational and axiomatic approaches to semantics are omitted completely; there are already excellent books covering these approaches, and addressing them here would detract from this book's strongly pragmatic, implementation-oriented perspective. The rich connections between type systems and logic are suggested in a few places but not developed in detail; while important, these would take us too far afield. Many advanced features of programming lan-

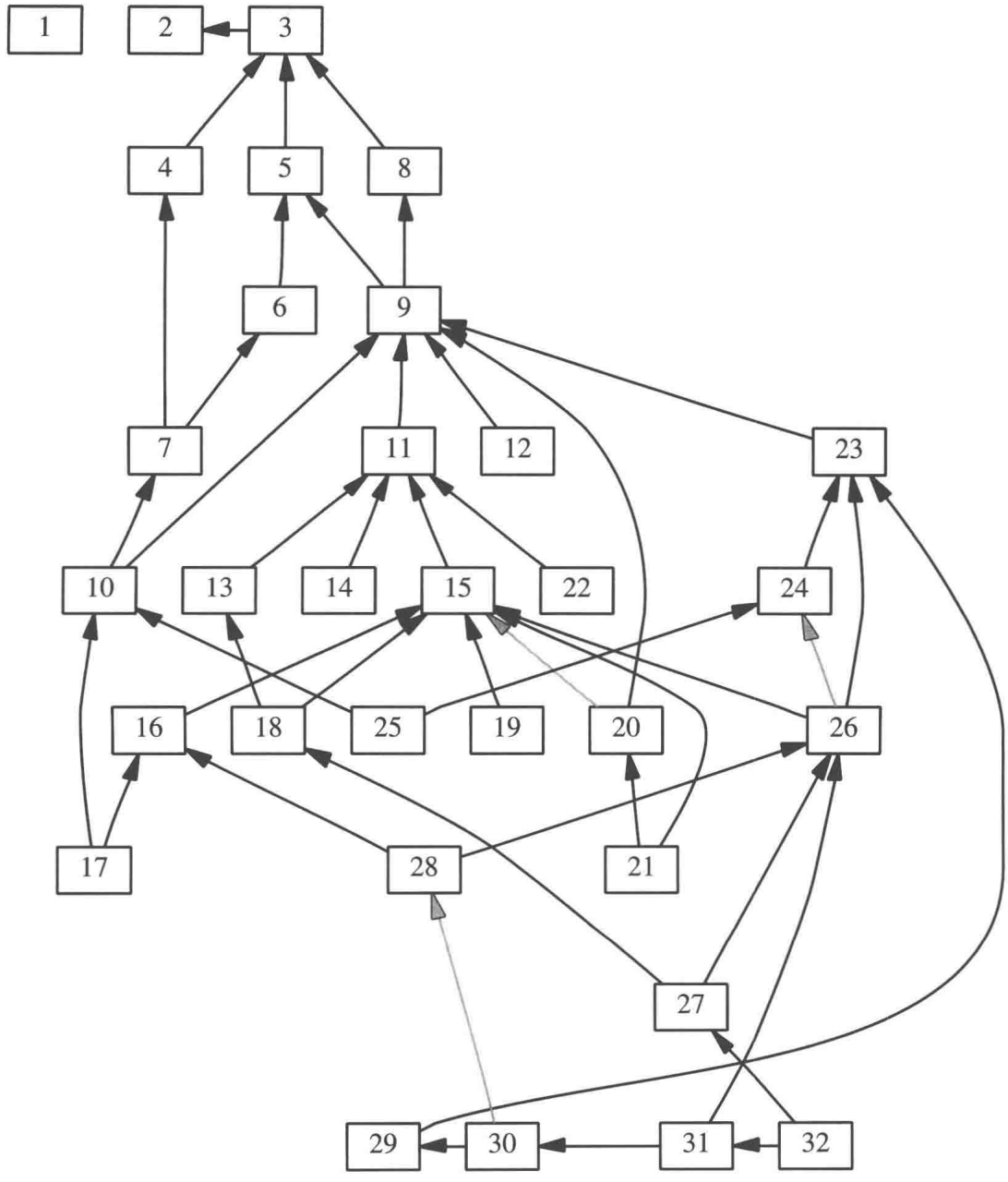

Figure P-1: Chapter dependencies

guages and type systems are mentioned only in passing, e.g, dependent types, intersection types, and the Curry-Howard correspondence; short sections on these topics provide starting points for further reading. Finally, except for a brief excursion into a Java-like core language (Chapter 19), the book focuses entirely on systems based on the lambda-calculus; however, the concepts and mechanisms developed in this setting can be transferred directly to related areas such as typed concurrent languages, typed assembly languages, and specialized object calculi.

Required Background

The text assumes no preparation in the theory of programming languages, but readers should start with a degree of mathematical maturity—in particular, rigorous undergraduate coursework in discrete mathematics, algorithms, and elementary logic.

Readers should be familiar with at least one higher-order functional programming language (Scheme, ML, Haskell, etc.), and with basic concepts of programming languages and compilers (abstract syntax, BNF grammars, evaluation, abstract machines, etc.). This material is available in many excellent undergraduate texts; I particularly like *Essentials of Programming Languages* by Friedman, Wand, and Haynes (2001) and *Programming Language Pragmatics* by Scott (1999). Experience with an object-oriented language such as Java (Arnold and Gosling, 1996) is useful in several chapters.

The chapters on concrete implementations of typecheckers present significant code fragments in OCaml (or Objective Caml), a popular dialect of ML. Prior knowledge of OCaml is helpful in these chapters, but not absolutely necessary; only a small part of the language is used, and features are explained at their first occurrence. These chapters constitute a distinct thread from the rest of the book and can be skipped completely if desired.

The best textbook on OCaml at the moment is Cousineau and Mauny's (1998). The tutorial materials packaged with the OCaml distribution (available at `http://caml.inria.fr` and `http://www.ocaml.org`) are also very readable.

Readers familiar with the other major dialect of ML, Standard ML, should have no trouble following the OCaml code fragments. Popular textbooks on Standard ML include those by Paulson (1996) and Ullman (1997).

Course Outlines

An intermediate or advanced graduate course should be able to cover most of the book in a semester. Figure P-2 gives a sample syllabus from an upper-

level course for doctoral students at the University of Pennsylvania (two 90-minute lectures a week, assuming minimal prior preparation in programming language theory but moving quickly).

For an undergraduate or an introductory graduate course, there are a number of possible paths through the material. A course on *type systems in programming* would concentrate on the chapters that introduce various typing features and illustrate their uses and omit most of the metatheory and implementation chapters. Alternatively, a course on *basic theory and implementation of type systems* would progress through all the early chapters, probably skipping Chapter 12 (and perhaps 18 and 21) and sacrificing the more advanced material toward the end of the book. Shorter courses can also be constructed by selecting particular chapters of interest using the dependency diagram in Figure P-1.

The book is also suitable as the main text for a more general graduate course in theory of programming languages. Such a course might spend half to two-thirds of a semester working through the better part of the book and devote the rest to, say, a unit on the theory of concurrency based on Milner's pi-calculus book (1999), an introduction to Hoare Logic and axiomatic semantics (e.g. Winskel, 1993), or a survey of advanced language features such as continuations or module systems.

In a course where term projects play a major role, it may be desirable to postpone some of the theoretical material (e.g., normalization, and perhaps some of the chapters on metatheory) so that a broad range of examples can be covered before students choose project topics.

Exercises

Most chapters include extensive exercises—some designed for pencil and paper, some involving programming examples *in* the calculi under discussion, and some concerning extensions to the ML implementations *of* these calculi. The estimated difficulty of each exercise is indicated using the following scale:

⋆	Quick check	30 seconds to 5 minutes
⋆⋆	Easy	≤ 1 hour
⋆⋆⋆	Moderate	≤ 3 hours
⋆⋆⋆⋆	Challenging	> 3 hours

Exercises marked ⋆ are intended as real-time checks of important concepts. Readers are strongly encouraged to pause for each one of these before moving on to the material that follows. In each chapter, a roughly homework-assignment-sized set of exercises is labeled RECOMMENDED.

LECTURE	TOPIC	READING
1.	Course overview; history; administrivia	1, (2)
2.	Preliminaries: syntax, operational semantics	3, 4
3.	Introduction to the lambda-calculus	5.1, 5.2
4.	Formalizing the lambda-calculus	5.3, 6, 7
5.	Types; the simply typed lambda-calculus	8, 9, 10
6.	Simple extensions; derived forms	11
7.	More extensions	11
8.	Normalization	12
9.	References; exceptions	13, 14
10.	Subtyping	15
11.	Metatheory of subtyping	16, 17
12.	Imperative objects	18
13.	Featherweight Java	19
14.	Recursive types	20
15.	Metatheory of recursive types	21
16.	Metatheory of recursive types	21
17.	Type reconstruction	22
18.	Universal polymorphism	23
19.	Existential polymorphism; ADTs	24, (25)
20.	Bounded quantification	26, 27
21.	Metatheory of bounded quantification	28
22.	Type operators	29
23.	Metatheory of F_ω	30
24.	Higher-order subtyping	31
25.	Purely functional objects	32
26.	Overflow lecture	

Figure P-2: Sample syllabus for an advanced graduate course

Complete solutions to most of the exercises are provided in Appendix A. To save readers the frustration of searching for solutions to the few exercises for which solutions are not available, those exercises are marked ↛.

Typographic Conventions

Most chapters introduce the features of some type system in a discursive style, then define the system formally as a collection of inference rules in one or more figures. For easy reference, these definitions are usually presented in full, including not only the new rules for the features under discussion at the moment, but also the rest of the rules needed to constitute a complete

calculus. The new parts are set on a gray background to make the "delta" from previous systems visually obvious.

An unusual feature of the book's production is that all the examples are mechanically typechecked during typesetting: a script goes through each chapter, extracts the examples, generates and compiles a custom typechecker containing the features under discussion, applies it to the examples, and inserts the checker's responses in the text. The system that does the hard parts of this, called *TinkerType,* was developed by Michael Levin and myself (2001).
Funding for this research was provided by the National Science Foundation, through grants CCR-9701826, *Principled Foundations for Programming with Objects,* and CCR-9912352, *Modular Type Systems.*

Electronic Resources

A web site associated with this book can be found at the following URL:

```
http://www.cis.upenn.edu/~bcpierce/tapl
```

Resources available on this site include errata for the text, suggestions for course projects, pointers to supplemental material contributed by readers, and a collection of implementations (typecheckers and simple interpreters) of the calculi covered in each chapter of the text.

These implementations offer an environment for experimenting with the examples in the book and testing solutions to exercises. They have also been polished for readability and modifiability and have been used successfully by students in my courses as the basis of both small implementation exercises and larger course projects. The implementations are written in OCaml. The OCaml compiler is available at no cost through `http://caml.inria.fr` and installs very easily on most platforms.

Readers should also be aware of the Types Forum, an email list covering all aspects of type systems and their applications. The list is moderated to ensure reasonably low volume and a high signal-to-noise ratio in announcements and discussions. Archives and subscription instructions can be found at `http://www.cis.upenn.edu/~bcpierce/types`.

Acknowledgments

Readers who find value in this book owe their biggest debt of gratitude to four mentors—Luca Cardelli, Bob Harper, Robin Milner, and John Reynolds—who taught me most of what I know about programming languages and types.

The rest I have learned mostly through collaborations; besides Luca, Bob, Robin, and John, my partners in these investigations have included Martín

Abadi, Gordon Plotkin, Randy Pollack, David N. Turner, Didier Rémy, Davide Sangiorgi, Adriana Compagnoni, Martin Hofmann, Giuseppe Castagna, Martin Steffen, Kim Bruce, Naoki Kobayashi, Haruo Hosoya, Atsushi Igarashi, Philip Wadler, Peter Buneman, Vladimir Gapeyev, Michael Levin, Peter Sewell, Jérôme Vouillon, and Eijiro Sumii. These collaborations are the foundation not only of my understanding, but also of my pleasure in the topic.

The structure and organization of this text have been improved by discussions on pedagogy with Thorsten Altenkirch, Bob Harper, and John Reynolds, and the text itself by corrections and comments from Jim Alexander, Penny Anderson, Josh Berdine, Tony Bonner, John Tang Boyland, Dave Clarke, Diego Dainese, Olivier Danvy, Matthew Davis, Vladimir Gapeyev, Bob Harper, Eric Hilsdale, Haruo Hosoya, Atsushi Igarashi, Robert Irwin, Takayasu Ito, Assaf Kfoury, Michael Levin, Vassily Litvinov, Pablo López Olivas, Dave MacQueen, Narciso Marti-Oliet, Philippe Meunier, Robin Milner, Matti Nykänen, Gordon Plotkin, John Prevost, Fermin Reig, Didier Rémy, John Reynolds, James Riely, Ohad Rodeh, Jürgen Schlegelmilch, Alan Schmitt, Andrew Schoonmaker, Olin Shivers, Perdita Stevens, Chris Stone, Eijiro Sumii, Val Tannen, Jérôme Vouillon, and Philip Wadler. (I apologize if I've inadvertently omitted anybody from this list.) Luca Cardelli, Roger Hindley, Dave MacQueen, John Reynolds, and Jonathan Seldin offered insiders' perspectives on some tangled historical points.

The participants in my graduate seminars at Indiana University in 1997 and 1998 and at the University of Pennsylvania in 1999 and 2000 soldiered through early versions of the manuscript; their reactions and comments gave me crucial guidance in shaping the book as you see it. Bob Prior and his team from The MIT Press expertly guided the manuscript through the many phases of the publication process. The book's design is based on LaTeX macros developed by Christopher Manning for The MIT Press.

Proofs of programs are too boring for the social process of mathematics to work. —*Richard DeMillo, Richard Lipton, and Alan Perlis, 1979*

... So don't rely on social processes for verification. —*David Dill, 1999*

Formal methods will never have a significant impact until they can be used by people that don't understand them. —*attributed to Tom Melham*