Design Concepts in Programming Languages

Franklyn Turbak and David Gifford
with Mark A. Sheldon

The MIT Press
Cambridge, Massachusetts
London, England

©2008 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

MIT Press books may be purchased at special quantity discounts for business or sales promotional use. For information, please email special_sales@mitpress.mit.edu or write to Special Sales Department, The MIT Press, 55 Hayward Street, Cambridge, MA 02142.

This book was set in LaTeX by the authors, and was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Turbak, Franklyn A.
Design concepts in programming languages / Franklyn A. Turbak and David K. Gifford, with Mark A. Sheldon.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-262-20175-9 (hardcover : alk. paper)
1. Programming languages (Electronic computers). I. Gifford, David K., 1954–. II. Sheldon, Mark A. III. Title.
QA76.7.T845 2008
005.1—dc22 2008013841

10 9 8 7 6 5 4 3 2 1

Contents

Preface xix

Acknowledgments xxi

I Foundations 1

1 Introduction 3
1.1 Programming Languages 3
1.2 Syntax, Semantics, and Pragmatics 4
1.3 Goals 6
1.4 PostFix: A Simple Stack Language 8
1.4.1 Syntax 8
1.4.2 Semantics 9
1.4.3 The Pitfalls of Informal Descriptions 14
1.5 Overview of the Book 15

2 Syntax 19
2.1 Abstract Syntax 20
2.2 Concrete Syntax 22
2.3 S-Expression Grammars Specify ASTs 23
2.3.1 S-Expressions 23
2.3.2 The Structure of S-Expression Grammars 24
2.3.3 Phrase Tags 30
2.3.4 Sequence Patterns 30
2.3.5 Notational Conventions 32
2.3.6 Mathematical Foundation of Syntactic Domains 36
2.4 The Syntax of PostFix 39

3 Operational Semantics 45
3.1 The Operational Semantics Game 45
3.2 Small-step Operational Semantics (SOS) 49
3.2.1 Formal Framework 49
3.2.2 Example: An SOS for PostFix 52
3.2.3 Rewrite Rules 54
3.2.4 Operational Execution 58

3.2.5 Progress Rules 62
3.2.6 Context-based Semantics 71
3.3 Big-step Operational Semantics 75
3.4 Operational Reasoning 79
3.5 Deterministic Behavior of EL 80
3.6 Termination of PostFix Programs 84
3.6.1 Energy 84
3.6.2 The Proof of Termination 86
3.6.3 Structural Induction 88
3.7 Safe PostFix Transformations 89
3.7.1 Observational Equivalence 89
3.7.2 Transform Equivalence 92
3.7.3 Transform Equivalence Implies Observational Equivalence 96
3.8 Extending PostFix 100

4 Denotational Semantics 113
4.1 The Denotational Semantics Game 113
4.2 A Denotational Semantics for EL 117
4.2.1 Step 1: Restricted ELMM 117
4.2.2 Step 2: Full ELMM 120
4.2.3 Step 3: ELM 124
4.2.4 Step 4: EL 127
4.2.5 A Denotational Semantics Is Not a Program 128
4.3 A Denotational Semantics for PostFix 131
4.3.1 A Semantic Algebra for PostFix 131
4.3.2 A Meaning Function for PostFix 134
4.3.3 Semantic Functions for PostFix: the Details 142
4.4 Denotational Reasoning 145
4.4.1 Program Equality 145
4.4.2 Safe Transformations: A Denotational Approach 147
4.4.3 Technical Difficulties 150
4.5 Relating Operational and Denotational Semantics 150
4.5.1 Soundness 151
4.5.2 Adequacy 157
4.5.3 Full Abstraction 159
4.5.4 Operational versus Denotational: A Comparison 161

5 Fixed Points 163
5.1 The Fixed Point Game 163
5.1.1 Recursive Definitions 163
5.1.2 Fixed Points 166
5.1.3 The Iterative Fixed Point Technique 168
5.2 Fixed Point Machinery 174
5.2.1 Partial Orders 174
5.2.2 Complete Partial Orders (CPOs) 182
5.2.3 Pointedness 185
5.2.4 Monotonicity and Continuity 187
5.2.5 The Least Fixed Point Theorem 190
5.2.6 Fixed Point Examples 191
5.2.7 Continuity and Strictness 197
5.3 Reflexive Domains 201
5.4 Summary 203

II Dynamic Semantics 205

6 FL: A Functional Language 207
6.1 Decomposing Language Descriptions 207
6.2 The Structure of FL 208
6.2.1 FLK: The Kernel of the FL Language 209
6.2.2 FL Syntactic Sugar 218
6.2.3 The FL Standard Library 235
6.2.4 Examples 239
6.3 Variables and Substitution 244
6.3.1 Terminology 244
6.3.2 Abstract Syntax DAGs and Stoy Diagrams 248
6.3.3 Alpha-Equivalence 250
6.3.4 Renaming and Variable Capture 251
6.3.5 Substitution 253
6.4 An Operational Semantics for FLK 258
6.4.1 FLK Evaluation 258
6.4.2 FLK Simplification 270
6.5 A Denotational Semantics for FLK 275
6.5.1 Semantic Algebra 275
6.5.2 Valuation Functions 280
6.6 The Lambda Calculus 290

6.6.1 Syntax of the Lambda Calculus 291
6.6.2 Operational Semantics of the Lambda Calculus 291
6.6.3 Denotational Semantics of the Lambda Calculus 296
6.6.4 Representational Games 297

7 Naming 307
7.1 Parameter Passing 309
7.1.1 Call-by-Name vs. Call-by-Value: The Operational View 310
7.1.2 Call-by-Name vs. Call-by-Value: The Denotational View 316
7.1.3 Nonstrict versus Strict Pairs 318
7.1.4 Handling `rec` in a CBV Language 320
7.1.5 Thunking 324
7.1.6 Call-by-Denotation 328
7.2 Name Control 332
7.2.1 Hierarchical Scoping: Static and Dynamic 334
7.2.2 Multiple Namespaces 347
7.2.3 Nonhierarchical Scope 352
7.3 Object-oriented Programming 362
7.3.1 HOOK: An Object-oriented Kernel 362
7.3.2 HOOPLA 368
7.3.3 Semantics of HOOK 370

8 State 383
8.1 FL Is a Stateless Language 384
8.2 Simulating State in FL 390
8.2.1 Iteration 390
8.2.2 Single-Threaded Data Flow 392
8.2.3 Monadic Style 394
8.2.4 Imperative Programming 397
8.3 Mutable Data: FLIC 397
8.3.1 Mutable Cells 397
8.3.2 Examples of Imperative Programming 400
8.3.3 An Operational Semantics for FLICK 405
8.3.4 A Denotational Semantics for FLICK 411
8.3.5 Call-by-Name versus Call-by-Value Revisited 425
8.3.6 Referential Transparency, Interference, and Purity 427
8.4 Mutable Variables: FLAVAR 429
8.4.1 Mutable Variables 429
8.4.2 FLAVAR 430
8.4.3 Parameter-passing Mechanisms for FLAVAR 432

9 Control 443
- 9.1 Motivation: Control Contexts and Continuations 443
- 9.2 Using Procedures to Model Control 446
  - 9.2.1 Representing Continuations as Procedures 446
  - 9.2.2 Continuation-Passing Style (CPS) 449
  - 9.2.3 Multiple-value Returns 450
  - 9.2.4 Nonlocal Exits 455
  - 9.2.5 Coroutines 457
  - 9.2.6 Error Handling 461
  - 9.2.7 Backtracking 465
- 9.3 Continuation-based Semantics of FLICK 471
  - 9.3.1 A Standard Semantics of FLICK 472
  - 9.3.2 A Computation-based Continuation Semantics of FLICK 482
- 9.4 Nonlocal Exits 493
  - 9.4.1 `label` and `jump` 494
  - 9.4.2 A Denotational Semantics for `label` and `jump` 497
  - 9.4.3 An Operational Semantics for `label` and `jump` 503
  - 9.4.4 `call-with-current-continuation` (`cwcc`) 505
- 9.5 Iterators: A Simple Coroutining Mechanism 506
- 9.6 Exception Handling 513
  - 9.6.1 `raise`, `handle`, and `trap` 515
  - 9.6.2 A Standard Semantics for Exceptions 519
  - 9.6.3 A Computation-based Semantics for Exceptions 524
  - 9.6.4 A Desugaring-based Implementation of Exceptions 527
  - 9.6.5 Examples Revisited 530

10 Data 539
- 10.1 Products 539
  - 10.1.1 Positional Products 541
  - 10.1.2 Named Products 549
  - 10.1.3 Nonstrict Products 551
  - 10.1.4 Mutable Products 561
- 10.2 Sums 567
- 10.3 Sum of Products 577
- 10.4 Data Declarations 583
- 10.5 Pattern Matching 590
  - 10.5.1 Introduction to Pattern Matching 590
  - 10.5.2 A Desugaring-based Semantics of `match` 594
  - 10.5.3 Views 605

III Static Semantics 615

11 Simple Types 617
- 11.1 Static Semantics 617
- 11.2 What Is a Type? 620
- 11.3 Dimensions of Types 622
  - 11.3.1 Dynamic versus Static Types 623
  - 11.3.2 Explicit versus Implicit Types 625
  - 11.3.3 Simple versus Expressive Types 627
- 11.4 μFLEX: A Language with Explicit Types 628
  - 11.4.1 Types 629
  - 11.4.2 Expressions 631
  - 11.4.3 Programs and Syntactic Sugar 634
  - 11.4.4 Free Identifiers and Substitution 636
- 11.5 Type Checking in μFLEX 640
  - 11.5.1 Introduction to Type Checking 640
  - 11.5.2 Type Environments 643
  - 11.5.3 Type Rules for μFLEX 645
  - 11.5.4 Type Derivations 648
  - 11.5.5 Monomorphism 655
- 11.6 Type Soundness 661
  - 11.6.1 What Is Type Soundness? 661
  - 11.6.2 An Operational Semantics for μFLEX 662
  - 11.6.3 Type Soundness of μFLEX 667
- 11.7 Types and Strong Normalization 673
- 11.8 Full FLEX: Typed Data and Recursive Types 675
  - 11.8.1 Typed Products 675
  - 11.8.2 Type Equivalence 679
  - 11.8.3 Typed Mutable Data 681
  - 11.8.4 Typed Sums 682
  - 11.8.5 Typed Lists 685
  - 11.8.6 Recursive Types 688
  - 11.8.7 Full FLEX Summary 696

12 Polymorphism and Higher-order Types 701
- 12.1 Subtyping 701
  - 12.1.1 FLEX/S: FLEX with Subtyping 702
  - 12.1.2 Dimensions of Subtyping 713
  - 12.1.3 Subtyping and Inheritance 723
- 12.2 Polymorphic Types 725

12.2.1 Monomorphic Types Are Not Expressive 725
12.2.2 Universal Polymorphism: FLEX/SP 727
12.2.3 Deconstructible Data Types 738
12.2.4 Bounded Quantification 745
12.2.5 Ad Hoc Polymorphism 748
12.3 Higher-order Types: Descriptions and Kinds 750
12.3.1 Descriptions: FLEX/SPD 750
12.3.2 Kinds and Kind Checking: FLEX/SPDK 758
12.3.3 Discussion 764

13 Type Reconstruction 769
13.1 Introduction 769
13.2 μFLARE: A Language with Implicit Types 772
13.2.1 μFLARE Syntax and Type Erasure 772
13.2.2 Static Semantics of μFLARE 774
13.2.3 Dynamic Semantics and Type Soundness of μFLARE 778
13.3 Type Reconstruction for μFLARE 781
13.3.1 Type Substitutions 781
13.3.2 Unification 783
13.3.3 The Type-Constraint-Set Abstraction 787
13.3.4 A Reconstruction Algorithm for μFLARE 790
13.4 Let Polymorphism 801
13.4.1 Motivation 801
13.4.2 A μFLARE Type System with Let Polymorphism 803
13.4.3 μFLARE Type Reconstruction with Let Polymorphism 808
13.5 Extensions 813
13.5.1 The Full FLARE Language 813
13.5.2 Mutable Variables 820
13.5.3 Products and Sums 821
13.5.4 Sum-of-products Data Types 826

14 Abstract Types 839
14.1 Data Abstraction 839
14.1.1 A Point Abstraction 840
14.1.2 Procedural Abstraction Is Not Enough 841
14.2 Dynamic Locks and Keys 843
14.3 Existential Types 847
14.4 Nonce Types 859
14.5 Dependent Types 869
14.5.1 A Dependent Package System 870

14.5.2 Design Issues with Dependent Types 877

15 Modules 889

15.1 An Overview of Modules and Linking 889

15.2 An Introduction to FLEX/M 891

15.3 Module Examples: Environments and Tables 901

15.4 Static Semantics of FLEX/M Modules 910

15.4.1 Scoping 910

15.4.2 Type Equivalence 911

15.4.3 Subtyping 912

15.4.4 Type Rules 912

15.4.5 Implicit Projection 918

15.4.6 Typed Pattern Matching 921

15.5 Dynamic Semantics of FLEX/M Modules 923

15.6 Loading Modules 925

15.6.1 Type Soundness of `load` via a Load-Time Check 927

15.6.2 Type Soundness of `load` via a Compile-Time Check 928

15.6.3 Referential Transparency of `load` for File-Value Coherence 930

15.7 Discussion 932

15.7.1 Scoping Limitations 932

15.7.2 Lack of Transparent and Translucent Types 933

15.7.3 The Coherence Problem 934

15.7.4 Purity Issues 937

16 Effects Describe Program Behavior 943

16.1 Types, Effects, and Regions: What, How, and Where 943

16.2 A Language with a Simple Effect System 945

16.2.1 Types, Effects, and Regions 945

16.2.2 Type and Effect Rules 951

16.2.3 Reconstructing Types and Effects: Algorithm Z 959

16.2.4 Effect Masking Hides Unobservable Effects 972

16.2.5 Effect-based Purity for Generalization 974

16.3 Using Effects to Analyze Program Behavior 978

16.3.1 Control Transfers 978

16.3.2 Dynamic Variables 983

16.3.3 Exceptions 985

16.3.4 Execution Cost Analysis 988

16.3.5 Storage Deallocation and Lifetime Analysis 991

16.3.6 Control Flow Analysis 995

16.3.7 Concurrent Behavior 996

16.3.8 Mobile Code Security 999

IV Pragmatics 1003

17 Compilation 1005
17.1 Why Do We Study Compilation? 1005
17.2 TORTOISE Architecture 1007
17.2.1 Overview of TORTOISE 1007
17.2.2 The Compiler Source Language: FLARE/V 1009
17.2.3 Purely Structural Transformations 1012
17.3 Transformation 1: Desugaring 1013
17.4 Transformation 2: Globalization 1014
17.5 Transformation 3: Assignment Conversion 1019
17.6 Transformation 4: Type/Effect Reconstruction 1025
17.6.1 Propagating Type and Effect Information 1026
17.6.2 Effect-based Code Optimization 1026
17.7 Transformation 5: Translation 1030
17.7.1 The Compiler Intermediate Language: FIL 1030
17.7.2 Translating FLARE to FIL 1036
17.8 Transformation 6: Renaming 1038
17.9 Transformation 7: CPS Conversion 1042
17.9.1 The Structure of TORTOISE CPS Code 1044
17.9.2 A Simple CPS Transformation 1049
17.9.3 A More Efficient CPS Transformation 1058
17.9.4 CPS-Converting Control Constructs 1070
17.10 Transformation 8: Closure Conversion 1075
17.10.1 Flat Closures 1076
17.10.2 Variations on Flat Closure Conversion 1085
17.10.3 Linked Environments 1090
17.11 Transformation 9: Lifting 1094
17.12 Transformation 10: Register Allocation 1098
17.12.1 The FIL$_{reg}$ Language 1098
17.12.2 A Register Allocation Algorithm 1102
17.12.3 The Expansion Phase 1104
17.12.4 The Register Conversion Phase 1104
17.12.5 The Spilling Phase 1112

18 Garbage Collection 1119
18.1 Why Garbage Collection? 1119
18.2 FRM: The FIL Register Machine 1122
18.2.1 The FRM Architecture 1122
18.2.2 FRM Descriptors 1123
18.2.3 FRM Blocks 1127
18.3 A Block Is Dead if It Is Unreachable 1130
18.3.1 Reference Counting 1131
18.3.2 Memory Tracing 1132
18.4 Stop-and-copy GC 1133
18.5 Garbage Collection Variants 1141
18.5.1 Mark-sweep GC 1141
18.5.2 Tag-free GC 1141
18.5.3 Conservative GC 1142
18.5.4 Other Variations 1142
18.6 Static Approaches to Automatic Deallocation 1144

A A Metalanguage 1147
A.1 The Basics 1147
A.1.1 Sets 1148
A.1.2 Boolean Operators and Predicates 1151
A.1.3 Tuples 1152
A.1.4 Relations 1153
A.2 Functions 1155
A.2.1 What Is a Function? 1156
A.2.2 Application 1158
A.2.3 More Function Terminology 1159
A.2.4 Higher-order Functions 1160
A.2.5 Multiple Arguments and Results 1161
A.2.6 Lambda Notation 1165
A.2.7 Recursion 1168
A.2.8 Lambda Notation Is Not Lisp! 1169
A.3 Domains 1171
A.3.1 Motivation 1171
A.3.2 Types 1172
A.3.3 Product Domains 1173
A.3.4 Sum Domains 1176
A.3.5 Sequence Domains 1181
A.3.6 Function Domains 1184

A.4 Metalanguage Summary 1186
A.4.1 The Metalanguage Kernel 1186
A.4.2 The Metalanguage Sugar 1188

B Our Pedagogical Languages 1197

References 1199

Index 1227