Advanced Topics in Types and Programming Languages

Advanced Topics in Types and Programming Languages

Benjamin C. Pierce, editor

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Lucida Bright by the editor and authors using the LaTeX document preparation system.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Advanced Topics in Types and programming languages / Benjamin C. Pierce, editor
p. cm.
Includes bibliographical references and index.
ISBN 0-262-16228-8 (hc.: alk. paper)
1. Programming languages (Electronic computers). I. Pierce, Benjamin C.
QA76.7.A36 2005
005.13—dc22
200457123

10 9 8 7 6 5 4 3 2 1

Contents

Preface

Overview

Work in type systems for programming languages now touches many parts of computer science, from language design and implementation to software engineering, network security, databases, and analysis of concurrent and distributed systems. The aim of this book, together with its predecessor, *Types and Programming Languages* (Pierce [2002]—henceforth *TAPL*) is to offer a comprehensive and accessible introduction to the area's central ideas, results, and techniques. The intended audience includes graduate students and researchers from other parts of computer science who want get up to speed in the area as a whole, as well as current researchers in programming languages who need comprehensible introductions to particular topics. Unlike *TAPL*, the present volume is conceived not as a unified text, but as a collection of more or less separate articles, authored by experts on their particular topics.

Required Background

Most of the material should be accessible to readers with a solid grasp of the basic notations and techniques of operational semantics and type systems—roughly, the first half of *TAPL*. Some chapters depend on more advanced topics from the second half of *TAPL* or earlier chapters of the present volume; these dependencies are indicated at the beginning of each chapter. Interchapter dependencies have been kept to a minimum to facilitate reading in any order.

Topics

Precise Type Analyses The first three chapters consider ways of extending simple type systems to give them a better grip on the run time behavior of programs. The first, **Substructural Type Systems,** by David Walker, surveys type systems based on analogies with "substructural" logics such as linear logic, in which one or more of the structural rules of conventional logics—which allow dropping, duplicating, and permuting assumptions—are omitted or allowed only under controlled circumstances. In substructural type systems, the type of a value is not only a description of its "shape," but also a capability for using it a certain number of times; this refinement plays a key role in advanced type systems being developed for a range of purposes, including static resource management and analyzing deadlocks and livelocks in concurrent systems. The chapter on **Dependent Types**, by David Aspinall and Martin Hofmann, describes a yet more powerful class of type systems, in which the behavior of computations on particular run-time values (not just generic "shapes") may be described at the type level. Dependent type systems blur the distinction between types and arbitrary correctness assertions, and between typechecking and theorem proving. The power of full dependent types has proved difficult to reconcile with language design desiderata such as automatic typechecking and the "phase distinction" between compile time and run time in compiled languages. Nevertheless, ideas of dependent typing have played a fruitful role in language design and theory over the years, offering a common conceptual foundation for numerous forms of "indexed" type systems. **Effect Types and Region-Based Memory Management**, by Fritz Henglein, Henning Makholm, and Henning Niss, introduces yet another idea for extending the reach of type systems: in addition to describing the shape of an expression's result (a static abstraction of the possible values that the expression may yield when evaluated), its type can also list a set of possible "effects," abstracting the possible computational effects (mutations to the store, input and output, etc.) that its evaluation may engender. Perhaps the most sophisticated application of this idea has been in memory management systems based on static "region inference," in which the effects manipulated by the typechecker track the program's ability to read and write in particular regions of the heap. For example, the ML Kit Compiler used a region analysis internally to implement the full Standard ML language without a garbage collector.

Types for Low-Level Languages The next part of the book addresses another research thrust that has generated considerable excitement over the past decade: the idea of adapting type system technologies originally developed for high-level languages to low-level languages such as assembly code and virtual machine bytecode. **Typed Assembly Language**, by Greg Morrisett, presents a low-level language with a type system based on the parametric polymorphism of System F and discusses how to construct a type-preserving compiler from a high-level language, through a series of *typed intermediate languages*, down to this typed assembly code. **Proof-Carrying Code**, by George Necula, presents a more general formulation in a logical setting with close ties to the dependent types described in Aspinall and Hofmann's chapter. The strength of this presentation is that it offers a natural transition from conventional type safety properties, such as memory safety, to more general security properties. A driving application area for both approaches is enforcing security guarantees when dealing with untrusted mobile code.

Types and Reasoning about Programs One attraction of rich type systems is that they support powerful methods of reasoning about programs—not only by compilers, but also by humans. One of the most useful, the technique of *logical relations*, is introduced in the chapter **Logical Relations and a Case Study in Equivalence Checking**, by Karl Crary. The extended example—proving the correctness of an algorithm for deciding a type-sensitive behavioral equivalence relation on terms in the simply typed lambda-calculus with a `Unit` type—foreshadows ideas developed further in Christopher Stone's chapter on type definitions. **Typed Operational Reasoning**, by Andrew Pitts, develops a more general theory of typed reasoning about program equivalence. Here the examples focus on proving representation independence properties for abstract data types in the setting of a rich language combining the universal and existential polymorphism of System F with records and recursive function definitions.

Types for Programming in the Large One of the most important projects in language *design* over the past decade and more has been the use of type-theory as a framework for the design of sophisticated *module systems*—languages for assembling large software systems from modular components. One highly developed line of work is embodied in the module systems found in modern ML dialects. **Design Considerations for ML-Style Module Systems**, by Robert Harper and Benjamin C. Pierce, offers an informal guided tour of the principal features of this class of module systems—a "big picture" introduction to a large but highly technical body of papers in the research literature. **Type Definitions**, by Christopher A. Stone, addresses the most critical and technically challenging feature of the type systems on which ML-style module systems are founded: *singleton kinds*, which allow type definitions to be internalized rather than being treated as meta-level abbreviations.

Type Inference The ML family of languages—including Standard ML, Objective Caml, and Moscow ML, as well as more distant relatives such as Haskell—

has for decades been a showcase for advances in typed language design and compiler implementation, and for the advantages of software construction in richly typed languages. One of the main reasons for the success of these languages is the combination of power and convenience offered by their *type inference* (or *type reconstruction*) algorithms. Basic ML type inference has been described in many places, but descriptions of the more advanced techniques used in production compilers for full-blown languages have until now been widely dispersed in the literature, when they were available at all. In **The Essence of ML Type Inference**, François Pottier and Didier Rémy offer a comprehensive, unified survey of the area.

Exercises

Most chapters include numerous exercises. 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. Some of the most important exercises are labeled RECOMMENDED.

Solutions to most of the exercises are provided in Appendix A. To save readers searching for solutions to exercises for which solutions are not available, these are marked ↛.

Electronic Resources

Additional materials associated with this book can be found at:

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

Resources available on this site will include errata for the text, pointers to supplemental material contributed by readers, and implementations associated with various chapters.

Acknowledgments

Many friends and colleagues have helped to improve the chapters as they developed. We are grateful to Amal Ahmed, Lauri Alanko, Jonathan Aldrich,