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
Contents

Preface ix

I Precise Type Analyses 1

1 Substructural Type Systems 3
   David Walker
   1.1 Structural Properties 4
   1.2 A Linear Type System 6
   1.3 Extensions and Variations 17
   1.4 An Ordered Type System 30
   1.5 Further Applications 36
   1.6 Notes 40

2 Dependent Types 45
   David Aspinall and Martin Hofmann
   2.1 Motivations 45
   2.2 Pure First-Order Dependent Types 50
   2.3 Properties 54
   2.4 Algorithmic Typing and Equality 56
   2.5 Dependent Sum Types 61
   2.6 The Calculus of Constructions 64
   2.7 Relating Abstractions: Pure Type Systems 71
   2.8 Programming with Dependent Types 74
   2.9 Implementation of Dependent Types 83
   2.10 Further Reading 86
3  Effect Types and Region-Based Memory Management 87
Fritz Henglein, Henning Makholm, and Henning Niss

3.1 Introduction and Overview 87
3.2 Value Flow by Typing with Labels 90
3.3 Effects 102
3.4 Region-Based Memory Management 106
3.5 The Tofte-Talpin Type System 114
3.6 Region Inference 123
3.7 More Powerful Models for Region-Based Memory Management 127
3.8 Practical Region-Based Memory Management Systems 133

II Types for Low-Level Languages 137

4  Typed Assembly Language 141
Greg Morrisett

4.1 TAL-0: Control-Flow-Safety 142
4.2 The TAL-0 Type System 146
4.3 TAL-1: Simple Memory-Safety 155
4.4 TAL-1 Changes to the Type System 161
4.5 Compiling to TAL-1 164
4.6 Scaling to Other Language Features 167
4.7 Some Real World Issues 172
4.8 Conclusions 175

5  Proof-Carrying Code 177
George Necula

5.1 Overview of Proof Carrying Code 177
5.2 Formalizing the Safety Policy 182
5.3 Verification-Condition Generation 187
5.4 Soundness Proof 199
5.5 The Representation and Checking of Proofs 204
5.6 Proof Generation 214
5.7 PCC beyond Types 216
5.8 Conclusion 219
### III Types and Reasoning about Programs  221

#### 6 Logical Relations and a Case Study in Equivalence Checking  223

**Karl Crary**

6.1 The Equivalence Problem  224  
6.2 Non-Type-Directed Equivalence Checking  225  
6.3 Type-Driven Equivalence  227  
6.4 An Equivalence Algorithm  228  
6.5 Completeness: A First Attempt  232  
6.6 Logical Relations  233  
6.7 A Monotone Logical Relation  236  
6.8 The Main Lemma  237  
6.9 The Fundamental Theorem  239  
6.10 Notes  243

#### 7 Typed Operational Reasoning  245

**Andrew Pitts**

7.1 Introduction  245  
7.2 Overview  246  
7.3 Motivating Examples  247  
7.4 The Language  253  
7.5 Contextual Equivalence  261  
7.6 An Operationally Based Logical Relation  266  
7.7 Operational Extensionality  279  
7.8 Notes  288

### IV Types for Programming in the Large  291

#### 8 Design Considerations for ML-Style Module Systems  293

**Robert Harper and Benjamin C. Pierce**

8.1 Basic Modularity  294  
8.2 Type Checking and Evaluation of Modules  298  
8.3 Compilation and Linking  302  
8.4 Phase Distinction  305  
8.5 Abstract Type Components  307  
8.6 Module Hierarchies  317  
8.7 Signature Families  320  
8.8 Module Families  324  
8.9 Advanced Topics  338  
8.10 Relation to Some Existing Languages  341  
8.11 History and Further Reading  343
9  Type Definitions  347
    Christopher A. Stone
    9.1 Definitions in the Typing Context  351
    9.2 Definitions in Module Interfaces  358
    9.3 Singleton Kinds  367
    9.4 Notes  384

V  Type Inference  387

10 The Essence of ML Type Inference  389
    François Pottier and Didier Rémy
    10.1 What Is ML?  389
    10.2 Constraints  407
    10.3 HM(X)  422
    10.4 Constraint Generation  429
    10.5 Type Soundness  434
    10.6 Constraint Solving  438
    10.7 From ML-the-Calculus to ML-the-Language  451
    10.8 Rows  460

A  Solutions to Selected Exercises  491

References  535

Index  567
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 to 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. Inter-chapter 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,