# Practical Foundations for Programming Languages

## SECOND EDITION

Robert Harper
Carnegie Mellon University

[Revision 1.43 of 04.15.2014.]

Preview

Copyright © 2014 by Robert Harper.

All Rights Reserved.

The electronic version of this work is licensed under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License. To view a copy of this license, visit

http://creativecommons.org/licenses/by-nc-nd/3.0/us/

or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Preview

# Preface

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), Constable (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, Iliano Cervesato, Lin Chase, Rowan Davies, Derek Dreyer, Dan Licata, Zhong Shao, Rob Simmons, and Todd Wilson for their extensive

PREVIEW

efforts in reading and criticizing the book. I also thank the following people for their suggestions: Joseph Abrahamson, Arbob Ahmad, Zena Ariola, Eric Bergstrome, Guy Blelloch, William Byrd, Luis Caires, Luca Cardelli, Manuel Chakravarty, Richard C. Cobbe, Karl Crary, 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 Jedynak, 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, 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, Carter Schonwald, Dale Schumacher, Dana Scott, Shayak Sen, Pawel Sobocinski, Kristina Sojakova, Daniel Spoonhower, Paulo Tanimoto, Peter Thiemann, Bernardo Toninho, Michael Tschantz, Kami Vaniea, Carsten Varming, David Walker, Dan Wang, Jack Wileden, 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 in Germany 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

Preview

# Contents

**Preface** iii

**I Judgments and Rules** 1

**1 Syntactic Objects** 3
1.1 Abstract Syntax Trees . . . 3
1.2 Abstract Binding Trees . . . 6
1.3 Notes . . . 10

**2 Inductive Definitions** 13
2.1 Judgments . . . 13
2.2 Inference Rules . . . 14
2.3 Derivations . . . 15
2.4 Rule Induction . . . 16
2.5 Iterated and Simultaneous Inductive Definitions . . . 18
2.6 Defining Functions by Rules . . . 19
2.7 Modes . . . 20
2.8 Notes . . . 21

**3 Hypothetical and General Judgments** 23
3.1 Hypothetical Judgments . . . 23
3.1.1 Derivability . . . 23
3.1.2 Admissibility . . . 25
3.2 Hypothetical Inductive Definitions . . . 26
3.3 General Judgments . . . 27
3.4 Generic Inductive Definitions . . . 29
3.5 Notes . . . 30

Preview

## II Statics and Dynamics 33

**4 Statics 35**
4.1 Syntax . . . 35
4.2 Type System . . . 36
4.3 Structural Properties . . . 37
4.4 Notes . . . 39

**5 Dynamics 41**
5.1 Transition Systems . . . 41
5.2 Structural Dynamics . . . 42
5.3 Contextual Dynamics . . . 44
5.4 Equational Dynamics . . . 46
5.5 Notes . . . 48

**6 Type Safety 51**
6.1 Preservation . . . 51
6.2 Progress . . . 52
6.3 Run-Time Errors . . . 53
6.4 Extrinsic and Intrinsic Safety . . . 54
6.5 Notes . . . 55

**7 Evaluation Dynamics 57**
7.1 Evaluation Dynamics . . . 57
7.2 Relating Structural and Evaluation Dynamics . . . 58
7.3 Type Safety, Revisited . . . 59
7.4 Cost Dynamics . . . 60
7.5 Notes . . . 61

## III Total Functions 63

**8 Function Definitions and Values 65**
8.1 First-Order Functions . . . 66
8.2 Higher-Order Functions . . . 67
8.3 Evaluation Dynamics and Definitional Equality . . . 69
8.4 Dynamic Scope . . . 70
8.5 Notes . . . 71

**9 Gödel's T 73**
9.1 Statics . . . 73
9.2 Dynamics . . . 74
9.3 Definability . . . 76
9.4 Undefinability . . . 77
9.5 Notes . . . 79


Preview

**IV Finite Data Types** **81**

**10 Product Types** **83**
10.1 Nullary and Binary Products . . . 83
10.2 Finite Products . . . 85
10.3 Primitive Mutual Recursion . . . 86
10.4 Notes . . . 87

**11 Sum Types** **89**
11.1 Nullary and Binary Sums . . . 89
11.2 Finite Sums . . . 91
11.3 Applications of Sum Types . . . 92
11.3.1 Void and Unit . . . 92
11.3.2 Booleans . . . 92
11.3.3 Enumerations . . . 93
11.3.4 Options . . . 94
11.4 Notes . . . 95

**12 Pattern Matching** **97**
12.1 A Pattern Language . . . 98
12.2 Statics . . . 98
12.3 Dynamics . . . 99
12.4 Exhaustiveness and Redundancy . . . 101
12.4.1 Match Constraints . . . 101
12.4.2 Enforcing Exhaustiveness and Redundancy . . . 102
12.4.3 Checking Exhaustiveness and Redundancy . . . 104
12.5 Notes . . . 105

**V Infinite Data Types** **107**

**13 Generic Programming** **109**
13.1 Introduction . . . 109
13.2 Polynomial Type Operators . . . 110
13.3 Positive Type Operators . . . 112
13.4 Notes . . . 113

**14 Inductive and Co-Inductive Types** **115**
14.1 Motivating Examples . . . 115
14.2 Statics . . . 118
14.2.1 Types . . . 118
14.2.2 Expressions . . . 119
14.3 Dynamics . . . 120
14.4 Solving Type Equations . . . 121

Preview

14.5 Notes . . . 121

VI Partial Functions 125

15 Plotkin's PCF 127
15.1 Statics . . . 129
15.2 Dynamics . . . 130
15.3 Definability . . . 131
15.4 Finite and Infinite Data Structures . . . 133
15.5 Totality and Partiality . . . 134
15.6 Notes . . . 135

16 Recursive Types 137
16.1 Solving Type Equations . . . 138
16.2 Inductive and Coinductive Types . . . 139
16.3 Self-Reference . . . 140
16.4 The Origin of State . . . 142
16.5 Notes . . . 143

VII Dynamic Types 145

17 Church's Untyped $\lambda$-Calculus 147
17.1 The $\lambda$-Calculus . . . 147
17.2 Definability . . . 148
17.3 Scott's Theorem . . . 150
17.4 Untyped Means Uni-Typed . . . 152
17.5 Notes . . . 153

18 Dynamic Typing 155
18.1 Dynamically Typed PCF . . . 156
18.2 Variations and Extensions . . . 159
18.3 Critique of Dynamic Typing . . . 161
18.4 Notes . . . 162

19 Hybrid Typing 165
19.1 A Hybrid Language . . . 165
19.2 Dynamic as Static Typing . . . 167
19.3 Optimization of Dynamic Typing . . . 168
19.4 Static Versus Dynamic Typing . . . 170
19.5 Notes . . . 171

Preview

## VIII Variable Types 173

**20 Girard's System F 175**
20.1 Polymorphic Abstraction . . . 176
20.2 Polymorphic Definability . . . 179
20.2.1 Products and Sums . . . 179
20.2.2 Natural Numbers . . . 180
20.3 Parametricity Overview . . . 181
20.4 Restricted Forms of Polymorphism . . . 182
20.4.1 Predicative Fragment . . . 182
20.4.2 Prenex Fragment . . . 183
20.4.3 Rank-Restricted Fragments . . . 185
20.5 Notes . . . 186

**21 Abstract Types 187**
21.1 Existential Types . . . 187
21.1.1 Statics . . . 188
21.1.2 Dynamics . . . 188
21.1.3 Safety . . . 189
21.2 Data Abstraction . . . 189
21.3 Definability of Existentials . . . 191
21.4 Representation Independence . . . 191
21.5 Notes . . . 193

**22 Constructors and Kinds 195**
22.1 Constructors and Kinds . . . 196
22.2 Expressions and Types . . . 198
22.3 Canonizing Substitution . . . 199
22.4 Canonization . . . 201
22.5 Notes . . . 203

## IX Subtyping 205

**23 Subtyping 207**
23.1 Subsumption . . . 207
23.2 Varieties of Subtyping . . . 208
23.2.1 Numeric Types . . . 208
23.2.2 Product Types . . . 209
23.2.3 Sum Types . . . 209
23.2.4 Dynamic Types . . . 210
23.3 Variance . . . 211
23.3.1 Product and Sum Types . . . 211
23.3.2 Partial Function Types . . . 211


Preview

23.3.3 Recursive Types . . . 212
23.3.4 Quantified Types . . . 214
23.4 Dynamics and Safety . . . 215
23.5 Notes . . . 216

**24 Type Refinements** **219**
24.1 Statics . . . 220
24.2 Boolean Blindness . . . 226
24.3 Refinement Safety . . . 228
24.4 Notes . . . 229

**25 Singleton Kinds** **233**
25.1 Overview . . . 234
25.2 Singletons . . . 235
25.3 Dependent Kinds . . . 236
25.4 Higher Singletons . . . 240
25.5 Notes . . . 241

**X Dynamic Dispatch** **243**

**26 Classes and Methods** **245**
26.1 The Dispatch Matrix . . . 246
26.2 Class-Based Organization . . . 247
26.3 Method-Based Organization . . . 249
26.4 Self-Reference . . . 250
26.5 Notes . . . 252

**27 Inheritance** **255**
27.1 Class and Method Extension . . . 255
27.2 Class-Based Inheritance . . . 256
27.3 Method-Based Inheritance . . . 257
27.4 Notes . . . 258

**XI Exceptions and Continuations** **261**

**28 Control Stacks** **263**
28.1 Machine Definition . . . 263
28.2 Safety . . . 265
28.3 Correctness of the Control Machine . . . 266
28.3.1 Completeness . . . 267
28.3.2 Soundness . . . 268
28.4 Notes . . . 269

Preview

**29 Exceptions** **271**
29.1 Simple Failures . . . 271
29.2 Value-Carrying Exceptions . . . 273
29.3 Exception Values . . . 274
29.4 Modal Separation of Exceptions . . . 276
29.5 Notes . . . 278

**30 Continuations** **281**
30.1 Overview . . . 281
30.2 Continuation Dynamics . . . 283
30.3 Coroutines from Continuations . . . 284
30.4 Notes . . . 288

**XII Types and Propositions** **289**

**31 Constructive Logic** **291**
31.1 Constructive Semantics . . . 292
31.2 Constructive Logic . . . 292
31.2.1 Provability . . . 293
31.2.2 Proof Terms . . . 295
31.3 Proof Dynamics . . . 296
31.4 Propositions as Types . . . 297
31.5 Notes . . . 297

**32 Classical Logic** **301**
32.1 Classical Logic . . . 302
32.1.1 Provability and Refutability . . . 302
32.1.2 Proofs and Refutations . . . 304
32.2 Deriving Elimination Forms . . . 306
32.3 Proof Dynamics . . . 307
32.4 Law of the Excluded Middle . . . 309
32.5 The Double-Negation Translation . . . 310
32.6 Notes . . . 311

**XIII Symbolic Data** **313**

**33 Symbols** **315**
33.1 Symbol Declaration . . . 316
33.1.1 Scoped Dynamics . . . 316
33.1.2 Scope-Free Dynamics . . . 317
33.2 Symbol References . . . 318
33.2.1 Statics . . . 318

Preview

33.2.2 Dynamics . . . 319
33.2.3 Safety . . . 319
33.3 Notes . . . 320

**34 Fluid Binding** **323**
34.1 Statics . . . 323
34.2 Dynamics . . . 324
34.3 Type Safety . . . 325
34.4 Some Subtleties . . . 326
34.5 Fluid References . . . 327
34.6 Notes . . . 329

**35 Dynamic Classification** **331**
35.1 Dynamic Classes . . . 331
35.1.1 Statics . . . 332
35.1.2 Dynamics . . . 332
35.1.3 Safety . . . 333
35.2 Class References . . . 333
35.3 Definability of Dynamic Classes . . . 334
35.4 Applications of Dynamic Classification . . . 335
35.4.1 Classifying Secrets . . . 335
35.4.2 Exception Values . . . 336
35.5 Notes . . . 336

**XIV Mutable State** **339**

**36 Reynolds's Algol** **341**
36.1 Basic Commands . . . 341
36.1.1 Statics . . . 342
36.1.2 Dynamics . . . 343
36.1.3 Safety . . . 345
36.2 Some Programming Idioms . . . 346
36.3 Typed Commands and Typed Assignables . . . 347
36.4 Notes . . . 349

**37 Assignable References** **353**
37.1 Capabilities . . . 353
37.2 Scoped Assignables . . . 354
37.3 Free Assignables . . . 356
37.4 Safety . . . 358
37.5 Benign Effects . . . 360
37.6 Notes . . . 362

Preview

## XV Laziness 365

**38 Lazy Evaluation 367**
38.1 By-Need Dynamics for PCF . . . 368
38.2 Safety for By-Need PCF . . . 371
38.3 By-Need Dynamics for FPC . . . 372
38.4 Suspensions . . . 373
38.5 Notes . . . 375

**39 Polarization 377**
39.1 Positive and Negative Types . . . 378
39.2 Focusing . . . 379
39.3 Statics . . . 379
39.4 Dynamics . . . 381
39.5 Safety . . . 382
39.6 Notes . . . 383

## XVI Parallelism 385

**40 Nested Parallelism 387**
40.1 Binary Fork-Join . . . 387
40.2 Cost Dynamics . . . 390
40.3 Multiple Fork-Join . . . 393
40.4 Provably Efficient Implementations . . . 395
40.5 Notes . . . 398

**41 Futures and Speculations 399**
41.1 Futures . . . 400
41.1.1 Statics . . . 400
41.1.2 Sequential Dynamics . . . 400
41.2 Speculations . . . 401
41.2.1 Statics . . . 401
41.2.2 Sequential Dynamics . . . 401
41.3 Parallel Dynamics . . . 402
41.4 Applications of Futures . . . 404
41.5 Notes . . . 405

## XVII Concurrency and Distribution 407

**42 Milner's Process Calculus 409**
42.1 Actions and Events . . . 409
42.2 Interaction . . . 411
42.3 Replication . . . 413


Preview

42.4 Allocating Channels . . . 414
42.5 Communication . . . 417
42.6 Channel Passing . . . 420
42.7 Universality . . . 422
42.8 Notes . . . 423

**43 Concurrent Algol** **425**
43.1 Concurrent Algol . . . 426
43.2 Broadcast Communication . . . 428
43.3 Selective Communication . . . 430
43.4 Free Assignables as Processes . . . 433
43.5 Notes . . . 434

**44 Distributed Algol** **435**
44.1 Statics . . . 435
44.2 Dynamics . . . 437
44.3 Safety . . . 438
44.4 Situated Types . . . 439
44.5 Notes . . . 442

**XVIII Separate Compilation and Modularity** **443**

**45 Components and Linking** **445**
45.1 Simple Units and Linking . . . 445
45.2 Initialization and Effects . . . 447
45.3 Notes . . . 448

**46 Type Abstractions and Type Classes** **449**
46.1 Type Abstraction . . . 450
46.2 Type Classes . . . 451
46.3 A Module Language . . . 454
46.4 First- and Second-Class . . . 458
46.5 Notes . . . 459

**47 Hierarchy and Parameterization** **461**
47.1 Hierarchy . . . 461
47.2 Parameterizaton . . . 464
47.3 Extending Modules with Hierarchies and Parameterization . . . 467
47.4 Applicative Functors . . . 469
47.5 Notes . . . 470

Preview

## XIX Equational Reasoning 471

**48 Equational Reasoning for T** **473**
48.1 Observational Equivalence . . . 473
48.2 Logical Equivalence . . . 477
48.3 Logical and Observational Equivalence Coincide . . . 478
48.4 Some Laws of Equality . . . 480
48.4.1 General Laws . . . 481
48.4.2 Equality Laws . . . 481
48.4.3 Induction Law . . . 482
48.5 Notes . . . 482

**49 Equational Reasoning for PCF** **483**
49.1 Observational Equivalence . . . 483
49.2 Logical Equivalence . . . 484
49.3 Logical and Observational Equivalence Coincide . . . 484
49.4 Compactness . . . 487
49.5 Lazy Natural Numbers . . . 489
49.6 Notes . . . 491

**50 Parametricity** **493**
50.1 Overview . . . 493
50.2 Observational Equivalence . . . 494
50.3 Logical Equivalence . . . 495
50.4 Parametricity Properties . . . 500
50.5 Representation Independence, Revisited . . . 503
50.6 Notes . . . 504

**51 Process Equivalence** **505**
51.1 Process Calculus . . . 505
51.2 Strong Equivalence . . . 507
51.3 Weak Equivalence . . . 510
51.4 Notes . . . 511

## XX Appendices 513

**A Finite Sets and Finite Functions** **515**

**B Solutions to Exercises** **517**

**C Errata** **561**


Preview

Preview

# Part I

# Judgments and Rules

Preview

Preview

# Chapter 1

# Syntactic Objects

Programming languages are languages, a means of expressing computations in a form comprehensible to both people and machines. The syntax of a language specifies the means by which various sorts of phrases (expressions, commands, declarations, and so forth) may be combined to form programs. But what sort of thing are these phrases? What is a program made of?

The informal concept of syntax may be seen to involve several distinct concepts. The *surface*, or *concrete*, *syntax* is concerned with how phrases are entered and displayed on a computer. The surface syntax is usually thought of as given by strings of characters from some alphabet (say, ASCII or Unicode). The *structural*, or *abstract*, *syntax* is concerned with the structure of phrases, specifically how they are composed from other phrases. At this level a phrase is a tree, called an *abstract syntax tree*, whose nodes are operators that combine several phrases to form another phrase. The *binding* structure of syntax is concerned with the introduction and use of identifiers: how they are declared, and how declared identifiers are to be used. At this level phrases are *abstract binding trees*, which enrich abstract syntax trees with the concepts of binding and scope.

We will not concern ourselves in this book with matters of concrete syntax, but will instead work at the level of abstract syntax. To prepare the ground for the rest of the book, we begin in this chapter by defininining abstract syntax trees and abstract binding trees and some functions and relations associated with them. The definitions are a bit technical, but are absolutely fundamental to what follows. It is probably best to skim this chapter on first reading, returning to it only as the need arises.

## 1.1 Abstract Syntax Trees

An *abstract syntax tree*, or *ast* for short, is an ordered tree whose leaves are *variables*, and whose interior nodes are *operators* whose *arguments* are its children. Ast's are classified into a variety of *sorts* corresponding to different forms of syntax. A *variable* stands for an unspecified, or generic, piece of syntax of a specified sort. Ast's may be combined by an *operator*, which has both a sort and an *arity*, a finite sequence of sorts specifying the number and sorts of its arguments. An operator of sort $s$ and arity $s_1, \ldots, s_n$ combines $n \geq 0$ ast's of sort $s_1, \ldots, s_n$, respectively, into a compound

Preview

ast of sort $s$. As a matter of terminology, a *nullary* operator is one that takes no arguments, a *unary* operator takes one, a *binary* operator two, and so forth.

The concept of a variable is central, and therefore deserves special emphasis. As in mathematics a variable is an *unknown* object drawn from some domain, its *range of significance*. In school mathematics the (often implied) range of significance is the set of real numbers. Here variables range over ast's of a specified sort. Being an unknown, the meaning of a variable is given by *substitution*, the process of "plugging in" an object from the domain for the variable in a formula. So, in school, we might plug in $\pi$ for $x$ in a polynomial, and calculate the result. Here we would plug in an ast of the appropriate sort for a variable in an ast to obtain another ast. The process of substitution is easily understood for ast's, because it amounts to a "physical" replacement of the variable by an ast within another ast. We will shortly consider a generalization of the concept of ast for which the substitution process is somewhat more complex, but the essential idea is the same, and bears repeating: *a variable is given meaning by substitution*.

For example, consider a simple language of expressions built from numbers, addition, and multiplication. The abstract syntax of such a language would consist of a single sort, Exp, and an infinite collection of operators that generate the forms of expression: $\texttt{num}[n]$ is a nullary operator of sort Exp whenever $n \in \mathbb{N}$; `plus` and `times` are binary operators of sort Exp whose arguments are both of sort Exp. The expression $2 + (3 \times x)$, which involves a variable, $x$, would be represented by the ast

$$\texttt{plus}(\texttt{num}[2]; \texttt{times}(\texttt{num}[3]; x))$$

of sort Exp, under the assumption that $x$ is also of this sort. Because, say, $\texttt{num}[4]$, is an ast of sort Exp, we may plug it in for $x$ in the above ast to obtain the ast

$$\texttt{plus}(\texttt{num}[2]; \texttt{times}(\texttt{num}[3]; \texttt{num}[4])),$$

which is written informally as $2 + (3 \times 4)$. We may, of course, plug in more complex ast's of sort Exp for $x$ to obtain other ast's as result.

The tree structure of ast's supports a very useful principle of reasoning, called *structural induction*. Suppose that we wish to prove that some property, $\mathcal{P}(a)$, holds of all ast's, $a$, of a given sort. To show this it is enough to consider all the ways in which $a$ may be generated, and show that the property holds in each case, under the assumption that it holds for each of its constituent ast's (if any). So, in the case of the sort Exp just described, we must show

1. The property holds for any variable, $x$, of sort Exp: $\mathcal{P}(x)$.
2. The property holds for any number, $\texttt{num}[n]$: for every $n \in \mathbb{N}$, $\mathcal{P}(\texttt{num}[n])$.
3. Assuming that the property holds for $a_1$ and $a_2$, show that it holds for $\texttt{plus}(a_1; a_2)$ and $\texttt{times}(a_1; a_2)$: if $\mathcal{P}(a_1)$ and $\mathcal{P}(a_2)$, then $\mathcal{P}(\texttt{plus}(a_1; a_2))$ and $\mathcal{P}(\texttt{times}(a_1; a_2))$.

Because these cases exhaust all possibilities for the formation of $a$, we are assured that $\mathcal{P}(a)$ holds for any ast $a$ of sort Exp.

For the sake of precision, and to prepare the ground for further developments, we will now give precise definitions of the foregoing concepts. Let $\mathcal{S}$ be a finite set of sorts. Let $\{ O_s \}_{s \in \mathcal{S}}$ be a



Preview

sort-indexed family of *operators*, $o$, of sort $s$ with arity $\mathsf{ar}(o) = (s_1, \ldots, s_n)$. Let $\{ \mathcal{X}_s \}_{s \in \mathcal{S}}$ be a sort-indexed family of *variables*, $x$, of each sort $s$. The family $\mathcal{A}[\mathcal{X}] = \{ \mathcal{A}[\mathcal{X}]_s \}_{s \in \mathcal{S}}$ of ast's of sort $s$ is defined to be the smallest (with respect to containment) family satisfying the following conditions:

1. A variable of sort $s$ is an ast of sort $s$: if $x \in \mathcal{X}_s$, then $x \in \mathcal{A}[\mathcal{X}]_s$.
2. Operators combine ast's: if $o$ is an operator of sort $s$ such that $\mathsf{ar}(o) = (s_1, \ldots, s_n)$, and if $a_1 \in \mathcal{A}[\mathcal{X}]_{s_1}, \ldots, a_n \in \mathcal{A}[\mathcal{X}]_{s_n}$, then $o(a_1; \ldots; a_n) \in \mathcal{A}[\mathcal{X}]_s$.

It follows from this definition that the principle of *structural induction* may be used to prove that some property, $\mathcal{P}$, holds of every ast. To show $\mathcal{P}(a)$ holds for every $a \in \mathcal{A}[\mathcal{X}]$, it is enough to show:

1. If $x \in \mathcal{X}_s$, then $\mathcal{P}_s(x)$.
2. If $o \in \mathcal{O}_s$ and $\mathsf{ar}(o) = (s_1, \ldots, s_n)$, then if $\mathcal{P}_{s_1}(a_1)$ and $\ldots$ and $\mathcal{P}_{s_n}(a_n)$, then $\mathcal{P}_s(o(a_1; \ldots; a_n))$.

For example, it is easy to prove by structural induction that $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{A}[\mathcal{Y}]$ whenever $\mathcal{X} \subseteq \mathcal{Y}$.

If $\mathcal{X}$ is a sort-indexed family of variables, we write $\mathcal{X}, x$, where $x$ is a variable of sort $s$ such that $x \notin \mathcal{X}_s$, to stand for the family of sets $\mathcal{Y}$ such that $\mathcal{Y}_s = \mathcal{X}_s \cup \{ x \}$ and $\mathcal{Y}_{s'} = \mathcal{X}_{s'}$ for all $s' \neq s$. The family $\mathcal{X}, x$, where $x$ is a variable of sort $s$, is said to be the family obtained by *adjoining* the variable $x$ to the family $\mathcal{X}$.

Variables are given meaning by *substitution*. If $x$ is a variable of sort $s$, $a \in \mathcal{A}[\mathcal{X}, x]_{s'}$, and $b \in \mathcal{A}[\mathcal{X}]_s$, then $[b/x]a \in \mathcal{A}[\mathcal{X}]_{s'}$ is defined to be the result of substituting $b$ for every occurrence of $x$ in $a$. The ast $a$ is called the *target*, and $x$ is called the *subject*, of the substitution. Substitution is defined by the following equations:

1. $[b/x]x = b$ and $[b/x]y = y$ if $x \neq y$.
2. $[b/x]o(a_1; \ldots; a_n) = o([b/x]a_1; \ldots; [b/x]a_n)$.

For example, we may check that

$$[\mathtt{num}[2]/x]\mathtt{plus}(x; \mathtt{num}[3]) = \mathtt{plus}(\mathtt{num}[2]; \mathtt{num}[3]).$$

We may prove by structural induction that substitution on ast's is well-defined.

**Theorem 1.1.** *If $a \in \mathcal{A}[\mathcal{X}, x]$, then for every $b \in \mathcal{A}[\mathcal{X}]$ there exists a unique $c \in \mathcal{A}[\mathcal{X}]$ such that $[b/x]a = c$*

*Proof.* By structural induction on $a$. If $a = x$, then $c = b$ by definition, otherwise if $a = y \neq x$, then $c = y$, also by definition. Otherwise, $a = o(a_1, \ldots, a_n)$, and we have by induction unique $c_1, \ldots, c_n$ such that $[b/x]a_1 = c_1$ and $\ldots$ $[b/x]a_n = c_n$, and so $c$ is $c = o(c_1; \ldots; c_n)$, by definition of substitution. □

In most cases it is possible to enumerate all of the operators that generate the ast's of a sort up front, as we have done in the foregoing examples. However, in some situations this is not possible—certain operators are available only within certain contexts. In such cases we cannot fix



Preview

the collection of operators, $\mathcal{O}$, in advance, but rather must allow it to be *extensible*. This is achieved by considering families of operators that are indexed by symbolic *parameters*, or *indices*, that serve as "names" for the instances. For example, in Chapter 35 we will consider a family of nullary operators, $\texttt{cls}[u]$, where $u$ is a symbolic parameter drawn from the set of *active* parameters. It is essential that distinct parameters determine distinct operators: if $u$ and $v$ are active parameters, and $u \neq v$, then $\texttt{cls}[u]$ and $\texttt{cls}[v]$ are different operators. Extensibility is achieved by introducing new active parameters. So, if $u$ is not active, then $\texttt{cls}[u]$ makes no sense, but if $u$ becomes active, then $\texttt{cls}[u]$ is a nullary operator.

Parameters must be carefully distinguished from variables. Whereas a variable stands for an unknown ast of its sort, a parameter *does not stand for anything*. It is a purely symbolic identifier whose only significance is whether it is the same or different as another parameter. Whereas variables are given meaning by substitution, it is not sensible to substitute for a parameter (we may only rename them). As a consequence, disequality of variables is not well-defined, because we may substitute the same tree for different variables, but disequality of parameters is of the essence.

To account for the set of active parameters, we will write $\mathcal{A}[\mathcal{U};\mathcal{X}]$ for the set of ast's with variables drawn from $\mathcal{X}$ and with parameters drawn from $\mathcal{U}$. Certain operators, such as $\texttt{cls}[u]$, are *parameterized* by parameters, $u$, of a given sort. The parameters are distinguished from the arguments by the square brackets around them. Instances of such operators are permitted only for parameters drawn from the active set, $\mathcal{U}$. So, for example, if $u \in \mathcal{U}$, then $\texttt{cls}[u]$ is a nullary operator, but if $u \notin \mathcal{U}$, then $\texttt{cls}[u]$ is not a valid operator. In the next section we will introduce the means of extending $\mathcal{U}$ to make operators available within that context.

## 1.2 Abstract Binding Trees

*Abstract binding trees*, or *abt's*, enrich ast's with the means to introduce new variables and parameters, called a *binding*, with a specified range of significance, called its *scope*. The scope of a binding is an abt within which the bound identifier may be used, either as a placeholder (in the case of a variable declaration) or as the index of some operator (in the case of a parameter declaration). Thus the set of active identifiers may be larger within a subtree of an abt than it is within the surrounding tree. Moreover, different subtrees may introduce identifiers with disjoint scopes. The crucial principle is that any use of an identifier should be understood as a reference, or abstract pointer, to its binding. One consequence is that the choice of identifiers is immaterial, so long as we can always associate a unique binding with each use of an identifier.

As a motivating example, consider the expression $\texttt{let}\ x\ \texttt{be}\ a_1\ \texttt{in}\ a_2$, which introduces a variable, $x$, for use within the expression $a_2$ to stand for the expression $a_1$. The variable $x$ is bound by the `let` expression for use within $a_2$; any use of $x$ within $a_1$ refers to a different variable that happens to have the same name. For example, in the expression $\texttt{let}\ x\ \texttt{be}\ 7\ \texttt{in}\ x + x$ occurrences of $x$ in the addition refer to the variable introduced by the `let`. On the other hand in the expression $\texttt{let}\ x\ \texttt{be}\ x * x\ \texttt{in}\ x + x$, occurrences of $x$ within the multiplication refer to a different variable than those occurring within the addition. The latter occurrences refer to the binding introduced by the `let`, whereas the former refer to some outer binding not displayed here.

The names of bound variables are immaterial insofar as they determine the same binding. So,



Preview

for example, the expression $\texttt{let}\ x\ \texttt{be}\ x * x\ \texttt{in}\ x + x$ could just as well have been written $\texttt{let}\ y\ \texttt{be}\ x * x\ \texttt{in}\ y + y$ without changing its meaning. In the former case the variable $x$ is bound within the addition, and in the latter it is the variable $y$, but the "pointer structure" remains the same. On the other hand the expression $\texttt{let}\ x\ \texttt{be}\ y * y\ \texttt{in}\ x + x$ has a different meaning to these two expressions, because now the variable $y$ within the multiplication refers to a different surrounding variable. Renaming of bound variables is constrained to the extent that it must not alter the reference structure of the expression. For example, the expression

$$\texttt{let}\ x\ \texttt{be}\ 2\ \texttt{in}\ \texttt{let}\ y\ \texttt{be}\ 3\ \texttt{in}\ x + x$$

has a different meaning than the expression

$$\texttt{let}\ y\ \texttt{be}\ 2\ \texttt{in}\ \texttt{let}\ y\ \texttt{be}\ 3\ \texttt{in}\ y + y,$$

because the $y$ in the expression $y + y$ in the second case refers to the inner declaration, not the outer one as before.

The concept of an ast may be enriched to account for binding and scope of a variable. These enriched ast's are called *abstract binding trees*, or *abt's* for short. Abt's generalize ast's by allowing an operator to bind any finite number (possibly zero) of variables in each argument position. An argument to an operator is called an *abstractor*, and has the form $x_1, \ldots, x_k.a$. The sequence of variables $x_1, \ldots, x_k$ are bound within the abt $a$. (When $k$ is zero, we elide the distinction between $.a$ and $a$ itself.) Written in the form of an abt, the expression $\texttt{let}\ x\ \texttt{be}\ a_1\ \texttt{in}\ a_2$ has the form $\texttt{let}(a_1; x.a_2)$, which more clearly specifies that the variable $x$ is bound within $a_2$, and not within $a_1$. We often write $\vec{x}$ to stand for a finite sequence $x_1, \ldots, x_n$ of distinct variables, and write $\vec{x}.a$ to mean $x_1, \ldots, x_n.a$.

To account for binding, the arity of an operator is generalized to consist of a finite sequence of *valences*. The length of the sequence determines the number of arguments, and each valence determines the sort of the argument and the number and sorts of the variables that are bound within it. A valence of the form $(s_1, \ldots, s_k)s$ specifies an argument of sort $s$ that binds $k$ variables of sorts $s_1, \ldots, s_k$ within it. We often write $\vec{s}$ for a finite sequence $s_1, \ldots, s_n$ of sorts, and we say that $\vec{x}$ is of sort $\vec{s}$ to mean that the two sequences have the same length and that each $x_i$ is of sort $s_i$.

Thus, for example, the arity of the operator `let` is $(\mathsf{Exp}, (\mathsf{Exp})\mathsf{Exp})$, which indicates that it takes two arguments described as follows:

1. The first argument is of sort Exp and binds no variables.
2. The second argument is of sort Exp and binds one variable of sort Exp.

The expression $\texttt{let}\ x\ \texttt{be}\ 2 + 2\ \texttt{in}\ x \times x$ may then be represented using this operator by the abt

$$\texttt{let}(\texttt{plus}(\texttt{num}[2]; \texttt{num}[2]); x.\texttt{times}(x; x)).$$

Let $\mathcal{O}$ be a sort-indexed family of operators, $o$, with arities, $\mathsf{ar}(o)$. For a given sort-indexed family, $\mathcal{X}$, of variables, the sort-indexed family of abt's, $\mathcal{B}[\mathcal{X}]$, is defined similarly to $\mathcal{A}[\mathcal{X}]$, except that the set of active variables changes for each argument according to which variables are bound within it. A first cut at the definition is as follows:

PREVIEW

1. If $x \in \mathcal{X}_s$, then $x \in \mathcal{B}[\mathcal{X}]_s$.
2. If $\mathsf{ar}(o) = ((\vec{s}_1)s_1, \ldots, (\vec{s}_n)s_n)$, and if, for each $1 \leq i \leq n$, $\vec{x}_i$ is of sort $\vec{s}_i$ and $a_i \in \mathcal{B}[\mathcal{X}, \vec{x}_i]_{s_i}$, then $o(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n) \in \mathcal{B}[\mathcal{X}]_s$.

The bound variables are adjoined to the set of active variables within each argument, with the sort of each variable determined by the valence of the operator.

This definition is *almost* correct, but fails to properly account for renaming of bound variables. An abt of the form $\mathtt{let}(a_1; x.\mathtt{let}(a_2; x.a_3))$ is ill-formed according to this definition, because the first binding adjoins $x$ to $\mathcal{X}$, which implies that the second cannot also adjoin $x$ to $\mathcal{X}, x$ without causing confusion. The solution is to ensure that each of the arguments is well-formed regardless of the choice of bound variable names. This is achieved by altering the second clause of the definition using renaming as follows:[1]

> If $\mathsf{ar}(o) = ((\vec{s}_1)s_1, \ldots, (\vec{s}_n)s_n)$, and if, for each $1 \leq i \leq n$ and for each renaming $\pi_i : \vec{x}_i \leftrightarrow \vec{x}_i'$, where $\vec{x}_i' \notin \mathcal{X}$, we have $\pi_i \cdot a_i \in \mathcal{B}[\mathcal{X}, \vec{x}_i']$, then $o(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n) \in \mathcal{B}[\mathcal{X}]_s$.

The renaming ensures that when we encounter nested binders we avoid collisions. This is called the *freshness condition on binders* because it ensures that all bound variables are "fresh" relative to the surrounding context.

The principle of structural induction extends to abt's, and is called *structural induction modulo renaming*. It states that to show that $\mathcal{P}[\mathcal{X}](a)$ holds for every $a \in \mathcal{B}[\mathcal{X}]$, it is enough to show the following:

1. if $x \in \mathcal{X}_s$, then $\mathcal{P}[\mathcal{X}]_s(x)$.
2. For every $o$ of sort $s$ and arity $((\vec{s}_1)s_1, \ldots, (\vec{s}_n)s_n)$, and if for each $1 \leq i \leq n$, we have $\mathcal{P}[\mathcal{X}, \vec{x}_i']_{s_i}(\pi_i \cdot a_i)$ for every $\pi_i : \vec{x}_i \leftrightarrow \vec{x}_i'$ with $\vec{x}_i' \notin \mathcal{X}$, then $\mathcal{P}[\mathcal{X}]_s(o(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n))$.

The renaming in the second condition ensures that the inductive hypothesis holds for *all* fresh choices of bound variable names, and not just the ones actually given in the abt.

As an example let us define the judgment $x \in a$, where $a \in \mathcal{B}[\mathcal{X}, x]$, to mean that $x$ *occurs free* in $a$. Informally, this means that $x$ is bound somewhere outside of $a$, rather than within $a$ itself. If $x$ is bound within $a$, then those occurrences of $x$ are different from those occurring outside the binding. The following definition ensures that this is the case:

1. $x \in x$.
2. $x \in o(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n)$ if there exists $1 \leq i \leq n$ such that for every fresh renaming $\pi : \vec{x}_i \leftrightarrow \vec{z}_i$ we have $x \in \pi \cdot a_i$.

The first condition states that $x$ is free in $x$, but not free in $y$ for any variable $y$ other than $x$. The second condition states that if $x$ is free in some argument, independently of the choice of bound variable names in that argument, then it is free in the overall abt. This implies, in particular, that $x$ is *not* free in $\mathtt{let}(\mathtt{zero}; x.x)$.

[1] The action of a renaming extends to abt's in the obvious way by replacing every occurrence of $x$ by $\pi(x)$, including any occurrences in the variable list of an abstractor as well as within its body.



Preview

The relation $a =_\alpha b$ of *α-equivalence* (so-called for historical reasons), is defined to mean that $a$ and $b$ are identical up to the choice of bound variable names. This relation is defined to be the strongest congruence containing the following two conditions:

1. $x =_\alpha x$.
2. $o(\vec{x}_1.a_1;\ldots;\vec{x}_n.a_n) =_\alpha o(\vec{x}'_1.a'_1;\ldots;\vec{x}'_n.a'_n)$ if for every $1 \leq i \leq n$, $\pi_i \cdot a_i =_\alpha \pi'_i \cdot a'_i$ for all fresh renamings $\pi_i : \vec{x}_i \leftrightarrow \vec{z}_i$ and $\pi'_i : \vec{x}'_i \leftrightarrow \vec{z}_i$.

The idea is that we rename $\vec{x}_i$ and $\vec{x}'_i$ consistently, avoiding confusion, and check that $a_i$ and $a'_i$ are $\alpha$-equivalent. If $a =_\alpha b$, then $a$ and $b$ are said to be *α-variants* of each other.

Some care is required in the definition of *substitution* of an abt $b$ of sort $s$ for free occurrences of a variable $x$ of sort $s$ in some abt $a$ of some sort, written $[b/x]a$. Substitution is partially defined by the following conditions:

1. $[b/x]x = b$, and $[b/x]y = y$ if $x \neq y$.
2. $[b/x]o(\vec{x}_1.a_1;\ldots;\vec{x}_n.a_n) = o(\vec{x}_1.a'_1;\ldots;\vec{x}_n.a'_n)$, where, for each $1 \leq i \leq n$, we require that $\vec{x}_i \notin b$, and we set $a'_i = [b/x]a_i$ if $x \notin \vec{x}_i$, and $a'_i = a_i$ otherwise.

If $x$ is bound in some argument to an operator, then substitution does not descend into its scope, for to do so would be to confuse two distinct variables. For this reason we must take care to define $a'_i$ in the second equation according to whether or not $x \in \vec{x}_i$. The requirement that $\vec{x}_i \notin b$ in the second equation is called *capture avoidance*. If some $x_{i,j}$ occurred free in $b$, then the result of the substitution $[b/x]a_i$ would in general contain $x_{i,j}$ free as well, but then forming $\vec{x}_i.[b/x]a_i$ would *incur capture* by changing the referent of $x_{i,j}$ to be the $j$th bound variable of the $i$th argument. In such cases *substitution is undefined* because we cannot replace $x$ by $b$ in $a_i$ without incurring capture.

One way around this is to alter the definition of substitution so that the bound variables in the result are chosen fresh by substitution. By the principle of structural induction we know inductively that, for any renaming $\pi_i : \vec{x}_i \leftrightarrow \vec{x}'_i$ with $\vec{x}'_i$ fresh, the substitution $[b/x](\pi_i \cdot a_i)$ is well-defined. Hence we may define

$$[b/x]o(\vec{x}_1.a_1;\ldots;\vec{x}_n.a_n) = o(\vec{x}'_1.[b/x](\pi_1 \cdot a_1);\ldots;\vec{x}'_n.[b/x](\pi_n \cdot a_n))$$

for some particular choice of fresh bound variable names (any choice will do). There is no longer any need to take care that $x \notin \vec{x}_i$ in each argument, because the freshness condition on binders ensures that this cannot occur, the variable $x$ already being active. Noting that

$$o(\vec{x}_1.a_1;\ldots;\vec{x}_n.a_n) =_\alpha o(\vec{x}'_1.\pi_1 \cdot a_1;\ldots;\vec{x}'_n.\pi_n \cdot a_n),$$

another way to avoid undefined substitutions is to first choose an $\alpha$-variant of the target of the substitution whose binders avoid any free variables in the substituting abt, and then perform substitution without fear of incurring capture. In other words substitution is totally defined on $\alpha$-equivalence classes of abt's.

To avoid all the bureaucracy of binding, we adopt the following *identification convention* throughout this book:

Preview

*Abstract binding trees are always to be identified up to $\alpha$-equivalence.*

That is, we implicitly work with $\alpha$-equivalence classes of abt's, rather than abt's themselves. We tacitly assert that all operations and relations on abt's respect $\alpha$-equivalence, so that they are properly defined on $\alpha$-equivalence classes of abt's. Whenever we examine an abt, we are choosing a representative of its $\alpha$-equivalence class, and we have no control over how the bound variable names are chosen. On the other hand experience shows that any operation or property of interest respects $\alpha$-equivalence, so there is no obstacle to achieving it. Indeed, we might say that a property or operation is legitimate exactly insofar as it respects $\alpha$-equivalence!

Parameters, as well as variables, may be bound within an argument of an operator. Such binders introduce a "new," or "fresh," parameter within the scope of the binder wherein it may be used to form further abt's. To allow for parameter declaration, the valence of an argument is generalized to indicate the sorts of the parameters bound within it, as well as the sorts of the variables, by writing $(\vec{s_1};\vec{s_2})s$, where $\vec{s_1}$ specifies the sorts of the parameters and $\vec{s_2}$ specifies the sorts of the variables. The sort-indexed family $\mathcal{B}[\mathcal{U};\mathcal{X}]$ is the set of abt's determined by a fixed set of operators using the parameters, $\mathcal{U}$, and the variables, $\mathcal{X}$. We rely on naming conventions to distinguish parameters from variables, reserving $u$ and $v$ for parameters, and $x$ and $y$ for variables.

## 1.3 Notes

The concept of abstract syntax has its orgins in the pioneering work of Church, Turing, and Gödel, who first considered the possibility of writing programs that act on representations of programs. Originally programs were represented by natural numbers, using encodings, now called *Gödel-numberings*, based on the prime factorization theorem. Any standard text on mathematical logic, such as Kleene (1952), contains a thorough account of such representations. The Lisp language (McCarthy, 1965; Allen, 1978) introduced a much more practical and direct representation of syntax as *symbolic expressions*. These ideas were developed further in the language ML (Gordon et al., 1979), which featured a type system capable of expressing abstract syntax trees. The AUTOMATH project (Nederpelt et al., 1994) introduced the idea of using Church's $\lambda$ notation (Church, 1941) to account for the binding and scope of variables. These ideas were developed further in LF (Harper et al., 1993).

## Exercises

**1.1**. Prove by structural induction on abstract syntax trees that if $\mathcal{X} \subseteq \mathcal{Y}$, then $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{A}[\mathcal{Y}]$.

**1.2**. Prove by structural induction modulo renaming on abstract binding trees that if $\mathcal{X} \subseteq \mathcal{Y}$, then $\mathcal{B}[\mathcal{X}] \subseteq \mathcal{B}[\mathcal{Y}]$.

**1.3**. Bound variables may be seen as the formal analogues of pronouns in natural languages. The binding occurrence of a variable at an abstractor fixes a "fresh" pronoun for use within its body that refers unambiguously to that variable (in contrast to English, in which the referent



Preview

of a pronoun can often be ambiguous). This suggests an alternative representation of abt's, called *abstract binding graphs*, or *abg's* for short, as directed graphs constructed as follows:

(a) Free variables are atomic nodes with no outgoing edges.

(b) Operators with $n$ arguments are $n$-ary nodes, with one outgoing edge directed at each of their children.

(c) Abstractors are unary nodes with one edge directed to the scope of the abstracted variable.

(d) Bound variables are back edges directed at the abstractor that introduced it.

Notice that ast's, thought of as abt's with no abstractors, are *acyclic* directed graphs (more precisely, variadic trees), whereas general abt's may be *cyclic*. Draw a few examples of abg's corresponding to the example abt's given in this chapter. Give a precise definition of the sort-indexed family $\mathcal{G}[\mathcal{X}]$ of abstract binding graphs. What representation would you use for bound variables (back edges)?

Preview

Preview

# Chapter 2

# Inductive Definitions

Inductive definitions are an indispensable tool in the study of programming languages. In this chapter we will develop the basic framework of inductive definitions, and give some examples of their use. An inductive definition consists of a set of *rules* for deriving *judgments*, or *assertions*, of a variety of forms. Judgments are statements about one or more syntactic objects of a specified sort. The rules specify necessary and sufficient conditions for the validity of a judgment, and hence fully determine its meaning.

## 2.1 Judgments

We start with the notion of a *judgment*, or *assertion*, about a syntactic object. We shall make use of many forms of judgment, including examples such as these:

| | |
|---|---|
| $n$ nat | $n$ is a natural number |
| $n = n_1 + n_2$ | $n$ is the sum of $n_1$ and $n_2$ |
| $\tau$ type | $\tau$ is a type |
| $e : \tau$ | expression $e$ has type $\tau$ |
| $e \Downarrow v$ | expression $e$ has value $v$ |

A judgment states that one or more syntactic objects have a property or stand in some relation to one another. The property or relation itself is called a *judgment form*, and the judgment that an object or objects have that property or stand in that relation is said to be an *instance* of that judgment form. A judgment form is also called a *predicate*, and the objects constituting an instance are its *subjects*. We write $a$ J for the judgment asserting that J holds of $a$. When it is not important to stress the subject of the judgment, we write $J$ to stand for an unspecified judgment. For particular judgment forms, we freely use prefix, infix, or mixfix notation, as illustrated by the above examples, in order to enhance readability.

Preview

## 2.2 Inference Rules

An *inductive definition* of a judgment form consists of a collection of *rules* of the form

$$\frac{J_1 \quad \ldots \quad J_k}{J} \tag{2.1}$$

in which $J$ and $J_1, \ldots, J_k$ are all judgments of the form being defined. The judgments above the horizontal line are called the *premises* of the rule, and the judgment below the line is called its *conclusion*. If a rule has no premises (that is, when $k$ is zero), the rule is called an *axiom*; otherwise it is called a *proper rule*.

An inference rule may be read as stating that the premises are *sufficient* for the conclusion: to show $J$, it is enough to show $J_1, \ldots, J_k$. When $k$ is zero, a rule states that its conclusion holds unconditionally. Bear in mind that there may be, in general, many rules with the same conclusion, each specifying sufficient conditions for the conclusion. Consequently, if the conclusion of a rule holds, then it is not necessary that the premises hold, for it might have been derived by another rule.

For example, the following rules constitute an inductive definition of the judgment $a$ nat:

$$\frac{}{\texttt{zero nat}} \tag{2.2a}$$

$$\frac{a \text{ nat}}{\texttt{succ}(a) \text{ nat}} \tag{2.2b}$$

These rules specify that $a$ nat holds whenever either $a$ is zero, or $a$ is $\texttt{succ}(b)$ where $b$ nat for some $b$. Taking these rules to be exhaustive, it follows that $a$ nat iff $a$ is a natural number.

Similarly, the following rules constitute an inductive definition of the judgment $a$ tree:

$$\frac{}{\texttt{empty tree}} \tag{2.3a}$$

$$\frac{a_1 \text{ tree} \quad a_2 \text{ tree}}{\texttt{node}(a_1;a_2) \text{ tree}} \tag{2.3b}$$

These rules specify that $a$ tree holds if either $a$ is empty, or $a$ is $\texttt{node}(a_1;a_2)$, where $a_1$ tree and $a_2$ tree. Taking these to be exhaustive, these rules state that $a$ is a binary tree, which is to say it is either empty, or a node consisting of two children, each of which is also a binary tree.

The judgment $a = b$ nat defining equality of $a$ nat and $b$ nat is inductively defined by the following rules:

$$\frac{}{\texttt{zero} = \texttt{zero nat}} \tag{2.4a}$$

$$\frac{a = b \text{ nat}}{\texttt{succ}(a) = \texttt{succ}(b) \text{ nat}} \tag{2.4b}$$

In each of the preceding examples we have made use of a notational convention for specifying an infinite family of rules by a finite number of patterns, or *rule schemes*. For example, rule (2.2b)



Preview

is a rule scheme that determines one rule, called an *instance* of the rule scheme, for each choice of object $a$ in the rule. We will rely on context to determine whether a rule is stated for a *specific* object, $a$, or is instead intended as a rule scheme specifying a rule for *each choice* of objects in the rule.

A collection of rules is considered to define the *strongest* judgment that is *closed under*, or *respects*, those rules. To be closed under the rules simply means that the rules are *sufficient* to show the validity of a judgment: $J$ holds *if* there is a way to obtain it using the given rules. To be the *strongest* judgment closed under the rules means that the rules are also *necessary*: $J$ holds *only if* there is a way to obtain it by applying the rules. The sufficiency of the rules means that we may show that $J$ holds by *deriving* it by composing rules. Their necessity means that we may reason about it using *rule induction*.

## 2.3 Derivations

To show that an inductively defined judgment holds, it is enough to exhibit a *derivation* of it. A derivation of a judgment is a finite composition of rules, starting with axioms and ending with that judgment. It may be thought of as a tree in which each node is a rule whose children are derivations of its premises. We sometimes say that a derivation of $J$ is evidence for the validity of an inductively defined judgment $J$.

We usually depict derivations as trees with the conclusion at the bottom, and with the children of a node corresponding to a rule appearing above it as evidence for the premises of that rule. Thus, if

$$\frac{J_1 \quad \dots \quad J_k}{J}$$

is an inference rule and $\nabla_1, \dots, \nabla_k$ are derivations of its premises, then

$$\frac{\nabla_1 \quad \dots \quad \nabla_k}{J}$$

is a derivation of its conclusion. In particular, if $k = 0$, then the node has no children.

For example, this is a derivation of $\mathtt{succ}(\mathtt{succ}(\mathtt{succ}(\mathtt{zero})))\ \mathtt{nat}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{zero\ nat}}}{\mathtt{succ(zero)\ nat}}}{\mathtt{succ(succ(zero))\ nat}}}{\mathtt{succ(succ(succ(zero)))\ nat}}\,. \tag{2.5}$$

Similarly, here is a derivation of $\mathtt{node}(\mathtt{node}(\mathtt{empty};\mathtt{empty});\mathtt{empty})\ \mathtt{tree}$:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{empty\ tree}} \quad \dfrac{}{\mathtt{empty\ tree}}}{\mathtt{node(empty;empty)\ tree}} \quad \dfrac{}{\mathtt{empty\ tree}}}{\mathtt{node(node(empty;empty);empty)\ tree}}\,. \tag{2.6}$$



Preview

To show that an inductively defined judgment is derivable we need only find a derivation for it. There are two main methods for finding derivations, called *forward chaining*, or *bottom-up construction*, and *backward chaining*, or *top-down construction*. Forward chaining starts with the axioms and works forward towards the desired conclusion, whereas backward chaining starts with the desired conclusion and works backwards towards the axioms.

More precisely, forward chaining search maintains a set of derivable judgments, and continually extends this set by adding to it the conclusion of any rule all of whose premises are in that set. Initially, the set is empty; the process terminates when the desired judgment occurs in the set. Assuming that all rules are considered at every stage, forward chaining will eventually find a derivation of any derivable judgment, but it is impossible (in general) to decide algorithmically when to stop extending the set and conclude that the desired judgment is not derivable. We may go on and on adding more judgments to the derivable set without ever achieving the intended goal. It is a matter of understanding the global properties of the rules to determine that a given judgment is not derivable.

Forward chaining is undirected in the sense that it does not take account of the end goal when deciding how to proceed at each step. In contrast, backward chaining is goal-directed. Backward chaining search maintains a queue of current goals, judgments whose derivations are to be sought. Initially, this set consists solely of the judgment we wish to derive. At each stage, we remove a judgment from the queue, and consider all rules whose conclusion is that judgment. For each such rule, we add the premises of that rule to the back of the queue, and continue. If there is more than one such rule, this process must be repeated, with the same starting queue, for each candidate rule. The process terminates whenever the queue is empty, all goals having been achieved; any pending consideration of candidate rules along the way may be discarded. As with forward chaining, backward chaining will eventually find a derivation of any derivable judgment, but there is, in general, no algorithmic method for determining in general whether the current goal is derivable. If it is not, we may futilely add more and more judgments to the goal set, never reaching a point at which all goals have been satisfied.

## 2.4 Rule Induction

Because an inductive definition specifies the *strongest* judgment closed under a collection of rules, we may reason about them by *rule induction*. The principle of rule induction states that to show that a property $\mathcal{P}$ holds of a judgment $J$ whenever $J$ is derivable, it is enough to show that $\mathcal{P}$ is *closed under*, or *respects*, the rules defining $J$. Writing $\mathcal{P}(J)$ to mean that the property $\mathcal{P}$ holds of the judgment $J$, we say that $\mathcal{P}$ respects the rule

$$\frac{J_1 \quad \ldots \quad J_k}{J}$$

if $\mathcal{P}(J)$ holds whenever $\mathcal{P}(J_1), \ldots, \mathcal{P}(J_k)$. The assumptions $\mathcal{P}(J_1), \ldots, \mathcal{P}(J_k)$ are called the *inductive hypotheses*, and $\mathcal{P}(J)$ is called the *inductive conclusion* of the inference.

The principle of rule induction is simply the expression of the definition of an inductively defined judgment form as the *strongest* judgment form closed under the rules comprising the definition. This means that the judgment form defined by a set of rules is both (a) closed under those



Preview

rules, and (b) sufficient for any other property also closed under those rules. The former means that a derivation is evidence for the validity of a judgment; the latter means that we may reason about an inductively defined judgment form by rule induction.

When specialized to rules (2.2), the principle of rule induction states that to show $\mathcal{P}(a\ \mathsf{nat})$ whenever $a\ \mathsf{nat}$, it is enough to show:

1. $\mathcal{P}(\mathtt{zero}\ \mathsf{nat})$.
2. for every $a$, if ($a\ \mathsf{nat}$ and) $\mathcal{P}(a\ \mathsf{nat})$, then $\mathcal{P}(\mathtt{succ}(a)\ \mathsf{nat})$.

This is just the familiar principle of *mathematical induction* arising as a special case of rule induction.

Similarly, rule induction for rules (2.3) states that to show $\mathcal{P}(a\ \mathsf{tree})$ whenever $a\ \mathsf{tree}$, it is enough to show

1. $\mathcal{P}(\mathtt{empty}\ \mathsf{tree})$.
2. for every $a_1$ and $a_2$, if $a_1\ \mathsf{tree}$ and $\mathcal{P}(a_1\ \mathsf{tree})$, and if $a_2\ \mathsf{tree}$ and $\mathcal{P}(a_2\ \mathsf{tree})$, then ($\mathtt{node}(a_1;a_2)\ \mathsf{tree}$ and) $\mathcal{P}(\mathtt{node}(a_1;a_2)\ \mathsf{tree})$.

This is called the principle of *tree induction*, and is once again an instance of rule induction.

We may also show by rule induction that the predecessor of a natural number is also a natural number. Although this may seem self-evident, the point of the example is to show how to derive this from first principles.

**Lemma 2.1.** *If* $\mathtt{succ}(a)\ \mathsf{nat}$, *then* $a\ \mathsf{nat}$.

*Proof.* It suffices to show that the property, $\mathcal{P}(a\ \mathsf{nat})$ stating that $a\ \mathsf{nat}$ and that $a = \mathtt{succ}(b)$ implies $b\ \mathsf{nat}$ is closed under rules (2.2).

**Rule** (2.2a) Clearly $\mathtt{zero}\ \mathsf{nat}$, and the second condition holds vacuously, because $\mathtt{zero}$ is not of the form $\mathtt{succ}(-)$.

**Rule** (2.2b) Inductively we know that $a\ \mathsf{nat}$ and that if $a$ is of the form $\mathtt{succ}(b)$, then $b\ \mathsf{nat}$. We are to show that $\mathtt{succ}(a)\ \mathsf{nat}$, which is immediate, and that if $\mathtt{succ}(a)$ is of the form $\mathtt{succ}(b)$, then $b\ \mathsf{nat}$, and we have $b\ \mathsf{nat}$ by the inductive hypothesis.

This completes the proof. □

Using rule induction we may show that equality, as defined by rules (2.4) is reflexive.

**Lemma 2.2.** *If* $a\ \mathsf{nat}$, *then* $a = a\ \mathsf{nat}$.

*Proof.* By rule induction on rules (2.2):

**Rule** (2.2a) Applying rule (2.4a) we obtain $\mathtt{zero} = \mathtt{zero}\ \mathsf{nat}$.

**Rule** (2.2b) Assume that $a = a\ \mathsf{nat}$. It follows that $\mathtt{succ}(a) = \mathtt{succ}(a)\ \mathsf{nat}$ by an application of rule (2.4b).

□



Preview

Similarly, we may show that the successor operation is injective.

**Lemma 2.3.** *If* $\mathtt{succ}(a_1) = \mathtt{succ}(a_2)$ nat, *then* $a_1 = a_2$ nat.

*Proof.* Similar to the proof of Lemma 2.1. □

## 2.5 Iterated and Simultaneous Inductive Definitions

Inductive definitions are often *iterated*, meaning that one inductive definition builds on top of another. In an iterated inductive definition the premises of a rule

$$\frac{J_1 \quad \ldots \quad J_k}{J}$$

may be instances of either a previously defined judgment form, or the judgment form being defined. For example, the following rules define the judgment $a$ list stating that $a$ is a list of natural numbers.

$$\frac{}{\mathtt{nil}\ \mathsf{list}} \tag{2.7a}$$

$$\frac{a\ \mathsf{nat} \quad b\ \mathsf{list}}{\mathtt{cons}(a;b)\ \mathsf{list}} \tag{2.7b}$$

The first premise of rule (2.7b) is an instance of the judgment form $a$ nat, which was defined previously, whereas the premise $b$ list is an instance of the judgment form being defined by these rules.

Frequently two or more judgments are defined at once by a *simultaneous inductive definition*. A simultaneous inductive definition consists of a set of rules for deriving instances of several different judgment forms, any of which may appear as the premise of any rule. Because the rules defining each judgment form may involve any of the others, none of the judgment forms may be taken to be defined prior to the others. Instead we must understand that all of the judgment forms are being defined at once by the entire collection of rules. The judgment forms defined by these rules are, as before, the strongest judgment forms that are closed under the rules. Therefore the principle of proof by rule induction continues to apply, albeit in a form that requires us to prove a property of each of the defined judgment forms simultaneously.

For example, consider the following rules, which constitute a simultaneous inductive definition of the judgments $a$ even, stating that $a$ is an even natural number, and $a$ odd, stating that $a$ is an odd natural number:

$$\frac{}{\mathtt{zero}\ \mathsf{even}} \tag{2.8a}$$

$$\frac{a\ \mathsf{odd}}{\mathtt{succ}(a)\ \mathsf{even}} \tag{2.8b}$$

$$\frac{a\ \mathsf{even}}{\mathtt{succ}(a)\ \mathsf{odd}} \tag{2.8c}$$

The principle of rule induction for these rules states that to show simultaneously that $\mathcal{P}(a\ \mathsf{even})$ whenever $a$ even and $\mathcal{P}(a\ \mathsf{odd})$ whenever $a$ odd, it is enough to show the following:



Preview

1. $\mathcal{P}(\mathtt{zero}\ \mathsf{even})$;
2. if $\mathcal{P}(a\ \mathsf{odd})$, then $\mathcal{P}(\mathtt{succ}(a)\ \mathsf{even})$;
3. if $\mathcal{P}(a\ \mathsf{even})$, then $\mathcal{P}(\mathtt{succ}(a)\ \mathsf{odd})$.

As a simple example, we may use simultaneous rule induction to prove that (1) if $a$ even, then $a$ nat, and (2) if $a$ odd, then $a$ nat. That is, we define the property $\mathcal{P}$ by (1) $\mathcal{P}(a\ \mathsf{even})$ iff $a$ nat, and (2) $\mathcal{P}(a\ \mathsf{odd})$ iff $a$ nat. The principle of rule induction for rules (2.8) states that it is sufficient to show the following facts:

1. $\mathtt{zero}$ nat, which is derivable by rule (2.2a).
2. If $a$ nat, then $\mathtt{succ}(a)$ nat, which is derivable by rule (2.2b).
3. If $a$ nat, then $\mathtt{succ}(a)$ nat, which is also derivable by rule (2.2b).

## 2.6 Defining Functions by Rules

A common use of inductive definitions is to define a function by giving an inductive definition of its *graph* relating inputs to outputs, and then showing that the relation uniquely determines the outputs for given inputs. For example, we may define the addition function on natural numbers as the relation $\mathsf{sum}(a;b;c)$, with the intended meaning that $c$ is the sum of $a$ and $b$, as follows:

$$\frac{b\ \mathsf{nat}}{\mathsf{sum}(\mathtt{zero};b;b)} \tag{2.9a}$$

$$\frac{\mathsf{sum}(a;b;c)}{\mathsf{sum}(\mathtt{succ}(a);b;\mathtt{succ}(c))} \tag{2.9b}$$

The rules define a ternary (three-place) relation, $\mathsf{sum}(a;b;c)$, among natural numbers $a$, $b$, and $c$. We may show that $c$ is determined by $a$ and $b$ in this relation.

**Theorem 2.4.** *For every $a$ nat and $b$ nat, there exists a unique $c$ nat such that $\mathsf{sum}(a;b;c)$.*

*Proof.* The proof decomposes into two parts:

1. (Existence) If $a$ nat and $b$ nat, then there exists $c$ nat such that $\mathsf{sum}(a;b;c)$.
2. (Uniqueness) If $\mathsf{sum}(a;b;c)$, and $\mathsf{sum}(a;b;c')$, then $c = c'$ nat.

For existence, let $\mathcal{P}(a\ \mathsf{nat})$ be the proposition *if $b$ nat then there exists $c$ nat such that $\mathsf{sum}(a;b;c)$*. We prove that if $a$ nat then $\mathcal{P}(a\ \mathsf{nat})$ by rule induction on rules (2.2). We have two cases to consider:

**Rule** (2.2a) We are to show $\mathcal{P}(\mathtt{zero}\ \mathsf{nat})$. Assuming $b$ nat and taking $c$ to be $b$, we obtain $\mathsf{sum}(\mathtt{zero};b;c)$ by rule (2.9a).



Preview

**Rule** (2.2b) Assuming $\mathcal{P}(a\ \mathsf{nat})$, we are to show $\mathcal{P}(\mathtt{succ}(a)\ \mathsf{nat})$. That is, we assume that if $b$ nat then there exists $c$ such that $\mathsf{sum}(a;b;c)$, and are to show that if $b'$ nat, then there exists $c'$ such that $\mathsf{sum}(\mathtt{succ}(a);b';c')$. To this end, suppose that $b'$ nat. Then by induction there exists $c$ such that $\mathsf{sum}(a;b';c)$. Taking $c' = \mathtt{succ}(c)$, and applying rule (2.9b), we obtain $\mathsf{sum}(\mathtt{succ}(a);b';c')$, as required.

For uniqueness, we prove that *if* $\mathsf{sum}(a;b;c_1)$, *then if* $\mathsf{sum}(a;b;c_2)$, *then* $c_1 = c_2$ nat by rule induction based on rules (2.9).

**Rule** (2.9a) We have $a = \mathtt{zero}$ and $c_1 = b$. By an inner induction on the same rules, we may show that if $\mathsf{sum}(\mathtt{zero};b;c_2)$, then $c_2$ is $b$. By Lemma 2.2 we obtain $b = b$ nat.

**Rule** (2.9b) We have that $a = \mathtt{succ}(a')$ and $c_1 = \mathtt{succ}(c_1')$, where $\mathsf{sum}(a';b;c_1')$. By an inner induction on the same rules, we may show that if $\mathsf{sum}(a;b;c_2)$, then $c_2 = \mathtt{succ}(c_2')$ nat where $\mathsf{sum}(a';b;c_2')$. By the outer inductive hypothesis $c_1' = c_2'$ nat and so $c_1 = c_2$ nat.

□

## 2.7 Modes

The statement that one or more arguments of a judgment is (perhaps uniquely) determined by its other arguments is called a *mode specification* for that judgment. For example, we have shown that every two natural numbers have a sum according to rules (2.9). This fact may be restated as a mode specification by saying that the judgment $\mathsf{sum}(a;b;c)$ has *mode* $(\forall, \forall, \exists)$. The notation arises from the form of the proposition it expresses: *for all* $a$ nat *and for all* $b$ nat, *there exists* $c$ nat *such that* $\mathsf{sum}(a;b;c)$. If we wish to further specify that $c$ is *uniquely* determined by $a$ and $b$, we would say that the judgment $\mathsf{sum}(a;b;c)$ has mode $(\forall, \forall, \exists!)$, corresponding to the proposition *for all* $a$ nat *and for all* $b$ nat, *there exists a unique* $c$ nat *such that* $\mathsf{sum}(a;b;c)$. If we wish only to specify that the sum is unique, *if it exists*, then we would say that the addition judgment has mode $(\forall, \forall, \exists^{\leq 1})$, corresponding to the proposition *for all* $a$ nat *and for all* $b$ nat *there exists at most one* $c$ nat *such that* $\mathsf{sum}(a;b;c)$.

As these examples illustrate, a given judgment may satisfy several different mode specifications. In general the universally quantified arguments are to be thought of as the *inputs* of the judgment, and the existentially quantified arguments are to be thought of as its *outputs*. We usually try to arrange things so that the outputs come after the inputs, but it is not essential that we do so. For example, addition also has the mode $(\forall, \exists^{\leq 1}, \forall)$, stating that the sum and the first addend uniquely determine the second addend, if there is any such addend at all. Put in other terms, this says that addition of natural numbers has a (partial) inverse, namely subtraction. We could equally well show that addition has mode $(\exists^{\leq 1}, \forall, \forall)$, which is just another way of stating that addition of natural numbers has a partial inverse.

Often there is an intended, or *principal*, mode of a given judgment, which we often foreshadow by our choice of notation. For example, when giving an inductive definition of a function, we often use equations to indicate the intended input and output relationships. For example, we may re-state the inductive definition of addition (given by rules (2.9)) using equations:

$$\frac{a\ \mathsf{nat}}{a + \mathtt{zero} = a\ \mathsf{nat}} \tag{2.10a}$$



Preview

$$\frac{a + b = c \ \mathsf{nat}}{a + \mathtt{succ}(b) = \mathtt{succ}(c) \ \mathsf{nat}} \tag{2.10b}$$

When using this notation we tacitly incur the obligation to prove that the mode of the judgment is such that the object on the right-hand side of the equations is determined as a function of those on the left. Having done so, we abuse notation, writing $a + b$ for the unique $c$ such that $a + b = c$ nat.

## 2.8 Notes

Aczel (1977) provides a thorough account of the theory of inductive definitions. The formulation given here is strongly influenced by Martin-Löf's development of the logic of judgments (Martin-Löf, 1983, 1987).

## Exercises

**2.1**. Give an inductive definition of the judgment $\mathsf{max}(m;n;p)$, where $m$ nat, $n$ nat, and $p$ nat, with the meaning that $p$ is the larger of $m$ and $n$. Prove that this judgment has the mode $(\forall, \forall, \exists!)$.

**2.2**. Consider the following rules, which define the judgment $\mathsf{hgt}(t;n)$ stating that the binary tree $t$ has *height* $n$.

$$\frac{}{\mathsf{hgt}(\mathtt{empty};\mathtt{zero})} \tag{2.11a}$$

$$\frac{\mathsf{hgt}(t_1;n_1) \quad \mathsf{hgt}(t_2;n_2) \quad \mathsf{max}(n_1;n_2;n)}{\mathsf{hgt}(\mathtt{node}(t_1;t_2);\mathtt{succ}(n))} \tag{2.11b}$$

Prove that the judgment $\mathtt{hgt}$ has the mode $(\forall, \exists!)$, with inputs being binary trees and outputs being natural numbers.

**2.3**. Given an inductive definition of *ordered variadic trees* whose nodes have a finite, but variable, number of children with a specified left-to-right ordering among them. Your solution should consist of a simultaneous definition of two judgments, $t$ tree, stating that $t$ is a variadic tree, and $f$ forest, stating that $f$ is a "forest" (finite sequence) of variadic trees.

**2.4**. Give an inductive definition of the height of a variadic tree of the kind defined in Exercise **2.3**. Your definition should make use of an auxiliary judgment defining the height of a forest of variadic trees, and will be defined simultaneously with the height of a variadic tree. Show that the two judgments so defined have mode $(\forall, \exists!)$, which is to say that they each define a function.

**2.5**. Give an inductive definition of the *binary natural numbers*, which are either zero, twice a binary number, or one more than twice a binary number. The size of such a representation is logarithmic, rather than linear, in the natural number it represents.

Preview

**2.6**. Give an inductive definition of addition of binary natural numbers as defined in Exercise **2.5**. *Hint*: Proceed by analyzing both arguments to the addition, and make use of an auxiliary function to compute the sucessor of a binary number. *Hint*: Alternatively, define both the sum and the sum-plus-one of two binary numbers mutually recursively.

Preview

# Chapter 3

# Hypothetical and General Judgments

A *hypothetical judgment* expresses an entailment between one or more hypotheses and a conclusion. We will consider two notions of entailment, called *derivability* and *admissibility*. Both express a form of entailment, but they differ in that derivability is stable under extension with new rules, admissibility is not. A *general judgment* expresses the universality, or genericity, of a judgment. There are two forms of general judgment, the *generic* and the *parametric*. The generic judgment expresses generality with respect to all substitution instances for variables in a judgment. The parametric judgment expresses generality with respect to renamings of symbols.

## 3.1 Hypothetical Judgments

The hypothetical judgment codifies the rules for expressing the validity of a conclusion conditional on the validity of one or more hypotheses. There are two forms of hypothetical judgment that differ according to the sense in which the conclusion is conditional on the hypotheses. One is stable under extension with additional rules, and the other is not.

### 3.1.1 Derivability

For a given set, $\mathcal{R}$, of rules, we define the *derivability* judgment, written $J_1, \ldots, J_k \vdash_{\mathcal{R}} K$, where each $J_i$ and $K$ are basic judgments, to mean that we may derive $K$ from the *expansion* $\mathcal{R} \cup \{ J_1, \ldots, J_k \}$ of the rules $\mathcal{R}$ with the additional axioms

$$\frac{}{J_1} \qquad \cdots \qquad \frac{}{J_k}.$$

We treat the *hypotheses*, or *antecedents*, of the judgment, $J_1, \ldots, J_k$ as "temporary axioms", and derive the *conclusion*, or *consequent*, by composing rules in $\mathcal{R}$. Thus, evidence for a hypothetical judgment consists of a derivation of the conclusion from the hypotheses using the rules in $\mathcal{R}$.

We use capital Greek letters, frequently $\Gamma$ or $\Delta$, to stand for a finite collection of basic judgments, and write $\mathcal{R} \cup \Gamma$ for the expansion of $\mathcal{R}$ with an axiom corresponding to each judgment in $\Gamma$. The

Preview

judgment $\Gamma \vdash_{\mathcal{R}} K$ means that $K$ is derivable from rules $\mathcal{R} \cup \Gamma$, and the judgment $\vdash_{\mathcal{R}} \Gamma$ means that $\vdash_{\mathcal{R}} J$ for each $J$ in $\Gamma$. An equivalent way of defining $J_1, \ldots, J_n \vdash_{\mathcal{R}} J$ is to say that the rule

$$\frac{J_1 \quad \ldots \quad J_n}{J} \tag{3.1}$$

is *derivable* from $\mathcal{R}$, which means that there is a derivation of $J$ composed of the rules in $\mathcal{R}$ augmented by treating $J_1, \ldots, J_n$ as axioms.

For example, consider the derivability judgment

$$a\ \mathsf{nat} \vdash_{(2.2)} \mathtt{succ}(\mathtt{succ}(a))\ \mathsf{nat} \tag{3.2}$$

relative to rules (2.2). This judgment is valid for *any* choice of object $a$, as evidenced by the derivation

$$\frac{\dfrac{a\ \mathsf{nat}}{\mathtt{succ}(a)\ \mathsf{nat}}}{\mathtt{succ}(\mathtt{succ}(a))\ \mathsf{nat}} \tag{3.3}$$

which composes rules (2.2), starting with $a$ nat as an axiom, and ending with $\mathtt{succ}(\mathtt{succ}(a))$ nat. Equivalently, the validity of (3.2) may also be expressed by stating that the rule

$$\frac{a\ \mathsf{nat}}{\mathtt{succ}(\mathtt{succ}(a))\ \mathsf{nat}} \tag{3.4}$$

is derivable from rules (2.2).

It follows directly from the definition of derivability that it is stable under extension with new rules.

**Theorem 3.1** (Stability)**.** *If* $\Gamma \vdash_{\mathcal{R}} J$*, then* $\Gamma \vdash_{\mathcal{R} \cup \mathcal{R}'} J$*.*

*Proof.* Any derivation of $J$ from $\mathcal{R} \cup \Gamma$ is also a derivation from $(\mathcal{R} \cup \mathcal{R}') \cup \Gamma$, because any rule in $\mathcal{R}$ is also a rule in $\mathcal{R} \cup \mathcal{R}'$. □

Derivability enjoys a number of *structural properties* that follow from its definition, independently of the rules, $\mathcal{R}$, in question.

**Reflexivity** Every judgment is a consequence of itself: $\Gamma, J \vdash_{\mathcal{R}} J$. Each hypothesis justifies itself as conclusion.

**Weakening** If $\Gamma \vdash_{\mathcal{R}} J$, then $\Gamma, K \vdash_{\mathcal{R}} J$. Entailment is not influenced by unexercised options.

**Transitivity** If $\Gamma, K \vdash_{\mathcal{R}} J$ and $\Gamma \vdash_{\mathcal{R}} K$, then $\Gamma \vdash_{\mathcal{R}} J$. If we replace an axiom by a derivation of it, the result is a derivation of its consequent without that hypothesis.

Reflexivity follows directly from the meaning of derivability. Weakening follows directly from the definition of derivability. Transitivity is proved by rule induction on the first premise.

Preview

### 3.1.2 Admissibility

*Admissibility*, written $\Gamma \models_{\mathcal{R}} J$, is a weaker form of hypothetical judgment stating that $\vdash_{\mathcal{R}} \Gamma$ implies $\vdash_{\mathcal{R}} J$. That is, the conclusion $J$ is derivable from rules $\mathcal{R}$ whenever the assumptions $\Gamma$ are all derivable from rules $\mathcal{R}$. In particular if any of the hypotheses are *not* derivable relative to $\mathcal{R}$, then the judgment is vacuously true. An equivalent way to define the judgment $J_1, \ldots, J_n \models_{\mathcal{R}} J$ is to state that the rule

$$\frac{J_1 \quad \ldots \quad J_n}{J} \tag{3.5}$$

is *admissible* relative to the rules in $\mathcal{R}$. This means that given any derivations of $J_1, \ldots, J_n$ using the rules in $\mathcal{R}$, we may construct a derivation of $J$ using the rules in $\mathcal{R}$.

For example, each instance of the admissibility judgment

$$\mathtt{succ}(a)\ \mathsf{nat} \models_{(2.2)} a\ \mathsf{nat} \tag{3.6}$$

obtained by replacing $a$ with some syntactic object is valid, because any derivation of $\mathtt{succ}(a)\ \mathsf{nat}$ from rules (2.2) must contain a sub-derivation of $a\ \mathsf{nat}$ from the same rules, which justifies the conclusion. The validity of (3.6) may equivalently be expressed by stating that each instance of the rule

$$\frac{\mathtt{succ}(a)\ \mathsf{nat}}{a\ \mathsf{nat}} \tag{3.7}$$

obtained by replacing $a$ with some syntactic object is admissible for rules (2.2).

In contrast to derivability the admissibility judgment is *not* stable under extension to the rules. For example, if we enrich rules (2.2) with the axiom

$$\frac{}{\mathtt{succ}(\mathtt{junk})\ \mathsf{nat}} \tag{3.8}$$

(where `junk` is some object for which `junk nat` is not derivable), then the admissibility (3.6) is *invalid*. This is because rule (3.8) has no premises, and there is no composition of rules deriving `junk nat`. Admissibility is as sensitive to which rules are *absent* from an inductive definition as it is to which rules are *present* in it.

The structural properties of derivability ensure that derivability is stronger than admissibility.

**Theorem 3.2.** *If $\Gamma \vdash_{\mathcal{R}} J$, then $\Gamma \models_{\mathcal{R}} J$.*

*Proof.* Repeated application of the transitivity of derivability shows that if $\Gamma \vdash_{\mathcal{R}} J$ and $\vdash_{\mathcal{R}} \Gamma$, then $\vdash_{\mathcal{R}} J$. □

To see that the converse fails, observe that there is no composition of rules such that

$$\mathtt{succ}(\mathtt{junk})\ \mathsf{nat} \vdash_{(2.2)} \mathtt{junk}\ \mathsf{nat},$$

yet the admissibility judgment

$$\mathtt{succ}(\mathtt{junk})\ \mathsf{nat} \models_{(2.2)} \mathtt{junk}\ \mathsf{nat}$$



Preview

holds vacuously.

Evidence for admissibility may be thought of as a mathematical function transforming derivations $\nabla_1, \ldots, \nabla_n$ of the hypotheses into a derivation $\nabla$ of the consequent. Therefore, the admissibility judgment enjoys the same structural properties as derivability, and hence is a form of hypothetical judgment:

**Reflexivity** If $J$ is derivable from the original rules, then $J$ is derivable from the original rules: $J \models_{\mathcal{R}} J$.

**Weakening** If $J$ is derivable from the original rules assuming that each of the judgments in $\Gamma$ are derivable from these rules, then $J$ must also be derivable assuming that $\Gamma$ and also $K$ are derivable from the original rules: if $\Gamma \models_{\mathcal{R}} J$, then $\Gamma, K \models_{\mathcal{R}} J$.

**Transitivity** If $\Gamma, K \models_{\mathcal{R}} J$ and $\Gamma \models_{\mathcal{R}} K$, then $\Gamma \models_{\mathcal{R}} J$. If the judgments in $\Gamma$ are derivable, so is $K$, by assumption, and hence so are the judgments in $\Gamma, K$, and hence so is $J$.

**Theorem 3.3.** *The admissibility judgment $\Gamma \models_{\mathcal{R}} J$ enjoys the structural properties of entailment.*

*Proof.* Follows immediately from the definition of admissibility as stating that if the hypotheses are derivable relative to $\mathcal{R}$, then so is the conclusion. □

If a rule, $r$, is admissible with respect to a rule set, $\mathcal{R}$, then $\vdash_{\mathcal{R},r} J$ is equivalent to $\vdash_{\mathcal{R}} J$. For if $\vdash_{\mathcal{R}} J$, then obviously $\vdash_{\mathcal{R},r} J$, by simply disregarding $r$. Conversely, if $\vdash_{\mathcal{R},r} J$, then we may replace any use of $r$ by its expansion in terms of the rules in $\mathcal{R}$. It follows by rule induction on $\mathcal{R}, r$ that every derivation from the expanded set of rules, $\mathcal{R}, r$, may be transformed into a derivation from $\mathcal{R}$ alone. Consequently, if we wish to show that $\mathcal{P}(J)$ whenever $\vdash_{\mathcal{R},r} J$, it is sufficient to show that $\mathcal{P}$ is closed under the rules $\mathcal{R}$ alone. That is, we need only consider the rules $\mathcal{R}$ in a proof by rule induction to derive $\mathcal{P}(J)$.

## 3.2 Hypothetical Inductive Definitions

It is useful to enrich the concept of an inductive definition to permit rules with derivability judgments as premises and conclusions. Doing so permits us to introduce *local hypotheses* that apply only in the derivation of a particular premise, and also allows us to constrain inferences based on the *global hypotheses* in effect at the point where the rule is applied.

A *hypothetical inductive definition* consists of a collection of *hypothetical rules* of the following form:

$$\frac{\Gamma\,\Gamma_1 \vdash J_1 \quad \ldots \quad \Gamma\,\Gamma_n \vdash J_n}{\Gamma \vdash J}\ . \tag{3.9}$$

The hypotheses $\Gamma$ are the *global hypotheses* of the rule, and the hypotheses $\Gamma_i$ are the *local hypotheses* of the $i$th premise of the rule. Informally, this rule states that $J$ is a derivable consequence of $\Gamma$ whenever each $J_i$ is a derivable consequence of $\Gamma$, augmented with the additional hypotheses $\Gamma_i$. Thus, one way to show that $J$ is derivable from $\Gamma$ is to show, in turn, that each $J_i$ is derivable from $\Gamma\,\Gamma_i$. The derivation of each premise involves a "context switch" in which we extend the global



Preview

hypotheses with the local hypotheses of that premise, establishing a new set of global hypotheses for use within that derivation.

In most cases a rule is stated for *all* choices of global context, in which case it is said to be *uniform*. A uniform rule may be given in the *implicit* form

$$\frac{\Gamma_1 \vdash J_1 \quad \ldots \quad \Gamma_n \vdash J_n}{J} \tag{3.10}$$

which stands for the collection of all rules of the form (3.9) in which the global hypotheses have been made explicit.

A hypothetical inductive definition is to be regarded as an ordinary inductive definition of a *formal derivability judgment* $\Gamma \vdash J$ consisting of a finite set of basic judgments, $\Gamma$, and a basic judgment, $J$. A collection of hypothetical rules, $\mathcal{R}$, defines the strongest formal derivability judgment that is *structural* and *closed* under rules $\mathcal{R}$. Structurality means that the formal derivability judgment must be closed under the following rules:

$$\frac{}{\Gamma, J \vdash J} \tag{3.11a}$$

$$\frac{\Gamma \vdash J}{\Gamma, K \vdash J} \tag{3.11b}$$

$$\frac{\Gamma \vdash K \quad \Gamma, K \vdash J}{\Gamma \vdash J} \tag{3.11c}$$

These rules ensure that formal derivability behaves like a hypothetical judgment. By a slight abuse of notation we write $\Gamma \vdash_{\mathcal{R}} J$ to indicate that $\Gamma \vdash J$ is derivable from rules $\mathcal{R}$.

The principle of *hypothetical rule induction* is just the principle of rule induction applied to the formal hypothetical judgment. So to show that $\mathcal{P}(\Gamma \vdash J)$ whenever $\Gamma \vdash_{\mathcal{R}} J$, it is enough to show that $\mathcal{P}$ is closed under both the rules of $\mathcal{R}$ and under the structural rules. Thus, for each rule of the form (3.9), whether structural or in $\mathcal{R}$, we must show that

$$\text{if } \mathcal{P}(\Gamma\, \Gamma_1 \vdash J_1) \text{ and } \ldots \text{ and } \mathcal{P}(\Gamma\, \Gamma_n \vdash J_n), \text{ then } \mathcal{P}(\Gamma \vdash J).$$

This is just a restatement of the principle of rule induction given in Chapter 2, specialized to the formal derivability judgment $\Gamma \vdash J$.

In practice we usually dispense with the structural rules by the method described in Section 3.1.2. By proving that the structural rules are admissible any proof by rule induction may restrict attention to the rules in $\mathcal{R}$ alone. If all of the rules of a hypothetical inductive definition are uniform, the structural rules (3.11b) and (3.11c) are readily seen to be admissible. Usually, rule (3.11a) must be postulated explictly as a rule, rather than shown to be admissible on the basis of the other rules.

## 3.3 General Judgments

General judgments codify the rules for handling variables in a judgment. As in mathematics in general, a variable is treated as an *unknown* ranging over a specified collection of objects. A *generic*



Preview

judgment expresses that a judgment holds for any choice of objects replacing designated variables in the judgment. Another form of general judgment codifies the handling of parameters. A *parametric* judgment expresses generality over any choice of fresh renamings of designated parameters of a judgment. To keep track of the active variables and parameters in a derivation, we write $\Gamma \vdash^{\mathcal{U};\mathcal{X}}_{\mathcal{R}} J$ to indicate that $J$ is derivable from $\Gamma$ according to rules $\mathcal{R}$, with objects consisting of abt's over parameters $\mathcal{U}$ and variables $\mathcal{X}$.

*Generic derivability* judgment is defined by

$$\vec{x} \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J \quad \text{iff} \quad \forall \pi : \vec{x} \leftrightarrow \vec{x}'\ \pi \cdot \Gamma \vdash^{\mathcal{X},\vec{x}'}_{\mathcal{R}} \pi \cdot J,$$

where the quantification is restricted to variables $\vec{x}'$ not already active in $\mathcal{X}$. Evidence for generic derivability consists of a *generic derivation*, $\nabla$, involving the variables $\vec{x}$ such that for every fresh renaming $\pi : \vec{x} \leftrightarrow \vec{x}'$, the derivation $\pi \cdot \nabla$ is evidence for $\pi \cdot \Gamma \vdash^{\mathcal{X},\vec{x}'}_{\mathcal{R}} \pi \cdot J$. The renaming ensures that the variables in the generic judgment are fresh (not already declared in $\mathcal{X}$), and that the meaning of the judgment is not dependent on the choice of variable names.

For example, the generic derivation, $\nabla$,

$$\dfrac{\dfrac{\overline{x\ \mathsf{nat}}}{\mathtt{succ}(x)\ \mathsf{nat}}}{\mathtt{succ}(\mathtt{succ}(x))\ \mathsf{nat}}$$

is evidence for the judgment,

$$x \mid x\ \mathsf{nat} \vdash^{\mathcal{X}}_{(2.2)} \mathtt{succ}(\mathtt{succ}(x))\ \mathsf{nat}.$$

This is because every fresh renaming of $x$ to $y$ in $\nabla$ results in a valid derivation of the corresponding renaming of the indicated judgment.

The generic derivability judgment enjoys the following *structural properties* governing the behavior of variables:

**Proliferation** If $\vec{x} \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$, then $\vec{x}, x \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$.

**Renaming** If $\vec{x}, x \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$, then $\vec{x}, x' \mid [x \leftrightarrow x'] \cdot \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} [x \leftrightarrow x'] \cdot J$ for any $x' \notin \mathcal{X}, \vec{x}$.

**Substitution** If $\vec{x}, x \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$ and $a \in \mathcal{B}[\mathcal{X}, \vec{x}]$, then $\vec{x} \mid [a/x]\Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} [a/x]J$.

(It is left implicit in the principle of substitution that sorts are to be respected in that the substituting object must be of the same sort as the variable which is being substituted.) Proliferation is guaranteed by the interpretation of rule schemes as ranging over all expansions of the universe. Renaming is built into the meaning of the generic judgment. Substitution holds as long as the rules themselves are closed under substitution. This need not be the case, but in practice this requirement is usually met.

*Parametric* derivability is defined analogously to generic derivability, albeit by generalizing over parameters, rather than variables. Parametric derivability is defined by

$$\vec{u}; \vec{x} \mid \Gamma \vdash^{\mathcal{U};\mathcal{X}}_{\mathcal{R}} J \quad \text{iff} \quad \forall \rho : \vec{u} \leftrightarrow \vec{u}'\ \forall \pi : \vec{x} \leftrightarrow \vec{x}'\ \rho \cdot \pi \cdot \Gamma \vdash^{\mathcal{U},\vec{u}';\mathcal{X},\vec{x}'}_{\mathcal{R}} \rho \cdot \pi \cdot J.$$



Preview

Evidence for parametric derivability consists of a derivation $\nabla$ involving the parameters $\vec{u}$ and variables $\vec{x}$ each of whose fresh renamings is a derivation of the corresponding renaming of the underlying hypothetical judgment.

Recalling from Chapter 1 that parameters admit disequality, we cannot expect any substitution principle for parameters to hold of a parametric derivability. It does, however, validate the structural properties of proliferation and renaming, because the presence of additional parameters does not affect the formation of an abt, and parametric derivability is defined to respect all fresh renamings of parameters.

## 3.4 Generic Inductive Definitions

A *generic inductive definition* admits generic hypothetical judgments in the premises of rules, with the effect of augmenting the variables, as well as the rules, within those premises. A *generic rule* has the form

$$\frac{\vec{x}\,\vec{x}_1 \mid \Gamma\,\Gamma_1 \vdash J_1 \quad \ldots \quad \vec{x}\,\vec{x}_n \mid \Gamma\,\Gamma_n \vdash J_n}{\vec{x} \mid \Gamma \vdash J}. \tag{3.12}$$

The variables $\vec{x}$ are the *global variables* of the inference, and, for each $1 \leq i \leq n$, the variables $\vec{x}_i$ are the *local variables* of the $i$th premise. In most cases a rule is stated for *all* choices of global variables and global hypotheses. Such rules may be given in *implicit form*,

$$\frac{\vec{x}_1 \mid \Gamma_1 \vdash J_1 \quad \ldots \quad \vec{x}_n \mid \Gamma_n \vdash J_n}{J}. \tag{3.13}$$

A generic inductive definition is just an ordinary inductive definition of a family of *formal generic judgments* of the form $\vec{x} \mid \Gamma \vdash J$. Formal generic judgments are identified up to renaming of variables, so that the latter judgment is treated as identical to the judgment $\vec{x}' \mid \pi \cdot \Gamma \vdash \pi \cdot J$ for any renaming $\pi : \vec{x} \leftrightarrow \vec{x}'$. If $\mathcal{R}$ is a collection of generic rules, we write $\vec{x} \mid \Gamma \vdash_{\mathcal{R}} J$ to mean that the formal generic judgment $\vec{x} \mid \Gamma \vdash J$ is derivable from rules $\mathcal{R}$.

When specialized to a collection of generic rules, the principle of rule induction states that to show $\mathcal{P}(\vec{x} \mid \Gamma \vdash J)$ whenever $\vec{x} \mid \Gamma \vdash_{\mathcal{R}} J$, it is enough to show that $\mathcal{P}$ is closed under the rules $\mathcal{R}$. Specifically, for each rule in $\mathcal{R}$ of the form (3.12), we must show that

$$\text{if } \mathcal{P}(\vec{x}\,\vec{x}_1 \mid \Gamma\,\Gamma_1 \vdash J_1) \ \ldots\ \mathcal{P}(\vec{x}\,\vec{x}_n \mid \Gamma\,\Gamma_n \vdash J_n) \text{ then } \mathcal{P}(\vec{x} \mid \Gamma \vdash J).$$

By the identification convention (stated in Chapter 1) the property $\mathcal{P}$ must respect renamings of the variables in a formal generic judgment.

To ensure that the formal generic judgment behaves like a generic judgment, we must always ensure that the following *structural rules* are admissible:

$$\frac{}{\vec{x} \mid \Gamma, J \vdash J} \tag{3.14a}$$

$$\frac{\vec{x} \mid \Gamma \vdash J}{\vec{x} \mid \Gamma, J' \vdash J} \tag{3.14b}$$



Preview

$$\frac{\vec{x} \mid \Gamma \vdash J}{\vec{x}, x \mid \Gamma \vdash J} \tag{3.14c}$$

$$\frac{\vec{x}, x' \mid [x \leftrightarrow x'] \cdot \Gamma \vdash [x \leftrightarrow x'] \cdot J}{\vec{x}, x \mid \Gamma \vdash J} \tag{3.14d}$$

$$\frac{\vec{x} \mid \Gamma \vdash J \quad \vec{x} \mid \Gamma, J \vdash J'}{\vec{x} \mid \Gamma \vdash J'} \tag{3.14e}$$

$$\frac{\vec{x}, x \mid \Gamma \vdash J \quad a \in \mathcal{B}[\vec{x}]}{\vec{x} \mid [a/x]\Gamma \vdash [a/x]J} \tag{3.14f}$$

The admissibility of rule (3.14a) is, in practice, ensured by explicitly including it. The admissibility of rules (3.14b) and (3.14c) is assured if each of the generic rules is uniform, because we may assimilate the additional parameter, $x$, to the global parameters, and the additional hypothesis, $J$, to the global hypotheses. The admissibility of rule (3.14d) is ensured by the identification convention for the formal generic judgment. rule (3.14f) must be verified explicitly for each inductive definition.

The concept of a generic inductive definition extends to parametric judgments as well. Briefly, rules are defined on formal parametric judgments of the form $\vec{u}; \vec{x} \mid \Gamma \vdash J$, with parameters $\vec{u}$, as well as variables, $\vec{x}$. Such formal judgments are identified up to renaming of its variables and its parameters to ensure that the meaning is independent of the choice of variable and parameter names.

## 3.5 Notes

The concepts of entailment and generality are fundamental to logic and programming languages. The formulation given here builds on Martin-Löf (1983, 1987) and Avron (1991). Hypothetical and general reasoning are consolidated into a single concept in the AUTOMATH languages (Nederpelt et al., 1994) and in the LF Logical Framework (Harper et al., 1993). These systems permit arbitrarily nested combinations of hypothetical and general judgments, whereas the present account considers only general hypothetical judgments over basic judgment forms. On the other hand we consider here parameters, as well as variables, which are not present in these previous accounts. Parametric judgements are required for specifying languages that admit the dynamic creation of "new" objects (see, for example, Chapter 36).

# Exercises

**3.1**. *Combinators* are inductively defined by the rule set $\mathcal{C}$ given as follows:

$$\frac{}{\mathtt{s}\ \mathsf{comb}} \tag{3.15a}$$

$$\frac{}{\mathtt{k}\ \mathsf{comb}} \tag{3.15b}$$

Preview

$$\frac{a_1\ \mathsf{comb} \quad a_2\ \mathsf{comb}}{\mathtt{ap}(a_1;a_2)\ \mathsf{comb}} \tag{3.15c}$$

Give an inductive definition of the *length* of a combinator defined as the number of occurrences of `S` and `K` within it.

**3.2**. The general judgment

$$x_1,\ldots,x_n \mid x_1\ \mathsf{comb},\ldots,x_n\ \mathsf{comb} \vdash_{\mathcal{C}} A\ \mathsf{comb}$$

states that $A$ is a combinator that may involve the variables $x_1,\ldots,x_n$. Prove that if $x \mid x\ \mathsf{comb} \vdash_{\mathcal{C}} a_2\ \mathsf{comb}$ and $a_1\ \mathsf{comb}$, then $[a_1/x]a_2\ \mathsf{comb}$ by induction on the derivation of the first hypothesis of the implication.

**3.3**. *Conversion*, or *equivalence*, of combinators is expressed by the judgment $A \equiv B$ defined by the rule set $\mathcal{E}$ extending $\mathcal{C}$ as follows:[1]

$$\frac{a\ \mathsf{comb}}{a \equiv a} \tag{3.16a}$$

$$\frac{}{\mathtt{k} \equiv \mathtt{k}} \tag{3.16b}$$

$$\frac{a_2 \equiv a_1}{a_1 \equiv a_2} \tag{3.16c}$$

$$\frac{a_1 \equiv a_2 \quad a_2 \equiv a_3}{a_1 \equiv a_3} \tag{3.16d}$$

$$\frac{a_1 \equiv a_1' \quad a_2 \equiv a_2'}{a_1\,a_2 \equiv a_1'\,a_2'} \tag{3.16e}$$

$$\frac{}{\mathtt{k}\,a_1\,a_2 \equiv a_1} \tag{3.16f}$$

$$\frac{}{\mathtt{s}\,a_1\,a_2\,a_3 \equiv (a_1\,a_3)\,(a_2\,a_3)} \tag{3.16g}$$

The no-doubt mysterious motivation for the last two equations will become clearer in a moment. For now, show that

$$x \mid x\ \mathsf{comb} \vdash_{\mathcal{C}\cup\mathcal{E}} \mathtt{s}\,\mathtt{k}\,\mathtt{k}\,x \equiv x.$$

**3.4**. Show that if $x \mid x\ \mathsf{comb} \vdash_{\mathcal{C}} a\ \mathsf{comb}$, then there is a combinator $a'$, written $[x]\,a$ and called *bracket abstraction*, such that

$$x \mid x\ \mathsf{comb} \vdash_{\mathcal{C}\cup\mathcal{E}} a'\,x \equiv a.$$

Consequently, by Exercise **3.2**, if $a''\ \mathsf{comb}$, then

$$([x]\,a)\,a'' \equiv [a''/x]a.$$

[1] The combinator $\mathtt{ap}(a_1;a_2)$ is written $a_1\,a_2$ for short, left-associatively when used in succession.



Preview

*Hint:* Proceed by rule induction on the derivation of the premise of the theorem. The motivation for the conversion axioms governing `k` and `s` should become clear while developing the proof of the desired equivalence.

**3.5**. Prove that bracket abstraction, as defined in Exercise **3.4**, is *non-compositional* by exhibiting $a$ and $b$ such that $a$ comb and

$$x\,y \mid x \text{ comb}\, y \text{ comb} \vdash_{\mathcal{C}} b \text{ comb}$$

such that $[a/y]([x]\,b) \neq [x]\,([a/y]b)$. *Hint*: Consider the case that $b$ is $y$.

Suggest a modification to the definition of bracket abstraction that is *compositional* by showing under the same conditions given above that

$$[a/y]([x]\,b) = [x]\,([a/y]b).$$

Preview

# Part II

# Statics and Dynamics

Preview

Preview

# Chapter 4

# Statics

Most programming languages exhibit a *phase distinction* between the *static* and *dynamic* phases of processing. The static phase consists of parsing and type checking to ensure that the program is well-formed; the dynamic phase consists of execution of well-formed programs. A language is said to be *safe* exactly when well-formed programs are well-behaved when executed.

The static phase is specified by a *statics* comprising a collection of rules for deriving *typing judgments* stating that an expression is well-formed of a certain type. Types mediate the interaction between the constituent parts of a program by "predicting" some aspects of the execution behavior of the parts so that we may ensure they fit together properly at run-time. Type safety tells us that these predictions are accurate; if not, the statics is considered to be improperly defined, and the language is deemed *unsafe* for execution.

In this chapter we present the statics of a simple expression language, **E**, as an illustration of the methodology that we shall employ throughout this book.

Preview

## 4.1 Syntax

When defining a language we shall be primarily concerned with its abstract syntax, specified by a collection of operators and their arities. The abstract syntax provides a systematic, unambiguous account of the hierarchical and binding structure of the language, and is therefore to be considered the official presentation of the language. However, for the sake of clarity, it is also useful to specify minimal concrete syntax conventions, without going through the trouble to set up a fully precise grammar for it.

We will accomplish both of these purposes with a *syntax chart*, whose meaning is best illus-

trated by example. The following chart summarizes the abstract and concrete syntax of **E**.

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | ::= | `num` | `num` | numbers |
| | | | `str` | `str` | strings |
| Exp | $e$ | ::= | $x$ | $x$ | variable |
| | | | `num`$[n]$ | $n$ | numeral |
| | | | `str`$[s]$ | $"s"$ | literal |
| | | | `plus`$(e_1; e_2)$ | $e_1 + e_2$ | addition |
| | | | `times`$(e_1; e_2)$ | $e_1 * e_2$ | multiplication |
| | | | `cat`$(e_1; e_2)$ | $e_1 \mathbin{\hat{}} e_2$ | concatenation |
| | | | `len`$(e)$ | $\lvert e \rvert$ | length |
| | | | `let`$(e_1; x.e_2)$ | `let` $x$ `be` $e_1$ `in` $e_2$ | definition |

This chart defines two sorts, Typ, ranged over by $\tau$, and Exp, ranged over by $e$. The chart defines a collection of operators and their arities. For example, the operator `let` has arity $(\mathsf{Exp}, (\mathsf{Exp})\mathsf{Exp})$, which specifies that it has two arguments of sort Exp, and binds a variable of sort Exp in the second argument.

## 4.2 Type System

The role of a type system is to impose constraints on the formations of phrases that are sensitive to the context in which they occur. For example, whether or not the expression `plus`$(x;$`num`$[n])$ is sensible depends on whether or not the variable $x$ is restricted to have type `num` in the surrounding context of the expression. This example is, in fact, illustrative of the general case, in that the *only* information required about the context of an expression is the type of the variables within whose scope the expression lies. Consequently, the statics of **E** consists of an inductive definition of generic hypothetical judgments of the form

$$\vec{x} \mid \Gamma \vdash e : \tau,$$

where $\vec{x}$ is a finite set of variables, and $\Gamma$ is a *typing context* consisting of hypotheses of the form $x : \tau$, one for each $x \in \mathcal{X}$. We rely on typographical conventions to determine the set of variables, using the letters $x$ and $y$ to stand for them. We write $x \notin dom(\Gamma)$ to indicate that there is no assumption in $\Gamma$ of the form $x : \tau$ for any type $\tau$, in which case we say that the variable $x$ is *fresh* for $\Gamma$.

The rules defining the statics of **E** are as follows:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{4.1a}$$

$$\frac{}{\Gamma \vdash \texttt{str}[s] : \texttt{str}} \tag{4.1b}$$

$$\frac{}{\Gamma \vdash \texttt{num}[n] : \texttt{num}} \tag{4.1c}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1; e_2) : \texttt{num}} \tag{4.1d}$$



Preview

$$\frac{\Gamma \vdash e_1 : \mathtt{num} \quad \Gamma \vdash e_2 : \mathtt{num}}{\Gamma \vdash \mathtt{times}(e_1; e_2) : \mathtt{num}} \tag{4.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{str} \quad \Gamma \vdash e_2 : \mathtt{str}}{\Gamma \vdash \mathtt{cat}(e_1; e_2) : \mathtt{str}} \tag{4.1f}$$

$$\frac{\Gamma \vdash e : \mathtt{str}}{\Gamma \vdash \mathtt{len}(e) : \mathtt{num}} \tag{4.1g}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}(e_1; x.e_2) : \tau_2} \tag{4.1h}$$

In rule (4.1h) we tacitly assume that the variable, $x$, is not already declared in $\Gamma$. This condition may always be met by choosing a suitable representative of the $\alpha$-equivalence class of the `let` expression.

It is easy to check that every expression has at most one type by *induction on typing*, which is rule induction applied to rules (4.1).

**Lemma 4.1** (Unicity of Typing)**.** *For every typing context $\Gamma$ and expression $e$, there exists at most one $\tau$ such that $\Gamma \vdash e : \tau$.*

*Proof.* By rule induction on rules (4.1), making use of the fact that variables have at most one type in any typing context. □

The typing rules are *syntax-directed* in the sense that there is exactly one rule for each form of expression. Consequently it is easy to give necessary conditions for typing an expression that invert the sufficient conditions expressed by the corresponding typing rule.

**Lemma 4.2** (Inversion for Typing)**.** *Suppose that $\Gamma \vdash e : \tau$. If $e = \mathtt{plus}(e_1; e_2)$, then $\tau = \mathtt{num}$, $\Gamma \vdash e_1 : \mathtt{num}$, and $\Gamma \vdash e_2 : \mathtt{num}$, and similarly for the other constructs of the language.*

*Proof.* These may all be proved by induction on the derivation of the typing judgment $\Gamma \vdash e : \tau$. □

In richer languages such inversion principles are more difficult to state and to prove.

## 4.3 Structural Properties

The statics enjoys the structural properties of the generic hypothetical judgment.

**Lemma 4.3** (Weakening)**.** *If $\Gamma \vdash e' : \tau'$, then $\Gamma, x : \tau \vdash e' : \tau'$ for any $x \notin dom(\Gamma)$ and any type $\tau$.*

*Proof.* By induction on the derivation of $\Gamma \vdash e' : \tau'$. We will give one case here, for rule (4.1h). We have that $e' = \mathtt{let}(e_1; z.e_2)$, where by the conventions on parameters we may assume $z$ is chosen such that $z \notin dom(\Gamma)$ and $z \neq x$. By induction we have

1. $\Gamma, x : \tau \vdash e_1 : \tau_1$,
2. $\Gamma, x : \tau, z : \tau_1 \vdash e_2 : \tau'$,



Preview

from which the result follows by rule (4.1h). □

**Lemma 4.4** (Substitution)**.** *If* $\Gamma, x : \tau \vdash e' : \tau'$ *and* $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash [e/x]e' : \tau'$*.*

*Proof.* By induction on the derivation of $\Gamma, x : \tau \vdash e' : \tau'$. We again consider only rule (4.1h). As in the preceding case, $e' = \texttt{let}(e_1; z.e_2)$, where $z$ may be chosen so that $z \neq x$ and $z \notin dom(\Gamma)$. We have by induction and Lemma 4.3 that

1. $\Gamma \vdash [e/x]e_1 : \tau_1$,
2. $\Gamma, z : \tau_1 \vdash [e/x]e_2 : \tau'$.

By the choice of $z$ we have

$$[e/x]\texttt{let}(e_1; z.e_2) = \texttt{let}([e/x]e_1; z.[e/x]e_2).$$

It follows by rule (4.1h) that $\Gamma \vdash [e/x]\texttt{let}(e_1; z.e_2) : \tau$, as desired. □

From a programming point of view, Lemma 4.3 allows us to use an expression in any context that binds its free variables: if $e$ is well-typed in a context $\Gamma$, then we may "import" it into any context that includes the assumptions $\Gamma$. In other words the introduction of new variables beyond those required by an expression, $e$, does not invalidate $e$ itself; it remains well-formed, with the same type.[1] More significantly, Lemma 4.4 expresses the concepts of *modularity* and *linking*. We may think of the expressions $e$ and $e'$ as two *components* of a larger system in which the component $e'$ is to be thought of as a *client* of the *implementation* $e$. The client declares a variable specifying the type of the implementation, and is type checked knowing only this information. The implementation must be of the specified type in order to satisfy the assumptions of the client. If so, then we may link them to form the composite system, $[e/x]e'$. This may itself be the client of another component, represented by a variable, $y$, that is replaced by that component during linking. When all such variables have been implemented, the result is a *closed expression* that is ready for execution (evaluation).

The converse of Lemma 4.4 is called *decomposition*. It states that any (large) expression may be decomposed into a client and implementor by introducing a variable to mediate their interaction.

**Lemma 4.5** (Decomposition)**.** *If* $\Gamma \vdash [e/x]e' : \tau'$*, then for every type* $\tau$ *such that* $\Gamma \vdash e : \tau$*, we have* $\Gamma, x : \tau \vdash e' : \tau'$*.*

*Proof.* The typing of $[e/x]e'$ depends only on the type of $e$ wherever it occurs, if at all. □

Lemma 4.5 tells us that any sub-expression may be isolated as a separate module of a larger system. This is especially useful when the variable $x$ occurs more than once in $e'$, because then one copy of $e$ suffices for all occurrences of $x$ in $e'$.

The statics of **E** given by rules (4.1) exemplifies a recurrent pattern. The constructs of a language are classified into one of two forms, the *introductory* and the *eliminatory*. The introductory forms

[1] This may seem so obvious as to be not worthy of mention, but, suprisingly, there are useful type systems that lack this property. Because they do not validate the structural principle of weakening, they are called *sub-structural* type systems.



Preview

for a type determine the *values*, or *canonical forms*, of that type. The eliminatory forms determine how to manipulate the values of a type to form a computation of another (possibly the same) type. In the language **E** the introductory forms for the type `num` are the numerals, and those for the type `str` are the literals. The eliminatory forms for the type `num` are addition and multiplication, and those for the type `str` are concatenation and length.

The importance of this classification will become apparent once we have defined the dynamics of the language in Chapter 5. Then we will see that the eliminatory forms are *inverse* to the introductory forms in that they "take apart" what the introductory forms have "put together." The coherence of the statics and dynamics of a language expresses the concept of *type safety*, the subject of Chapter 6.

## 4.4 Notes

The concept of the static semantics of a programming language was historically slow to develop, perhaps because the earliest languages had relatively few features and only very weak type systems. The concept of a static semantics in the sense considered here was introduced in the definition of the Standard ML programming language (Milner et al., 1997), building on much earlier work by Church and others on the typed $\lambda$-calculus (Barendregt, 1992). The concept of introduction and elimination, and the associated inversion principle, was introduced by Gentzen in his pioneering work on natural deduction (Gentzen, 1969). These principles were applied to the structure of programming languages by Martin-Löf (1984, 1980).

## Exercises

**4.1**. It is sometimes useful to give the typing judgment $\Gamma \vdash e : \tau$ an "operational" reading that specifies more precisely the flow of information required to derive a typing judgment (or determine that it is not derivable). The *analytic* mode corresponds to the context, expression, and type being given, with the goal to determine whether or not the typing judgment is derivable. The *synthetic* mode corresponds to the context and expression being given, with the goal to find the unique type $\tau$, if any, that is possessed by the expression in that context. These two readings can be made explicit as judgments of the form $e \downarrow \tau$, corresponding to the analytic mode, and $e \uparrow \tau$, corresponding to the synthetic mode.

Give a simultaneous inductive definition of these two judgments according to the following guidelines:

(a) Variables are introduced in synthetic form.

(b) If we can synthesize a unique type for an expression, then we can analyze it with respect to a given type by checking type equality.

(c) Definitions require care, because the type of the defined expression is not given, even when the type of the result is given.

There is room for variation; the point of the exercise is to explore the possibilities.

Preview

**4.2**. One way to limit the range of possibilities in the solution to Exercise **4.1** is to restrict and extend the syntax of the language so that every expression is either synthetic or analytic according to the following suggestions:

(a) Variables are analytic.

(b) Introductory forms are analytic, eliminatory are synthetic.

(c) An analytic expresssion can be made synthetic by introducing a *type cast* of the form $\mathtt{cast}[\tau](e)$ specifying that $e$ must check against the specified type $\tau$, which is synthesized for the whole expression.

(d) The defining expression of a definition must be synthetic, but the scope of the definition can be either synthetic or analytic.

Reformulate your solution to Exercise **4.1** to take account of these guidelines.

Preview

# Chapter 5

# Dynamics

The *dynamics* of a language is a description of how programs are to be executed. The most important way to define the dynamics of a language is by the method of *structural dynamics*, which defines a *transition system* that inductively specifies the step-by-step process of executing a program. Another method for presenting dynamics, called *contextual dynamics*, is a variation of structural dynamics in which the transition rules are specified in a slightly different manner. An *equational dynamics* presents the dynamics of a language equationally by a collection of rules for deducing when one program is *definitionally equal* to another.

## 5.1 Transition Systems

A *transition system* is specified by the following four forms of judgment:

1. $s$ state, asserting that $s$ is a *state* of the transition system.
2. $s$ final, where $s$ state, asserting that $s$ is a *final* state.
3. $s$ initial, where $s$ state, asserting that $s$ is an *initial* state.
4. $s \longmapsto s'$, where $s$ state and $s'$ state, asserting that state $s$ may transition to state $s'$.

In practice we always arrange things so that no transition is possible from a final state: if $s$ final, then there is no $s'$ state such that $s \longmapsto s'$. A state from which no transition is possible is sometimes said to be *stuck*. Whereas all final states are, by convention, stuck, there may be stuck states in a transition system that are not final. A transition system is *deterministic* iff for every state $s$ there exists at most one state $s'$ such that $s \longmapsto s'$, otherwise it is *non-deterministic*.

A *transition sequence* is a sequence of states $s_0, \ldots, s_n$ such that $s_0$ initial, and $s_i \longmapsto s_{i+1}$ for every $0 \leq i < n$. A transition sequence is *maximal* iff there is no $s$ such that $s_n \longmapsto s$, and it is *complete* iff it is maximal and, in addition, $s_n$ final. Thus every complete transition sequence is maximal, but maximal sequences are not necessarily complete. The judgment $s \downarrow$ means that there is a complete transition sequence starting from $s$, which is to say that there exists $s'$ final such that $s \longmapsto^* s'$.

Preview

The *iteration* of transition judgment, $s \longmapsto^* s'$, is inductively defined by the following rules:

$$\frac{}{s \longmapsto^* s} \tag{5.1a}$$

$$\frac{s \longmapsto s' \quad s' \longmapsto^* s''}{s \longmapsto^* s''} \tag{5.1b}$$

When applied to the definition of iterated transition, the principle of rule induction states that to show that $P(s, s')$ holds whenever $s \longmapsto^* s'$, it is enough to show these two properties of $P$:

1. $P(s, s)$.
2. if $s \longmapsto s'$ and $P(s', s'')$, then $P(s, s'')$.

The first requirement is to show that $P$ is reflexive. The second is to show that $P$ is *closed under head expansion*, or *closed under inverse evaluation*. Using this principle, it is easy to prove that $\longmapsto^*$ is reflexive and transitive.

The *n-times iterated* transition judgment, $s \longmapsto^n s'$, where $n \geq 0$, is inductively defined by the following rules.

$$\frac{}{s \longmapsto^0 s} \tag{5.2a}$$

$$\frac{s \longmapsto s' \quad s' \longmapsto^n s''}{s \longmapsto^{n+1} s''} \tag{5.2b}$$

**Theorem 5.1.** *For all states $s$ and $s'$, $s \longmapsto^* s'$ iff $s \longmapsto^k s'$ for some $k \geq 0$.*

*Proof.* From left to right, by induction on the definition of multi-step transition. From right to left, by mathematical induction on $k \geq 0$. □

## 5.2 Structural Dynamics

A *structural dynamics* for the language **E** is given by a transition system whose states are closed expressions. All states are initial. The final states are the *(closed) values*, which represent the completed computations. The judgment $e$ val, which states that $e$ is a value, is inductively defined by the following rules:

$$\frac{}{\mathtt{num}[n] \text{ val}} \tag{5.3a}$$

$$\frac{}{\mathtt{str}[s] \text{ val}} \tag{5.3b}$$

The transition judgment, $e \longmapsto e'$, between states is inductively defined by the following rules:

$$\frac{n_1 + n_2 = n \text{ nat}}{\mathtt{plus}(\mathtt{num}[n_1]; \mathtt{num}[n_2]) \longmapsto \mathtt{num}[n]} \tag{5.4a}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{plus}(e_1; e_2) \longmapsto \mathtt{plus}(e_1'; e_2)} \tag{5.4b}$$



Preview

$$\frac{e_1 \text{ val} \quad e_2 \longmapsto e_2'}{\texttt{plus}(e_1;e_2) \longmapsto \texttt{plus}(e_1;e_2')} \tag{5.4c}$$

$$\frac{s_1 \hat{} s_2 = s \text{ str}}{\texttt{cat}(\texttt{str}[s_1];\texttt{str}[s_2]) \longmapsto \texttt{str}[s]} \tag{5.4d}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{cat}(e_1;e_2) \longmapsto \texttt{cat}(e_1';e_2)} \tag{5.4e}$$

$$\frac{e_1 \text{ val} \quad e_2 \longmapsto e_2'}{\texttt{cat}(e_1;e_2) \longmapsto \texttt{cat}(e_1;e_2')} \tag{5.4f}$$

$$\left[\frac{e_1 \longmapsto e_1'}{\texttt{let}(e_1;x.e_2) \longmapsto \texttt{let}(e_1';x.e_2)}\right] \tag{5.4g}$$

$$\frac{[e_1 \text{ val}]}{\texttt{let}(e_1;x.e_2) \longmapsto [e_1/x]e_2} \tag{5.4h}$$

We have omitted rules for multiplication and computing the length of a string, which follow a similar pattern. Rules (5.4a), (5.4d), and (5.4h) are *instruction transitions*, because they correspond to the primitive steps of evaluation. The remaining rules are *search transitions* that determine the order in which instructions are executed.

The bracketed rule, rule (5.4g), and bracketed premise on rule (5.4h), are to be included for a *by-value* interpretation of `let`, and omitted for a *by-name* interpretation. The by-value intepretation evaluates an expression before binding it to the defined variable, whereas the by-name interpretation binds it in unevaluated form. The by-value interpretation saves work if the defined variable is used more than once, but wastes work if it is not used at all. Conversely, the by-name interpretation saves work if the defined variable is not used, and wastes work if it is used more than once.

A derivation sequence in a structural dynamics has a two-dimensional structure, with the number of steps in the sequence being its "width" and the derivation tree for each step being its "height." For example, consider the following evaluation sequence.

$$\begin{aligned}
&\texttt{let}(\texttt{plus}(\texttt{num}[1];\texttt{num}[2]);x.\texttt{plus}(\texttt{plus}(x;\texttt{num}[3]);\texttt{num}[4])) \\
&\quad\longmapsto\quad \texttt{let}(\texttt{num}[3];x.\texttt{plus}(\texttt{plus}(x;\texttt{num}[3]);\texttt{num}[4])) \\
&\quad\longmapsto\quad \texttt{plus}(\texttt{plus}(\texttt{num}[3];\texttt{num}[3]);\texttt{num}[4]) \\
&\quad\longmapsto\quad \texttt{plus}(\texttt{num}[6];\texttt{num}[4]) \\
&\quad\longmapsto\quad \texttt{num}[10]
\end{aligned}$$

Each step in this sequence of transitions is justified by a derivation according to rules (5.4). For example, the third transition in the preceding example is justified by the following derivation:

$$\frac{\dfrac{}{\texttt{plus}(\texttt{num}[3];\texttt{num}[3]) \longmapsto \texttt{num}[6]}\ (5.4a)}{\texttt{plus}(\texttt{plus}(\texttt{num}[3];\texttt{num}[3]);\texttt{num}[4]) \longmapsto \texttt{plus}(\texttt{num}[6];\texttt{num}[4])}\ (5.4b)$$

The other steps are similarly justified by a composition of rules.



Preview

The principle of rule induction for the structural dynamics of **E** states that to show $\mathcal{P}(e \longmapsto e')$ whenever $e \longmapsto e'$, it is sufficient to show that $\mathcal{P}$ is closed under rules (5.4). For example, we may show by rule induction that the structural dynamics of **E** is *determinate*, which means that an expression may make a transition to at most one other expression. The proof requires a simple lemma relating transition to values.

**Lemma 5.2** (Finality of Values)**.** *For no expression $e$ do we have both $e$ $\mathsf{val}$ and $e \longmapsto e'$ for some $e'$.*

*Proof.* By rule induction on rules (5.3) and (5.4). □

**Lemma 5.3** (Determinacy)**.** *If $e \longmapsto e'$ and $e \longmapsto e''$, then $e'$ and $e''$ are $\alpha$-equivalent.*

*Proof.* By rule induction on the premises $e \longmapsto e'$ and $e \longmapsto e''$, carried out either simultaneously or in either order. It is assumed that the primitive operators, such as addition, have a unique value when applied to values. □

Rules (5.4) exemplify the *inversion principle* of language design, which states that the eliminatory forms are *inverse* to the introductory forms of a language. The search rules determine the *principal arguments* of each eliminatory form, and the instruction rules specify how to evaluate an eliminatory form when all of its principal arguments are in introductory form. For example, rules (5.4) specify that both arguments of addition are principal, and specify how to evaluate an addition once its principal arguments are evaluated to numerals. The inversion principle is central to ensuring that a programming language is properly defined, the exact statement of which is given in Chapter 6.

## 5.3 Contextual Dynamics

A variant of structural dynamics, called *contextual dynamics*, is sometimes useful. There is no fundamental difference between contextual and structural dynamics, rather one of style. The main idea is to isolate instruction steps as a special form of judgment, called *instruction transition*, and to formalize the process of locating the next instruction using a device called an *evaluation context*. The judgment, $e$ $\mathsf{val}$, defining whether an expression is a value, remains unchanged.

The instruction transition judgment, $e_1 \to e_2$, for **E** is defined by the following rules, together with similar rules for multiplication of numbers and the length of a string.

$$\frac{m + n = p \ \mathsf{nat}}{\mathtt{plus}(\mathtt{num}[m];\mathtt{num}[n]) \to \mathtt{num}[p]} \tag{5.5a}$$

$$\frac{s \hat{} t = u \ \mathsf{str}}{\mathtt{cat}(\mathtt{str}[s];\mathtt{str}[t]) \to \mathtt{str}[u]} \tag{5.5b}$$

$$\frac{}{\mathtt{let}(e_1; x.e_2) \to [e_1/x]e_2} \tag{5.5c}$$

The judgment $\mathcal{E}$ $\mathsf{ectxt}$ determines the location of the next instruction to execute in a larger expression. The position of the next instruction step is specified by a "hole", written $\circ$, into which



Preview

the next instruction is placed, as we shall detail shortly. (The rules for multiplication and length are omitted for concision, as they are handled similarly.)

$$\frac{}{\circ\ \mathsf{ectxt}} \tag{5.6a}$$

$$\frac{\mathcal{E}_1\ \mathsf{ectxt}}{\mathtt{plus}(\mathcal{E}_1; e_2)\ \mathsf{ectxt}} \tag{5.6b}$$

$$\frac{e_1\ \mathsf{val} \quad \mathcal{E}_2\ \mathsf{ectxt}}{\mathtt{plus}(e_1; \mathcal{E}_2)\ \mathsf{ectxt}} \tag{5.6c}$$

The first rule for evaluation contexts specifies that the next instruction may occur "here", at the point of the occurrence of the hole. The remaining rules correspond one-for-one to the search rules of the structural dynamics. For example, rule (5.6c) states that in an expression $\mathtt{plus}(e_1; e_2)$, if the first argument, $e_1$, is a value, then the next instruction step, if any, lies at or within the second argument, $e_2$.

An evaluation context is to be thought of as a template that is instantiated by replacing the hole with an instruction to be executed. The judgment $e' = \mathcal{E}\{e\}$ states that the expression $e'$ is the result of filling the hole in the evaluation context $\mathcal{E}$ with the expression $e$. It is inductively defined by the following rules:

$$\frac{}{e = \circ\{e\}} \tag{5.7a}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\mathtt{plus}(e_1; e_2) = \mathtt{plus}(\mathcal{E}_1; e_2)\{e\}} \tag{5.7b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 = \mathcal{E}_2\{e\}}{\mathtt{plus}(e_1; e_2) = \mathtt{plus}(e_1; \mathcal{E}_2)\{e\}} \tag{5.7c}$$

There is one rule for each form of evaluation context. Filling the hole with $e$ results in $e$; otherwise we proceed inductively over the structure of the evaluation context.

Finally, the contextual dynamics for **E** is defined by a single rule:

$$\frac{e = \mathcal{E}\{e_0\} \quad e_0 \to e_0' \quad e' = \mathcal{E}\{e_0'\}}{e \longmapsto e'} \tag{5.8}$$

Thus, a transition from $e$ to $e'$ consists of (1) decomposing $e$ into an evaluation context and an instruction, (2) execution of that instruction, and (3) replacing the instruction by the result of its execution in the same spot within $e$ to obtain $e'$.

The structural and contextual dynamics define the same transition relation. For the sake of the proof, let us write $e \longmapsto_{\mathsf{s}} e'$ for the transition relation defined by the structural dynamics (rules (5.4)), and $e \longmapsto_{\mathsf{c}} e'$ for the transition relation defined by the contextual dynamics (rules (5.8)).

**Theorem 5.4.** *$e \longmapsto_{\mathsf{s}} e'$ if, and only if, $e \longmapsto_{\mathsf{c}} e'$.*



Preview

*Proof.* From left to right, proceed by rule induction on rules (5.4). It is enough in each case to exhibit an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \to e_0'$. For example, for rule (5.4a), take $\mathcal{E} = \circ$, and observe that $e \to e'$. For rule (5.4b), we have by induction that there exists an evaluation context $\mathcal{E}_1$ such that $e_1 = \mathcal{E}_1\{e_0\}$, $e_1' = \mathcal{E}_1\{e_0'\}$, and $e_0 \to e_0'$. Take $\mathcal{E} = \mathtt{plus}(\mathcal{E}_1; e_2)$, and observe that $e = \mathtt{plus}(\mathcal{E}_1; e_2)\{e_0\}$ and $e' = \mathtt{plus}(\mathcal{E}_1; e_2)\{e_0'\}$ with $e_0 \to e_0'$.

From right to left, observe that if $e \longmapsto_{\mathtt{c}} e'$, then there exists an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \to e_0'$. We prove by induction on rules (5.7) that $e \longmapsto_{\mathtt{s}} e'$. For example, for rule (5.7a), $e_0$ is $e$, $e_0'$ is $e'$, and $e \to e'$. Hence $e \longmapsto_{\mathtt{s}} e'$. For rule (5.7b), we have that $\mathcal{E} = \mathtt{plus}(\mathcal{E}_1; e_2)$, $e_1 = \mathcal{E}_1\{e_0\}$, $e_1' = \mathcal{E}_1\{e_0'\}$, and $e_1 \longmapsto_{\mathtt{s}} e_1'$. Therefore $e$ is $\mathtt{plus}(e_1; e_2)$, $e'$ is $\mathtt{plus}(e_1'; e_2)$, and therefore by rule (5.4b), $e \longmapsto_{\mathtt{s}} e'$.

□

Because the two transition judgments coincide, contextual dynamics may be seen as an alternative way of presenting a structural dynamics. It has two advantages over structural dynamics, one relatively superficial, one rather less so. The superficial advantage stems from writing rule (5.8) in the simpler form

$$\frac{e_0 \to e_0'}{\mathcal{E}\{e_0\} \longmapsto \mathcal{E}\{e_0'\}}. \tag{5.9}$$

This formulation is superficially simpler in that it does not make explicit how an expression is to be decomposed into an evaluation context and a reducible expression. The deeper advantage of contextual dynamics is that all transitions are between complete programs. One need never consider a transition between expressions of any type other than the ultimate observable type. This simplifies certain arguments, notably the proof of Lemma 49.16.

## 5.4 Equational Dynamics

Another formulation of the dynamics of a language is based on regarding computation as a form of equational deduction, much in the style of elementary algebra. For example, in algebra we may show that the polynomials $x^2 + 2x + 1$ and $(x + 1)^2$ are equivalent by a simple process of calculation and re-organization using the familiar laws of addition and multiplication. The same laws are sufficient to determine the value of any polynomial, given the values of its variables. So, for example, we may plug in 2 for $x$ in the polynomial $x^2 + 2x + 1$ and calculate that $2^2 + 2 \times 2 + 1 = 9$, which is indeed $(2 + 1)^2$. This gives rise to a model of computation in which we may determine the value of a polynomial for a given value of its variable by substituting the given value for the variable and proving that the resulting expression is equal to its value.

Very similar ideas give rise to the concept of *definitional*, or *computational*, *equivalence* of expressions in **E**, which we write as $\mathcal{X} \mid \Gamma \vdash e \equiv e' : \tau$, where $\Gamma$ consists of one assumption of the form $x : \tau$ for each $x \in \mathcal{X}$. We only consider definitional equality of well-typed expressions, so that when considering the judgment $\Gamma \vdash e \equiv e' : \tau$, we tacitly assume that $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$. Here, as usual, we omit explicit mention of the parameters, $\mathcal{X}$, when they can be determined from the forms of the assumptions $\Gamma$.



Preview

Definitional equality of expressions in **E** under the by-name interpretation of `let` is inductively defined by the following rules:

$$\frac{}{\Gamma \vdash e \equiv e : \tau} \tag{5.10a}$$

$$\frac{\Gamma \vdash e' \equiv e : \tau}{\Gamma \vdash e \equiv e' : \tau} \tag{5.10b}$$

$$\frac{\Gamma \vdash e \equiv e' : \tau \quad \Gamma \vdash e' \equiv e'' : \tau}{\Gamma \vdash e \equiv e'' : \tau} \tag{5.10c}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \mathtt{num} \quad \Gamma \vdash e_2 \equiv e_2' : \mathtt{num}}{\Gamma \vdash \mathtt{plus}(e_1; e_2) \equiv \mathtt{plus}(e_1'; e_2') : \mathtt{num}} \tag{5.10d}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \mathtt{str} \quad \Gamma \vdash e_2 \equiv e_2' : \mathtt{str}}{\Gamma \vdash \mathtt{cat}(e_1; e_2) \equiv \mathtt{cat}(e_1'; e_2') : \mathtt{str}} \tag{5.10e}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{let}(e_1; x.e_2) \equiv \mathtt{let}(e_1'; x.e_2') : \tau_2} \tag{5.10f}$$

$$\frac{n_1 + n_2 = n \ \mathsf{nat}}{\Gamma \vdash \mathtt{plus}(\mathtt{num}[n_1]; \mathtt{num}[n_2]) \equiv \mathtt{num}[n] : \mathtt{num}} \tag{5.10g}$$

$$\frac{s_1 \hat{} s_2 = s \ \mathsf{str}}{\Gamma \vdash \mathtt{cat}(\mathtt{str}[s_1]; \mathtt{str}[s_2]) \equiv \mathtt{str}[s] : \mathtt{str}} \tag{5.10h}$$

$$\frac{}{\Gamma \vdash \mathtt{let}(e_1; x.e_2) \equiv [e_1 / x]e_2 : \tau} \tag{5.10i}$$

Rules (5.10a) through (5.10c) state that definitional equality is an *equivalence relation*. Rules (5.10d) through (5.10f) state that it is a *congruence relation*, which means that it is compatible with all expression-forming constructs in the language. Rules (5.10g) through (5.10i) specify the meanings of the primitive constructs of **E**. For the sake of concision, rules (5.10) may be characterized as defining the *strongest congruence* closed under rules (5.10g), (5.10h), and (5.10i).

Rules (5.10) are sufficient to allow us to calculate the value of an expression by an equational deduction similar to that used in high school algebra. For example, we may derive the equation

$$\mathtt{let}\ x\ \mathtt{be}\ 1 + 2\ \mathtt{in}\ x + 3 + 4 \equiv 10 : \mathtt{num}$$

by applying rules (5.10). Here, as in general, there may be many different ways to derive the same equation, but we need find only one derivation in order to carry out an evaluation.

Definitional equality is rather weak in that many equivalences that we might intuitively think are true are not derivable from rules (5.10). A prototypical example is the putative equivalence

$$x_1 : \mathtt{num}, x_2 : \mathtt{num} \vdash x_1 + x_2 \equiv x_2 + x_1 : \mathtt{num}, \tag{5.11}$$

which, intuitively, expresses the commutativity of addition. Although we shall not prove this here, this equivalence is *not* derivable from rules (5.10). And yet we *may* derive all of its closed instances,

$$n_1 + n_2 \equiv n_2 + n_1 : \mathtt{num}, \tag{5.12}$$



Preview

where $n_1$ nat and $n_2$ nat are particular numbers.

The "gap" between a general law, such as Equation (5.11), and all of its instances, given by Equation (5.12), may be filled by enriching the notion of equivalence to include a principle of proof by mathematical induction. Such a notion of equivalence is sometimes called *semantic equivalence*, because it expresses relationships that hold by virtue of the dynamics of the expressions involved. (Semantic equivalence is developed rigorously for a related language in Chapter 48.)

Definitional equality is sometimes called *symbolic evaluation*, because it allows any subexpression to be replaced by the result of evaluating it according to the rules of the dynamics of the language.

**Theorem 5.5.** *For the expression language* **E**, $e \equiv e' : \tau$ *iff there exists* $e_0$ val *such that* $e \longmapsto^* e_0$ *and* $e' \longmapsto^* e_0$.

*Proof.* The proof from right to left is direct, because every transition step is a valid equation. The converse follows from the following, more general, proposition, which is proved by induction on rules (5.10): if $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e \equiv e' : \tau$, then whenever $e_1 : \tau_1, e'_1 : \tau_1, \ldots, e_n : \tau_n, e'_n : \tau_n$, if for each $1 \leq i \leq n$ the expressions $e_i$ and $e'_i$ evalute to a common value $v_i$, then there exists $e_0$ val such that

$$[e_1, \ldots, e_n / x_1, \ldots, x_n]e \longmapsto^* e_0$$

and

$$[e'_1, \ldots, e'_n / x_1, \ldots, x_n]e' \longmapsto^* e_0.$$

□

## 5.5 Notes

The use of transition systems to specify the behavior of programs goes back to the early work of Church and Turing on computability. Turing's approach emphasized the concept of an abstract machine consisting of a finite program together with unbounded memory. Computation proceeds by changing the memory in accordance with the instructions in the program. Much early work on the operational semantics of programming languages, such as the SECD machine (Landin, 1965), emphasized machine models. Church's approach emphasized the language for expressing computations, and defined execution in terms of the programs themselves, rather than in terms of auxiliary concepts such as memories or tapes. Plotkin's elegant formulation of structural operational semantics (Plotkin, 1981), which we use heavily throughout this book, was inspired by Church's and Landin's ideas (Plotkin, 2004). Contextual semantics, which was introduced by Felleisen and Hieb (1992), may be seen as an alternative formulation of structural semantics in which "search rules" are replaced by "context matching". Computation viewed as equational deduction goes back to the early work of Herbrand, Gödel, and Church.

## Exercises

Preview

**5.1**. Prove that if $s \longmapsto^* s'$ and $s' \longmapsto^* s''$, then $s \longmapsto^* s''$.

**5.2**. Complete the proof of Theorem 5.1 along the lines suggested there.

**5.3**. Complete the proof of Theorem 5.5 along the lines suggested there.

Preview

Preview

# Chapter 6

# Type Safety

Most contemporary programming languages are *safe* (or, *type safe*, or *strongly typed*). Informally, this means that certain kinds of mismatches cannot arise during execution. For example, type safety for **E** states that it will never arise that a number is to be added to a string, or that two numbers are to be concatenated, neither of which is meaningful.

In general type safety expresses the coherence between the statics and the dynamics. The statics may be seen as predicting that the value of an expression will have a certain form so that the dynamics of that expression is well-defined. Consequently, evaluation cannot "get stuck" in a state for which no transition is possible, corresponding in implementation terms to the absence of "illegal instruction" errors at execution time. This is proved by showing that each step of transition preserves typability and by showing that typable states are well-defined. Consequently, evaluation can never "go off into the weeds," and hence can never encounter an illegal instruction.

More precisely, type safety for the language **E** may be stated as follows:

**Theorem 6.1** (Type Safety)**.**

1. *If* $e : \tau$ *and* $e \longmapsto e'$*, then* $e' : \tau$*.*
2. *If* $e : \tau$*, then either* $e$ val*, or there exists* $e'$ *such that* $e \longmapsto e'$*.*

The first part, called *preservation*, says that the steps of evaluation preserve typing; the second, called *progress*, ensures that well-typed expressions are either values or can be further evaluated. Safety is the conjunction of preservation and progress.

We say that an expression, $e$, is *stuck* iff it is not a value, yet there is no $e'$ such that $e \longmapsto e'$. It follows from the safety theorem that a stuck state is necessarily ill-typed. Or, putting it the other way around, that well-typed states do not get stuck.

## 6.1 Preservation

The preservation theorem for **E** defined in Chapters 4 and 5 is proved by rule induction on the transition system (rules (5.4)).

Preview

**Theorem 6.2** (Preservation). *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

*Proof.* We will give the proof in two cases, leaving the rest to the reader. Consider rule (5.4b),

$$\frac{e_1 \longmapsto e_1'}{\mathtt{plus}(e_1; e_2) \longmapsto \mathtt{plus}(e_1'; e_2)}\ .$$

Assume that $\mathtt{plus}(e_1; e_2) : \tau$. By inversion for typing, we have that $\tau = \mathtt{num}$, $e_1 : \mathtt{num}$, and $e_2 : \mathtt{num}$. By induction we have that $e_1' : \mathtt{num}$, and hence $\mathtt{plus}(e_1'; e_2) : \mathtt{num}$. The case for concatenation is handled similarly.

Now consider rule (5.4h),

$$\frac{}{\mathtt{let}(e_1; x.e_2) \longmapsto [e_1/x]e_2}\ .$$

Assume that $\mathtt{let}(e_1; x.e_2) : \tau_2$. By the inversion lemma 4.2, $e_1 : \tau_1$ for some $\tau_1$ such that $x : \tau_1 \vdash e_2 : \tau_2$. By the substitution lemma 4.4 $[e_1/x]e_2 : \tau_2$, as desired.

It is easy to check that the primitive operations are all type-preserving; for example, if $a$ nat and $b$ nat and $a + b = c$ nat, then $c$ nat.

□

The proof of preservation is naturally structured as an induction on the transition judgment, because the argument hinges on examining all possible transitions from a given expression. In some cases we may manage to carry out a proof by structural induction on $e$, or by an induction on typing, but experience shows that this often leads to awkward arguments, or, in some cases, cannot be made to work at all.

## 6.2 Progress

The progress theorem captures the idea that well-typed programs cannot "get stuck". The proof depends crucially on the following lemma, which characterizes the values of each type.

**Lemma 6.3** (Canonical Forms). *If $e$ val and $e : \tau$, then*

1. *If $\tau = \mathtt{num}$, then $e = \mathtt{num}[n]$ for some number $n$.*
2. *If $\tau = \mathtt{str}$, then $e = \mathtt{str}[s]$ for some string $s$.*

*Proof.* By induction on rules (4.1) and (5.3). □

Progress is proved by rule induction on rules (4.1) defining the statics of the language.

**Theorem 6.4** (Progress). *If $e : \tau$, then either $e$ val, or there exists $e'$ such that $e \longmapsto e'$.*

*Proof.* The proof proceeds by induction on the typing derivation. We will consider only one case, for rule (4.1d),

$$\frac{e_1 : \mathtt{num} \quad e_2 : \mathtt{num}}{\mathtt{plus}(e_1; e_2) : \mathtt{num}}\ ,$$



Preview

where the context is empty because we are considering only closed terms.

By induction we have that either $e_1$ val, or there exists $e_1'$ such that $e_1 \longmapsto e_1'$. In the latter case it follows that $\texttt{plus}(e_1; e_2) \longmapsto \texttt{plus}(e_1'; e_2)$, as required. In the former we also have by induction that either $e_2$ val, or there exists $e_2'$ such that $e_2 \longmapsto e_2'$. In the latter case we have that $\texttt{plus}(e_1; e_2) \longmapsto \texttt{plus}(e_1; e_2')$, as required. In the former, we have, by the Canonical Forms Lemma 6.3, $e_1 = \texttt{num}[n_1]$ and $e_2 = \texttt{num}[n_2]$, and hence

$$\texttt{plus}(\texttt{num}[n_1]; \texttt{num}[n_2]) \longmapsto \texttt{num}[n_1 + n_2].$$

□

Because the typing rules for expressions are syntax-directed, the progress theorem could equally well be proved by induction on the structure of $e$, appealing to the inversion theorem at each step to characterize the types of the parts of $e$. But this approach breaks down when the typing rules are not syntax-directed, that is, when there may be more than one rule for a given expression form. No difficulty arises if the proof proceeds by induction on the typing rules.

Summing up, the combination of preservation and progress together constitute the proof of safety. The progress theorem ensures that well-typed expressions do not "get stuck" in an ill-defined state, and the preservation theorem ensures that if a step is taken, the result remains well-typed (with the same type). Thus the two parts work hand-in-hand to ensure that the statics and dynamics are coherent, and that no ill-defined states can ever be encountered while evaluating a well-typed expression.

## 6.3 Run-Time Errors

Suppose that we wish to extend **E** with, say, a quotient operation that is undefined for a zero divisor. The natural typing rule for quotients is given by the following rule:

$$\frac{e_1 : \texttt{num} \quad e_2 : \texttt{num}}{\texttt{div}(e_1; e_2) : \texttt{num}}.$$

But the expression $\texttt{div}(\texttt{num}[3]; \texttt{num}[0])$ is well-typed, yet stuck! We have two options to correct this situation:

1. Enhance the type system, so that no well-typed program may divide by zero.
2. Add dynamic checks, so that division by zero signals an error as the outcome of evaluation.

Either option is, in principle, viable, but the most common approach is the second. The first requires that the type checker prove that an expression be non-zero before permitting it to be used in the denominator of a quotient. It is difficult to do this without ruling out too many programs as ill-formed. This is because we cannot reliably predict statically whether an expression will turn out to be non-zero when executed (because this is an undecidable property). We therefore consider the second approach, which is typical of current practice.

The general idea is to distinguish *checked* from *unchecked* errors. An unchecked error is one that is ruled out by the type system. No run-time checking is performed to ensure that such an



Preview

error does not occur, because the type system rules out the possibility of it arising. For example, the dynamics need not check, when performing an addition, that its two arguments are, in fact, numbers, as opposed to strings, because the type system ensures that this is the case. On the other hand the dynamics for quotient *must* check for a zero divisor, because the type system does not rule out the possibility.

One approach to modelling checked errors is to give an inductive definition of the judgment $e$ err stating that the expression $e$ incurs a checked run-time error, such as division by zero. Here are some representative rules that would appear in a full inductive definition of this judgment:

$$\frac{e_1 \ \mathsf{val}}{\mathtt{div}(e_1;\mathtt{num}[0]) \ \mathsf{err}} \tag{6.1a}$$

$$\frac{e_1 \ \mathsf{err}}{\mathtt{div}(e_1;e_2) \ \mathsf{err}} \tag{6.1b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{err}}{\mathtt{div}(e_1;e_2) \ \mathsf{err}} \tag{6.1c}$$

Rule (6.1a) signals an error condition for division by zero. The other rules propagate this error upwards: if an evaluated sub-expression is a checked error, then so is the overall expression.

Once the error judgment is available, we may also consider an expression, `error`, which forcibly induces an error, with the following static and dynamic semantics:

$$\frac{}{\Gamma \vdash \mathtt{error} : \tau} \tag{6.2a}$$

$$\frac{}{\mathtt{error} \ \mathsf{err}} \tag{6.2b}$$

The preservation theorem is not affected by the presence of checked errors. However, the statement (and proof) of progress is modified to account for checked errors.

**Theorem 6.5** (Progress With Error)**.** *If $e : \tau$, then either $e$ err, or $e$ val, or there exists $e'$ such that $e \longmapsto e'$.*

*Proof.* The proof is by induction on typing, and proceeds similarly to the proof given earlier, except that there are now three cases to consider at each point in the proof. □

## 6.4 Extrinsic and Intrinsic Safety

The treatment of safety in this chapter is based on viewing the dynamics as *extrinsic* to the statics of the language. The dynamics is defined on expressions irrespective of their typing, or even whether they are well-typed. The rules are guided solely by the form of expression, regardless of any static formation restrictions that might be imposed on them. However, the progress and preservation theorems assure us that only well-typed expressions are ever relevant to evaluating well-typed programs. For, by preservation, if the initial state is well-typed, then all states reachable by a



Preview

sequence of transitions are themselves well-typed, and, by progress, well-typed states are either final or amenable to further transition.

This suggests an alternative formulation of type safety in which the statics is taken as primary, determining the class of well-formed expressions by a collection of context-sensitive typing rules. The dynamics is then defined *only* on well-formed expressions in the sense that transition is a relation on two closed expressions of the same type. This *intrinsic* formulation of the dynamics obviates the need for a preservation theorem: a transition relating a well-typed to an ill-typed expression is ruled out *a priori*. All that remains is the progress theorem, itself is a consequence of the canonical forms lemma, which characterizes the shapes of the well-formed values of a type.

Once safety has been proved, the extrinsic formulation and the intrinsic formulation coincide. The difference is solely a matter of methodology. The extrinsic formulation applies whenever type information is irrelevant to the execution. This need not always be the case. Consequently, we shall at times use either formulation, or a mixture of the two, as the situation demands.

## 6.5 Notes

The concept of type safety as it is understood today was first formulated by Milner (1978), who invented the slogan "well-typed programs do not go wrong." Whereas Milner relied on an explicit notion of "going wrong" to express the concept of a type error, Wright and Felleisen (1994) observed that we can instead show that ill-defined states cannot arise in a well-typed program, giving rise to the slogan "well-typed programs do not get stuck." However, their formulation relied on an analysis showing that no stuck state is well-typed. This analysis is replaced by the progress theorem given here, which relies on the concept of canonical forms introduced by Martin-Löf (1980).

## Exercises

**6.1**. Complete the proof of Theorem 6.2 in full detail.

**6.2**. Complete the proof of Theorem 6.4 in full detail.

**6.3**. Give several cases of the proof of Theorem 6.5 to illustrate how checked errors are accounted for in type safety proofs.

Preview

Preview

# Chapter 7

# Evaluation Dynamics

In Chapter 5 we defined the evaluation of expressions in **E** using the method of structural dynamics. This approach is useful as a foundation for proving properties of a language, but other methods are often more appropriate for other purposes, such as writing user manuals. Another method, called *evaluation dynamics* presents the dynamics as a relation between a phrase and its value, without detailing how it is to be determined in a step-by-step manner. Evaluation dynamics suppresses the step-by-step details of determining the value of an expression, and hence does not provide any useful notion of the time complexity of a program. *Cost dynamics* rectifies this by augmenting evaluation dynamics with a *cost measure*. Various cost measures may be assigned to an expression. One example is the number of steps in the structural dynamics required for an expression to reach a value.

## 7.1 Evaluation Dynamics

An *evaluation dynamics*, consists of an inductive definition of the evaluation judgment, $e \Downarrow v$, stating that the closed expression, $e$, evaluates to the value, $v$. The evaluation dynamics of **E** is defined by the following rules:

$$\frac{}{\mathtt{num}[n] \Downarrow \mathtt{num}[n]} \tag{7.1a}$$

$$\frac{}{\mathtt{str}[s] \Downarrow \mathtt{str}[s]} \tag{7.1b}$$

$$\frac{e_1 \Downarrow \mathtt{num}[n_1] \quad e_2 \Downarrow \mathtt{num}[n_2] \quad n_1 + n_2 = n \ \mathsf{nat}}{\mathtt{plus}(e_1; e_2) \Downarrow \mathtt{num}[n]} \tag{7.1c}$$

$$\frac{e_1 \Downarrow \mathtt{str}[s_1] \quad e_2 \Downarrow \mathtt{str}[s_2] \quad s_1 \hat{} s_2 = s \ \mathsf{str}}{\mathtt{cat}(e_1; e_2) \Downarrow \mathtt{str}[s]} \tag{7.1d}$$

$$\frac{e \Downarrow \mathtt{str}[s] \quad |s| = n \ \mathsf{nat}}{\mathtt{len}(e) \Downarrow \mathtt{num}[n]} \tag{7.1e}$$

Preview

$$\frac{[e_1/x]e_2 \Downarrow v_2}{\texttt{let}(e_1; x.e_2) \Downarrow v_2} \tag{7.1f}$$

The value of a `let` expression is determined by substitution of the binding into the body. The rules are therefore not syntax-directed, because the premise of rule (7.1f) is not a sub-expression of the expression in the conclusion of that rule.

Rule (7.1f) specifies a by-name interpretation of definitions. For a by-value interpretation the following rule should be used instead:

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v_2}{\texttt{let}(e_1; x.e_2) \Downarrow v_2} \tag{7.2}$$

Because the evaluation judgment is inductively defined, we prove properties of it by rule induction. Specifically, to show that the property $\mathcal{P}(e \Downarrow v)$ holds, it is enough to show that $\mathcal{P}$ is closed under rules (7.1):

1. Show that $\mathcal{P}(\texttt{num}[n] \Downarrow \texttt{num}[n])$.
2. Show that $\mathcal{P}(\texttt{str}[s] \Downarrow \texttt{str}[s])$.
3. Show that $\mathcal{P}(\texttt{plus}(e_1; e_2) \Downarrow \texttt{num}[n])$, if $\mathcal{P}(e_1 \Downarrow \texttt{num}[n_1])$, $\mathcal{P}(e_2 \Downarrow \texttt{num}[n_2])$, and $n_1 + n_2 = n$ nat.
4. Show that $\mathcal{P}(\texttt{cat}(e_1; e_2) \Downarrow \texttt{str}[s])$, if $\mathcal{P}(e_1 \Downarrow \texttt{str}[s_1])$, $\mathcal{P}(e_2 \Downarrow \texttt{str}[s_2])$, and $s_1 \hat{} s_2 = s$ str.
5. Show that $\mathcal{P}(\texttt{let}(e_1; x.e_2) \Downarrow v_2)$, if $\mathcal{P}([e_1/x]e_2 \Downarrow v_2)$.

This induction principle is *not* the same as structural induction on $e$ itself, because the evaluation rules are not syntax-directed.

**Lemma 7.1.** *If $e \Downarrow v$, then $v$ val.*

*Proof.* By induction on rules (7.1). All cases except rule (7.1f) are immediate. For the latter case, the result follows directly by an appeal to the inductive hypothesis for the premise of the evaluation rule. □

## 7.2 Relating Structural and Evaluation Dynamics

We have given two different forms of dynamics for **E**. It is natural to ask whether they are equivalent, but to do so first requires that we consider carefully what we mean by equivalence. The structural dynamics describes a step-by-step process of execution, whereas the evaluation dynamics suppresses the intermediate states, focusing attention on the initial and final states alone. This suggests that the appropriate correspondence is between *complete* execution sequences in the structural dynamics and the evaluation judgment in the evaluation dynamics.

**Theorem 7.2.** *For all closed expressions $e$ and values $v$, $e \longmapsto^* v$ iff $e \Downarrow v$.*



Preview

How might we prove such a theorem? We will consider each direction separately. We consider the easier case first.

**Lemma 7.3.** *If $e \Downarrow v$, then $e \longmapsto^* v$.*

*Proof.* By induction on the definition of the evaluation judgment. For example, suppose that $\mathtt{plus}(e_1;e_2) \Downarrow \mathtt{num}[n]$ by the rule for evaluating additions. By induction we know that $e_1 \longmapsto^* \mathtt{num}[n_1]$ and $e_2 \longmapsto^* \mathtt{num}[n_2]$. We reason as follows:

$$\begin{array}{rcl} \mathtt{plus}(e_1;e_2) & \longmapsto^* & \mathtt{plus}(\mathtt{num}[n_1];e_2) \\ & \longmapsto^* & \mathtt{plus}(\mathtt{num}[n_1];\mathtt{num}[n_2]) \\ & \longmapsto & \mathtt{num}[n_1+n_2] \end{array}$$

Therefore $\mathtt{plus}(e_1;e_2) \longmapsto^* \mathtt{num}[n_1+n_2]$, as required. The other cases are handled similarly. □

For the converse, recall from Chapter 5 the definitions of multi-step evaluation and complete evaluation. Because $v \Downarrow v$ whenever $v$ val, it suffices to show that evaluation is closed under converse evaluation:[1]

**Lemma 7.4.** *If $e \longmapsto e'$ and $e' \Downarrow v$, then $e \Downarrow v$.*

*Proof.* By induction on the definition of the transition judgment. For example, suppose that $\mathtt{plus}(e_1;e_2) \longmapsto \mathtt{plus}(e_1';e_2)$, where $e_1 \longmapsto e_1'$. Suppose further that $\mathtt{plus}(e_1';e_2) \Downarrow v$, so that $e_1' \Downarrow \mathtt{num}[n_1]$, $e_2 \Downarrow \mathtt{num}[n_2]$, $n_1+n_2=n$ nat, and $v$ is $\mathtt{num}[n]$. By induction $e_1 \Downarrow \mathtt{num}[n_1]$, and hence $\mathtt{plus}(e_1;e_2) \Downarrow \mathtt{num}[n]$, as required. □

## 7.3 Type Safety, Revisited

Type safety is defined in Chapter 6 as the conjunction of preservation and progress (Theorem 6.1). These concepts are meaningful when applied to a dynamics given by a transition system, as we shall do throughout this book. But what if we had instead given the dynamics as an evaluation relation? How is type safety to be proved in that case?

The answer, unfortunately, is that we cannot. Although there is an analogue of the preservation property for an evaluation dynamics, there is no clear analogue of the progress property. Preservation may be stated as saying that if $e \Downarrow v$ and $e : \tau$, then $v : \tau$. This can be readily proved by induction on the evaluation rules. But what is the analogue of progress? We might be tempted to phrase progress as saying that if $e : \tau$, then $e \Downarrow v$ for some $v$. Although this property is true for **E**, it demands much more than just progress — it requires that every expression evaluate to a value! If **E** were extended to admit operations that may result in an error (as discussed in Section 6.3), or to admit non-terminating expressions, then this property would fail, even though progress would remain valid.

One possible attitude towards this situation is to simply conclude that type safety cannot be properly discussed in the context of an evaluation dynamics, but only by reference to a structural dynamics. Another point of view is to instrument the dynamics with explicit checks for dynamic

[1] Converse evaluation is also known as *head expansion*.

Preview

type errors, and to show that any expression with a dynamic type fault must be statically ill-typed. Re-stated in the contrapositive, this means that a statically well-typed program cannot incur a dynamic type error. A difficulty with this point of view is that we must explicitly account for a form of error solely to prove that it cannot arise! Nevertheless, we will press on to show how a semblance of type safety can be established using evaluation dynamics.

The main idea is to define a judgment $e\Uparrow$ stating, in the jargon of the literature, that the expression *e goes wrong* when executed. The exact definition of "going wrong" is given by a set of rules, but the intention is that it should cover all situations that correspond to type errors. The following rules are representative of the general case:

$$\frac{}{\mathtt{plus}(\mathtt{str}[s]; e_2)\Uparrow} \tag{7.3a}$$

$$\frac{e_1\ \mathsf{val}}{\mathtt{plus}(e_1; \mathtt{str}[s])\Uparrow} \tag{7.3b}$$

These rules explicitly check for the misapplication of addition to a string; similar rules govern each of the primitive constructs of the language.

**Theorem 7.5.** *If $e\Uparrow$, then there is no $\tau$ such that $e : \tau$.*

*Proof.* By rule induction on rules (7.3). For example, for rule (7.3a), we observe that $\mathtt{str}[s] : \mathtt{str}$, and hence $\mathtt{plus}(\mathtt{str}[s]; e_2)$ is ill-typed. □

**Corollary 7.6.** *If $e : \tau$, then $\neg(e\Uparrow)$.*

Apart from the inconvenience of having to define the judgment $e\Uparrow$ only to show that it is irrelevant for well-typed programs, this approach suffers a very significant methodological weakness. If we should omit one or more rules defining the judgment $e\Uparrow$, the proof of Theorem 7.5 remains valid; there is nothing to ensure that we have included sufficiently many checks for run-time type errors. We can prove that the ones we define cannot arise in a well-typed program, but we cannot prove that we have covered all possible cases. By contrast the structural dynamics does not specify any behavior for ill-typed expressions. Consequently, any ill-typed expression will "get stuck" without our explicit intervention, and the progress theorem rules out all such cases. Moreover, the transition system corresponds more closely to implementation—a compiler need not make any provisions for checking for run-time type errors. Instead, it relies on the statics to ensure that these cannot arise, and assigns no meaning to any ill-typed program. Execution is therefore more efficient, and the language definition is simpler.

## 7.4 Cost Dynamics

A structural dynamics provides a natural notion of *time complexity* for programs, namely the number of steps required to reach a final state. An evaluation dynamics, on the other hand, does not provide such a direct notion of complexity. Because the individual steps required to complete an evaluation are suppressed, we cannot directly read off the number of steps required to evaluate to a value. Instead we must augment the evaluation relation with a cost measure, resulting in a *cost dynamics*.



Preview

Evaluation judgments have the form $e \Downarrow^k v$, with the meaning that $e$ evaluates to $v$ in $k$ steps.

$$\frac{}{\mathtt{num}[n] \Downarrow^0 \mathtt{num}[n]} \tag{7.4a}$$

$$\frac{e_1 \Downarrow^{k_1} \mathtt{num}[n_1] \quad e_2 \Downarrow^{k_2} \mathtt{num}[n_2]}{\mathtt{plus}(e_1; e_2) \Downarrow^{k_1+k_2+1} \mathtt{num}[n_1 + n_2]} \tag{7.4b}$$

$$\frac{}{\mathtt{str}[s] \Downarrow^0 \mathtt{str}[s]} \tag{7.4c}$$

$$\frac{e_1 \Downarrow^{k_1} s_1 \quad e_2 \Downarrow^{k_2} s_2}{\mathtt{cat}(e_1; e_2) \Downarrow^{k_1+k_2+1} \mathtt{str}[s_1 \hat{} s_2]} \tag{7.4d}$$

$$\frac{[e_1/x]e_2 \Downarrow^{k_2} v_2}{\mathtt{let}(e_1; x.e_2) \Downarrow^{k_2+1} v_2} \tag{7.4e}$$

For a by-value interpretation of `let`, rule (7.4e) should be replaced by the following rule:

$$\frac{e_1 \Downarrow^{k_1} v_1 \quad [v_1/x]e_2 \Downarrow^{k_2} v_2}{\mathtt{let}(e_1; x.e_2) \Downarrow^{k_1+k_2+1} v_2} \tag{7.5}$$

**Theorem 7.7.** *For any closed expression $e$ and closed value $v$ of the same type, $e \Downarrow^k v$ iff $e \longmapsto^k v$.*

*Proof.* From left to right proceed by rule induction on the definition of the cost dynamics. From right to left proceed by induction on $k$, with an inner rule induction on the definition of the structural dynamics. □

## 7.5 Notes

The structural similarity between evaluation dynamics and typing rules was first developed in the definition of Standard ML (Milner et al., 1997). The advantage of evaluation semantics is that it directly defines the relation of interest, that between a program and its outcome. The disadvantage is that it is not as well-suited to metatheory as structural semantics, precisely because it glosses over the fine structure of computation. The concept of a cost dynamics was introduced by Blelloch and Greiner (1996) in their study of parallelism (discussed further in Chapter 40).

## Exercises

**7.1.** Show that evaluation is deterministic: if $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 = v_2$.

**7.2.** Complete the proof of Lemma 7.3.

**7.3.** Complete the proof of Lemma 7.4. Then show that if $e \longmapsto^* e'$ with $e'$ val, then $e \Downarrow e'$.



Preview

**7.4**. Augment the evaluation dynamics with checked errors, along the lines sketched in Chapter 5, using $e\Uparrow$ to indicate that $e$ incurs a checked (or an unchecked) error. What remains unsatisfactory about the type safety proof? Can you think of a better alternative?

**7.5**. Consider generic hypothetical judgments of the form

$$x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v$$

where $v_1$ val, $\ldots$, $v_n$ val, and $v$ val. The hypotheses, written $\Delta$, are called the *environment* of the evaluation; they provide the values of the free variables in $e$. The hypothetical judgment $\Delta \vdash e \Downarrow v$ is called an *environmental evaluation dynamics*.

Give a hypothetical inductive definition of the environmental evaluation dynamics *without making any use of substitution*. In particular, you should include the rule

$$\frac{}{\Delta, x \Downarrow v \vdash x \Downarrow v}$$

defining the evaluation of a free variable.

Show that $x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v$ iff $[v_1, \ldots, v_n / x_1, \ldots, x_n]e \Downarrow v$ (using the by-value form of evaluation).

Preview

# Part III

# Total Functions

Preview

Preview

# Chapter 8

# Function Definitions and Values

In the language **E** we may perform calculations such as the doubling of a given expression, but we cannot express doubling as a concept in itself. To capture the general pattern of doubling, we abstract away from the particular number being doubled using a *variable* to stand for a fixed, but unspecified, number, to express the doubling of an arbitrary number. Any particular instance of doubling may then be obtained by substituting a numeric expression for that variable. In general an expression may involve many distinct variables, necessitating that we specify which of several possible variables is varying in a particular context, giving rise to a *function* of that variable.

In this chapter we will consider two extensions of **E** with functions. The first, and perhaps most obvious, extension is by adding *function definitions* to the language. A function is defined by binding a name to an abt with a bound variable that serves as the argument of that function. A function is *applied* by substituting a particular expression (of suitable type) for the bound variable, obtaining an expression.

The domain and range of defined functions are limited to the types `nat` and `str`, because these are the only types of expression. Such functions are called *first-order* functions, in contrast to *higher-order functions*, which permit functions as arguments and results of other functions. Because the domain and range of a function are types, this requires that we introduce *function types* whose elements are functions. Consequently, we may form functions of *higher type*, those whose domain and range may themselves be function types.

Historically the introduction of higher-order functions was responsible for a mistake in language design that subsequently was re-characterized as a feature, called *dynamic binding*. Dynamic binding arises from getting the definition of substitution wrong by failing to avoid capture. This makes the names of bound variables important, in violation of the fundamental principle of binding stating that the names of bound variables are unimportant.

Preview

## 8.1 First-Order Functions

The language **ED** is the extension of **E** with function definitions and function applications as described by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Exp | $e$ | ::= | $\texttt{apply}[f](e)$ | $f(e)$ | application |
| | | | $\texttt{fun}[\tau_1;\tau_2](x_1.e_2; f.e)$ | $\texttt{fun}\, f(x_1:\tau_1):\tau_2 = e_2\, \texttt{in}\, e$ | definition |

The expression $\texttt{fun}[\tau_1;\tau_2](x_1.e_2; f.e)$ binds the function name $f$ within $e$ to the pattern $x_1.e_2$, which has parameter $x_1$ and definition $e_2$. The domain and range of the function are, respectively, the types $\tau_1$ and $\tau_2$. The expression $\texttt{apply}[f](e)$ instantiates the binding of $f$ with the argument $e$.

The statics of **ED** defines two forms of judgment:

1. Expression typing, $e : \tau$, stating that $e$ has type $\tau$;
2. Function typing, $f(\tau_1) : \tau_2$, stating that $f$ is a function with argument type $\tau_1$ and result type $\tau_2$.

The judgment $f(\tau_1) : \tau_2$ is called the *function header* of $f$; it specifies the domain type and the range type of a function.

The statics of **ED** is defined by the following rules:

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2 \quad \Gamma, f(\tau_1) : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{fun}[\tau_1;\tau_2](x_1.e_2; f.e) : \tau} \tag{8.1a}$$

$$\frac{\Gamma \vdash f(\tau_1) : \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{apply}[f](e) : \tau_2} \tag{8.1b}$$

*Function substitution*, written $[\![x.e/f]\!]e'$, is defined by induction on the structure of $e'$ much like the definition of ordinary substitution. However, a function name, $f$, is not a form of expression, but rather can only occur in an application of the form $\texttt{apply}[f](e)$. Function substitution for such expressions is defined by the following rule:

$$\overline{[\![x.e/f]\!]\texttt{apply}[f](e') = \texttt{let}([\![x.e/f]\!]e'; x.e)} \tag{8.2}$$

At application sites to $f$ with argument $e'$, function substitution yields a `let` expression that binds $x$ to the result of expanding any further applications to $f$ within $e'$.

**Lemma 8.1.** *If* $\Gamma, f(\tau_1) : \tau_2 \vdash e : \tau$ *and* $\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2$, *then* $\Gamma \vdash [\![x_1.e_2/f]\!]e : \tau$.

*Proof.* By rule induction on the first premise, similarly to the proof of Lemma 4.4. □

The dynamics of **ED** is defined using function substitution:

$$\overline{\texttt{fun}[\tau_1;\tau_2](x_1.e_2; f.e) \longmapsto [\![x_1.e_2/f]\!]e} \tag{8.3}$$



Preview

Because function substitution replaces all applications of $f$ by appropriate `let` expressions, there is no need to give a rule for application expressions (essentially, they behave like variables that are replaced during evaluation, rather than like a primitive operation of the language.)

The safety of **ED** may, with some effort, be derived from the safety theorem for higher-order functions, which we discuss next.

## 8.2 Higher-Order Functions

The similarity between variable definitions and function definitions in **ED** is striking. Is it possible to consolidate them? The gap that must be bridged is the segregation of functions from expressions. A function name $f$ is bound to an abstractor $x.e$ specifying a pattern that is instantiated when $f$ is applied. To reduce function definitions to ordinary definitions, we *reify* the abstractor into a form of expression, called a $\lambda$-*abstraction*, written $\mathtt{lam}[\tau_1](x.e)$. Applications generalize to $\mathtt{ap}(e_1;e_2)$, where $e_1$ is an expression denoting a function, and not just a function name. $\lambda$-abstraction and application are the introduction and elimination forms for the *function type*, $\mathtt{arr}(\tau_1;\tau_2)$, which classifies functions with domain $\tau_1$ and range $\tau_2$.

The language **EF** is the enrichment of **E** with function types, as specified by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | ::= | $\mathtt{arr}(\tau_1;\tau_2)$ | $\tau_1 \to \tau_2$ | function |
| Exp | $e$ | ::= | $\mathtt{lam}[\tau](x.e)$ | $\lambda\,(x{:}\tau)\,e$ | abstraction |
| | | | $\mathtt{ap}(e_1;e_2)$ | $e_1(e_2)$ | application |

Because functions are a form of expression that can be used like any other, they are said to be *first-class*, whereas the restriction of functions to declarations renders them *second-class*. First-class functions may be passed as arguments to, and returned as results from, other functions. First-class functions are thefore *higher-order*, rather than *first-order*. It is possible to extend second-class (declared) functions to take functions as arguments, but it is not possible to admit functions as results without allowing functions to be first-class.

The statics of **EF** is given by extending rules (4.1) with the following rules:

$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e) : \mathtt{arr}(\tau_1;\tau_2)} \tag{8.4a}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{8.4b}$$

**Lemma 8.2** (Inversion)**.** *Suppose that* $\Gamma \vdash e : \tau$.

1. *If* $e = \mathtt{lam}[\tau_1](x.e_2)$*, then* $\tau = \mathtt{arr}(\tau_1;\tau_2)$ *and* $\Gamma, x:\tau_1 \vdash e_2 : \tau_2$.
2. *If* $e = \mathtt{ap}(e_1;e_2)$*, then there exists* $\tau_2$ *such that* $\Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau)$ *and* $\Gamma \vdash e_2 : \tau_2$.

*Proof.* The proof proceeds by rule induction on the typing rules. Observe that for each rule, exactly one case applies, and that the premises of the rule in question provide the required result. □

**Lemma 8.3** (Substitution)**.** *If* $\Gamma, x:\tau \vdash e' : \tau'$*, and* $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash [e/x]e' : \tau'$.

Preview

*Proof.* By rule induction on the derivation of the first judgment. □

The dynamics of **EF** extends that of **E** with the following additional rules:

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{8.5a}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{ap}(e_1;e_2) \longmapsto \mathtt{ap}(e_1';e_2)} \tag{8.5b}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathtt{ap}(e_1;e_2) \longmapsto \mathtt{ap}(e_1;e_2')}\right] \tag{8.5c}$$

$$\frac{[e_2\ \mathsf{val}]}{\mathtt{ap}(\mathtt{lam}[\tau_2](x.e_1);e_2) \longmapsto [e_2/x]e_1} \tag{8.5d}$$

The bracketed rule and premise are to be included for a *call-by-value* interpretation of function application, and excluded for a *call-by-name* interpretation.[1]

In the presence of first-class functions, function declarations can be reduced to ordinary definitions by replacing the function definition $\mathtt{fun}\, f(x_1 : \tau_1) : \tau_2 = e_2\ \mathtt{in}\, e$ by the ordinary definition $\mathtt{let}\, \lambda\,(x{:}\tau_1)\, e_2\, \mathtt{be}\, f\ \mathtt{in}\, e$, and replacing the second-class function application $f(e)$ by the first-class function application $f(e)$. Because $\lambda$-abstractions are values, it makes no difference whether the definition is evaluated by-value or by-name for this replacement to make sense. However, using ordinary definitions we may, for example, give a name to a partially applied function, as in the following example:

$$\begin{array}{l}\mathtt{let}\ k\ \mathtt{be}\ \lambda\,(x_1{:}\mathtt{num})\,\lambda\,(x_2{:}\mathtt{num})\,x_1\\ \mathtt{in\ let}\ kz\ \mathtt{be}\ k(0)\ \mathtt{in}\ kz(3) + kz(5)\,.\end{array}$$

Without first-class functions, we cannot even form the function $k$, which returns a function as result when applied to its first argument.

**Theorem 8.4** (Preservation)**.** *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

*Proof.* The proof is by induction on rules (8.5), which define the dynamics of the language.

Consider rule (8.5d),

$$\frac{}{\mathtt{ap}(\mathtt{lam}[\tau_2](x.e_1);e_2) \longmapsto [e_2/x]e_1}\ .$$

Suppose that $\mathtt{ap}(\mathtt{lam}[\tau_2](x.e_1);e_2) : \tau_1$. By Lemma 8.2 we have $e_2 : \tau_2$ and $x : \tau_2 \vdash e_1 : \tau_1$, so by Lemma 8.3 $[e_2/x]e_1 : \tau_1$.

The other rules governing application are handled similarly. □

[1] Although the term "call-by-value" is accurately descriptive, the origin of the term "call-by-name" remains shrouded in mystery.

Preview

**Lemma 8.5** (Canonical Forms). *If $e : \mathtt{arr}(\tau_1; \tau_2)$ and $e$ val, then $e = \lambda\,(x{:}\tau_1)\,e_2$ for some variable $x$ and expression $e_2$ such that $x : \tau_1 \vdash e_2 : \tau_2$.*

*Proof.* By induction on the typing rules, using the assumption $e$ val. □

**Theorem 8.6** (Progress). *If $e : \tau$, then either $e$ val, or there exists $e'$ such that $e \longmapsto e'$.*

*Proof.* The proof is by induction on rules (8.4). Note that because we consider only closed terms, there are no hypotheses on typing derivations.

Consider rule (8.4b) (under the by-name interpretation). By induction either $e_1$ val or $e_1 \longmapsto e_1'$. In the latter case we have $\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1'; e_2)$. In the former case, we have by Lemma 8.5 that $e_1 = \mathtt{lam}[\tau_2](x.e)$ for some $x$ and $e$. But then $\mathtt{ap}(e_1; e_2) \longmapsto [e_2/x]e$. □

## 8.3 Evaluation Dynamics and Definitional Equality

An inductive definition of the evaluation judgment $e \Downarrow v$ for **EF** is given by the following rules:

$$\frac{}{\mathtt{lam}[\tau](x.e) \Downarrow \mathtt{lam}[\tau](x.e)} \tag{8.6a}$$

$$\frac{e_1 \Downarrow \mathtt{lam}[\tau](x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1; e_2) \Downarrow v} \tag{8.6b}$$

It is easy to check that if $e \Downarrow v$, then $v$ val, and that if $e$ val, then $e \Downarrow e$.

**Theorem 8.7.** *$e \Downarrow v$ iff $e \longmapsto^* v$ and $v$ val.*

*Proof.* In the forward direction we proceed by rule induction on rules (8.6), following along similar lines as the proof of Theorem 7.2.

In the reverse direction we proceed by rule induction on rules (5.1). The proof relies on an analogue of Lemma 7.4, which states that evaluation is closed under converse execution, which is proved by induction on rules (8.5). □

Definitional equality for the call-by-name dynamics of **EF** is defined by extension of rules (5.10).

$$\frac{}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau](x.e_2); e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{8.7a}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_2 \to \tau \quad \Gamma \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1; e_2) \equiv \mathtt{ap}(e_1'; e_2') : \tau} \tag{8.7b}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) \equiv \mathtt{lam}[\tau_1](x.e_2') : \tau_1 \to \tau_2} \tag{8.7c}$$

Definitional equality for call-by-value requires a small bit of additional machinery. The main idea is to restrict rule (8.7a) to require that the argument be a value. However, to be fully expressive, we must also widen the concept of a value to include all variables that are in scope, so that

Preview

rule (8.7a) would apply even when the argument is a variable. The justification for this is that in call-by-value, the parameter of a function stands for the value of its argument, and not for the argument itself. The call-by-value definitional equality judgment has the form

$$\Gamma \vdash e_1 \equiv e_2 : \tau,$$

where $\Gamma$ consists of paired hypotheses $x : \tau, x\ \mathsf{val}$ stating, for each variable $x$ in scope, its type and that it is a value. We write $\Gamma \vdash e\ \mathsf{val}$ to indicate that $e$ is a value under these hypotheses, so that, for example, $x : \tau, x\ \mathsf{val} \vdash x\ \mathsf{val}$. (The typing hypothesis is irrelevant, but harmless, to the value judgment.)

## 8.4 Dynamic Scope

The dynamics of function application given by rules (8.5) is defined only for expressions without free variables. When a function is applied, the argument is substituted for the function parameter, ensuring that the result remains closed. Moreover, because substitution of closed expressions can never incur capture, the scopes of variables are not disturbed by the dynamics, ensuring that the principles of binding and scope described in Chapter 1 are respected. This treatment of variables is called *static scoping*, or *static binding*, to contrast it with an alternative approach that we now describe.

Another approach, called *dynamic scoping*, or *dynamic binding*, is sometimes advocated as an alternative to static binding. Evaluation is defined for expressions that may contain free variables. Evaluation of a variable is undefined; it is an error to ask for the value of an unbound variable. Function application is defined similarly to static binding, *except* that when a function is called, the argument *replaces* the parameter in the body, possibly *incurring*, rather than avoiding, capture of free variables in the argument. (As we will explain shortly, this behavior is considered to be a feature, not a bug!)

The difference between replacement and substitution may be illustrated by example. Let $e$ be the expression $\lambda\,(x{:}\mathtt{str})\,y + |x|$ in which the variable $y$ occurs free, and let $e'$ be the expression $\lambda\,(y{:}\mathtt{str})\,f(y)$ with free variable $f$. If we *substitute* $e$ for $f$ in $e'$ we obtain an expression of the form

$$\lambda\,(y'{:}\mathtt{str})\,(\lambda\,(x{:}\mathtt{str})\,y + |x|)(y'),$$

where the bound variable, $y$, in $e$ has been renamed to some fresh variable $y'$ so as to avoid capture. If we instead *replace* $f$ by $e$ in $e'$ we obtain

$$\lambda\,(y{:}\mathtt{str})\,(\lambda\,(x{:}\mathtt{str})\,y + |x|)(y)$$

in which $y$ is no longer free: it has been captured during replacement.

The implications of this seemingly small change to the dynamics of the language are far-reaching. The most obvious implication is that the language is not type safe. In the above example we have that $y : \mathtt{nat} \vdash e : \mathtt{str} \to \mathtt{nat}$, and that $f : \mathtt{str} \to \mathtt{nat} \vdash e' : \mathtt{str} \to \mathtt{nat}$. It follows that $y : \mathtt{nat} \vdash [e/f]e' : \mathtt{str} \to \mathtt{nat}$, but it is easy to see that the result of replacing $f$ by $e$ in $e'$ is ill-typed, regardless of what assumption we make about $y$. The difficulty, of course, is that the bound occurrence of $y$ in $e'$ has type $\mathtt{str}$, whereas the free occurrence in $e$ must have type $\mathtt{nat}$ in order for $e$ to be well-formed.



Preview

One way around this difficulty is to ignore types altogether, and rely on run-time checks to ensure that "stuck" states become checked errors. (See Chapter 18 for a full exploration of this approach.) But even if we ignore worries about safety, we are still left with the serious problem that the names of bound variables matter, and cannot be altered without changing the meaning of a program. So, for example, to use expression $e'$, we must bear in mind that the parameter, $f$, occurs within the scope of a binder for $y$, a fact that is not revealed by the type of $e'$ (and certainly not if we disregard types entirely!) If we change $e'$ so that it binds a different variable, say $z$, then we must correspondingly change $e$ to ensure that it refers to $z$, and not $y$, in order to preserve the overall behavior of the system of two expressions. This means that $e$ and $e'$ must be developed in tandem, violating a basic principle of modular decomposition. (For more on dynamic scope, please see Chapter 34.)

## 8.5 Notes

Nearly all programming languages provide some form of function definition mechanism of the kind illustrated here. The main point of the present account is to demonstrate that a more natural, and more powerful, approach is to separate the generic concept of a definition from the specific concept of a function. Function types codify the general notion in a systematic manner that encompasses function definitions as a special case, and moreover, admits passing functions as arguments and returning them as results without special provision. The essential contribution of Church's $\lambda$-calculus (Church, 1941) was to take the notion of function as primary, and to demonstrate that nothing more is needed to obtain a fully expressive programming language.

## Exercises

**8.1**. Formulate an environmental evaluation dynamics (see Exercise 7.5) for **ED**. *Hint*: Introduce a new form of judgment for evaluation of function identifiers.

**8.2**. Consider an environmental dynamics for **EF**, which includes higher-order functions. What difficulties do you encounter? Can you think of a way to evade these difficulties? *Hint*: One approach is to "substitute away" all free variables in a $\lambda$-abstraction at the point at which it is evaluated. The second is to "freeze" the values of each of the free variables in a $\lambda$-abstraction, and to "thaw" them whenever such a function is applied. What problems do you encounter in each case?

**8.3**. Dynamic scoping conflicts with the identification convention stated in Chapter 1. Give an example of an expression $e$ that evaluates, under dynamic scoping, to a value $v$, and an expression $e'$ such that $e =_\alpha e'$, and yet $e'$ evaluates to $v'$, under dynamic scoping, with $v \neq_\alpha v'$.

Preview

Preview

# Chapter 9

# Gödel's T

The language **T**, well-known as *Gödel's T*, is the combination of function types with the type of natural numbers. In contrast to **E**, which equips the naturals with some arbitrarily chosen arithmetic operations, the language **T** provides a general mechanism, called *primitive recursion*, from which these primitives may be defined. Primitive recursion captures the essential inductive character of the natural numbers, and hence may be seen as an intrinsic termination proof for each program in the language. Consequently, we may only define *total* functions in the language, those that always return a value for each argument. In essence every program in **T** "comes equipped" with a proof of its termination. Although this may seem like a shield against infinite loops, it is also a weapon that can be used to show that some programs cannot be written in **T**. To do so would require a master termination proof for every possible program in the language, something that we shall prove does not exist.

## 9.1 Statics

The syntax of **T** is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | ::= | $\mathtt{nat}$ | $\mathtt{nat}$ | naturals |
| | | | $\mathtt{arr}(\tau_1;\tau_2)$ | $\tau_1 \to \tau_2$ | function |
| Exp | $e$ | ::= | $x$ | $x$ | variable |
| | | | $\mathtt{z}$ | $\mathtt{z}$ | zero |
| | | | $\mathtt{s}(e)$ | $\mathtt{s}(e)$ | successor |
| | | | $\mathtt{rec}(e_0; x.y.e_1; e)$ | $\mathtt{rec}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x)\, \mathtt{with}\, y \hookrightarrow e_1\}$ | recursion |
| | | | $\mathtt{lam}[\tau](x.e)$ | $\lambda\,(x{:}\tau)\, e$ | abstraction |
| | | | $\mathtt{ap}(e_1;e_2)$ | $e_1(e_2)$ | application |

We write $\overline{n}$ for the expression $\mathtt{s}(\ldots\mathtt{s}(\mathtt{z}))$, in which the successor is applied $n \geq 0$ times to zero. The expression $\mathtt{rec}(e_0; x.y.e_1; e)$ is called the *recursor*. It represents the $e$-fold iteration of the transformation $x.y.e_1$ starting from $e_0$. The bound variable $x$ represents the predecessor and the bound

Preview

variable $y$ represents the result of the $x$-fold iteration. The "`with`" clause in the concrete syntax for the recursor binds the variable $y$ to the result of the recursive call, as will become apparent shortly.

Sometimes the *iterator*, $\mathtt{iter}(e_0; y.e_1; e)$, is considered as an alternative to the recursor. It has essentially the same meaning as the recursor, except that only the result of the recursive call is bound to $y$ in $e_1$, and no binding is made for the predecessor. Clearly the iterator is a special case of the recursor, because we can always ignore the predecessor binding. Conversely, the recursor is definable from the iterator, provided that we have product types (Chapter 10) at our disposal. To define the recursor from the iterator, we simultaneously compute the predecessor while iterating the specified computation.

The statics of **T** is given by the following typing rules:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{9.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{9.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}} \tag{9.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}(e_0; x.y.e_1; e) : \tau} \tag{9.1d}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e) : \mathtt{arr}(\tau_2; \tau_2)} \tag{9.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{arr}(\tau_2; \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1; e_2) : \tau} \tag{9.1f}$$

As usual, admissibility of the structural rule of substitution is crucially important.

**Lemma 9.1.** *If* $\Gamma \vdash e : \tau$ *and* $\Gamma, x : \tau \vdash e' : \tau'$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

## 9.2 Dynamics

The closed values of **T** are defined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{9.2a}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{s}(e)\ \mathsf{val}} \tag{9.2b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{9.2c}$$

Preview

The premise of rule (9.2b) is to be included for an *eager* interpretation of successor, and excluded for a *lazy* interpretation.

The transition rules for the dynamics of **T** are as follows:

$$\left[\frac{e \longmapsto e'}{\mathtt{s}(e) \longmapsto \mathtt{s}(e')}\right] \tag{9.3a}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1'; e_2)} \tag{9.3b}$$

$$\left[\frac{e_1 \text{ val} \quad e_2 \longmapsto e_2'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1; e_2')}\right] \tag{9.3c}$$

$$\frac{[e_2 \text{ val}]}{\mathtt{ap}(\mathtt{lam}[\tau](x.e); e_2) \longmapsto [e_2/x]e} \tag{9.3d}$$

$$\frac{e \longmapsto e'}{\mathtt{rec}(e_0; x.y.e_1; e) \longmapsto \mathtt{rec}(e_0; x.y.e_1; e')} \tag{9.3e}$$

$$\frac{}{\mathtt{rec}(e_0; x.y.e_1; \mathtt{z}) \longmapsto e_0} \tag{9.3f}$$

$$\frac{\mathtt{s}(e) \text{ val}}{\mathtt{rec}(e_0; x.y.e_1; \mathtt{s}(e)) \longmapsto [e, \mathtt{rec}(e_0; x.y.e_1; e)/x, y]e_1} \tag{9.3g}$$

The bracketed rules and premises are to be included for an eager successor and call-by-value application, and omitted for a lazy successor and call-by-name application. Rules (9.3f) and (9.3g) specify the behavior of the recursor on $\mathtt{z}$ and $\mathtt{s}(e)$. In the former case the recursor reduces to $e_0$, and in the latter case the variable $x$ is bound to the predecessor, $e$, and $y$ is bound to the (unevaluated) recursion on $e$. If the value of $y$ is not required in the rest of the computation, the recursive call will not be evaluated.

**Lemma 9.2** (Canonical Forms)**.** *If $e : \tau$ and $e$ val, then*

1. *If $\tau = \mathtt{nat}$, then $e = \mathtt{s}(\mathtt{s}(\ldots \mathtt{z}))$ for some number $n \geq 0$ occurrences of the successor starting with zero.*
2. *If $\tau = \tau_1 \to \tau_2$, then $e = \lambda\,(x{:}\tau_1)\, e_2$ for some $e_2$.*

**Theorem 9.3** (Safety)**.** 1. *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

2. *If $e : \tau$, then either $e$ val or $e \longmapsto e'$ for some $e'$*

Preview

## 9.3 Definability

A mathematical function $f : \mathbb{N} \to \mathbb{N}$ on the natural numbers is *definable* in **T** iff there exists an expression $e_f$ of type $\mathtt{nat} \to \mathtt{nat}$ such that for every $n \in \mathbb{N}$,

$$e_f(\overline{n}) \equiv \overline{f(n)} : \mathtt{nat}. \tag{9.4}$$

That is, the numeric function $f : \mathbb{N} \to \mathbb{N}$ is definable iff there is an expression $e_f$ of type $\mathtt{nat} \to \mathtt{nat}$ such that, when applied to the numeral representing the argument $n \in \mathbb{N}$, the application is definitionally equal to the numeral corresponding to $f(n) \in \mathbb{N}$.

Definitional equality for **T**, written $\Gamma \vdash e \equiv e' : \tau$, is the strongest congruence containing these axioms:

$$\frac{}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau_1](x.e_2); e_1) \equiv [e_1/x]e_2 : \tau} \tag{9.5a}$$

$$\frac{}{\Gamma \vdash \mathtt{rec}(e_0; x.y.e_1; \mathtt{z}) \equiv e_0 : \tau} \tag{9.5b}$$

$$\frac{}{\Gamma \vdash \mathtt{rec}(e_0; x.y.e_1; \mathtt{s}(e)) \equiv [e, \mathtt{rec}(e_0; x.y.e_1; e)/x, y]e_1 : \tau} \tag{9.5c}$$

For example, the doubling function, $d(n) = 2 \times n$, is definable in **T** by the expression $e_d : \mathtt{nat} \to \mathtt{nat}$ given by

$$\lambda\,(x{:}\mathtt{nat})\,\mathtt{rec}\,x\,\{\mathtt{z} \hookrightarrow \mathtt{z} \mid \mathtt{s}(u)\,\mathtt{with}\,v \hookrightarrow \mathtt{s}(\mathtt{s}(v))\}.$$

To check that this defines the doubling function, we proceed by induction on $n \in \mathbb{N}$. For the basis, it is easy to check that

$$e_d(\overline{0}) \equiv \overline{0} : \mathtt{nat}.$$

For the induction, assume that

$$e_d(\overline{n}) \equiv \overline{d(n)} : \mathtt{nat}.$$

Then calculate using the rules of definitional equality:

$$\begin{aligned} e_d(\overline{n+1}) &\equiv \mathtt{s}(\mathtt{s}(e_d(\overline{n}))) \\ &\equiv \mathtt{s}(\mathtt{s}(\overline{2 \times n})) \\ &= \overline{2 \times (n+1)} \\ &= \overline{d(n+1)}. \end{aligned}$$

As another example, consider the following function, called *Ackermann's function*, defined by the following equations:

$$\begin{aligned} A(0, n) &= n + 1 \\ A(m+1, 0) &= A(m, 1) \\ A(m+1, n+1) &= A(m, A(m+1, n)). \end{aligned}$$

Preview

This function grows very quickly. For example, $A(4,2) \approx 2^{65,536}$, which is often cited as being much larger than the number of atoms in the universe! Yet we can show that the Ackermann function is total by a lexicographic induction on the pair of arguments $(m,n)$. On each recursive call, either $m$ decreases, or else $m$ remains the same, and $n$ decreases, so inductively the recursive calls are well-defined, and hence so is $A(m,n)$.

A *first-order primitive recursive function* is a function of type $\mathtt{nat} \to \mathtt{nat}$ that is defined using the recursor, but without using any higher order functions. Ackermann's function is defined so that it is not first-order primitive recursive, but is higher-order primitive recursive. The key to showing that it is definable in **T** is to observe that $A(m+1,n)$ iterates $n$ times the function $A(m,-)$, starting with $A(m,1)$. As an auxiliary, let us define the higher-order function

$$\mathtt{it} : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$$

to be the $\lambda$-abstraction

$$\lambda\,(f{:}\mathtt{nat} \to \mathtt{nat})\,\lambda\,(n{:}\mathtt{nat})\,\mathtt{rec}\,n\,\{\mathtt{z} \hookrightarrow \mathtt{id} \mid \mathtt{s}(_)\,\mathtt{with}\,g \hookrightarrow f \circ g\},$$

where $\mathtt{id} = \lambda\,(x{:}\mathtt{nat})\,x$ is the identity, and $f \circ g = \lambda\,(x{:}\mathtt{nat})\,f(g(x))$ is the composition of $f$ and $g$. It is easy to check that

$$\mathtt{it}(f)(\overline{n})(\overline{m}) \equiv f^{(n)}(\overline{m}) : \mathtt{nat},$$

where the latter expression is the $n$-fold composition of $f$ starting with $\overline{m}$. We may then define the Ackermann function

$$e_a : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$$

to be the expression

$$\lambda\,(m{:}\mathtt{nat})\,\mathtt{rec}\,m\,\{\mathtt{z} \hookrightarrow \mathtt{s} \mid \mathtt{s}(_)\,\mathtt{with}\,f \hookrightarrow \lambda\,(n{:}\mathtt{nat})\,\mathtt{it}(f)(n)(f(\overline{1}))\}.$$

It is instructive to check that the following equivalences are valid:

$$e_a(\overline{0})(\overline{n}) \equiv \mathtt{s}(\overline{n}) \tag{9.6}$$

$$e_a(\overline{m+1})(\overline{0}) \equiv e_a(\overline{m})(\overline{1}) \tag{9.7}$$

$$e_a(\overline{m+1})(\overline{n+1}) \equiv e_a(\overline{m})(e_a(\mathtt{s}(\overline{m}))(\overline{n})). \tag{9.8}$$

That is, the Ackermann function is definable in **T**.

## 9.4 Undefinability

It is impossible to define an infinite loop in **T**.

**Theorem 9.4.** *If $e : \tau$, then there exists $v$ val such that $e \equiv v : \tau$.*

*Proof.* See Corollary 48.15. □



Preview

Consequently, values of function type in **T** behave like mathematical functions: if $e : \tau_1 \to \tau_2$ and $e_1 : \tau_1$, then $e(e_1)$ evaluates to a value of type $\tau_2$. Moreover, if $e : \mathtt{nat}$, then there exists a natural number $n$ such that $e \equiv \overline{n} : \mathtt{nat}$.

Using this, we can show, using a technique called *diagonalization*, that there are functions on the natural numbers that are not definable in **T**. We make use of a technique, called *Gödel-numbering*, that assigns a unique natural number to each closed expression of **T**. This allows us to manipulate expressions as data values in **T**, and hence permits **T** to compute with its own programs.[1]

The essence of Gödel-numbering is captured by the following simple construction on abstract syntax trees. (The generalization to abstract binding trees is slightly more difficult, the main complication being to ensure that all $\alpha$-equivalent expressions are assigned the same Gödel number.) Recall that a general ast, $a$, has the form $o(a_1, \ldots, a_k)$, where $o$ is an operator of arity $k$. Fix an enumeration of the operators so that every operator has an index $i \in \mathbb{N}$, and let $m$ be the index of $o$ in this enumeration. Define the *Gödel number* $\ulcorner a \urcorner$ of $a$ to be the number

$$2^m \, 3^{n_1} \, 5^{n_2} \, \ldots \, p_k^{n_k},$$

where $p_k$ is the $k$th prime number (so that $p_0 = 2$, $p_1 = 3$, and so on), and $n_1, \ldots, n_k$ are the Gödel numbers of $a_1, \ldots, a_k$, respectively. This assigns a natural number to each ast. Conversely, given a natural number, $n$, we may apply the prime factorization theorem to "parse" $n$ as a unique abstract syntax tree. (If the factorization is not of the appropriate form, which can only be because the arity of the operator does not match the number of factors, then $n$ does not code any ast.)

Now, using this representation, we may define a (mathematical) function $f_{univ} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that, for any $e : \mathtt{nat} \to \mathtt{nat}$, $f_{univ}(\ulcorner e \urcorner)(m) = n$ iff $e(\overline{m}) \equiv \overline{n} : \mathtt{nat}$.[2] The determinacy of the dynamics, together with Theorem 9.4, ensure that $f_{univ}$ is a well-defined function. It is called the *universal function* for **T** because it specifies the behavior of any expression $e$ of type $\mathtt{nat} \to \mathtt{nat}$. Using the universal function, let us define an auxiliary mathematical function, called the *diagonal function* $\delta : \mathbb{N} \to \mathbb{N}$, by the equation $\delta(m) = f_{univ}(m)(m)$. This function is chosen so that $\delta(\ulcorner e \urcorner) = n$ iff $e(\overline{\ulcorner e \urcorner}) \equiv \overline{n} : \mathtt{nat}$. (The motivation for this definition will become apparent in a moment.)

The function $f_{univ}$ is not definable in **T**. Suppose that it were definable by the expression $e_{univ}$, then the diagonal function $\delta$ would be definable by the expression

$$e_\delta = \lambda\,(m{:}\mathtt{nat})\, e_{univ}(m)(m).$$

But in that case we would have the equations

$$\begin{aligned} e_\delta(\overline{\ulcorner e \urcorner}) &\equiv e_{univ}(\overline{\ulcorner e \urcorner})(\overline{\ulcorner e \urcorner}) \\ &\equiv e(\overline{\ulcorner e \urcorner}). \end{aligned}$$

Now let $e_\Delta$ be the function expression

$$\lambda\,(x{:}\mathtt{nat})\, \mathtt{s}(e_\delta(x)),$$

[1] The same technique lies at the heart of the proof of Gödel's celebrated incompleteness theorem. The undefinability of certain functions on the natural numbers within **T** may be seen as a form of incompleteness similar to that considered by Gödel.

[2] The value of $f_{univ}(k)(m)$ may be chosen arbitrarily to be zero when $k$ is not the code of any expression $e$.



Preview

so that we may deduce

$$\begin{aligned} e_\Delta(\overline{\ulcorner e_\Delta \urcorner}) &\equiv \mathtt{s}(e_\delta(\overline{\ulcorner e_\Delta \urcorner})) \\ &\equiv \mathtt{s}(e_\Delta(\overline{\ulcorner e_\Delta \urcorner})). \end{aligned}$$

But the termination theorem implies that there exists $n$ such that $e_\Delta(\overline{\ulcorner e_\Delta \urcorner}) \equiv \overline{n}$, and hence we have $\overline{n} \equiv \mathtt{s}(\overline{n})$, which is impossible.

We say that a language $\mathcal{L}$ is *universal* if it is possible to write an interpreter for $\mathcal{L}$ in $\mathcal{L}$ itself. It is intuitively evident that $f_{univ}$ is computable in the sense that we can define it in *some* sufficiently powerful programming language. But the preceding argument shows that **T** is not up to the task; it is not a universal programming language. Examination of the foregoing proof reveals an inescapable tradeoff: by insisting that all expressions terminate, we necessarily lose universality—there are computable functions that are not definable in the language.

## 9.5 Notes

**T** was introduced by Gödel in his study of the consistency of arithmetic (Gödel, 1980). Gödel showed how to "compile" proofs in arithmetic into well-typed terms of the language **T**, and to reduce the consistency problem for arithmetic to the termination of programs in **T**. This was perhaps the first programming language whose design was directly influenced by the verification (of termination) of its programs.

## Exercises

**9.1**. Prove Lemma 9.2.

**9.2**. Prove Theorem 9.3.

**9.3**. Attempt to prove that if $e : \mathtt{nat}$ is closed, then there exists $n$ such that $e \longmapsto^* \overline{n}$. Where does the proof break down?

**9.4**. Attempt to prove termination for all well-typed closed terms: if $e : \tau$, then there exists $e'$ val such that $e \longmapsto^* e'$. You are free to appeal to Lemma 9.2 and Theorem 9.3 as necessary. Where does the attempt break down? Can you think of a stronger inductive hypothesis that might evade the difficulty?

**9.5**. Define a closed term $e$ of type $\tau$ in **T** to be *hereditarily terminating at type* $\tau$ by induction on the structure of $\tau$ as follows:

(a) If $\tau = \mathtt{nat}$, then $e$ is defined to be hereditarily terminating at type $\tau$ iff $e$ is terminating (that is, iff $e \longmapsto^* \overline{n}$ for some $n$.)

(b) If $\tau = \tau_1 \to \tau_2$, then $e$ is defined to be hereditarily terminating iff whenever $e_1$ is hereditarily terminating at type $\tau_1$, then $e(e_1)$ is hereditarily terminating at type $\tau_2$.



Preview

Attempt to prove hereditary termination for well-typed terms: if $e : \tau$, then $e$ is hereditarily terminating at type $\tau$. The stronger inductive hypothesis bypasses the difficulty encountered in Exercise 9.4, but introduces another obstacle. What is the complication? Can you think of an even stronger inductive hypothesis that would suffice for the proof?

**9.6**. Show that if $e$ is hereditarily terminating at type $\tau$, $e' : \tau$, and $e' \longmapsto e$, then $e$ is also hereditarily terminating at type $\tau$. (The need for this result will be apparent from the solution to Exercise 9.5.)

**9.7**. Define an open, well-typed term

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$$

to be *open hereditarily terminating* iff every substitution instance

$$[e_1, \ldots, e_n / x_1, \ldots, x_n]e$$

is closed hereditarily terminating at type $\tau$ whenever each $e_i$ is closed hereditarily terminating at type $\tau_i$ for each $1 \leq i \leq n$. Derive Exercise 9.3 from this result.

Preview

# Part IV

# Finite Data Types

Preview

Preview

# Chapter 10

# Product Types

The *binary product* of two types consists of *ordered pairs* of values, one from each type in the order specified. The associated eliminatory forms are *projections*, which select the first and second component of a pair. The *nullary product*, or *unit*, type consists solely of the unique "null tuple" of no values, and has no associated eliminatory form. The product type admits both a *lazy* and an *eager* dynamics. According to the lazy dynamics, a pair is a value without regard to whether its components are values; they are not evaluated until (if ever) they are accessed and used in another computation. According to the eager dynamics, a pair is a value only if its components are values; they are evaluated when the pair is created.

More generally, we may consider the *finite product*, $\langle \tau_i \rangle_{i \in I}$, indexed by a finite set of *indices*, $I$. The elements of the finite product type are *I-indexed tuples* whose $i$th component is an element of the type $\tau_i$, for each $i \in I$. The components are accessed by *I-indexed projection* operations, generalizing the binary case. Special cases of the finite product include *n-tuples*, indexed by sets of the form $I = \{0, \ldots, n-1\}$, and *labeled tuples*, or *records*, indexed by finite sets of symbols. Similarly to binary products, finite products admit both an eager and a lazy interpretation.

## 10.1 Nullary and Binary Products

The abstract syntax of products is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{unit}$ | $\mathtt{unit}$ | nullary product |
| | | | $\mathtt{prod}(\tau_1; \tau_2)$ | $\tau_1 \times \tau_2$ | binary product |
| Exp | $e$ | $::=$ | $\mathtt{triv}$ | $\langle\rangle$ | null tuple |
| | | | $\mathtt{pair}(e_1; e_2)$ | $\langle e_1, e_2 \rangle$ | ordered pair |
| | | | $\mathtt{pr}[\mathtt{l}](e)$ | $e \cdot \mathtt{l}$ | left projection |
| | | | $\mathtt{pr}[\mathtt{r}](e)$ | $e \cdot \mathtt{r}$ | right projection |

There is no elimination form for the unit type, there being nothing to extract from the null tuple.

Preview

The statics of product types is given by the following rules.

$$\frac{}{\Gamma \vdash \langle\rangle : \mathtt{unit}} \tag{10.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2} \tag{10.1b}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot \mathtt{l} : \tau_1} \tag{10.1c}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot \mathtt{r} : \tau_2} \tag{10.1d}$$

The dynamics of product types is specified by the following rules:

$$\frac{}{\langle\rangle \ \mathsf{val}} \tag{10.2a}$$

$$\frac{[e_1 \ \mathsf{val}] \quad [e_2 \ \mathsf{val}]}{\langle e_1, e_2\rangle \ \mathsf{val}} \tag{10.2b}$$

$$\left[\frac{e_1 \longmapsto e_1'}{\langle e_1, e_2\rangle \longmapsto \langle e_1', e_2\rangle}\right] \tag{10.2c}$$

$$\left[\frac{e_1 \ \mathsf{val} \quad e_2 \longmapsto e_2'}{\langle e_1, e_2\rangle \longmapsto \langle e_1, e_2'\rangle}\right] \tag{10.2d}$$

$$\frac{e \longmapsto e'}{e \cdot \mathtt{l} \longmapsto e' \cdot \mathtt{l}} \tag{10.2e}$$

$$\frac{e \longmapsto e'}{e \cdot \mathtt{r} \longmapsto e' \cdot \mathtt{r}} \tag{10.2f}$$

$$\frac{[e_1 \ \mathsf{val}] \quad [e_2 \ \mathsf{val}]}{\langle e_1, e_2\rangle \cdot \mathtt{l} \longmapsto e_1} \tag{10.2g}$$

$$\frac{[e_1 \ \mathsf{val}] \quad [e_2 \ \mathsf{val}]}{\langle e_1, e_2\rangle \cdot \mathtt{r} \longmapsto e_2} \tag{10.2h}$$

The bracketed rules and premises are to be omitted for a lazy dynamics, and included for an eager dynamics of pairing.

The safety theorem applies to both the eager and the lazy dynamics, with the proof proceeding along similar lines in each case.

**Theorem 10.1** (Safety)**.** *1. If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

*2. If $e : \tau$ then either $e$ $\mathsf{val}$ or there exists $e'$ such that $e \longmapsto e'$.*

*Proof.* Preservation is proved by induction on transition defined by rules (10.2). Progress is proved by induction on typing defined by rules (10.1). □

Preview

## 10.2 Finite Products

The syntax of finite product types is given by the following grammar:

$$\begin{array}{llllll} \mathsf{Typ} & \tau & ::= & \mathtt{prod}(\{i \hookrightarrow \tau_i\}_{i \in I}) & \langle \tau_i \rangle_{i \in I} & \text{product} \\ \mathsf{Exp} & e & ::= & \mathtt{tpl}(\{i \hookrightarrow e_i\}_{i \in I}) & \langle e_i \rangle_{i \in I} & \text{tuple} \\ & & & \mathtt{pr}[i](e) & e \cdot i & \text{projection} \end{array}$$

The variable $I$ stands for a finite *index set* over which products are formed. The type $\mathtt{prod}(\{i \hookrightarrow \tau_i\}_{i \in I})$, or $\prod_{i \in I} \tau_i$ for short, is the type of *$I$-tuples* of expressions $e_i$ of type $\tau_i$, one for each $i \in I$. An $I$-tuple has the form $\mathtt{tpl}(\{i \hookrightarrow e_i\}_{i \in I})$, or $\langle e_i \rangle_{i \in I}$ for short, and for each $i \in I$ the $i$th projection from an $I$-tuple, $e$, is written $\mathtt{pr}[i](e)$, or $e \cdot i$ for short.

When $I = \{ i_1, \ldots, i_n \}$, the $I$-tuple type may be written in the form

$$\langle i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n \rangle$$

in which we make explicit the association of a type to each index $i \in I$. Similarly, we may write

$$\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle$$

for the $I$-tuple whose $i$th component is $e_i$.

Finite products generalize empty and binary products by choosing $I$ to be empty or the two-element set $\{ \mathtt{l}, \mathtt{r} \}$, respectively. In practice $I$ is often chosen to be a finite set of symbols that serve as labels for the components of the tuple so as to enhance readability.

The statics of finite products is given by the following rules:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle : \langle i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n \rangle} \tag{10.3a}$$

$$\frac{\Gamma \vdash e : \langle i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n \rangle \quad (1 \leq k \leq n)}{\Gamma \vdash e \cdot i_k : \tau_k} \tag{10.3b}$$

In rule (10.3b) the index $i_k \in I$ is a *particular* element of the index set $I$, whereas in rule (10.3a), the indices $i_1, \ldots, i_n$ range over the entire index set $I$.

The dynamics of finite products is given by the following rules:

$$\frac{[e_1 \ \mathsf{val} \quad \ldots \quad e_n \ \mathsf{val}]}{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle \ \mathsf{val}} \tag{10.4a}$$

$$\left[ \frac{\left\{ \begin{array}{c} e_1 \ \mathsf{val} \quad \ldots \quad e_{j-1} \ \mathsf{val} \quad e'_1 = e_1 \quad \ldots \quad e'_{j-1} = e_{j-1} \\ e_j \longmapsto e'_j \quad e'_{j+1} = e_{j+1} \quad \ldots \quad e'_n = e_n \end{array} \right\}}{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle \longmapsto \langle i_1 \hookrightarrow e'_1, \ldots, i_n \hookrightarrow e'_n \rangle} \right] \tag{10.4b}$$

$$\frac{e \longmapsto e'}{e \cdot i \longmapsto e' \cdot i} \tag{10.4c}$$



Preview

$$\frac{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle \ \mathsf{val}}{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle \cdot i_k \longmapsto e_k} \tag{10.4d}$$

As formulated, rule (10.4b) specifies that the components of a tuple are to be evaluated in *some* sequential order, without specifying the order in which the components are considered. It is not hard, but a bit technically complicated, to impose an evaluation order by imposing a total ordering on the index set and evaluating components according to this ordering.

**Theorem 10.2** (Safety)**.** *If $e : \tau$, then either $e$ val or there exists $e'$ such that $e' : \tau$ and $e \longmapsto e'$.*

*Proof.* The safety theorem may be decomposed into progress and preservation lemmas, which are proved as in Section 10.1. □

## 10.3 Primitive Mutual Recursion

In the presence of products we may simplify the primitive recursion construct of **T** so that only the recursive result on the predecessor, and not the predecessor itself, is passed to the successor branch. Writing this as $\mathtt{iter}(e_0; x.e_1; e)$, we may define $\mathtt{rec}(e_0; x.y.e_1; e)$ to be $e' \cdot \mathtt{r}$, where $e'$ is the expression

$$\mathtt{iter}(\langle \mathtt{z}, e_0 \rangle; x'.\langle \mathtt{s}(x' \cdot \mathtt{l}), [x' \cdot \mathtt{r}/x]e_1 \rangle; e).$$

The idea is to compute inductively both the number, $n$, and the result of the recursive call on $n$, from which we can compute both $n+1$ and the result of an additional recursion using $e_1$. The base case is computed directly as the pair of zero and $e_0$. It is easy to check that the statics and dynamics of the recursor are preserved by this definition.

We may also use product types to implement *mutual primitive recursion*, in which we define two functions simultaneously by primitive recursion. For example, consider the following recursion equations defining two mathematical functions on the natural numbers:

$$\begin{aligned} e(0) &= 1 \\ o(0) &= 0 \\ e(n+1) &= o(n) \\ o(n+1) &= e(n) \end{aligned}$$

Intuitively, $e(n)$ is non-zero if and only if $n$ is even, and $o(n)$ is non-zero if and only if $n$ is odd.

To define these functions in **T** enriched with products, we first define an auxiliary function $e_{\mathtt{eo}}$ of type

$$\mathtt{nat} \to (\mathtt{nat} \times \mathtt{nat})$$

that computes both results simultaneously by swapping back and forth on recursive calls:

$$\lambda\,(n{:}\mathtt{nat} \times \mathtt{nat})\,\mathtt{iter}\,n\,\{\mathtt{z} \hookrightarrow \langle 1, 0\rangle \mid \mathtt{s}(b) \hookrightarrow \langle b \cdot \mathtt{r}, b \cdot \mathtt{l}\rangle\}.$$

We may then define $e_{\mathtt{ev}}$ and $e_{\mathtt{od}}$ as follows:

$$\begin{aligned} e_{\mathtt{ev}} &\triangleq \lambda\,(n{:}\mathtt{nat})\,e_{\mathtt{eo}}(n) \cdot \mathtt{l} \\ e_{\mathtt{od}} &\triangleq \lambda\,(n{:}\mathtt{nat})\,e_{\mathtt{eo}}(n) \cdot \mathtt{r}. \end{aligned}$$

Preview

## 10.4 Notes

Product types are the essence of structured data. All languages have some form of product type, but frequently in a form that is combined with other, separable, concepts. Common manifestations of products include: (1) functions with "multiple arguments" or "multiple results"; (2) "objects" represented as tuples of mutually recursive functions; (3) "structures," which are tuples with mutable components. There are many papers on finite product types, which include record types as a special case. Pierce (2002) provides a thorough account of record types, and their subtyping properties (for which, see Chapter 23). Allen et al. (2006) analyzes many of the key ideas in the framework of dependent type theory.

## Exercises

**10.1**. A *database schema* may be thought of as a finite product type $\prod_{i \in I} \tau$, in which the *columns*, or *attributes*. are labelled by the indices $I$ whose values are restricted to *atomic* types, such as `nat` and `str`. A value of a schema type is called a *tuple*, or *instance*, of that schema. A *database* may be thought of as a finite sequence of such tuples, called the *rows* of the database. Give a representation of a database using function, product, and natural numbers types, and define the *project* operation that sends a database with columns $I$ to a database with columns $I' \subseteq I$ by restricting each row to the specified columns.

**10.2**. Rather than choose between a lazy and an eager dynamics for products, we can instead distinguish two forms of product type, called the *positive* and the *negative*. The statics of the negative product is given by rules (10.1), and the dynamics is taken to be lazy. The statics of the positive product, written $\tau_1 \otimes \tau_2$, is given by the following rules:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{fuse}(e_1; e_2) : \tau_1 \otimes \tau_2} \tag{10.5a}$$

$$\frac{\Gamma \vdash e_0 : \tau_1 \otimes \tau_2 \quad \Gamma\, x_1 : \tau_1\, x_2 : \tau_2 \vdash e : \tau}{\Gamma \vdash \mathtt{split}(e_0; x_1, x_2.e) : \tau} \tag{10.5b}$$

The dynamics of `fuse`, the introductory form for the positive pair, is taken to be eager, essentially because the eliminatory form, `split`, extracts both components simultaneously.

Show that the negative product is definable in terms of the positive product using the unit and function types to express the lazy semantics of negative pairing. Show that the positive product is definable in terms of the negative product, provided that we have at our disposal a `let` expression with a by-value dynamics so that we may ensure eager evaluation of positive pairs.

**10.3**. Specializing Exercise **10.2** to nullary products, we obtain a positive and a negative unit type. The negative unit type is given by rules (10.1), with no eliminatory forms and one introductory form. Give the appropriate statics and dynamics for a positive unit type, and show that the positive and negative unit types are interdefinable without any further assumptions.

Preview

Preview

# Chapter 11

# Sum Types

Most data structures involve alternatives such as the distinction between a leaf and an interior node in a tree, or a choice in the outermost form of a piece of abstract syntax. Importantly, the choice determines the structure of the value. For example, nodes have children, but leaves do not, and so forth. These concepts are expressed by *sum types*, specifically the *binary sum*, which offers a choice of two things, and the *nullary sum*, which offers a choice of no things. *Finite sums* generalize nullary and binary sums to permit an arbitrary number of cases indexed by a finite index set. As with products, sums come in both eager and lazy variants, differing in how values of sum type are defined.

## 11.1 Nullary and Binary Sums

The abstract syntax of sums is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{void}$ | $\mathtt{void}$ | nullary sum |
| | | | $\mathtt{sum}(\tau_1;\tau_2)$ | $\tau_1 + \tau_2$ | binary sum |
| Exp | $e$ | $::=$ | $\mathtt{abort}[\tau](e)$ | $\mathtt{abort}(e)$ | abort |
| | | | $\mathtt{in}[\tau_1;\tau_2][\mathtt{l}](e)$ | $\mathtt{l}\cdot e$ | left injection |
| | | | $\mathtt{in}[\tau_1;\tau_2][\mathtt{r}](e)$ | $\mathtt{r}\cdot e$ | right injection |
| | | | $\mathtt{case}(e;x_1.e_1;x_2.e_2)$ | $\mathtt{case}\,e\,\{\mathtt{l}\cdot x_1 \hookrightarrow e_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow e_2\}$ | case analysis |

The nullary sum represents a choice of zero alternatives, and hence admits no introductory form. The eliminatory form, $\mathtt{abort}(e)$, aborts the computation in the event that $e$ evaluates to a value, which it cannot do. The elements of the binary sum type are labeled to indicate whether they are drawn from the left or the right summand, either $\mathtt{l}\cdot e$ or $\mathtt{r}\cdot e$. A value of the sum type is eliminated by case analysis.

The statics of sum types is given by the following rules.

$$\frac{\Gamma \vdash e : \mathtt{void}}{\Gamma \vdash \mathtt{abort}(e) : \tau} \tag{11.1a}$$

Preview

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathtt{l} \cdot e : \tau_1 + \tau_2} \tag{11.1b}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathtt{r} \cdot e : \tau_1 + \tau_2} \tag{11.1c}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}\, e \, \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} : \tau} \tag{11.1d}$$

For the sake of readability, in rules (11.1b) and (11.1c) we have written $\mathtt{l} \cdot e$ and $\mathtt{r} \cdot e$ in place of the abstract syntax $\mathtt{in}[\tau_1;\tau_2][\mathtt{l}](e)$ and $\mathtt{in}[\tau_1;\tau_2][\mathtt{r}](e)$, which includes the types $\tau_1$ and $\tau_2$ explicitly. In rule (11.1d) both branches of the case analysis must have the same type. Because a type expresses a static "prediction" on the form of the value of an expression, and because an expression of sum type could evaluate to either form at run-time, we must insist that both branches yield the same type.

The dynamics of sums is given by the following rules:

$$\frac{e \longmapsto e'}{\mathtt{abort}(e) \longmapsto \mathtt{abort}(e')} \tag{11.2a}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{l} \cdot e\ \mathsf{val}} \tag{11.2b}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{r} \cdot e\ \mathsf{val}} \tag{11.2c}$$

$$\left[\frac{e \longmapsto e'}{\mathtt{l} \cdot e \longmapsto \mathtt{l} \cdot e'}\right] \tag{11.2d}$$

$$\left[\frac{e \longmapsto e'}{\mathtt{r} \cdot e \longmapsto \mathtt{r} \cdot e'}\right] \tag{11.2e}$$

$$\frac{e \longmapsto e'}{\mathtt{case}\, e \, \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \longmapsto \mathtt{case}\, e' \, \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\}} \tag{11.2f}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{case}\, \mathtt{l} \cdot e \, \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \longmapsto [e/x_1]e_1} \tag{11.2g}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{case}\, \mathtt{r} \cdot e \, \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \longmapsto [e/x_2]e_2} \tag{11.2h}$$

The bracketed premises and rules are to be included for an eager dynamics, and excluded for a lazy dynamics.

The coherence of the statics and dynamics is stated and proved as usual.

**Theorem 11.1** (Safety)**.** *1. If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

*2. If $e : \tau$, then either $e$ val or $e \longmapsto e'$ for some $e'$.*

*Proof.* The proof proceeds by induction on rules (11.2) for preservation, and by induction on rules (11.1) for progress. □

Preview

## 11.2 Finite Sums

Just as we may generalize nullary and binary products to finite products, so may we also generalize nullary and binary sums to finite sums. The syntax for finite sums is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{sum}(\{i \hookrightarrow \tau_i\}_{i \in I})$ | $[\tau_i]_{i \in I}$ | sum |
| Exp | $e$ | $::=$ | $\mathtt{in}[\vec{\tau}][i](e)$ | $i \cdot e$ | injection |
| | | | $\mathtt{case}(e; \{i \hookrightarrow x_i.e_i\}_{i \in I})$ | $\mathtt{case}\, e\, \{i \cdot x_i \hookrightarrow e_i\}_{i \in I}$ | case analysis |

The variable $I$ stands for a finite index set over which sums are formed. The notation $\vec{\tau}$ stands for a finite function $\{i \hookrightarrow \tau_i\}_{i \in I}$ for some index set $I$. The type $\mathtt{sum}(\{i \hookrightarrow \tau_i\}_{i \in I})$, or $\sum_{i \in I} \tau_i$ for short, is the type of $I$-classified values of the form $\mathtt{in}[I][i](e_i)$, or $i \cdot e_i$ for short, where $i \in I$ and $e_i$ is an expression of type $\tau_i$. An $I$-classified value is analyzed by an $I$-way case analysis of the form $\mathtt{case}(e; \{i \hookrightarrow x_i.e_i\}_{i \in I})$.

When $I = \{ l_1, \ldots, l_n \}$, the type of $I$-classified values may be written

$$[i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n]$$

specifying the type associated with each class $l_i \in I$. Correspondingly, the $I$-way case analysis has the form

$$\mathtt{case}\, e\, \{i_1 \cdot x_1 \hookrightarrow e_1 \mid \ldots \mid i_n \cdot x_n \hookrightarrow e_n\}.$$

Finite sums generalize empty and binary sums by choosing $I$ to be empty or the two-element set $\{ \mathtt{l}, \mathtt{r} \}$, respectively. In practice $I$ is often chosen to be a finite set of symbols that serve as symbolic names for the classes so as to enhance readability.

The statics of finite sums is defined by the following rules:

$$\frac{\Gamma \vdash e : \tau_k \quad (1 \leq k \leq n)}{\Gamma \vdash i_k \cdot e : [i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n]} \tag{11.3a}$$

$$\frac{\Gamma \vdash e : [i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n] \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \ldots \quad \Gamma, x_n : \tau_n \vdash e_n : \tau}{\Gamma \vdash \mathtt{case}\, e\, \{i_1 \cdot x_1 \hookrightarrow e_1 \mid \ldots \mid i_n \cdot x_n \hookrightarrow e_n\} : \tau} \tag{11.3b}$$

These rules generalize the statics for nullary and binary sums given in Section 11.1.

The dynamics of finite sums is defined by the following rules:

$$\frac{[e\ \mathsf{val}]}{i \cdot e\ \mathsf{val}} \tag{11.4a}$$

$$\left[\frac{e \longmapsto e'}{i \cdot e \longmapsto i \cdot e'}\right] \tag{11.4b}$$

$$\frac{e \longmapsto e'}{\mathtt{case}\, e\, \{i \cdot x_i \hookrightarrow e_i\}_{i \in I} \longmapsto \mathtt{case}\, e'\, \{i \cdot x_i \hookrightarrow e_i\}_{i \in I}} \tag{11.4c}$$

$$\frac{i \cdot e\ \mathsf{val}}{\mathtt{case}\, i \cdot e\, \{i \cdot x_i \hookrightarrow e_i\}_{i \in I} \longmapsto [e/x_i]e_i} \tag{11.4d}$$

These again generalize the dynamics of binary sums given in Section 11.1.

Preview

**Theorem 11.2** (Safety)**.** *If $e : \tau$, then either $e$ val or there exists $e' : \tau$ such that $e \longmapsto e'$.*

*Proof.* The proof is similar to that for the binary case, as described in Section 11.1. □

## 11.3 Applications of Sum Types

Sum types have numerous uses, several of which we outline here. More interesting examples arise once we also have induction and recursive types, which are introduced in Parts V and Part VI.

### 11.3.1 Void and Unit

It is instructive to compare the types `unit` and `void`, which are often confused with one another. The type `unit` has exactly one element, $\langle\rangle$, whereas the type `void` has no elements at all. Consequently, if $e$ : `unit`, then if $e$ evaluates to a value, it must be $\langle\rangle$ — in other words, $e$ has *no interesting value*. On the other hand, if $e$ : `void`, then *$e$ must not yield a value;* if it were to have a value, it would have to be a value of type `void`, of which there are none. This shows that what is called the `void` type in many languages is really the type `unit` because it indicates that an expression has no interesting value, not that it has no value at all!

### 11.3.2 Booleans

Perhaps the simplest example of a sum type is the familiar type of Booleans, whose syntax is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | ::= | `bool` | `bool` | booleans |
| Exp | $e$ | ::= | `true` | `true` | truth |
| | | | `false` | `false` | falsity |
| | | | `if`$(e; e_1; e_2)$ | `if` $e$ `then` $e_1$ `else` $e_2$ | conditional |

The expression `if`$(e; e_1; e_2)$ branches on the value of $e$ : `bool`.

The statics of Booleans is given by the following typing rules:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \tag{11.5a}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}} \tag{11.5b}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \tau} \tag{11.5c}$$

The dynamics is given by the following value and transition rules:

$$\frac{}{\texttt{true}\ \mathsf{val}} \tag{11.6a}$$

Preview

$$\frac{}{\texttt{false val}} \tag{11.6b}$$

$$\frac{}{\texttt{if true then } e_1 \texttt{ else } e_2 \longmapsto e_1} \tag{11.6c}$$

$$\frac{}{\texttt{if false then } e_1 \texttt{ else } e_2 \longmapsto e_2} \tag{11.6d}$$

$$\frac{e \longmapsto e'}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \longmapsto \texttt{if } e' \texttt{ then } e_1 \texttt{ else } e_2} \tag{11.6e}$$

The type `bool` is definable in terms of binary sums and nullary products:

$$\texttt{bool} = \texttt{unit} + \texttt{unit} \tag{11.7a}$$
$$\texttt{true} = \texttt{l} \cdot \langle\rangle \tag{11.7b}$$
$$\texttt{false} = \texttt{r} \cdot \langle\rangle \tag{11.7c}$$
$$\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 = \texttt{case } e \, \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \tag{11.7d}$$

In the last equation above the variables $x_1$ and $x_2$ are chosen arbitrarily such that $x_1 \notin e_1$ and $x_2 \notin e_2$. It is a simple matter to check that the evident statics and dynamics of the type `bool` are engendered by these definitions.

### 11.3.3 Enumerations

More generally, sum types may be used to define *finite enumeration* types, those whose values are one of an explicitly given finite set, and whose elimination form is a case analysis on the elements of that set. For example, the type `suit`, whose elements are ♣, ♢, ♡, and ♠, has as elimination form the case analysis

$$\texttt{case } e \, \{\clubsuit \hookrightarrow e_0 \mid \diamondsuit \hookrightarrow e_1 \mid \heartsuit \hookrightarrow e_2 \mid \spadesuit \hookrightarrow e_3\},$$

which distinguishes among the four suits. Such finite enumerations are easily representable as sums. For example, we may define $\texttt{suit} = [\texttt{unit}]_{_ \in I}$, where $I = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$ and the type family is constant over this set. The case analysis form for a labeled sum is almost literally the desired case analysis for the given enumeration, the only difference being the binding for the uninteresting value associated with each summand, which we may ignore.

Other examples of enumeration types abound. For example, most languages have a type `char` of characters, which is a large enumeration type containing all possible `Unicode` (or other such standard classification) of characters. Each character is assigned a *code* (such as `UTF-8`) that is used for interchange among programs. The type `char` is equipped with operations such as $\texttt{chcode}(n)$ that yield the `char` associated to the code $n$, and $\texttt{codech}(c)$ that yield the code of character $c$. Using the linear ordering on codes we may define a total ordering of characters, called the *collating sequence* determined by that code.

Preview

### 11.3.4 Options

Another use of sums is to define the *option* types, which have the following syntax:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | ::= | $\mathtt{opt}(\tau)$ | $\tau\,\mathtt{opt}$ | option |
| Exp | $e$ | ::= | $\mathtt{null}$ | $\mathtt{null}$ | nothing |
| | | | $\mathtt{just}(e)$ | $\mathtt{just}(e)$ | something |
| | | | $\mathtt{ifnull}[\tau](e; e_1; x.e_2)$ | $\mathtt{check}\,e\,\{\mathtt{null} \hookrightarrow e_1 \mid \mathtt{just}(x) \hookrightarrow e_2\}$ | null test |

The type $\mathtt{opt}(\tau)$ represents the type of "optional" values of type $\tau$. The introductory forms are $\mathtt{null}$, corresponding to "no value", and $\mathtt{just}(e)$, corresponding to a specified value of type $\tau$. The elimination form discriminates between the two possibilities.

The option type is definable from sums and nullary products according to the following equations:[1]

$$\tau\,\mathtt{opt} = \mathtt{unit} + \tau \tag{11.8a}$$

$$\mathtt{null} = \mathtt{l} \cdot \langle\rangle \tag{11.8b}$$

$$\mathtt{just}(e) = \mathtt{r} \cdot e \tag{11.8c}$$

$$\mathtt{check}\,e\,\{\mathtt{null} \hookrightarrow e_1 \mid \mathtt{just}(x_2) \hookrightarrow e_2\} = \mathtt{case}\,e\,\{\mathtt{l} \cdot _ \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \tag{11.8d}$$

We leave it to the reader to examine the statics and dynamics implied by these definitions.

The option type is the key to understanding a common misconception, the *null pointer fallacy*. This fallacy, which is particularly common in object-oriented languages, is based on two related errors. The first error is to deem the values of certain types to be mysterious entities called *pointers*, based on suppositions about how these values might be represented at run-time, rather than on the semantics of the type itself. The second error compounds the first. A particular value of a pointer type is distinguished as *the null pointer*, which, unlike the other elements of that type, does not designate a value of that type at all, but rather rejects all attempts to use it as such.

To help avoid such failures, such languages usually include a function, say $\mathtt{null} : \tau \to \mathtt{bool}$, that yields $\mathtt{true}$ if its argument is null, and $\mathtt{false}$ otherwise. This allows the programmer to take steps to avoid using null as a value of the type it purports to inhabit. Consequently, programs are riddled with conditionals of the form

$$\mathtt{if}\,\mathtt{null}(e)\,\mathtt{then}\ldots \textit{error}\ldots\,\mathtt{else}\ldots \textit{proceed}\ldots. \tag{11.9}$$

Despite this, "null pointer" exceptions at run-time are rampant, in part because it is quite easy to overlook the need for such a test, and in part because detection of a null pointer leaves little recourse other than abortion of the program.

The underlying problem may be traced to the failure to distinguish the type $\tau$ from the type $\tau\,\mathtt{opt}$. Rather than think of the elements of type $\tau$ as pointers, and thereby have to worry about the null pointer, we instead distinguish between a *genuine* value of type $\tau$ and an *optional* value of

[1] We often write an underscore in place of a bound variable that is not used within its scope.



Preview

type $\tau$. An optional value of type $\tau$ may or may not be present, but, if it is, the underlying value is truly a value of type $\tau$ (and cannot be null). The elimination form for the option type,

$$\texttt{check}\, e\, \{\texttt{null} \hookrightarrow e_{error} \mid \texttt{just}(x) \hookrightarrow e_{ok}\}, \tag{11.10}$$

propagates the information that $e$ is present into the non-null branch by binding a genuine value of type $\tau$ to the variable $x$. The case analysis effects a change of type from "optional value of type $\tau$" to "genuine value of type $\tau$", so that within the non-null branch no further null checks, explicit or implicit, are required. Observe that such a change of type is not achieved by the simple Boolean-valued test exemplified by expression (11.9); the advantage of option types is precisely that they do so.

## 11.4 Notes

Heterogeneous data structures are ubiquitous. Sums codify heterogeneity, yet few languages support them in the form given here. The best approximation in commercial languages is the concept of a class in object-oriented programming. A class is an injection into a sum type, and dispatch is case analysis on the class of the data object. (See Chapter 26 for more on this correspondence.) The absence of sums is the origin of C.A.R. Hoare's self-described "billion dollar mistake," the null pointer (Hoare, 2009). Bad language designs put the burden of managing "null" values entirely at run-time, rather than making the possibility or more importantly the *impossibility*, of "null" apparent at compile time.

## Exercises

**11.1**. Complete the definition of a finite enumeration type sketched in Section 11.3.3. Derive enumeration types from finite sum types.

**11.2**. The essence of Hoare's mistake is the misidentification of the type $\tau\,\texttt{opt}$ with the type $\texttt{bool} \times \tau$. Values of the latter type are pairs consisting of a boolean "flag" and a value of type $\tau$. The idea is that the flag indicates whether or not the associated value is "present". When the flag is `true`, the second component is held to be present, and, when the flag is `false`, the second component is held to be absent.

Analyze Hoare's mistake by attempting to define $\tau\,\texttt{opt}$ to be the type $\texttt{bool} \times \tau$ by filling in the following chart:

$$\begin{aligned} \texttt{null} &\triangleq \textbf{?} \\ \texttt{just}(e) &\triangleq \textbf{?} \\ \texttt{check}\, e\, \{\texttt{null} \hookrightarrow e_1 \mid \texttt{just}(x) \hookrightarrow e_2\} &\triangleq \textbf{?} \end{aligned}$$

Argue that *even if* we adopt Hoare's convention of admitting a "null" value of every type, the chart cannot be properly filled.

Preview

**11.3**. Databases exhibit a version of the "null pointer" problem that arises when not every tuple provides a value for every attribute (for example, a person's middle name might be absent). More generally, many commercial databases are limited to a single atomic type for each attribute, presenting problems when the value of that attribute may be of several types (for example, one may have different sorts of postal codes depending on the country). Consider how to address these problems from the point of view of Exercise **10.1**. Suggest how to handle null values and heterogeneous values that avoids some of the complications that arise in traditional formulations of databases.

Preview

# Chapter 12

# Pattern Matching

Pattern matching is a natural and convenient generalization of the elimination forms for product and sum types. For example, rather than write

$$\mathtt{let}\ x\ \mathtt{be}\ e\ \mathtt{in}\ x \cdot \mathtt{l} + x \cdot \mathtt{r}$$

to add the components of a pair, $e$, of natural numbers, we may instead write

$$\mathtt{match}\ e\ \{\langle x_1, x_2\rangle \hookrightarrow x_1 + x_2\},$$

using pattern matching to name the components of the pair and refer to them directly. The first argument to the `match` is called the *match value* and the second argument consist of a finite sequence of *rules*, separated by vertical bars. In this example there is only one rule, but as we shall see shortly there is, in general, more than one rule in a given `match` expression. Each rule consists of a *pattern*, possibly involving variables, and an expression that may involve those variables. The value of the `match` is determined by considering each rule in the order given to determine the first rule whose pattern matches the match value. If such a rule is found, the value of the `match` is the value of the expression part of the matching rule, with the variables of the pattern replaced by the corresponding components of the match value.

Pattern matching becomes more interesting, and useful, when combined with sums. The patterns $\mathtt{l} \cdot x$ and $\mathtt{r} \cdot x$ match the corresponding values of sum type. These may be used in combination with other patterns to express complex decisions about the structure of a value. For example, the following `match` expresses the computation that, when given a pair of type $(\mathtt{unit} + \mathtt{unit}) \times \mathtt{nat}$, either doubles or squares its second component depending on the form of its first component:

$$\mathtt{match}\ e\ \{\langle \mathtt{l} \cdot \langle\rangle, x\rangle \hookrightarrow x + x \mid \langle \mathtt{r} \cdot \langle\rangle, y\rangle \hookrightarrow y * y\}. \tag{12.1}$$

It is an instructive exercise to express the same computation using only the primitives for sums and products given in Chapters 10 and 11.

In this chapter we study pattern for eager product and sum types in the context of an eager dynamics of **PCF**. The lazy dynamics is explored in the exercises.

Preview

## 12.1 A Pattern Language

The language **Pat** is the extension of **PCF** with the following constructs:

| | | | | | |
|---|---|---|---|---|---|
| Exp | $e$ | $::=$ | $\texttt{match}(e; rs)$ | $\texttt{match}\, e\, \{rs\}$ | case analysis |
| Rules | $rs$ | $::=$ | $\texttt{rules}[n](r_1; \ldots; r_n)$ | $r_1 \mid \ldots \mid r_n$ | $(n \geq 0)$ |
| Rule | $r$ | $::=$ | $\texttt{rule}[k](p; x_1, \ldots, x_k.e)$ | $p \hookrightarrow e$ | $(k \geq 0)$ |
| Pat | $p$ | $::=$ | $\texttt{wild}$ | $_$ | wild card |
| | | | $x$ | $x$ | variable |
| | | | $\texttt{triv}$ | $\langle\rangle$ | unit |
| | | | $\texttt{pair}(p_1; p_2)$ | $\langle p_1, p_2 \rangle$ | pair |
| | | | $\texttt{in}[\texttt{l}](p)$ | $\texttt{l} \cdot p$ | left injection |
| | | | $\texttt{in}[\texttt{r}](p)$ | $\texttt{r} \cdot p$ | right injection |

The operator `match` has two operands, the expression to match and a series of rules. A sequence of rules is constructed using the operator $\texttt{rules}[n]$, which has $n \geq 0$ operands. Each rule is constructed by the operator $\texttt{rule}[k]$, which specifies that it has two operands, binding $k$ variables in the second.

## 12.2 Statics

The statics of **Pat** makes use of a special form of hypothetical judgment, written

$$x_1 : \tau_1, \ldots, x_k : \tau_k \Vdash p : \tau,$$

with almost the same meaning as

$$x_1 : \tau_1, \ldots, x_k : \tau_k \vdash p : \tau,$$

except that each variable is required to be used *exactly once* in $p$. When reading the judgment $\Lambda \Vdash p : \tau$ it is helpful to think of $\Lambda$ as an *output*, and $p$ and $\tau$ as *inputs*. Given $p$ and $\tau$, the rules determine the hypotheses $\Lambda$ such that $\Lambda \Vdash p : \tau$.

$$\frac{}{x : \tau \Vdash x : \tau} \tag{12.2a}$$

$$\frac{}{\emptyset \Vdash _ : \tau} \tag{12.2b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle : \texttt{unit}} \tag{12.2c}$$

$$\frac{\Lambda_1 \Vdash p_1 : \tau_1 \quad \Lambda_2 \Vdash p_2 : \tau_2 \quad dom(\Lambda_1) \cap dom(\Lambda_2) = \emptyset}{\Lambda_1\, \Lambda_2 \Vdash \langle p_1, p_2 \rangle : \tau_1 \times \tau_2} \tag{12.2d}$$

$$\frac{\Lambda_1 \Vdash p : \tau_1}{\Lambda_1 \Vdash \texttt{l} \cdot p : \tau_1 + \tau_2} \tag{12.2e}$$

$$\frac{\Lambda_2 \Vdash p : \tau_2}{\Lambda_2 \Vdash \texttt{r} \cdot p : \tau_1 + \tau_2} \tag{12.2f}$$

Preview

Rule (12.2a) states that a variable is a pattern of type $\tau$. Rule (12.2d) states that a pair pattern consists of two patterns with disjoint variables.

The typing judgments for a rule,

$$p \hookrightarrow e : \tau \rightsquigarrow \tau',$$

and for a sequence of rules,

$$r_1 \mid \ldots \mid r_n : \tau \rightsquigarrow \tau',$$

specify that rules transform a value of type $\tau$ into a value of type $\tau'$. These judgments are inductively defined as follows:

$$\frac{\Lambda \Vdash p : \tau \quad \Gamma\, \Lambda \vdash e : \tau'}{\Gamma \vdash p \hookrightarrow e : \tau \rightsquigarrow \tau'} \tag{12.3a}$$

$$\frac{\Gamma \vdash r_1 : \tau \rightsquigarrow \tau' \quad \ldots \quad \Gamma \vdash r_n : \tau \rightsquigarrow \tau'}{\Gamma \vdash r_1 \mid \ldots \mid r_n : \tau \rightsquigarrow \tau'} \tag{12.3b}$$

Using the typing judgments for rules, the typing rule for a `match` expression may be stated quite easily:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash rs : \tau \rightsquigarrow \tau'}{\Gamma \vdash \texttt{match}\, e\, \{rs\} : \tau'} \tag{12.4}$$

## 12.3 Dynamics

A *substitution*, $\theta$, is a finite mapping from variables to values. If $\theta$ is the substitution $\{x_0 \hookrightarrow e_0, \ldots, x_{k-1} \hookrightarrow e_{k-1}\}$, we write $\hat{\theta}(e)$ for $[e_1, \ldots, e_k / x_1, \ldots, x_k]e$. The judgment $\theta : \Lambda$ is inductively defined by the following rules:

$$\frac{}{\emptyset : \emptyset} \tag{12.5a}$$

$$\frac{\theta : \Lambda \quad e : \tau}{\theta \otimes x \hookrightarrow e : \Lambda, x : \tau} \tag{12.5b}$$

The judgment $\theta \Vdash p \lhd e$ states that the pattern, $p$, matches the value, $e$, as witnessed by the substitution, $\theta$, defined on the variables of $p$. This judgment is inductively defined by the following rules:

$$\frac{}{x \hookrightarrow e \Vdash x \lhd e} \tag{12.6a}$$

$$\frac{}{\emptyset \Vdash _ \lhd e} \tag{12.6b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle \lhd \langle\rangle} \tag{12.6c}$$

$$\frac{\theta_1 \Vdash p_1 \lhd e_1 \quad \theta_2 \Vdash p_2 \lhd e_2 \quad dom(\theta_1) \cap dom(\theta_2) = \emptyset}{\theta_1 \otimes \theta_2 \Vdash \langle p_1, p_2 \rangle \lhd \langle e_1, e_2 \rangle} \tag{12.6d}$$

$$\frac{\theta \Vdash p \lhd e}{\theta \Vdash \mathtt{l} \cdot p \lhd \mathtt{l} \cdot e} \tag{12.6e}$$

$$\frac{\theta \Vdash p \lhd e}{\theta \Vdash \mathtt{r} \cdot p \lhd \mathtt{r} \cdot e} \tag{12.6f}$$

Preview

These rules simply collect the bindings for the pattern variables required to form a substitution witnessing the success of the matching process.

The judgment $e \perp p$ states that $e$ does not match the pattern $p$. It is inductively defined by the following rules:

$$\frac{e_1 \perp p_1}{\langle e_1, e_2 \rangle \perp \langle p_1, p_2 \rangle} \tag{12.7a}$$

$$\frac{e_2 \perp p_2}{\langle e_1, e_2 \rangle \perp \langle p_1, p_2 \rangle} \tag{12.7b}$$

$$\frac{}{\mathtt{l} \cdot e \perp \mathtt{r} \cdot p} \tag{12.7c}$$

$$\frac{e \perp p}{\mathtt{l} \cdot e \perp \mathtt{l} \cdot p} \tag{12.7d}$$

$$\frac{}{\mathtt{r} \cdot e \perp \mathtt{l} \cdot p} \tag{12.7e}$$

$$\frac{e \perp p}{\mathtt{r} \cdot e \perp \mathtt{r} \cdot p} \tag{12.7f}$$

Neither a variable nor a wildcard nor a null-tuple can mismatch any value of appropriate type. A pair can only mismatch a pair pattern due to a mismatch in one of its components. An injection into a sum type can mismatch the opposite injection, or it can mismatch the same injection by having its argument mismatch the argument pattern.

**Theorem 12.1.** *Suppose that $e : \tau$, $e$ val, and $\Lambda \Vdash p : \tau$. Then either there exists $\theta$ such that $\theta : \Lambda$ and $\theta \Vdash p \triangleleft e$, or $e \perp p$.*

*Proof.* By rule induction on rules (12.2), making use of the canonical forms lemma to characterize the shape of $e$ based on its type. □

The dynamics of the `match` expression is given in terms of the pattern match and mismatch judgments as follows:

$$\frac{e \longmapsto e'}{\mathtt{match}\, e\, \{rs\} \longmapsto \mathtt{match}\, e'\, \{rs\}} \tag{12.8a}$$

$$\frac{e\ \mathsf{val}}{\mathtt{match}\, e\, \{\}\ \mathsf{err}} \tag{12.8b}$$

$$\frac{e\ \mathsf{val} \quad \theta \Vdash p_0 \triangleleft e}{\mathtt{match}\, e\, \{p_0 \hookrightarrow e_0 \mid rs\} \longmapsto \hat{\theta}(e_0)} \tag{12.8c}$$

$$\frac{e\ \mathsf{val} \quad e \perp p_0 \quad \mathtt{match}\, e\, \{rs\} \longmapsto e'}{\mathtt{match}\, e\, \{p_0 \hookrightarrow e_0 \mid rs\} \longmapsto e'} \tag{12.8d}$$

Rule (12.8b) specifies that evaluation results in a checked error once all rules are exhausted. Rules (12.8c) specifies that the rules are to be considered in order. If the match value, $e$, matches the pattern, $p_0$, of the initial rule in the sequence, then the result is the corresponding instance of $e_0$; otherwise, matching continues by considering the remaining rules.

Preview

**Theorem 12.2** (Preservation). *If $e \longmapsto e'$ and $e : \tau$, then $e' : \tau$.*

*Proof.* By induction on the derivation of $e \longmapsto e'$. □

## 12.4 Exhaustiveness and Redundancy

Although it is possible to state and prove a progress theorem for **Pat** as defined in Section 12.1, it would not have much force, because the statics does not rule out pattern matching failure. What is missing is enforcement of the *exhaustiveness* of a sequence of rules, which ensures that every value of the domain type of a sequence of rules must match some rule in the sequence. In addition it would be useful to rule out *redundancy* of rules, which arises when a rule can only match values that are also matched by a preceding rule. Because pattern matching considers rules in the order in which they are written, such a rule can never be executed, and hence can be safely eliminated.

### 12.4.1 Match Constraints

To express exhaustiveness and irredundancy, we introduce a language of *match constraints* that identify a subset of the closed values of a type. With each rule we associate a constraint that classifies the values that are matched by that rule. A sequence of rules is *exhaustive* if every value of the domain type of the rules satisfies the match constraint of some rule in the sequence. A rule in a sequence is *redundant* if every value that satisfies its match constraint also satisfies the match constraint of some preceding rule.

The language of match constraints is defined by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Constr | $\xi$ | ::= | $\mathtt{all}[\tau]$ | $\top$ | truth |
| | | | $\mathtt{and}(\xi_1;\xi_2)$ | $\xi_1 \wedge \xi_2$ | conjunction |
| | | | $\mathtt{nothing}[\tau]$ | $\bot$ | falsity |
| | | | $\mathtt{or}(\xi_1;\xi_2)$ | $\xi_1 \vee \xi_2$ | disjunction |
| | | | $\mathtt{in}[\tau_1;\tau_2][\mathtt{l}](\xi_1)$ | $\mathtt{l} \cdot \xi_1$ | left injection |
| | | | $\mathtt{in}[\tau_1;\tau_2][\mathtt{r}](\xi_2)$ | $\mathtt{r} \cdot \xi_2$ | right injection |
| | | | $\mathtt{triv}$ | $\langle\rangle$ | unit |
| | | | $\mathtt{pair}(\xi_1;\xi_2)$ | $\langle \xi_1, \xi_2 \rangle$ | pair |

It is easy to define the judgment $\xi : \tau$ specifying that the constraint $\xi$ constrains values of type $\tau$.

The *De Morgan Dual*, $\overline{\xi}$, of a match constraint, $\xi$, is defined by the following rules:

$$\begin{aligned}
\overline{\top} &= \bot \\
\overline{\xi_1 \wedge \xi_2} &= \overline{\xi_1} \vee \overline{\xi_2} \\
\overline{\bot} &= \top \\
\overline{\xi_1 \vee \xi_2} &= \overline{\xi_1} \wedge \overline{\xi_2} \\
\overline{\mathtt{l} \cdot \xi_1} &= \mathtt{l} \cdot \overline{\xi_1} \vee \mathtt{r} \cdot \top \\
\overline{\mathtt{r} \cdot \xi_1} &= \mathtt{r} \cdot \overline{\xi_1} \vee \mathtt{l} \cdot \top
\end{aligned}$$

Preview

$$\overline{\langle\rangle} = \bot$$
$$\overline{\langle \xi_1, \xi_2 \rangle} = \langle \overline{\xi_1}, \xi_2 \rangle \vee \langle \xi_1, \overline{\xi_2} \rangle \vee \langle \overline{\xi_1}, \overline{\xi_2} \rangle$$

Intuitively, the dual of a match constraint expresses the negation of that constraint. In the case of the last four rules it is important to keep in mind that these constraints apply only to specific types.

The *satisfaction* judgment, $e \models \xi$, is defined for values $e$ and constraints $\xi$ of the same type by the following rules:

$$\frac{}{e \models \top} \tag{12.9a}$$

$$\frac{e \models \xi_1 \quad e \models \xi_2}{e \models \xi_1 \wedge \xi_2} \tag{12.9b}$$

$$\frac{e \models \xi_1}{e \models \xi_1 \vee \xi_2} \tag{12.9c}$$

$$\frac{e \models \xi_2}{e \models \xi_1 \vee \xi_2} \tag{12.9d}$$

$$\frac{e_1 \models \xi_1}{\mathtt{l} \cdot e_1 \models \mathtt{l} \cdot \xi_1} \tag{12.9e}$$

$$\frac{e_2 \models \xi_2}{\mathtt{r} \cdot e_2 \models \mathtt{r} \cdot \xi_2} \tag{12.9f}$$

$$\frac{}{\langle\rangle \models \langle\rangle} \tag{12.9g}$$

$$\frac{e_1 \models \xi_1 \quad e_2 \models \xi_2}{\langle e_1, e_2 \rangle \models \langle \xi_1, \xi_2 \rangle} \tag{12.9h}$$

The De Morgan dual construction negates a constraint.

**Lemma 12.3.** *If $\xi$ is a constraint on values of type $\tau$, then $e \models \overline{\xi}$ if, and only if, $e \not\models \xi$.*

We define the *entailment* of two constraints, $\xi_1 \models \xi_2$ to mean that $e \models \xi_2$ whenever $e \models \xi_1$. By Lemma 12.3 we have that $\xi_1 \models \xi_2$ iff $\models \overline{\xi_1} \vee \xi_2$. We often write $\xi_1, \ldots, \xi_n \models \xi$ for $\xi_1 \wedge \ldots \wedge \xi_n \models \xi$ so that in particular $\models \xi$ means $e \models \xi$ for every value $e : \tau$.

### 12.4.2 Enforcing Exhaustiveness and Redundancy

To enforce exhaustiveness and irredundancy the statics of pattern matching is augmented with constraints that express the set of values matched by a given set of rules. A sequence of rules is *exhaustive* if every value of suitable type satisfies the associated constraint. A rule is *redundant* relative to the preceding rules if every value satisfying its constraint satisfies one of the preceding constraints. A sequence of rules is *irredundant* iff no rule is redundant relative to the rules that precede it in the sequence.

Preview

The judgment $\Lambda \Vdash p : \tau\ [\xi]$ augments the judgment $\Lambda \Vdash p : \tau$ with a match constraint characterizing the set of values of type $\tau$ matched by the pattern $p$. It is inductively defined by the following rules:

$$\frac{}{x : \tau \Vdash x : \tau\ [\top]} \tag{12.10a}$$

$$\frac{}{\emptyset \Vdash _ : \tau\ [\top]} \tag{12.10b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle : \mathtt{unit}\ [\langle\rangle]} \tag{12.10c}$$

$$\frac{\Lambda_1 \Vdash p : \tau_1\ [\xi_1]}{\Lambda_1 \Vdash \mathtt{l} \cdot p : \tau_1 + \tau_2\ [\mathtt{l} \cdot \xi_1]} \tag{12.10d}$$

$$\frac{\Lambda_2 \Vdash p : \tau_2\ [\xi_2]}{\Lambda_2 \Vdash \mathtt{r} \cdot p : \tau_1 + \tau_2\ [\mathtt{r} \cdot \xi_2]} \tag{12.10e}$$

$$\frac{\Lambda_1 \Vdash p_1 : \tau_1\ [\xi_1] \quad \Lambda_2 \Vdash p_2 : \tau_2\ [\xi_2] \quad dom(\Lambda_1) \cap dom(\Lambda_2) = \emptyset}{\Lambda_1\, \Lambda_2 \Vdash \langle p_1, p_2\rangle : \tau_1 \times \tau_2\ [\langle \xi_1, \xi_2\rangle]} \tag{12.10f}$$

**Lemma 12.4.** *Suppose that $\Lambda \Vdash p : \tau\ [\xi]$. For every $e : \tau$ such that $e$ val, $e \models \xi$ iff $\theta \Vdash p \triangleleft e$ for some $\theta$, and $e \not\models \xi$ iff $e \perp p$.*

The judgment $\Gamma \vdash r : \tau \rightsquigarrow \tau'\ [\xi]$ augments the formation judgment for a rule with a match constraint characterizing the pattern component of the rule. The judgment $\Gamma \vdash rs : \tau \rightsquigarrow \tau'\ [\xi]$ augments the formation judgment for a sequence of rules with a match constraint characterizing the values matched by some rule in the given rule sequence.

$$\frac{\Lambda \Vdash p : \tau\ [\xi] \quad \Gamma\, \Lambda \vdash e : \tau'}{\Gamma \vdash p \hookrightarrow e : \tau \rightsquigarrow \tau'\ [\xi]} \tag{12.11a}$$

$$\frac{(\forall 1 \leq i \leq n)\ \xi_i \not\models \xi_1 \vee \ldots \vee \xi_{i-1} \qquad \Gamma \vdash r_1 : \tau \rightsquigarrow \tau'\ [\xi_1] \quad \ldots \quad \Gamma \vdash r_n : \tau \rightsquigarrow \tau'\ [\xi_n]}{\Gamma \vdash r_1 \mid \ldots \mid r_n : \tau \rightsquigarrow \tau'\ [\xi_1 \vee \ldots \vee \xi_n]} \tag{12.11b}$$

Rule (12.11b) requires that each successive rule not be redundant relative to the preceding rules. The overall constraint associated to the rule sequence specifies that every value of type $\tau$ satisfy the constraint associated with some rule.

The typing rule for `match` expressions demands that the rules that comprise it be exhaustive:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash rs : \tau \rightsquigarrow \tau'\ [\xi] \quad \models \xi}{\Gamma \vdash \mathtt{match}\, e\, \{rs\} : \tau'} \tag{12.12}$$

Rule (12.11b) ensures that $\xi$ is a disjunction of the match constraints associated to the constituent rules of the match expression. The requirement that $\xi$ be valid amounts to requiring that every value of type $\tau$ satisfies the constraint of at least one rule of the match.

**Theorem 12.5.** *If $e : \tau$, then either $e$ val or there exists $e'$ such that $e \longmapsto e'$.*

*Proof.* The exhaustiveness check in rule (12.12) ensures that if $e$ val and $e : \tau$, then $e \models \xi$. The form of $\xi$ given by rule (12.11b) ensures that $e \models \xi_i$ for some constraint $\xi_i$ corresponding to the $i$th rule. By Lemma 12.4 the value $e$ must match the $i$th rule, which is enough to ensure progress. □

Preview

### 12.4.3 Checking Exhaustiveness and Redundancy

Checking exhaustiveness and redundancy reduces to showing that the constraint validity judgment $\models \xi$ is decidable. We will prove this by defining a judgment $\Xi$ incon, where $\Xi$ is a finite set of constraints of the same type, with the meaning that no value of this type satisfies all of the constraints in $\Xi$. We will then show that either $\Xi$ incon or not.

The rules defining inconsistency of a finite set, $\Xi$, of constraints of the same type are as follows:

$$\frac{\Xi \text{ incon}}{\Xi, \top \text{ incon}} \tag{12.13a}$$

$$\frac{\Xi, \xi_1, \xi_2 \text{ incon}}{\Xi, \xi_1 \wedge \xi_2 \text{ incon}} \tag{12.13b}$$

$$\frac{}{\Xi, \bot \text{ incon}} \tag{12.13c}$$

$$\frac{\Xi, \xi_1 \text{ incon} \quad \Xi, \xi_2 \text{ incon}}{\Xi, \xi_1 \vee \xi_2 \text{ incon}} \tag{12.13d}$$

$$\frac{}{\Xi, \mathtt{l} \cdot \xi_1, \mathtt{r} \cdot \xi_2 \text{ incon}} \tag{12.13e}$$

$$\frac{\Xi \text{ incon}}{\mathtt{l} \cdot \Xi \text{ incon}} \tag{12.13f}$$

$$\frac{\Xi \text{ incon}}{\mathtt{r} \cdot \Xi \text{ incon}} \tag{12.13g}$$

$$\frac{\Xi_1 \text{ incon}}{\langle \Xi_1, \Xi_2 \rangle \text{ incon}} \tag{12.13h}$$

$$\frac{\Xi_2 \text{ incon}}{\langle \Xi_1, \Xi_2 \rangle \text{ incon}} \tag{12.13i}$$

In rule (12.13f) we write $\mathtt{l} \cdot \Xi$ for the finite set of constraints $\mathtt{l} \cdot \xi_1, \ldots, \mathtt{l} \cdot \xi_n$, where $\Xi = \xi_1, \ldots, \xi_n$, and similarly in rules (12.13g), (12.13h), and (12.13i).

**Lemma 12.6.** *It is decidable whether or not $\Xi$ incon.*

*Proof.* The premises of each rule involves only constraints that are proper components of the constraints in the conclusion. Consequently, we can simplify $\Xi$ by inverting each of the applicable rules until no rule applies, then determine whether or not the resulting set, $\Xi'$, is contradictory in the sense that it contains $\bot$ or both $\mathtt{l} \cdot \xi$ and $\mathtt{r} \cdot \xi'$ for some $\xi$ and $\xi'$. □

**Lemma 12.7.** *$\Xi$ incon iff $\Xi \models \bot$.*

*Proof.* From left to right we proceed by induction on rules (12.13). From right to left we may show that if $\Xi$ incon is not derivable, then there exists a value $e$ such that $e \models \Xi$, and hence $\Xi \not\models \bot$. □

Preview

## 12.5 Notes

Pattern-matching against heterogeneous structured data was first explored in the context of logic programming languages, such as Prolog (Kowalski, 1988; Colmerauer and Roussel, 1993), but with an execution model based on proof search. Pattern matching in the form described here is present in the functional languages Miranda (Turner, 1987), Hope (Burstall et al., 1980), Standard ML (Milner et al., 1997), Caml (Cousineau and Mauny, 1998), and Haskell (Jones, 2003).

## Exercises

**12.1**. Extend pattern matching to admit patterns `z` and $\texttt{s}(p)$ over the type `nat` of natural numbers.

**12.2**. Show that *multi-argument functions* and *keyword parameter passing* as found in many languages are both instances of pattern matching, and hence need not be considered as primitive notions.

**12.3**. Consider the interaction between laziness and pattern matching. Suppose that pairs were evaluated lazily, rather than eagerly. Give a precise dynamics for pattern matching with lazy pairs. Consider carefully the patterns $\langle x_1, x_2 \rangle$, $\langle x_1, _ \rangle$, $\langle _, x_2 \rangle$, and $\langle _, _ \rangle$. Which components should be evaluated in each case, and why? What if a variable is bound by the pattern but not used in the branch in which it is bound?

**12.4**. Exercise **12.3** suggests that there may be a "maximally lazy" sequential dynamics for pattern matching that avoids evaluating an expression unless it is absolutely needed. Indeed, you may already believe that your solution to Exercise **12.3** enjoys this property. Disabuse yourself of this notion by defining using pattern matching a three-argument function, sometimes called Gustav's function, that evaluates to `true` iff *any two* of its arguments evaluate to `true`, and to `false` otherwise (that is, if all evaluate to `false`). Show that no matter what order of pattern matching you use, there will be an input for which more work is done than is strictly necessary to determine the correct output.

Preview

Preview

# Part V

# Infinite Data Types

Preview

Preview

# Chapter 13

# Generic Programming

## 13.1 Introduction

Many programs can be seen as instances of a general pattern applied to a particular situation. Very often the pattern is determined by the types of the data involved. For example, in Chapter 9 the pattern of computing by recursion over a natural number is isolated as the defining characteristic of the type of natural numbers. This concept will itself emerge as an instance of the concept of *type-generic*, or just *generic*, programming.

Suppose that we have a function, $f$, of type $\rho \to \rho'$ that transforms values of type $\rho$ into values of type $\rho'$. For example, $f$ might be the doubling function on natural numbers. We wish to extend $f$ to a transformation from type $[\rho/t]\tau$ to type $[\rho'/t]\tau$ by applying $f$ to various spots in the input where a value of type $\rho$ occurs to obtain a value of type $\rho'$, leaving the rest of the data structure alone. For example, $\tau$ might be $\texttt{bool} \times \rho$, in which case $f$ could be extended to a function of type $\texttt{bool} \times \rho \to \texttt{bool} \times \rho'$ that sends the pairs $\langle a, b\rangle$ to the pair $\langle a, f(b)\rangle$.

This example glosses over a significant problem of ambiguity of the extension. Given a function $f$ of type $\rho \to \rho'$, it is not obvious in general how to extend it to a function mapping $[\rho/t]\tau$ to $[\rho'/t]\tau$. The problem is that it is not clear which of many occurrences of $\rho$ in $[\rho/t]\tau$ are to be transformed by $f$, even if there is only one occurrence of $\rho$. To avoid ambiguity we need a way to mark which occurrences of $\rho$ in $[\rho/t]\tau$ are to be transformed, and which are to be left fixed. This can be achieved by isolating the *type operator*, $t.\tau$, which is a type expression in which a designated variable, $t$, marks the spots at which we wish the transformation to occur. Given $t.\tau$ and $f : \rho \to \rho'$, we can extend $f$ unambiguously to a function of type $[\rho/t]\tau \to [\rho'/t]\tau$.

The technique of using a type operator to determine the behavior of a piece of code is called *generic programming*. The power of generic programming depends on which forms of type operator are considered. The simplest case is that of a *polynomial* type operator, one constructed from sum and product of types, including their nullary forms. These may be extended to *positive* type operators, which also permit restricted forms of function types.

Preview

## 13.2 Polynomial Type Operators

A *type operator* is a type equipped with a designated variable whose occurrences mark the positions in the type where a transformation is to be applied. A type operator is represented by an abstractor $t.\tau$ such that $t$ type $\vdash \tau$ type. An example of a type operator is the abstractor

$$t.\mathtt{unit} + (\mathtt{bool} \times t)$$

in which occurrences of $t$ mark the spots in which a transformation is to be applied. An *instance* of the type operator $t.\tau$ is obtained by substituting a type, $\rho$, for the variable, $t$, within the type $\tau$.

The *polynomial* type operators are those constructed from the type variable, $t$, the types `void` and `unit`, and the product and sum type constructors, $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$. More precisely, the judgment $t.\tau$ poly is inductively defined by the following rules:

$$\frac{}{t.t \text{ poly}} \tag{13.1a}$$

$$\frac{}{t.\mathtt{unit} \text{ poly}} \tag{13.1b}$$

$$\frac{t.\tau_1 \text{ poly} \quad t.\tau_2 \text{ poly}}{t.\tau_1 \times \tau_2 \text{ poly}} \tag{13.1c}$$

$$\frac{}{t.\mathtt{void} \text{ poly}} \tag{13.1d}$$

$$\frac{t.\tau_1 \text{ poly} \quad t.\tau_2 \text{ poly}}{t.\tau_1 + \tau_2 \text{ poly}} \tag{13.1e}$$

Exercise **13.1** asks for a proof that polynomial type operators are closed under substitution.

The *generic extension* of a polynomial type operator is a form of expression with the following syntax

$$\mathsf{Exp} \quad e \quad ::= \quad \mathtt{map}[t.\tau](x.e';e) \quad \mathtt{map}_{t.\tau}(x.e';e) \quad \text{generic extension.}$$

Its statics is given as follows:

$$\frac{t.\tau \text{ poly} \quad \Gamma, x : \rho \vdash e' : \rho' \quad \Gamma \vdash e : [\rho/t]\tau}{\Gamma \vdash \mathtt{map}[t.\tau](x.e';e) : [\rho'/t]\tau} \tag{13.2}$$

Thinking of the abstractor $x.e'$ as a transformation sending $x$ of type $\rho$ to $e'$ of type $\rho'$, the generic extension of $t.\tau$ is a transformation sending values of type $[\rho/t]\tau$ to values of type $[\rho'/t]\tau$ that applies the given transformation to its argument everywhere that $t$ occurs within its type. In other words the abstractor $t.\tau$ is to be thought of as a "type template" in which certain spots, marked by occurrences of $t$ in $\tau$, indicate where to apply the transformation $x.e$ to the argument of the generic extension.



Preview

This informal description of the generic extension of a polynomial type operator is made precise by the following formal dynamics.

$$\frac{}{\mathtt{map}[t.t](x.e';e) \longmapsto [e/x]e'} \tag{13.3a}$$

$$\frac{}{\mathtt{map}[t.\mathtt{unit}](x.e';e) \longmapsto \langle\rangle} \tag{13.3b}$$

$$\frac{}{\begin{array}{c}\mathtt{map}[t.\tau_1 \times \tau_2](x.e';e) \\ \longmapsto \\ \langle \mathtt{map}[t.\tau_1](x.e';e\cdot \mathtt{l}), \mathtt{map}[t.\tau_2](x.e';e\cdot \mathtt{r})\rangle\end{array}} \tag{13.3c}$$

$$\frac{}{\mathtt{map}[t.\mathtt{void}](x.e';e) \longmapsto \mathtt{abort}(e)} \tag{13.3d}$$

$$\frac{}{\begin{array}{c}\mathtt{map}[t.\tau_1 + \tau_2](x.e';e) \\ \longmapsto \\ \mathtt{case}\, e\, \{\mathtt{l}\cdot x_1 \hookrightarrow \mathtt{l}\cdot \mathtt{map}[t.\tau_1](x.e';x_1) \mid \mathtt{r}\cdot x_2 \hookrightarrow \mathtt{r}\cdot \mathtt{map}[t.\tau_2](x.e';x_2)\}\end{array}} \tag{13.3e}$$

Rule (13.3a) applies the transformation $x.e'$ to $e$ itself, because the operator $t.t$ specifies that the transformation is to be perfomed directly. Rule (13.3b) states that the empty tuple is transformed to itself. Rule (13.3c) states that to transform $e$ according to the operator $t.\tau_1 \times \tau_2$, the first component of $e$ is transformed according to $t.\tau_1$ and the second component of $e$ is transformed according to $t.\tau_2$. Rule (13.3d) states that the transformation of a value of type `void` aborts, because there can be no such values. Rule (13.3e) states that to transform $e$ according to $t.\tau_1 + \tau_2$, case analyze $e$ and reconstruct it after transforming the injected value according to $t.\tau_1$ or $t.\tau_2$.

Consider the type operator $t.\tau$ given by $t.\mathtt{unit} + (\mathtt{bool} \times t)$. Let $x.e$ be the abstractor $x.\mathtt{s}(x)$, which increments a natural number. Using rules (13.3) we may derive that

$$\mathtt{map}_{t.\tau}(x.e;\mathtt{r}\cdot\langle \mathtt{true}, n\rangle) \longmapsto^* \mathtt{r}\cdot\langle \mathtt{true}, n+1\rangle.$$

The natural number in the second component of the pair is incremented, because the type variable, $t$, occurs in that position in the type operator $t.\tau$.

**Theorem 13.1** (Preservation). *If* $\mathtt{map}[t.\tau](x.e';e) : \tau'$ *and* $\mathtt{map}[t.\tau](x.e';e) \longmapsto e''$, *then* $e'' : \tau'$.

*Proof.* By inversion of rule (13.2) we have

1. $t$ type $\vdash \tau$ type;
2. $x : \rho \vdash e' : \rho'$ for some $\rho$ and $\rho'$;



Preview

3. $e : [\rho/t]\tau$;

4. $\tau'$ is $[\rho'/t]\tau$.

We proceed by cases on rules (13.3). For example, consider rule (13.3c). It follows from inversion that $\mathtt{map}[t.\tau_1](x.e';e \cdot \mathtt{l}) : [\rho'/t]\tau_1$, and similarly that $\mathtt{map}[t.\tau_2](x.e';e \cdot \mathtt{r}) : [\rho'/t]\tau_2$. It is easy to check that

$$\langle \mathtt{map}[t.\tau_1](x.e';e \cdot \mathtt{l}), \mathtt{map}[t.\tau_2](x.e';e \cdot \mathtt{r}) \rangle$$

has type $[\rho'/t](\tau_1 \times \tau_2)$, as required. □

## 13.3 Positive Type Operators

The *positive* type operators extend the polynomial type operators to admit restricted forms of function type. Specifically, $t.\tau_1 \to \tau_2$ is a positive type operator, provided that (1) $t$ *does not occur* in $\tau_1$, and (2) $t.\tau_2$ is a positive type operator. In general, any occurrences of a type variable $t$ in the domain of a function type are said to be *negative occurrences*, whereas any occurrences of $t$ within the range of a function type, or within a product or sum type, are said to be *positive occurrences*.[1] A positive type operator is one for which only positive occurrences of the parameter, $t$, are permitted. Positive type operators, like polynomial type operators, are closed under substitution.

To make the definition of positive type operator precise, we define the judgment $t.\tau$ pos, which states that the abstractor, $t.\tau$, is such by the following rules:

$$\frac{}{t.t \text{ pos}} \tag{13.4a}$$

$$\frac{}{t.\mathtt{unit} \text{ pos}} \tag{13.4b}$$

$$\frac{t.\tau_1 \text{ pos} \quad t.\tau_2 \text{ pos}}{t.\tau_1 \times \tau_2 \text{ pos}} \tag{13.4c}$$

$$\frac{}{t.\mathtt{void} \text{ pos}} \tag{13.4d}$$

$$\frac{t.\tau_1 \text{ pos} \quad t.\tau_2 \text{ pos}}{t.\tau_1 + \tau_2 \text{ pos}} \tag{13.4e}$$

$$\frac{\tau_1 \text{ type} \quad t.\tau_2 \text{ pos}}{t.\tau_1 \to \tau_2 \text{ pos}} \tag{13.4f}$$

In rule (13.4f), the type variable $t$ is excluded from the domain of the function type by demanding that it be well-formed without regard to $t$.

[1] The origin of this terminology seems to be that a function type $\tau_1 \to \tau_2$ is analogous to the implication $\phi_1 \supset \phi_2$, which is classically equivalent to $\neg\phi_1 \vee \phi_2$, so that occurrences in the domain are under the negation.

Preview

The generic extension of a positive type operator is defined similarly to that of a polynomial type operator, with the following dynamics on function types:

$$\overline{\mathtt{map}^+[t.\tau_1 \to \tau_2](x.e';e) \longmapsto \lambda\,(x_1{:}\tau_1)\,\mathtt{map}^+[t.\tau_2](x.e';e(x_1))} \tag{13.5}$$

Because $t$ is not permitted to occur within the domain type, the type of the result is $\tau_1 \to [\rho'/t]\tau_2$, assuming that $e$ is of type $\tau_1 \to [\rho/t]\tau_2$. It is easy to verify preservation for the generic extension of a positive type operator.

It is instructive to consider what goes wrong if we try to extend the generic extension to an arbitrary type operator, without any positivity restriction. Consider the type operator $t.\tau_1 \to \tau_2$, without restriction on $t$, and suppose that $x : \rho \vdash e' : \rho'$. The generic extension $\mathtt{map}[t.\tau_1 \to \tau_2](x.e';e)$ should have type $[\rho'/t]\tau_1 \to [\rho'/t]\tau_2$, given that $e$ has type $[\rho/t]\tau_1 \to [\rho/t]\tau_2$. The extension should yield a function of the form

$$\lambda\,(x_1{:}[\rho'/t]\tau_1)\ldots(e(\ldots(x_1)))$$

in which we apply $e$ to a transformation of $x_1$ and then transform the result. The trouble is that we are given, inductively, that $\mathtt{map}[t.\tau_1](x.e';-)$ transforms values of type $[\rho/t]\tau_1$ into values of type $[\rho'/t]\tau_1$, but *we need to go the other way around* in order to make $x_1$ suitable as an argument for $e$.

## 13.4 Notes

The generic extension of a type operator is an example of the concept of a *functor* in category theory (MacLane, 1998). Generic programming is essentially *functorial* programming, exploiting the functorial action of polynomial type operators (Hinze and Jeuring, 2003).

## Exercises

**13.1**. Prove that if $t.\tau$ poly and $t'.\tau'$ poly, then $t.[\tau/t']\tau'$ poly.

**13.2**. Show that the generic extension of a constant type operator is essentially the identity in that it sends each closed value to itself. More precisely, show that, for each value $e$ of type $\tau$, the expression

$$\mathtt{map}_{_\tau}(x.e';e)$$

evaluates to $e$, regardless of the choice of $e'$. For simplicity, assume an eager dynamics for products and sums, and consider only polynomial type operators. What is the complication with extending this observation to positive type operators?

**13.3**. Consider Exercises **10.1** and **11.3** in which a database schema is represented by a finite product type indexed by the attributes of the schema, and a database with that schema is a finite sequence of instances of tuples of that type. Show that any database transformation that applies a function to one or more of the columns of each row of a database may be programmed in two steps using generic programming according to the following plan:



Preview

(a) Specify a type operator whose type parameter indicates which columns are to be be transformed. All indicated columns must be of the same type in order for the transformation to make sense.

(b) Specify the transformation on the type of column that will be applied to each tuple of the database to obtain an updated database.

(c) Form the generic extension of the type operator with the given transformation, and apply it to the given database.

For specificity, consider a schema whose attributes, $I$, include the attributes `first` and `last`, both of type `str`. Let $c : \texttt{str} \to \texttt{str}$ be a function that capitalizes strings according to some convention. Use generic programming to capitalize the `first` and `last` attributes of each row of a given database with the specified schema.

**13.4**. Whereas $t$ occurs negatively in the type $t \to \texttt{bool}$, and does not occur only positively $(t \to \texttt{bool}) \to \texttt{bool}$, we may say that $t$ does occur *non-negatively* in the latter type. This example illustrates that occurrences of $t$ in the domain of a function are negative, and that occurrences in the domain of the domain of a function are non-negative. Every positive occurrence counts as non-negative, but not every non-negative occurrence is considered to be positive.[2] Give a simultaneous induction definition of the negative and non-negative type operators. Check that the type operator $t.(t \to \texttt{bool}) \to \texttt{bool}$ is non-negative according to your definition.

**13.5**. Using the definitions of negative and non-negative type operators requested in Exercise **13.4**, give the definition of the generic extension of a non-negative type operator. Specifically, define simultaneously $\texttt{map}^{--}[t.\tau](x.e';e)$ and $\texttt{map}^{-}[t.\tau](x.e;e')$ by induction on the structure of $\tau$ with the statics give by these rules:

$$\frac{t.\tau \text{ non-neg} \quad \Gamma, x : \rho \vdash e' : \rho' \quad \Gamma \vdash e : [\rho/t]\tau}{\Gamma \vdash \texttt{map}^{--}[t.\tau](x.e';e) : [\rho'/t]\tau} \tag{13.6a}$$

$$\frac{t.\tau \text{ non-neg} \quad \Gamma, x : \rho \vdash e' : \rho' \quad \Gamma \vdash e : [\rho'/t]\tau}{\Gamma \vdash \texttt{map}^{-}[t.\tau](x.e';e) : [\rho/t]\tau} \tag{13.6b}$$

Note well the reversal of the types of $e$ and the overall type in these two rules. Determine the generic extension of the type operator $t.(t \to \texttt{bool}) \to \texttt{bool}$.

[2] In the literature what we have called "positive" is usually called "strictly positive", and what we have called "non-negative" is usually called "positive".



Preview

# Chapter 14

# Inductive and Co-Inductive Types

The *inductive* and the *coinductive* types are two important forms of recursive type. Inductive types correspond to *least*, or *initial*, solutions of certain type equations, and coinductive types correspond to their *greatest*, or *final*, solutions. Intuitively, the elements of an inductive type are those that may be obtained by a finite composition of its introductory forms. Consequently, if we specify the behavior of a function on each of the introductory forms of an inductive type, then its behavior is determined for all values of that type. Such a function is called a *recursor*, or *catamorphism*. Dually, the elements of a coinductive type are those that behave properly in response to a finite composition of its elimination forms. Consequently, if we specify the behavior of an element on each elimination form, then we have fully specified a value of that type. Such an element is called an *generator*, or *anamorphism*.

## 14.1 Motivating Examples

The most important example of an inductive type is the type of natural numbers as formalized in Chapter 9. The type $\mathtt{nat}$ is defined to be the *least* type containing $\mathtt{z}$ and closed under $\mathtt{s}(-)$. The minimality condition is witnessed by the existence of the iterator, $\mathtt{iter}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}$, which transforms a natural number into a value of type $\tau$, given its value for zero, and a transformation from its value on a number to its value on the successor of that number. This operation is well-defined precisely because there are no other natural numbers.

With a view towards deriving the type $\mathtt{nat}$ as a special case of an inductive type, it is useful to consolidate zero and successor into a single introductory form, and to correspondingly consolidate the basis and inductive step of the iterator. The following rules specify the statics of this reformulation:

$$\frac{\Gamma \vdash e : \mathtt{unit} + \mathtt{nat}}{\Gamma \vdash \mathtt{fold}_{\mathtt{nat}}(e) : \mathtt{nat}} \tag{14.1a}$$

$$\frac{\Gamma, x : \mathtt{unit} + \tau \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{rec}_{\mathtt{nat}}(x.e_1; e_2) : \tau} \tag{14.1b}$$

Preview

The expression $\mathtt{fold}_{\mathtt{nat}}(e)$ is the unique introductory form of the type nat. Using this, the expression z is defined to be $\mathtt{fold}_{\mathtt{nat}}(\mathtt{l} \cdot \langle\rangle)$, and $\mathtt{s}(e)$ is defined to be $\mathtt{fold}_{\mathtt{nat}}(\mathtt{r} \cdot e)$. The recursor, $\mathtt{rec}_{\mathtt{nat}}(x.e_1; e_2)$, takes as argument the abstractor $x.e_1$ that consolidates the basis and inductive step into a single computation that, given a value of type $\mathtt{unit} + \tau$, yields a value of type $\tau$. Intuitively, if $x$ is replaced by the value $\mathtt{l} \cdot \langle\rangle$, then $e_1$ computes the base case of the recursion, and if $x$ is replaced by the value $\mathtt{r} \cdot e$, then $e_1$ computes the inductive step as a function of the result, $e$, of the recursive call.

The dynamics of the consolidated representation of natural numbers is given by the following rules:

$$\frac{}{\mathtt{fold}_{\mathtt{nat}}(e)\ \mathsf{val}} \tag{14.2a}$$

$$\frac{e_2 \longmapsto e_2'}{\mathtt{rec}_{\mathtt{nat}}(x.e_1; e_2) \longmapsto \mathtt{rec}_{\mathtt{nat}}(x.e_1; e_2')} \tag{14.2b}$$

$$\frac{}{\begin{array}{c}\mathtt{rec}_{\mathtt{nat}}(x.e_1; \mathtt{fold}_{\mathtt{nat}}(e_2)) \\ \longmapsto \\ [\mathtt{map}[t.\mathtt{unit} + t](y.\mathtt{rec}_{\mathtt{nat}}(x.e_1; y); e_2)/x]e_1\end{array}} \tag{14.2c}$$

Rule (14.2c) makes use of (polynomial) generic extension (see Chapter 13) to apply the recursor to the predecessor, if any, of a natural number. If we expand the definition of the generic extension in place, we obtain this rule:

$$\frac{}{\begin{array}{c}\mathtt{rec}_{\mathtt{nat}}(x.e_1; \mathtt{fold}_{\mathtt{nat}}(e_2)) \\ \longmapsto \\ [\mathtt{case}\, e_2\, \{\mathtt{l} \cdot _ \hookrightarrow \mathtt{l} \cdot \langle\rangle \mid \mathtt{r} \cdot y \hookrightarrow \mathtt{r} \cdot \mathtt{rec}_{\mathtt{nat}}(x.e_1; y)\}/x]e_1\end{array}}$$

Exercise **14.1** asks for a derivation of the iterator, as defined in Chapter 9, from the recursor just given.

An illustrative example of a coinductive type is the type of *streams* of natural numbers. A stream is an infinite sequence of natural numbers such that an element of the stream can be computed only after computing all preceding elements in that stream. That is, the computations of successive elements of the stream are sequentially dependent in that the computation of one element influences the computation of the next. In this sense the introductory form for streams is dual to the eliminatory form for natural numbers.

A stream is characterized by its behavior under the elimination forms for the stream type: $\mathtt{hd}(e)$ returns the next, or head, element of the stream, and $\mathtt{tl}(e)$ returns the tail of the stream, the stream resulting when the head element is removed. A stream is introduced by a *generator*, the dual of an recursor, that determines the head and the tail of the stream in terms of the current state of the stream, which is represented by a value of some type. The statics of streams is given by the

Preview

following rules:

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{hd}(e) : \mathtt{nat}} \tag{14.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{tl}(e) : \mathtt{stream}} \tag{14.3b}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_1 : \mathtt{nat} \quad \Gamma, x : \tau \vdash e_2 : \tau}{\Gamma \vdash \mathtt{strgen}\, x\, \mathtt{is}\, e\, \mathtt{in} <\mathtt{hd} \hookrightarrow e_1, \mathtt{tl} \hookrightarrow e_2> : \mathtt{stream}} \tag{14.3c}$$

In rule (14.3c) the current state of the stream is given by the expression $e$ of some type $\tau$, and the head and tail of the stream are determined by the expressions $e_1$ and $e_2$, respectively, as a function of the current state. (The notation for the generator is chosen to emphasize that every stream has both a head *and* a tail.)

The dynamics of streams is given by the following rules:

$$\frac{}{\mathtt{strgen}\, x\, \mathtt{is}\, e\, \mathtt{in} <\mathtt{hd} \hookrightarrow e_1, \mathtt{tl} \hookrightarrow e_2> \mathsf{val}} \tag{14.4a}$$

$$\frac{e \longmapsto e'}{\mathtt{hd}(e) \longmapsto \mathtt{hd}(e')} \tag{14.4b}$$

$$\frac{}{\mathtt{hd}(\mathtt{strgen}\, x\, \mathtt{is}\, e\, \mathtt{in} <\mathtt{hd} \hookrightarrow e_1, \mathtt{tl} \hookrightarrow e_2>) \longmapsto [e/x]e_1} \tag{14.4c}$$

$$\frac{e \longmapsto e'}{\mathtt{tl}(e) \longmapsto \mathtt{tl}(e')} \tag{14.4d}$$

$$\begin{array}{c} \mathtt{tl}(\mathtt{strgen}\, x\, \mathtt{is}\, e\, \mathtt{in} <\mathtt{hd} \hookrightarrow e_1, \mathtt{tl} \hookrightarrow e_2>) \\ \longmapsto \\ \mathtt{strgen}\, x\, \mathtt{is}\, [e/x]e_2\, \mathtt{in} <\mathtt{hd} \hookrightarrow e_1, \mathtt{tl} \hookrightarrow e_2> \end{array} \tag{14.4e}$$

Rules (14.4c) and (14.4e) express the dependency of the head and tail of the stream on its current state. Observe that the tail is obtained by applying the generator to the new state determined by $e_2$ as a function of the current state.

To derive streams as a special case of a coinductive type, we consolidate the head and the tail into a single eliminatory form, and reorganize the generator correspondingly. This leads to the following statics:

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{unfold}_{\mathtt{stream}}(e) : \mathtt{nat} \times \mathtt{stream}} \tag{14.5a}$$

$$\frac{\Gamma, x : \tau \vdash e_1 : \mathtt{nat} \times \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{gen}_{\mathtt{stream}}(x.e_1; e_2) : \mathtt{stream}} \tag{14.5b}$$

Rule (14.5a) states that a stream may be unfolded into a pair consisting of its head, a natural number, and its tail, another stream. The head, $\mathtt{hd}(e)$, and tail, $\mathtt{tl}(e)$, of a stream, $e$, are defined to be the projections $\mathtt{unfold}_{\mathtt{stream}}(e) \cdot \mathtt{l}$ and $\mathtt{unfold}_{\mathtt{stream}}(e) \cdot \mathtt{r}$, respectively. Rule (14.5b) states

Preview

that a stream may be generated from the state element, $e_2$, by an expression $e_1$ that yields the head element and the next state as a function of the current state.

The dynamics of streams is given by the following rules:

$$\frac{}{\mathtt{gen}_{\mathtt{stream}}(x.e_1;e_2)\ \mathsf{val}} \tag{14.6a}$$

$$\frac{e \longmapsto e'}{\mathtt{unfold}_{\mathtt{stream}}(e) \longmapsto \mathtt{unfold}_{\mathtt{stream}}(e')} \tag{14.6b}$$

$$\frac{}{\begin{array}{c}\mathtt{unfold}_{\mathtt{stream}}(\mathtt{gen}_{\mathtt{stream}}(x.e_1;e_2)) \\ \longmapsto \\ \mathtt{map}[t.\mathtt{nat}\times t](y.\mathtt{gen}_{\mathtt{stream}}(x.e_1;y);[e_2/x]e_1)\end{array}} \tag{14.6c}$$

Rule (14.6c) uses generic extension to generate a new stream whose state is the second component of $[e_2/x]e_1$. Expanding the generic extension we obtain the following reformulation of this rule:

$$\frac{}{\begin{array}{c}\mathtt{unfold}_{\mathtt{stream}}(\mathtt{gen}_{\mathtt{stream}}(x.e_1;e_2)) \\ \longmapsto \\ \langle([e_2/x]e_1)\cdot\mathtt{l},\mathtt{gen}_{\mathtt{stream}}(x.e_1;([e_2/x]e_1)\cdot\mathtt{r})\rangle\end{array}}$$

Exercise **14.2** asks for a derivation of $\mathtt{strgen}\,x\,\mathtt{is}\,e\,\mathtt{in}\,{<}\mathtt{hd}\hookrightarrow e_1,\mathtt{tl}\hookrightarrow e_2{>}$ from the coinductive generation form.

## 14.2 Statics

We may now give a general account of inductive and coinductive types, which are defined in terms of positive type operators. We will consider a variant of **T**, which we will call **M**, with natural numbers replaced by functions, products, sums, and a rich class of inductive and coinductive types.

### 14.2.1 Types

The syntax of inductive and coinductive types involves *type variables*, which are, of course, variables ranging over types. The abstract syntax of inductive and coinductive types is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | ::= | $t$ | $t$ | self-reference |
| | | | $\mathtt{ind}(t.\tau)$ | $\mu(t.\tau)$ | inductive |
| | | | $\mathtt{coi}(t.\tau)$ | $\nu(t.\tau)$ | coinductive |



Preview

*Type formation* judgments have the form

$$t_1 \text{ type}, \ldots, t_n \text{ type} \vdash \tau \text{ type},$$

where $t_1, \ldots, t_n$ are type names. We let $\Delta$ range over finite sets of hypotheses of the form $t$ type, where $t$ is a type name. The type formation judgment is inductively defined by the following rules:

$$\frac{}{\Delta, t \text{ type} \vdash t \text{ type}} \tag{14.7a}$$

$$\frac{}{\Delta \vdash \mathtt{unit} \text{ type}} \tag{14.7b}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \mathtt{prod}(\tau_1; \tau_2) \text{ type}} \tag{14.7c}$$

$$\frac{}{\Delta \vdash \mathtt{void} \text{ type}} \tag{14.7d}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \mathtt{sum}(\tau_1; \tau_2) \text{ type}} \tag{14.7e}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \mathtt{arr}(\tau_1; \tau_2) \text{ type}} \tag{14.7f}$$

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta \vdash t.\tau \text{ pos}}{\Delta \vdash \mathtt{ind}(t.\tau) \text{ type}} \tag{14.7g}$$

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta \vdash t.\tau \text{ pos}}{\Delta \vdash \mathtt{coi}(t.\tau) \text{ type}} \tag{14.7h}$$

### 14.2.2 Expressions

The abstract syntax of **M** is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Exp | $e$ | ::= | $\mathtt{fold}[t.\tau](e)$ | $\mathtt{fold}_{t.\tau}(e)$ | constructor |
| | | | $\mathtt{rec}[t.\tau](x.e_1; e_2)$ | $\mathtt{rec}(x.e_1; e_2)$ | recursor |
| | | | $\mathtt{unfold}[t.\tau](e)$ | $\mathtt{unfold}_{t.\tau}(e)$ | destructor |
| | | | $\mathtt{gen}[t.\tau](x.e_1; e_2)$ | $\mathtt{gen}(x.e_1; e_2)$ | generator |

The subscripts on the concrete syntax forms are often omitted when they are clear from context.

The statics for **M** is given by the following typing rules:

$$\frac{\Gamma \vdash e : [\mathtt{ind}(t.\tau)/t]\tau}{\Gamma \vdash \mathtt{fold}[t.\tau](e) : \mathtt{ind}(t.\tau)} \tag{14.8a}$$

$$\frac{\Gamma, x : [\tau'/t]\tau \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \mathtt{ind}(t.\tau)}{\Gamma \vdash \mathtt{rec}[t.\tau](x.e_1; e_2) : \tau'} \tag{14.8b}$$

$$\frac{\Gamma \vdash e : \mathtt{coi}(t.\tau)}{\Gamma \vdash \mathtt{unfold}[t.\tau](e) : [\mathtt{coi}(t.\tau)/t]\tau} \tag{14.8c}$$

$$\frac{\Gamma \vdash e_2 : \tau_2 \quad \Gamma, x : \tau_2 \vdash e_1 : [\tau_2/t]\tau}{\Gamma \vdash \mathtt{gen}[t.\tau](x.e_1; e_2) : \mathtt{coi}(t.\tau)} \tag{14.8d}$$

Preview

## 14.3 Dynamics

The dynamics of **M** is given in terms of the positive generic extension operation described in Chapter 13. The following rules specify a lazy dynamics for **M**:

$$\frac{}{\mathtt{fold}[t.\tau](e)\ \mathsf{val}} \tag{14.9a}$$

$$\frac{e_2 \longmapsto e_2'}{\mathtt{rec}[t.\tau](x.e_1; e_2) \longmapsto \mathtt{rec}[t.\tau](x.e_1; e_2')} \tag{14.9b}$$

$$\frac{}{\begin{array}{c}\mathtt{rec}[t.\tau](x.e_1; \mathtt{fold}[t.\tau](e_2)) \\ \longmapsto \\ [\mathtt{map}^+[t.\tau](y.\mathtt{rec}[t.\tau](x.e_1; y); e_2)/x]e_1\end{array}} \tag{14.9c}$$

$$\frac{}{\mathtt{gen}[t.\tau](x.e_1; e_2)\ \mathsf{val}} \tag{14.9d}$$

$$\frac{e \longmapsto e'}{\mathtt{unfold}[t.\tau](e) \longmapsto \mathtt{unfold}[t.\tau](e')} \tag{14.9e}$$

$$\frac{}{\begin{array}{c}\mathtt{unfold}[t.\tau](\mathtt{gen}[t.\tau](x.e_1; e_2)) \\ \longmapsto \\ \mathtt{map}^+[t.\tau](y.\mathtt{gen}[t.\tau](x.e_1; y); [e_2/x]e_1)\end{array}} \tag{14.9f}$$

Rule (14.9c) states that to evaluate the recursor on a value of recursive type, we inductively apply the recursor as guided by the type operator to the value, and then perform the inductive step on the result. Rule (14.9f) is simply the dual of this rule for coinductive types.

**Lemma 14.1.** *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

*Proof.* By rule induction on rules (14.9). □

**Lemma 14.2.** *If $e : \tau$, then either $e$ val or there exists $e'$ such that $e \longmapsto e'$.*

*Proof.* By rule induction on rules (14.8). □

Although a proof of this fact lies beyond our current reach, all programs in **M** terminate

**Theorem 14.3** (Termination for **M**)**.** *If $e : \tau$, then there exists $e'$ val such that $e \longmapsto^* e'$.*

It may, at first, seem surprising that a language with infinite data structures, such as streams, can enjoy such a termination property. But bear in mind that infinite data structures, such as streams, are represented as in a continuing state of creation, rather than as a completed whole.



Preview

## 14.4 Solving Type Equations

For a positive type operator $t.\tau$, we may say that the inductive type, $\mu(t.\tau)$, and the coinductive type, $\nu(t.\tau)$, are both *solutions* (up to isomorphims) of the type equation $t \cong \tau$:

$$\mu(t.\tau) \cong [\mu(t.\tau)/t]\tau$$
$$\nu(t.\tau) \cong [\nu(t.\tau)/t]\tau.$$

Intuitively speaking, this means that every value of an inductive type is the folding of a value of the unfolding of the inductive type, and that, similarly, every value of the unfolding of a coinductive type is the unfolding of a value of the coinductive type itself.

Whereas both are solutions to the same type equation, they are not isomorphic to each other. To see why, consider the inductive type $\mathtt{nat} \triangleq \mu(t.\mathtt{unit} + t)$ and the coinductive type $\mathtt{conat} \triangleq \nu(t.\mathtt{unit} + t)$. Intuitively, `nat` is the smallest (most restrictive) type containing zero, given by $\mathtt{fold}(\mathtt{l} \cdot \langle\rangle)$, and closed under formation of the successor of any other $e$ of type `nat`, given by $\mathtt{fold}(\mathtt{r} \cdot e)$. Dually, `conat` is the largest (most permissive) type of expressions $e$ for which the unfolding, $\mathtt{unfold}(e)$, is either zero, given by $\mathtt{l} \cdot \langle\rangle$, or to the successor of some other $e'$ of this type, given by $\mathtt{r} \cdot e'$.

Informally, the type `nat` may be considered to be the type of "finite numbers" and the type `conat` may be considered to be the type of "potentially infinite numbers." Because every finite number is a potentially infinite number that happens not to be infinite, we may embed the type `nat` into the type `conat` using the recursor associated with `nat`. Now consider the potentially infinite number, $\omega$, of type `conat` given by the generator $\mathtt{gen}(x.\mathtt{r} \cdot x; \langle\rangle)$. The number $\omega$ unfolds to the successor of itself, and hence may be said to be actually infinite, and therefore not the embedding of any finite natural number. This shows that `conat` is a "larger" solution to the type equation $t \cong \mathtt{unit} + t$ than `nat`. Indeed, there is a sense (not developed here) in which the latter is the smallest possible solution, and the former the largest possible solution, to the same type equation.

Summing up, the mere fact of being a solution to a type equation does not uniquely characterize a type: there can be many different solutions to the same type equation. However, this property depends on the fact that we are working with a language of total functions and terminating computations. As we shall see in Chapters 15 and 16, the situation changes significantly. Type equations have unique solutions, but not every computation terminates.

## 14.5 Notes

The language **M** is named after Mendler, on whose work the present treatment is based (Mendler, 1987). Mendler's work is grounded in category theory, specifically the concept of an algebra for a functor (MacLane, 1998; Taylor, 1999). The functorial action of a type constructor (described in Chapter 13) plays a central role. Inductive types are initial algebras and coinductive types are final coalgebras for the functor determined by a (polynomial or positive) type operator.

## Exercises

Preview

**14.1**. Derive the iterator, $\texttt{iter}\, e\, \{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(x) \hookrightarrow e_1\}$, described in Chapter 9 from the recursor for the inductive type of natural numbers given in Section 14.1.

**14.2**. Derive the stream generator, $\texttt{strgen}\, x\, \texttt{is}\, e\, \texttt{in} <\texttt{hd} \hookrightarrow e_1, \texttt{tl} \hookrightarrow e_2>$ from the generator for the coinductive stream type given in Section 14.1.

**14.3**. Consider the type $\texttt{seq} \triangleq \texttt{nat} \to \texttt{nat}$ of infinite sequences of natural numbers. Every stream may be turned into a sequence by the following function:

$$\lambda\,(\texttt{stream}{:}s)\,\lambda\,(n{:}\texttt{nat})\,\texttt{hd}(\texttt{iter}\, n\, \{\texttt{z} \hookrightarrow s \mid \texttt{s}(x) \hookrightarrow \texttt{tl}(x)\}).$$

Show that every sequence may be turned into a stream whose $n$th element is the $n$th element of the given sequence.

**14.4**. The type of lists of natural numbers is defined by the following introductory and eliminatory forms:

$$\frac{}{\Gamma \vdash \texttt{nil} : \texttt{natlist}} \tag{14.10a}$$

$$\frac{\Gamma \vdash e_1 : \texttt{nat} \quad \Gamma \vdash e_2 : \texttt{natlist}}{\Gamma \vdash \texttt{cons}(e_1; e_2) : \texttt{natlist}} \tag{14.10b}$$

$$\frac{\Gamma \vdash e : \texttt{natlist} \quad \Gamma \vdash e_0 : \tau \quad \Gamma\, x : \texttt{nat}\, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \texttt{rec}\, e\, \{\texttt{nil} \hookrightarrow e_0 \mid \texttt{cons}(x; y) \hookrightarrow e_1\} : \tau} \tag{14.10c}$$

The associated dynamics, whether eager or lazy, may be derived from that of the recursor for the type `nat` given in Chapter 9. Give a definition of `natlist` as an inductive type, including the definitions of its associated introductory and eliminatory forms. Check that they validate the expected dynamics.

**14.5**. Consider the type `itree` of possibly infinite binary trees with the following introduction and elimination forms:

$$\frac{\Gamma \vdash e : \texttt{itree}}{\Gamma \vdash \texttt{view}(e) : (\texttt{itree} \times \texttt{itree})\, \texttt{opt}} \tag{14.11a}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma\, x : \tau \vdash e' : (\tau \times \tau)\, \texttt{opt}}{\Gamma \vdash \texttt{itgen}\, x\, \texttt{is}\, e\, \texttt{in}\, e' : \texttt{itree}} \tag{14.11b}$$

Because a possibly infinite tree must be considered only in a state of generation, viewing a tree exposes only its top-level structure, an optional pair of possibly infinite trees.[1] If the view is `null`, the tree is empty, and if it is $\texttt{just}(e_1)e_2$, then it is non-empty, with children given by $e_1$ and $e_2$. To generate an infinite tree, choose a type, $\tau$, of its state of generation, and provide its current state, $e$, and a state transformation, $e'$, that, when applied to the current state, announces whether or not generation is complete, and, if not, provides the state for each of the children.

(a) Give a precise dynamics for the `itree` operations as just described informally. *Hint*: use generic programming!

[1] See Chapter 11 for the definition of option types.

Preview

(b) Reformulate the type `itree` as a coinductive type, and derive the statics and dynamics of its introduction and elimination forms.

Preview

Preview

# Part VI

# Partial Functions

Preview

Preview

# Chapter 15

# Plotkin's PCF

We introduced the language **T** as a basis for discussing total computations, those for which the type system guarantees termination. The language **M** generalizes **T** to admit inductive and coinductive types, while preserving totality. In this chapter we introduce **PCF** as a basis for discussing partial computations, those that may not terminate when evaluated, even when they are well-typed. At first blush this may seem like a disadvantage, but as we shall see in Chapter 16 it admits greater expressive power than is possible in **T**.

The source of partiality in **PCF** is the concept of *general recursion*, which permits the solution of equations between expressions. The price for admitting solutions to all such equations is that computations may not terminate—the solution to some equations might be undefined (divergent). In **PCF** it is the responsibility of the programmer to ensure that a computation terminates; the type system does not ensure that this is so. The advantage of this approach is that the termination proof does not have to be embedded into the code itself, resulting in significantly shorter programs.

For example, consider the equations

$$\begin{aligned} f(0) &\triangleq 1 \\ f(n+1) &\triangleq (n+1) \times f(n). \end{aligned}$$

Intuitively, these equations define the factorial function. They may be seen as a system of simultaneous equations in the unknown, $f$, which ranges over functions on the natural numbers. The function we seek is a *solution* to these equations—a specific function $f : \mathbb{N} \to \mathbb{N}$ such that the above conditions are satisfied.

The solution to such a system of equations may be characterized as the fixed point of an associated functional (higher-order function). To see this, let us re-write these equations in another form:

$$f(n) \triangleq \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1. \end{cases}$$

Preview

Re-writing yet again, we seek $f$ given by

$$n \mapsto \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1. \end{cases}$$

Now define the *functional* $F$ by the equation $F(f) = f'$, where $f'$ is given by

$$n \mapsto \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1. \end{cases}$$

Note well that the condition on $f'$ is expressed in terms of $f$, the argument to the functional $F$, and not in terms of $f'$ itself! The function $f$ we seek is a *fixed point* of $F$, a function $f : \mathbb{N} \to \mathbb{N}$ such that $f = F(f)$. In other words $e$ may be defined to be $\mathit{fix}(F)$, where *fix* is a higher-order operator on functionals $F$ that computes a fixed point for it.

Why should an operator such as $F$ have a fixed point? The key is that functions in **PCF** are *partial*, which means that they may diverge on some (or even all) inputs. Consequently, a fixed point of a functional, $F$, may be obtained as the limit of a series of approximations of the desired solution obtained by iterating $F$. Let us say that a partial function, $\phi$ on the natural numbers, is an *approximation* to a total function, $f$, if $\phi(m) = n$ implies that $f(m) = n$. Let $\perp\colon \mathbb{N} \rightharpoonup \mathbb{N}$ be the totally undefined partial function—$\perp(n)$ is undefined for every $n \in \mathbb{N}$. Intuitively, this is the "worst" approximation to the desired solution, $f$, of the recursion equations given above. Given any approximation, $\phi$, of $f$, we may "improve" it to $\phi' = F(\phi)$. Intuitively, $\phi'$ is defined on 0 and on $m + 1$ for every $m \geq 0$ on which $\phi$ is defined. Continuing in this manner, $\phi'' = F(\phi') = F(F(\phi))$ is an improvement on $\phi'$, and hence a further improvement on $\phi$. If we start with $\perp$ as the initial approximation to $f$, then pass to the limit

$$\lim_{i \geq 0} F^{(i)}(\perp),$$

we will obtain the least approximation to $f$ that is defined for every $m \in \mathbb{N}$, and hence is the function $f$ itself. Turning this around, if the limit exists, it must be the solution we seek.

Because this construction works for any functional $F$, we conclude that *all* such operators have fixed points, and hence that *all* systems of equations such as the one given above have solutions. The solution is given by general recursion, but there is no guarantee that it is a total function (defined on all elements of its domain). For the above example it happens to be the case, because we can prove by induction that this is so, but in general the solution is a partial function that may diverge on some inputs. It is our task as programmers to ensure that the functions defined by general recursion are total, or at least that we have a grasp of those inputs for which it is well-defined.

Preview

## 15.1 Statics

The syntax of **PCF** is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{nat}$ | $\mathtt{nat}$ | naturals |
| | | | $\mathtt{parr}(\tau_1;\tau_2)$ | $\tau_1 \rightharpoonup \tau_2$ | partial function |
| Exp | $e$ | $::=$ | $x$ | $x$ | variable |
| | | | $\mathtt{z}$ | $\mathtt{z}$ | zero |
| | | | $\mathtt{s}(e)$ | $\mathtt{s}(e)$ | successor |
| | | | $\mathtt{ifz}(e;e_0;x.e_1)$ | $\mathtt{ifz}\,e\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}$ | zero test |
| | | | $\mathtt{lam}[\tau](x.e)$ | $\lambda\,(x{:}\tau)\,e$ | abstraction |
| | | | $\mathtt{ap}(e_1;e_2)$ | $e_1(e_2)$ | application |
| | | | $\mathtt{fix}[\tau](x.e)$ | $\mathtt{fix}\,x:\tau\,\mathtt{is}\,e$ | recursion |

The expression $\mathtt{fix}[\tau](x.e)$ is called *general recursion*; it is discussed in more detail below. The expression $\mathtt{ifz}(e;e_0;x.e_1)$ branches according to whether $e$ evaluates to $\mathtt{z}$ or not, binding the predecessor to $x$ in the case that it is not.

The statics of **PCF** is inductively defined by the following rules:

$$\frac{}{\Gamma, x:\tau \vdash x:\tau} \tag{15.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z}:\mathtt{nat}} \tag{15.1b}$$

$$\frac{\Gamma \vdash e:\mathtt{nat}}{\Gamma \vdash \mathtt{s}(e):\mathtt{nat}} \tag{15.1c}$$

$$\frac{\Gamma \vdash e:\mathtt{nat} \quad \Gamma \vdash e_0:\tau \quad \Gamma, x:\mathtt{nat} \vdash e_1:\tau}{\Gamma \vdash \mathtt{ifz}(e;e_0;x.e_1):\tau} \tag{15.1d}$$

$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e):\mathtt{parr}(\tau_1;\tau_2)} \tag{15.1e}$$

$$\frac{\Gamma \vdash e_1:\mathtt{parr}(\tau_2;\tau) \quad \Gamma \vdash e_2:\tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2):\tau} \tag{15.1f}$$

$$\frac{\Gamma, x:\tau \vdash e:\tau}{\Gamma \vdash \mathtt{fix}[\tau](x.e):\tau} \tag{15.1g}$$

Rule (15.1g) reflects the self-referential nature of general recursion. To show that $\mathtt{fix}[\tau](x.e)$ has type $\tau$, we *assume* that it is the case by assigning that type to the variable, $x$, which stands for the recursive expression itself, and checking that the body, $e$, has type $\tau$ under this very assumption.

The structural rules, including in particular substitution, are admissible for the static semantics.

**Lemma 15.1.** *If* $\Gamma, x:\tau \vdash e':\tau'$, $\Gamma \vdash e:\tau$, *then* $\Gamma \vdash [e/x]e':\tau'$.

Preview

## 15.2 Dynamics

The dynamic semantics of **PCF** is defined by the judgments $e$ val, specifying the closed values, and $e \longmapsto e'$, specifying the steps of evaluation.

The judgment $e$ val is defined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{15.2a}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{s}(e)\ \mathsf{val}} \tag{15.2b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{15.2c}$$

The bracketed premise on rule (15.2b) is to be included for the *eager* interpretation of the successor operation, and omitted for the *lazy* interpretation. (See Chapter 38 for a further discussion of laziness.)

The transition judgment $e \longmapsto e'$ is defined by the following rules:

$$\left[\frac{e \longmapsto e'}{\mathtt{s}(e) \longmapsto \mathtt{s}(e')}\right] \tag{15.3a}$$

$$\frac{e \longmapsto e'}{\mathtt{ifz}(e; e_0; x.e_1) \longmapsto \mathtt{ifz}(e'; e_0; x.e_1)} \tag{15.3b}$$

$$\frac{}{\mathtt{ifz}(\mathtt{z}; e_0; x.e_1) \longmapsto e_0} \tag{15.3c}$$

$$\frac{\mathtt{s}(e)\ \mathsf{val}}{\mathtt{ifz}(\mathtt{s}(e); e_0; x.e_1) \longmapsto [e/x]e_1} \tag{15.3d}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1'; e_2)} \tag{15.3e}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1; e_2')}\right] \tag{15.3f}$$

$$\frac{[e_2\ \mathsf{val}]}{\mathtt{ap}(\mathtt{lam}[\tau](x.e); e_2) \longmapsto [e_2/x]e} \tag{15.3g}$$

$$\frac{}{\mathtt{fix}[\tau](x.e) \longmapsto [\mathtt{fix}[\tau](x.e)/x]e} \tag{15.3h}$$

The bracketed rule (15.3a) is to be included for an eager interpretation of the successor, and omitted otherwise. Bracketed rule (15.3f) and the bracketed premise on rule (15.3g) are to be included for a call-by-value interpretation, and omitted for a call-by-name interpretation, of function application. Rule (15.3h) implements self-reference by substituting the recursive expression itself for the variable $x$ in its body; this is called *unwinding* the recursion.

Preview

**Theorem 15.2** (Safety)**.**

1. *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*
2. *If $e : \tau$, then either $e$ val or there exists $e'$ such that $e \longmapsto e'$.*

*Proof.* The proof of preservation is by induction on the derivation of the transition judgment. Consider rule (15.3h). Suppose that $\mathtt{fix}[\tau](x.e) : \tau$. By inversion and substitution we have $[\mathtt{fix}[\tau](x.e)/x]e : \tau$, as required. The proof of progress proceeds by induction on the derivation of the typing judgment. For example, for rule (15.1g) the result follows immediately because we may make progress by unwinding the recursion. □

It is easy to check directly that if $e$ val, then $e$ is irreducible in that there is no $e'$ such that $e \longmapsto e'$. The safety theorem implies the converse, namely that an irreducible expression is a value, provided that it is closed and well-typed.

Definitional equality for the call-by-name variant of **PCF**, written $\Gamma \vdash e_1 \equiv e_2 : \tau$, is defined to be the strongest congruence containing the following axioms:

$$\frac{}{\Gamma \vdash \mathtt{ifz}(\mathtt{z}; e_0; x.e_1) \equiv e_0 : \tau} \tag{15.4a}$$

$$\frac{}{\Gamma \vdash \mathtt{ifz}(\mathtt{s}(e); e_0; x.e_1) \equiv [e/x]e_1 : \tau} \tag{15.4b}$$

$$\frac{}{\Gamma \vdash \mathtt{fix}[\tau](x.e) \equiv [\mathtt{fix}[\tau](x.e)/x]e : \tau} \tag{15.4c}$$

$$\frac{}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}[\tau_1](x.e_2); e_1) \equiv [e_1/x]e_2 : \tau} \tag{15.4d}$$

These rules are sufficient to calculate the value of any closed expression of type `nat`: if $e : \mathtt{nat}$, then $e \equiv \overline{n} : \mathtt{nat}$ iff $e \longmapsto^* \overline{n}$.

## 15.3 Definability

General recursion is a very flexible programming technique that permits a wide variety of functions to be defined within **PCF**. The drawback is that, in contrast to primitive recursion, the termination of a recursively defined function is not intrinsic to the program itself, but rather must be proved extrinsically by the programmer. The benefit is a much greater freedom in writing programs.

Let us write $\mathtt{fun}\, x(y{:}\tau_1){:}\tau_2\, \mathtt{is}\, e$ for a recursive function within whose body, $e : \tau_2$, are bound two variables, $y : \tau_1$ standing for the argument and $x : \tau_1 \rightharpoonup \tau_2$ standing for the function itself. The dynamic semantics of this construct is given by the axiom

$$\frac{}{(\mathtt{fun}\, x(y{:}\tau_1){:}\tau_2\, \mathtt{is}\, e)(e_1) \longmapsto [\mathtt{fun}\, x(y{:}\tau_1){:}\tau_2\, \mathtt{is}\, e, e_1/x, y]e}\ .$$



Preview

That is, to apply a recursive function, we substitute the recursive function itself for $x$ and the argument for $y$ in its body.

Recursive functions may be defined in **PCF** using a combination of recursion and functions, writing

$$\mathtt{fix}\, x : \tau_1 \rightharpoonup \tau_2 \,\mathtt{is}\, \lambda\, (y{:}\tau_1)\, e$$

for $\mathtt{fun}\, x(y{:}\tau_1){:}\tau_2\, \mathtt{is}\, e$. It is a good exercise to check that the static and dynamic semantics of recursive functions are derivable from this definition.

The primitive recursion construct of **T** is defined in **PCF** using recursive functions by taking the expression

$$\mathtt{rec}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x)\, \mathtt{with}\, y \hookrightarrow e_1\}$$

to stand for the application, $e'(e)$, where $e'$ is the general recursive function

$$\mathtt{fun}\, f(u{:}\mathtt{nat}){:}\tau\, \mathtt{is}\, \mathtt{ifz}\, u\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow [f(x)/y]e_1\}.$$

The static and dynamic semantics of primitive recursion are derivable in **PCF** using this expansion.

In general, functions definable in **PCF** are partial in that they may be undefined for some arguments. A partial (mathematical) function, $\phi : \mathbb{N} \rightharpoonup \mathbb{N}$, is *definable* in **PCF** iff there is an expression $e_\phi : \mathtt{nat} \rightharpoonup \mathtt{nat}$ such that $\phi(m) = n$ iff $e_\phi(\overline{m}) \equiv \overline{n} : \mathtt{nat}$. So, for example, if $\phi$ is the totally undefined function, then $e_\phi$ is any function that loops without returning whenever it is called.

It is informative to classify those partial functions $\phi$ that are definable in **PCF**. These are the so-called *partial recursive functions*, which are defined to be the primitive recursive functions augmented by the *minimization* operation: given $\phi(m, n)$, define $\psi(n)$ to be the least $m \geq 0$ such that (1) for $m' < m$, $\phi(m', n)$ is defined and non-zero, and (2) $\phi(m, n) = 0$. If no such $m$ exists, then $\psi(n)$ is undefined.

**Theorem 15.3.** *A partial function $\phi$ on the natural numbers is definable in* **PCF** *iff it is partial recursive.*

*Proof sketch.* Minimization is readily definable in **PCF**, so it is at least as powerful as the set of partial recursive functions. Conversely, we may, with considerable tedium, define an evaluator for expressions of **PCF** as a partial recursive function, using Gödel-numbering to represent expressions as numbers. Consequently, **PCF** does not exceed the power of the set of partial recursive functions. □

Church's Law states that the partial recursive functions coincide with the set of effectively computable functions on the natural numbers—those that can be carried out by a program written in any programming language currently available or that will ever be available.[1] Therefore **PCF** is as powerful as any other programming language with respect to the set of definable functions on the natural numbers.

The universal function, $\phi_{univ}$, for **PCF** is the partial function on the natural numbers defined by

$$\phi_{univ}(\ulcorner e \urcorner)(m) = n \text{ iff } e(\overline{m}) \equiv \overline{n} : \mathtt{nat}.$$

[1] See Chapter 17 for further discussion of Church's Law.

Preview

In contrast to **T**, the universal function $\phi_{univ}$ for **PCF** is partial (may be undefined for some inputs). It is, in essence, an interpreter that, given the code $\overline{\ulcorner e \urcorner}$ of a closed expression of type $\mathtt{nat} \rightharpoonup \mathtt{nat}$, simulates the dynamic semantics to calculate the result, if any, of applying it to the $\overline{m}$, obtaining $\overline{n}$. Because this process may fail to terminate, the universal function is not defined for all inputs.

By Church's Law the universal function is definable in **PCF**. In contrast, we proved in Chapter 9 that the analogous function is *not* definable in **T** using the technique of diagonalization. It is instructive to examine why that argument does not apply in the present setting. As in Section 9.4, we may derive the equivalence

$$e_\Delta(\overline{\ulcorner e_\Delta \urcorner}) \equiv \mathtt{s}(e_\Delta(\overline{\ulcorner e_\Delta \urcorner}))$$

for **PCF**. But now, instead of concluding that the universal function, $e_{univ}$, does not exist as we did for **T**, we instead must conclude for **PCF** that $e_{univ}$ diverges on the code for $e_\Delta$ applied to its own code.

## 15.4 Finite and Infinite Data Structures

Finite data types (products and sums), including their use in pattern matching and generic programming, carry over verbatim to **PCF**. However, the distinction between the eager and lazy dynamics for these constructs becomes much more significant. Rather than being a matter of preference, the decision to use an eager or lazy dynamics is a significant matter of semantics: the "same" types have very different meanings under a lazy dynamics than under an eager dynamics. For example, the elements of a product type in an eager language are pairs of values of the component types. In a lazy language they are instead pairs of unevaluated, possibly divergent, computations of the component types, a very different thing indeed. And similarly for sums.

The situation grows more acute for infinite types such as the type `nat` of "natural numbers." The scare quotes are warranted, because the "same" type has a very different meaning under an eager dynamics than under a lazy dynamics. In the former case the type `nat` is, indeed, the authentic type of natural numbers—the least type containing zero and closed under successor. The principle of mathematical induction is valid for reasoning about the type `nat` in an eager dynamics. It corresponds to the inductive type $\mathtt{nat}_\mathtt{i}$ defined in Chapter 14.

On the other hand, under a lazy dynamics the type `nat` is no longer the type of natural numbers at all. For example, it includes the value

$$\omega \triangleq \mathtt{fix}\, x : \mathtt{nat}\, \mathtt{is}\, \mathtt{s}(x),$$

which has itself as predecessor! It is, intuitively, an "infinite stack of successors", growing without end. It is obviously not a natural number (it is larger than all of them), so the principal of mathematical induction does not apply. In a lazy setting `nat` should be renamed `lnat` to remind us of the distinction; it corresponds to the type $\mathtt{nat}_\mathtt{f}$ defined in Chapter 14.

Unfortunately it is common in lazy languages (those with a lazy dynamics) to use misleading terminology borrowed from eager languages. There is a type `nat`, but it is *not* the natural numbers. There is a type `natlist`, but it is *not* a type of lists! And so on. Rather than being a deliberate



Preview

deception, the misleading terminology arises from a much more significant and fundamental mistake, the very concept of a lazy language itself, for to do is crippling. As we shall see in Chapter 38, laziness is but a *mode of use* of eagerness.

## 15.5 Totality and Partiality

The advantage of a total programming language such as **T** is that it ensures, by type checking, that every program terminates, and that every function is total. There is simply no way to have a well-typed program that goes into an infinite loop. This may seem appealing, until one considers that the upper bound on the time to termination can be quite large, so large that it might as well diverge as far as we humans are concerned. But never mind that, let us grant for the moment that it is a virtue of **T** that it precludes divergence. Why, then, bother with a language such as **PCF** that does not rule out divergence? After all, infinite loops are invariably bugs,[2] so why not rule them out by type checking? The notion seems appealing until one actually tries to write a program in a language such as **T**.

Consider the computation of the greatest common divisor (gcd) of two natural numbers. This can be easily programmed in **PCF** by solving the following equations using general recursion:

$$\begin{aligned} gcd(m,0) &= m \\ gcd(0,n) &= n \\ gcd(m,n) &= gcd(m-n,n) \quad \text{if } m > n \\ gcd(m,n) &= gcd(m,n-m) \quad \text{if } m < n \end{aligned}$$

The type of *gcd* defined in this manner is $(\mathtt{nat} \times \mathtt{nat}) \rightharpoonup \mathtt{nat}$, which suggests that it may not terminate for some inputs. But we may prove by induction on the sum of the pair of arguments that it is, in fact, a total function.

Now consider programming this function in **T**. It is, in fact, programmable using only primitive recursion, but the code to do it is rather painful (try it!). One way to see the problem is that in **T** the only form of looping is one that reduces a natural number by one on each recursive call; it is not (directly) possible to make a recursive call on a smaller number other than the immediate predecessor. In fact one may code up more general patterns of terminating recursion using only primitive recursion as a primitive, but if you examine the details, you will see that doing so comes at a significant price in performance and program complexity. Program complexity can be mitigated by building libraries that codify standard patterns of reasoning whose cost of development should be amortized over all programs, not just one in particular. But there is still the problem of performance. Indeed, the encoding of more general forms of recursion into primitive recursion means that, deep within the encoding, there must be a "timer" that goes down by ones to ensure that the program terminates. The result will be that programs written with such libraries will not be nearly as fast as they ought to be.

[2] Do not be fooled by glib arguments about useful programs, such as operating systems, that "run forever". For example, infinite streams are programmable in **M**, in which all functions terminate. Computing infinitely does not mean running forever, it just means "for as long as one wishes, without bound."

Preview

But, one may argue, **T** is simply not a serious language. A more serious total programming language would admit sophisticated patterns of control without performance penalty. Indeed, one could easily envision representing the natural numbers in binary, rather than unary, and allowing recursive calls to be made by halving to achieve logarithmic complexity. This is surely possible, as are numerous other such techniques. Could we not then have a practical language that rules out divergence?

We can, but at a cost. We have already seen one limitation of total programming languages: they are not universal. You cannot write an interpreter for **T** within **T**, and this limitation extends to any total language whatever. If this does not seem important, then consider the *Blum Size Theorem (BST)*, which places another limitation on total languages. Fix *any* total language, $\mathcal{L}$, that permits writing functions on the natural numbers. Pick any blowup factor, say $2^{2^n}$, or however expansive you wish to be. The BST states that there is a total function on the natural numbers that is programmable in $\mathcal{L}$, but whose shortest program in $\mathcal{L}$ is larger by the given blowup factor than its shortest program in **PCF**!

The underlying idea of the proof is that *in a total language the proof of termination of a program must be baked into the code itself*, whereas *in a partial language the termination proof is an external verification condition left to the programmer*. Roughly speaking, there are, and always will be, programs whose termination proof is rather complicated to express, if you fix in advance the means by which it may be proved total. (In **T** it was primitive recursion, but one can be more ambitious, yet still get caught by the BST.) But if you leave room for ingenuity, then programs can be short, precisely because they do not have to embed the proof of their termination in their own running code.

## 15.6 Notes

The characterization of functions as fixed points is based on Kleene's fixed point theorem for complete partial orders applied to the subset ordering of partial functions. The language **PCF** is derived from Plotkin (1977). Plotkin introduced PCF to study the relationship between its operational and denotational semantics. Subsequently, many authors have used PCF as the basis for the study of many issues in semantics. It has become, in effect, the *E. coli* of programming languages.

## Exercises

**15.1**. Consider the problem considered in Section 10.3 of how to define the mutually recursive "even" and "odd" functions. There we gave a solution in terms of primitive recursion. Your are, instead, to give a solution in terms of general recursion. *Hint*: consider that a pair of mutually recursive functions is a recursive pair of functions.

**15.2**. Show that minimization, as explained immediately before the statement of Theorem 15.3, is definable in **PCF**.

**15.3**. Consider the partial function $\phi_{halts}$ such that if $e : \mathtt{nat} \rightharpoonup \mathtt{nat}$, then $\phi_{halts}(\ulcorner e \urcorner)$ evaluates to zero iff $e(\ulcorner e \urcorner)$ converges, and evaluates to one otherwise. Prove that $\phi_{halts}$ is not definable in **PCF**.



Preview

**15.4**. Suppose that we changed the specification of minimization given prior to Theorem 15.3 so that $\psi(n)$ is simply defined to be the least $m$ such that $\phi(m,n) = 0$, and is undefined if no such $m$ exists. Is this "simplified" form of minimization definable in **PCF**?

**15.5**. Suppose that we wished to define, in the lazy variant of **PCF**, a version of the *parallel or* function specified a function of two arguments that returns $\mathtt{z}$ if either of its arguments is $\mathtt{z}$, and $\mathtt{s}(\mathtt{z})$ otherwise. That is, we wish to find an expression, $e$, satisfying the following properties:

$$e(e_1)(e_2) \longmapsto^* \mathtt{z} \text{ if } e_1 \longmapsto^* \mathtt{z}$$
$$e(e_1)(e_2) \longmapsto^* \mathtt{z} \text{ if } e_2 \longmapsto^* \mathtt{z}$$
$$e(e_1)(e_2) \longmapsto^* \mathtt{s}(\mathtt{z}) \text{ otherwise}$$

Thus, $e$ defines a total function of its two arguments, *even if* one of the arguments diverges. Clearly such a function cannot be defined in the call-by-value variant of **PCF**, but can it be defined in the call-by-name variant? If so, show how; if not, prove that it cannot be, and suggest an extension of **PCF** that would allow it to be defined.

**15.6**. We appealed to Church's Law to argue that the universal function for **PCF** is definable in **PCF**. Examine what is behind this claim by considering two aspects of the problem: (1) Gödel-numbering, the representation of abstract syntax by a number; (2) evaluation, the process of interpreting a function on its inputs. Part (1) is a purely technical issue arising from the limited data structures available in **PCF**. Part (2) is the heart of the matter; explore its implementation relative to a solution to Part (1).

Preview

# Chapter 16

# Recursive Types

In this chapter we study a variant of **PCF**, called **FPC**, in which we have partial functions, products, sums, and *recursive types*. Recursive types are solutions to type equations $t \cong \tau$ in which there is no restriction on where $t$ may occur in $\tau$. Equivalently, a recursive type is a *fixed point* up to isomorphism of the associated unrestricted type operator $t.\tau$. By removing the restrictions on the type operator we may consider the solution of a type equation such as $t \cong t \rightharpoonup t$, which describes a type that is isomorphic to the type of partial functions define on itself. If types were sets, such an equation could not be solved, because there are more partial functions on a set than there are elements of that set. But *types are not sets*: they classify *computable* functions, not *arbitrary* functions. With types we may solve such "dubious" type equations, even though we cannot expect to do so with sets. The penalty is that we must admit the possibility of non-termination. For one thing, type equations involving functions have solutions only if the functions involved are permitted to be partial. For another, as we shall see in Section 16.3, having solutions to all type equations implies that divergent (and other, useful, self-referential) computations exist.

A benefit of working in the setting of partial functions is that type equations have *unique* solutions (up to isomorphism). Therefore it makes sense, as we shall do in this chapter, to speak of *the* solution to a type equation. But what about the *distinct* solutions to a type equation given in Chapter 14? These turn out to coincide for any fixed dynamics, but give rise to different solutions according to whether the dynamics is eager or lazy (as illustrated in Section 15.4 for the special case of the natural numbers). Under a lazy dynamics (in which all constructs are evaluated lazily), recursive types have a coinductive flavor, and the inductive analogues are inaccessible. Under an eager dynamics (in which all contructs are evaluated eagerly), recursive types have an inductive flavor. But the coinductive analogues are accessible as well, using function types to selectively impose laziness.[1] It follows that the eager dynamics is *more expressive* than the lazy dynamics, because it is impossible to go the other way around (one cannot define inductive types in a lazy language).

[1] This suggests that there could be a type distinction between, say, eager and lazy products, as suggested in Exercise **10.2**. This possibility is developed systematically in Chapter 39.

Preview

## 16.1 Solving Type Equations

The language **FPC** has products, sums, and partial functions inherited from the preceding development, extended with the new concept of recursive types. The syntax of recursive types is defined as follows:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $t$ | $t$ | self-reference |
| | | | $\mathtt{rec}(t.\tau)$ | $\mathtt{rec}\, t\, \mathtt{is}\, \tau$ | recursive type |
| Exp | $e$ | $::=$ | $\mathtt{fold}[t.\tau](e)$ | $\mathtt{fold}(e)$ | fold |
| | | | $\mathtt{unfold}(e)$ | $\mathtt{unfold}(e)$ | unfold |

The subscript on the concrete syntax of `fold` is often omitted when it is clear from context.

Recursive types have the same general form as the inductive and coinductive types discussed in Chapter 14, but without restriction on the type operator involved. The formation of a recursive type is governed by the following rule:

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathtt{rec}(t.\tau)\ \mathsf{type}} \tag{16.1}$$

The statics of folding and unfolding is given by the following rules:

$$\frac{\Gamma \vdash e : [\mathtt{rec}(t.\tau)/t]\tau}{\Gamma \vdash \mathtt{fold}[t.\tau](e) : \mathtt{rec}(t.\tau)} \tag{16.2a}$$

$$\frac{\Gamma \vdash e : \mathtt{rec}(t.\tau)}{\Gamma \vdash \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{16.2b}$$

The dynamics of folding and unfolding is given by these rules:

$$\frac{[e\ \mathsf{val}]}{\mathtt{fold}[t.\tau](e)\ \mathsf{val}} \tag{16.3a}$$

$$\left[\frac{e \longmapsto e'}{\mathtt{fold}[t.\tau](e) \longmapsto \mathtt{fold}[t.\tau](e')}\right] \tag{16.3b}$$

$$\frac{e \longmapsto e'}{\mathtt{unfold}(e) \longmapsto \mathtt{unfold}(e')} \tag{16.3c}$$

$$\frac{\mathtt{fold}[t.\tau](e)\ \mathsf{val}}{\mathtt{unfold}(\mathtt{fold}[t.\tau](e)) \longmapsto e} \tag{16.3d}$$

The bracketed premise and rule are to be included for an *eager* interpretation of the introduction form, and omitted for a *lazy* interpretation. As mentioned in the introduction, the choice of eager or lazy dynamics affects the nature of the recursive types we obtain.

**Theorem 16.1** (Safety)**.** *1. If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

*2. If $e : \tau$, then either $e$ val, or there exists $e'$ such that $e \longmapsto e'$.*

Preview

## 16.2 Inductive and Coinductive Types

One important application of recursive types is the representation of inductive data types such as the type of natural numbers. Using an *eager* dynamics for **FPC**, the recursive type

$$\rho = \mathtt{rec}\, t\, \mathtt{is}\, [\mathtt{z} \hookrightarrow \mathtt{unit}, \mathtt{s} \hookrightarrow t]$$

satisfies the type equation

$$\rho \cong [\mathtt{z} \hookrightarrow \mathtt{unit}, \mathtt{s} \hookrightarrow \rho],$$

and is isomorphic to the type of eager natural numbers. The introduction and elimination forms are defined on $\rho$ by the following equations:[2]

$$\begin{aligned}
\mathtt{z} &\triangleq \mathtt{fold}(\mathtt{z} \cdot \langle\rangle) \\
\mathtt{s}(e) &\triangleq \mathtt{fold}(\mathtt{s} \cdot e) \\
\mathtt{ifz}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\} &\triangleq \mathtt{case}\, \mathtt{unfold}(e)\, \{\mathtt{z} \cdot _ \hookrightarrow e_0 \mid \mathtt{s} \cdot x \hookrightarrow e_1\}.
\end{aligned}$$

It is a good exercise to check that the eager dynamics of natural numbers in **PCF** is correctly simulated by these definitions.

On the other hand, under a lazy dynamics for **FPC**, the same recursive type, $\rho'$,

$$\mathtt{rec}\, t\, \mathtt{is}\, [\mathtt{z} \hookrightarrow \mathtt{unit}, \mathtt{s} \hookrightarrow t],$$

satisfies the same type equation,

$$\rho' \cong [\mathtt{z} \hookrightarrow \mathtt{unit}, \mathtt{s} \hookrightarrow \rho'],$$

but is not the type of natural numbers! Rather, it is the type `lnat` of lazy natural numbers introduced in Section 15.4. As discussed there, the type $\rho'$ contains the "infinite number" $\omega$, which is of course not a natural number.

Similarly, using an eager dynamics for **FPC**, the type `natlist` of lists of natural numbers may be defined by the recursive type

$$\mathtt{rec}\, t\, \mathtt{is}\, [\mathtt{n} \hookrightarrow \mathtt{unit}, \mathtt{c} \hookrightarrow \mathtt{nat} \times t],$$

which satisfies the type equation

$$\mathtt{natlist} \cong [\mathtt{n} \hookrightarrow \mathtt{unit}, \mathtt{c} \hookrightarrow \mathtt{nat} \times \mathtt{natlist}].$$

The list introduction operations are represented by the following equations:

$$\begin{aligned}
\mathtt{nil} &\triangleq \mathtt{fold}(\mathtt{n} \cdot \langle\rangle) \\
\mathtt{cons}(e_1; e_2) &\triangleq \mathtt{fold}(\mathtt{c} \cdot \langle e_1, e_2\rangle).
\end{aligned}$$

[2] The "underscore" stands for a variable that does not occur free in $e_0$.



Preview

A conditional list elimination form is represented by the following equation:

$$\texttt{case}\, e\, \{\texttt{nil} \hookrightarrow e_0 \mid \texttt{cons}(x; y) \hookrightarrow e_1\} \triangleq \texttt{case}\, \texttt{unfold}(e)\, \{\texttt{n} \cdot _ \hookrightarrow e_0 \mid \texttt{c} \cdot \langle x, y\rangle \hookrightarrow e_1\},$$

in which we have used pattern-matching syntax to bind the components of a pair for the sake of clarity.

Now consider the *same* recursive type, but in the context of a lazy dynamics for **FPC**. What type is it? If all constructs are evaluated lazily, then a value of the recursive type

$$\texttt{rec}\, t\, \texttt{is}\, [\texttt{n} \hookrightarrow \texttt{unit}, \texttt{c} \hookrightarrow \texttt{nat} \times t],$$

has the form $\texttt{fold}(e)$, where $e$ is an unevaluated computation of the sum type, whose values are injections of unevaluated computations of either the unit type or of the product type $\texttt{nat} \times t$. And the latter consists of pairs of an unevaluated computation of a (lazy!) natural number, and an unevaluated computation of another value of this type. In particular, this type contains infinite lists whose tails go on without end, as well as finite lists that eventually reach an end. The type is, in fact, a version of the type of infinite streams defined in Chapter 14, rather than a type of finite lists as is the case under an eager dynamics.

It is common in textbooks to depict data structures using "box-and-pointer" diagrams. This works well in the eager setting, provided that no functions are involved. For example, an eager list of eager natural numbers may be depicted using this notation. We may think of `fold` as an abstract pointer to a tagged cell consisting of either (a) the tag `n` with no associated data, or (b) the tag `c` attached to a pair consisting of an authentic natural number and another list, which must be an abstract pointer of the same type. But this notation does not scale well to types involving functions, or to languages with a lazy dynamics. For example, the recursive type of "lists" in lazy **FPC** cannot be depicted using boxes and pointers, precisely because of the unevaluated computations occuring in values of this type. It is a mistake to limit one's conception of data structures to those that can be drawn on the blackboard using boxes and pointers or similar informal notations. There is no substitute for a programming language to express data structures fully and accurate.

It is deceiving that the "same" recursive type can have two different meanings according to whether the underlying dynamics is eager or lazy. For example, it is common for lazy languages to use the name "list" for the recursive type of streams, or the name "nat" for the type of lazy natural numbers. This is a highly misleading choice of terminology, especially considering that such languages do not (and can not) have a proper type of finite lists or a type of natural numbers. *Caveat emptor!*

## 16.3 Self-Reference

In the general recursive expression, $\texttt{fix}[\tau](x.e)$, the variable, $x$, stands for the expression itself. This is ensured by the unrolling transition

$$\texttt{fix}[\tau](x.e) \longmapsto [\texttt{fix}[\tau](x.e)/x]e,$$

which substitutes the expression itself for $x$ in its body during execution. It is useful to think of $x$ as an *implicit argument* to $e$ that is instantiated to itself whenever the expression is used. In



Preview

many well-known languages this implicit argument has a special name, such as `this` or `self`, to emphasize its self-referential interpretation.

Using this intuition as a guide, we may derive general recursion from recursive types. This derivation shows that general recursion may, like other language features, be seen as a manifestation of type structure, rather than included as an *ad hoc* language feature. The derivation is based on isolating a type of self-referential expressions given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{self}(\tau)$ | $\tau\ \mathtt{self}$ | self-referential type |
| Exp | $e$ | $::=$ | $\mathtt{self}[\tau](x.e)$ | $\mathtt{self}\ x\ \mathtt{is}\ e$ | self-referential expression |
| | | | $\mathtt{unroll}(e)$ | $\mathtt{unroll}(e)$ | unroll self-reference |

The statics of these constructs is given by the following rules:

$$\frac{\Gamma, x : \mathtt{self}(\tau) \vdash e : \tau}{\Gamma \vdash \mathtt{self}[\tau](x.e) : \mathtt{self}(\tau)} \tag{16.4a}$$

$$\frac{\Gamma \vdash e : \mathtt{self}(\tau)}{\Gamma \vdash \mathtt{unroll}(e) : \tau} \tag{16.4b}$$

The dynamics is given by the following rule for unrolling the self-reference:

$$\frac{}{\mathtt{self}[\tau](x.e)\ \mathsf{val}} \tag{16.5a}$$

$$\frac{e \longmapsto e'}{\mathtt{unroll}(e) \longmapsto \mathtt{unroll}(e')} \tag{16.5b}$$

$$\frac{}{\mathtt{unroll}(\mathtt{self}[\tau](x.e)) \longmapsto [\mathtt{self}[\tau](x.e)/x]e} \tag{16.5c}$$

The main difference, compared to general recursion, is that we distinguish a type of self-referential expressions, rather than impose self-reference at every type. However, as we shall see shortly, the self-referential type is sufficient to implement general recursion, so the difference is largely one of technique.

The type $\mathtt{self}(\tau)$ is definable from recursive types. As suggested earlier, the key is to consider a self-referential expression of type $\tau$ to be a function of the expression itself. That is, we seek to define the type $\mathtt{self}(\tau)$ so that it satisfies the isomorphism

$$\mathtt{self}(\tau) \cong \mathtt{self}(\tau) \rightharpoonup \tau.$$

This means that we seek a fixed point of the type operator $t.t \rightharpoonup \tau$, where $t \notin \tau$ is a type variable standing for the type in question. The required fixed point is just the recursive type

$$\mathtt{rec}(t.t \rightharpoonup \tau),$$

which we take as the definition of $\mathtt{self}(\tau)$.

Preview

The self-referential expression $\texttt{self}[\tau](x.e)$ is then defined to be the expression

$$\texttt{fold}(\lambda\,(x{:}\texttt{self}(\tau))\,e).$$

We may easily check that rule (16.4a) is derivable according to this definition. The expression $\texttt{unroll}(e)$ is correspondingly defined to be the expression

$$\texttt{unfold}(e)(e).$$

It is easy to check that rule (16.4b) is derivable from this definition. Moreover, we may check that

$$\texttt{unroll}(\texttt{self}[\tau](y.e)) \longmapsto^* [\texttt{self}[\tau](y.e)/y]e.$$

This completes the derivation of the type $\texttt{self}(\tau)$ of self-referential expressions of type $\tau$.

One consequence of admitting the self-referential type $\texttt{self}(\tau)$ is that we may use it to define general recursion for *any* type. To be precise, we may define $\texttt{fix}[\tau](x.e)$ to stand for the expression

$$\texttt{unroll}(\texttt{self}[\tau](y.[\texttt{unroll}(y)/x]e))$$

in which we have unrolled the recursion at each occurrence of $x$ within $e$. It is easy to check that this verifies the statics of general recursion given in Chapter 15. Moreover, it also validates the dynamics, as evidenced by the following derivation:

$$\begin{aligned}\texttt{fix}[\tau](x.e) &= \texttt{unroll}(\texttt{self}[\tau](y.[\texttt{unroll}(y)/x]e))\\ &\longmapsto^* [\texttt{unroll}(\texttt{self}[\tau](y.[\texttt{unroll}(y)/x]e))/x]e\\ &= [\texttt{fix}[\tau](x.e)/x]e.\end{aligned}$$

It follows that recursive types may be used to define a non-terminating expression of every type, namely $\texttt{fix}[\tau](x.x)$.

## 16.4 The Origin of State

The notion of *state* in a computation—which will be discussed thoroughly in Part XIV—has its origins in the concept of recursion, or self-reference, which, as we have just seen, arises from the concept of recursive types. For example, you may be familiar with the concept of a *flip-flop* or a *latch* at the hardware level. These are circuits built from combinational logic elements (typically, nor or nand gates) that have the characteristic that they maintain an alterable state over time. An RS latch, for example, maintains its output at the logical level of zero or one in response to a signal on the R or S inputs, respectively, after a brief settling delay. This behavior is achieved using *feedback*, which is just a form of self-reference, or recursion: the output of the gate is fed back into its input so as to convey the current state of the gate to the logic that determines its next state.

We can implement an RS latch using recursive types. The idea is to use self-reference to model the passage of time, with the current output being a function of its input and its previous outputs. Specifically, an RS latch is a value of type $\tau_{rsl}$ given by

$$\texttt{rec}\,t\,\texttt{is}\,\langle \texttt{X} \hookrightarrow \texttt{bool}, \texttt{Q} \hookrightarrow \texttt{bool}, \texttt{N} \hookrightarrow t\rangle.$$



Preview

The X and Q components of the latch represent its current outputs (of which Q represents the current state of the latch), and the N component represents the next state of the latch. If $e$ is of type $\tau_{rsl}$, then we define $e$@X to mean $\mathtt{unfold}(e) \cdot \mathtt{X}$, and define $e$@Q and $e$@N similarly. The expressions $e$@X and $e$@Q evaluate to the boolean outputs of the latch $e$, and $e$@N evaluates to another latch representing its evolution over time based on these inputs.

For given values, $r$ and $s$, a new latch is computed from an old latch by the recursive function *rsl* defined as follows:[3]

$$\mathtt{fix}\, rsl\, \mathtt{is}\, \lambda\, (l{:}\tau_{rsl})\, e_{rsl},$$

where $e_{rsl}$ is the expression

$$\mathtt{fix}\, this\, \mathtt{is}\, \mathtt{fold}(\langle \mathtt{X} \hookrightarrow e_{nor}(\langle s, l\,@\,\mathtt{Q}\rangle), \mathtt{Q} \hookrightarrow e_{nor}(\langle r, l\,@\,\mathtt{X}\rangle), \mathtt{N} \hookrightarrow rsl(this)\rangle),$$

where $e_{nor}$ is the obvious binary function on booleans. The outputs of the latch are computed as a function of the $r$ and $s$ inputs and the ouputs of the previous state of the latch. To get the construction started, we define an initial state of the latch in which the outputs are arbitrarily set to false, and whose next state is determined by applying the recursive function *rsl* to that state:

$$\mathtt{fix}\, this\, \mathtt{is}\, \mathtt{fold}(\langle \mathtt{X} \hookrightarrow \mathtt{false}, \mathtt{Q} \hookrightarrow \mathtt{false}, \mathtt{N} \hookrightarrow rsl(this)\rangle).$$

Selection of the N component causes the outputs to be recalculated based on the current outputs. Notice the role of self-reference in maintaining the state of the latch.

## 16.5 Notes

The systematic study of recursive types in programming was initiated by Scott (1976, 1982) to provide a mathematical model of the untyped $\lambda$-calculus. The derivation of recursion from recursive types is essentially an application of Scott's theory to find the interpretation of a fixed point combinator in a model of the $\lambda$-calculus given by a recursive type. The category-theoretic view of recursive types was developed by Wand (1979) and Smyth and Plotkin (1982). Implementing state using self-reference is fundamental to digital logic. Abadi and Cardelli (1996) and Cook (2009), among others, explore similar ideas to model objects. The account of signals as streams is inspired by the pioneering work of Kahn (MacQueen, 2009). The language name **FPC** is borrowed from Gunter (1992).

## Exercises

**16.1**. Show that the recursive type $\mathtt{D} \triangleq \mathtt{rec}\, t\, \mathtt{is}\, t \rightharpoonup t$ is non-trivial by interpreting the sk-combinators defined in Exercise **3.1** into it. Specifically, define elements $\mathtt{k} : \mathtt{D}$ and $\mathtt{s} : \mathtt{D}$ and a (left-associative) "application" function

$$x : \mathtt{D}\, y : \mathtt{D} \vdash x \cdot y : \mathtt{D}$$

such that

[3] For convenience we assume that fold is evaluated lazily.



Preview

(a) $\mathtt{k} \cdot x \cdot y \longmapsto^* x$;

(b) $\mathtt{s} \cdot x \cdot y \cdot z \longmapsto^* (x \cdot z) \cdot (y \cdot z)$.

**16.2**. Recursive types admit the structure of both inductive and coinductive types. Consider the recursive type $\tau \triangleq \mathtt{rec}\, t\, \mathtt{is}\, \tau'$ and the associated inductive and coinductive types $\mu(t.\tau')$ and $\nu(t.\tau')$. Complete the following chart consistently with the statics of inductive and coinductive types on the left-hand side and with the statics of recursive types on the right:

$$\begin{aligned} \mathtt{fold}_{t.t\,\mathtt{opt}}(e) &\triangleq \mathtt{fold}(e) \\ \mathtt{rec}(x.e'; e) &\triangleq \mathtt{?} \\ \mathtt{unfold}_{t.t\,\mathtt{opt}}(e) &\triangleq \mathtt{unfold}(e) \\ \mathtt{gen}(x.e'; e) &\triangleq \mathtt{?} \end{aligned}$$

Check that the statics is derivable under these definitions. *Hint*: you will need to use general recursion on the right to fill in the missing cases. You may also find it useful to use generic programming.

Now consider the dynamics of these definitions, under both an eager and a lazy interpretation. What happens in each case?

**16.3**. Define the type `signal` of *signals* to be the coinductive type of infinite streams of booleans (bits). Define a *signal transducer* to be a function of type $\mathtt{signal} \rightharpoonup \mathtt{signal}$. Combinational logic gates, such as the NOR gate, may be defined as signal transducers. Give a coinductive definition of the type `signal`, and define NOR as a signal transducer. Be sure to take account of the underlying dynamics of **PCF**.

The passage from combinational to digital logic (circuit elements that maintain state) relies on self-reference. For example, an RS latch may be built from NOR two nor gates in this manner. Define an RS latch using general recursion and two of the NOR gates just defined.

**16.4**. The type $\tau_{rsl}$ given in Section 16.4 above may be interpreted as the type of streams of pairs of booleans. Give another formulation of an RS latch as a value of type $\tau_{rsl}$, but this time using the coinductive interpretation of the recursive type proposed in Exercise **16.2** (using the lazy dynamics for **FPC**). Expand and simplify this definition using your solution to Exercise **16.2**, and compare it with the formulation given in Section 16.4. *Hint*: the internal state of the stream should be a pair of booleans corresponding to the `X` and `Q` outputs of the latch.

Preview

# Part VII

# Dynamic Types

Preview

Preview

# Chapter 17

# Church's Untyped $\lambda$-Calculus

In this chapter we study the premier example of a uni-typed programming language, the *(untyped) $\lambda$-calculus*. This formalism was introduced by Church in the 1930's as a universal language of computable functions. It is distinctive for its austere elegance. The $\lambda$-calculus has but one "feature", the higher-order function. Everything is a function, hence every expression may be applied to an argument, which must itself be a function, with the result also being a function. To borrow a turn of phrase, in the $\lambda$-calculus it's functions all the way down.

## 17.1 The $\lambda$-Calculus

The abstract syntax of the untyped $\lambda$-calculus, called $\mathbf{\Lambda}$, is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Exp | $u$ | $::=$ | $x$ | $x$ | variable |
| | | | $\lambda(x.u)$ | $\lambda\,(x)\,u$ | $\lambda$-abstraction |
| | | | $\mathsf{ap}(u_1; u_2)$ | $u_1(u_2)$ | application |

The statics of $\mathbf{\Lambda}$ is defined by general hypothetical judgments of the form $x_1\ \mathsf{ok}, \ldots, x_n\ \mathsf{ok} \vdash u\ \mathsf{ok}$, stating that $u$ is a well-formed expression involving the variables $x_1$, ..., $x_n$. (As usual, we omit explicit mention of the parameters when they can be determined from the form of the hypotheses.) This relation is inductively defined by the following rules:

$$\frac{}{\Gamma, x\ \mathsf{ok} \vdash x\ \mathsf{ok}} \tag{17.1a}$$

$$\frac{\Gamma \vdash u_1\ \mathsf{ok} \quad \Gamma \vdash u_2\ \mathsf{ok}}{\Gamma \vdash u_1(u_2)\ \mathsf{ok}} \tag{17.1b}$$

$$\frac{\Gamma, x\ \mathsf{ok} \vdash u\ \mathsf{ok}}{\Gamma \vdash \lambda\,(x)\,u\ \mathsf{ok}} \tag{17.1c}$$

Preview

The dynamics of $\mathbf{\Lambda}$ is given equationally, rather than via a transition system. Definitional equality for $\mathbf{\Lambda}$ is a judgment of the form $\Gamma \vdash u \equiv u'$, where $\Gamma = x_1 \ \mathsf{ok}, \ldots, x_n \ \mathsf{ok}$ for some $n \geq 0$, and $u$ and $u'$ are terms having at most the variables $x_1, \ldots, x_n$ free. It is inductively defined by the following rules:

$$\frac{}{\Gamma, u \ \mathsf{ok} \vdash u \equiv u} \tag{17.2a}$$

$$\frac{\Gamma \vdash u \equiv u'}{\Gamma \vdash u' \equiv u} \tag{17.2b}$$

$$\frac{\Gamma \vdash u \equiv u' \quad \Gamma \vdash u' \equiv u''}{\Gamma \vdash u \equiv u''} \tag{17.2c}$$

$$\frac{\Gamma \vdash u_1 \equiv u_1' \quad \Gamma \vdash u_2 \equiv u_2'}{\Gamma \vdash u_1(u_2) \equiv u_1'(u_2')} \tag{17.2d}$$

$$\frac{\Gamma, x \ \mathsf{ok} \vdash u \equiv u'}{\Gamma \vdash \lambda\,(x)\,u \equiv \lambda\,(x)\,u'} \tag{17.2e}$$

$$\frac{\Gamma, x \ \mathsf{ok} \vdash u_2 \ \mathsf{ok} \quad \Gamma \vdash u_1 \ \mathsf{ok}}{\Gamma \vdash (\lambda\,(x)\,u_2)(u_1) \equiv [u_1/x]u_2} \tag{17.2f}$$

We often write just $u \equiv u'$ when the variables involved need not be emphasized or are clear from context.

## 17.2 Definability

Interest in the untyped $\lambda$-calculus stems from its surprising expressiveness. It is a *Turing-complete* language in the sense that it has the same capability to express computations on the natural numbers as does any other known programming language. Church's Law states that any conceivable notion of computable function on the natural numbers is equivalent to the $\lambda$-calculus. This is certainly true for all *known* means of defining computable functions on the natural numbers. The force of Church's Law is that it postulates that all future notions of computation will be equivalent in expressive power (measured by definability of functions on the natural numbers) to the $\lambda$-calculus. Church's Law is therefore a *scientific law* in the same sense as, say, Newton's Law of Universal Gravitation, which makes a prediction about all future measurements of the acceleration in a gravitational field.[1]

We will sketch a proof that the untyped $\lambda$-calculus is as powerful as the language PCF described in Chapter 15. The main idea is to show that the PCF primitives for manipulating the natural numbers are definable in the untyped $\lambda$-calculus. This means, in particular, that we must show that the natural numbers are definable as $\lambda$-terms in such a way that case analysis, which

[1] Unfortunately, it is common in Computer Science to put forth as "laws" assertions that are not scientific laws at all. For example, Moore's Law is merely an observation about a near-term trend in microprocessor fabrication that is certainly not valid over the long term, and Amdahl's Law is but a simple truth of arithmetic. Worse, Church's Law, which is a proper scientific law, is usually called *Church's Thesis*, which, to the author's ear, suggests something less than the full force of a scientific law.

Preview

discriminates between zero and non-zero numbers, is definable. The principal difficulty is with computing the predecessor of a number, which requires a bit of cleverness. Finally, we show how to represent general recursion, completing the proof.

The first task is to represent the natural numbers as certain $\lambda$-terms, called the *Church numerals*.

$$\overline{0} \triangleq \lambda\,(b)\,\lambda\,(s)\,b \tag{17.3a}$$

$$\overline{n+1} \triangleq \lambda\,(b)\,\lambda\,(s)\,s(\overline{n}(b)(s)) \tag{17.3b}$$

It follows that

$$\overline{n}(u_1)(u_2) \equiv u_2(\ldots(u_2(u_1))),$$

the $n$-fold application of $u_2$ to $u_1$. That is, $\overline{n}$ iterates its second argument (the induction step) $n$ times, starting with its first argument (the basis).

Using this definition it is not difficult to define the basic functions of arithmetic. For example, successor, addition, and multiplication are defined by the following untyped $\lambda$-terms:

$$\mathtt{succ} \triangleq \lambda\,(x)\,\lambda\,(b)\,\lambda\,(s)\,s(x(b)(s)) \tag{17.4}$$

$$\mathtt{plus} \triangleq \lambda\,(x)\,\lambda\,(y)\,y(x)(\mathtt{succ}) \tag{17.5}$$

$$\mathtt{times} \triangleq \lambda\,(x)\,\lambda\,(y)\,y(\overline{0})(\mathtt{plus}(x)) \tag{17.6}$$

It is easy to check that $\mathtt{succ}(\overline{n}) \equiv \overline{n+1}$, and that similar correctness conditions hold for the representations of addition and multiplication.

To define $\mathtt{ifz}(u; u_0; x.u_1)$ requires a bit of ingenuity. The key is to define the "cut-off predecessor", `pred`, such that

$$\mathtt{pred}(\overline{0}) \equiv \overline{0} \tag{17.7}$$

$$\mathtt{pred}(\overline{n+1}) \equiv \overline{n}. \tag{17.8}$$

To compute the predecessor using Church numerals, we must show how to compute the result for $\overline{n+1}$ as a function of its value for $\overline{n}$. At first glance this seems simple—just take the successor—until we consider the base case, in which we define the predecessor of $\overline{0}$ to be $\overline{0}$. This invalidates the obvious strategy of taking successors at inductive steps, and necessitates some other approach.

What to do? A useful intuition is to think of the computation in terms of a pair of "shift registers" satisfying the invariant that on the $n$th iteration the registers contain the predecessor of $n$ and $n$ itself, respectively. Given the result for $n$, namely the pair $(n-1, n)$, we pass to the result for $n+1$ by shifting left and incrementing to obtain $(n, n+1)$. For the base case, we initialize the registers with $(0, 0)$, reflecting the stipulation that the predecessor of zero be zero. To compute the predecessor of $n$ we compute the pair $(n-1, n)$ by this method, and return the first component.

To make this precise, we must first define a Church-style representation of ordered pairs.

$$\langle u_1, u_2\rangle \triangleq \lambda\,(f)\,f(u_1)(u_2) \tag{17.9}$$

$$u \cdot \mathtt{l} \triangleq u(\lambda\,(x)\,\lambda\,(y)\,x) \tag{17.10}$$

$$u \cdot \mathtt{r} \triangleq u(\lambda\,(x)\,\lambda\,(y)\,y) \tag{17.11}$$



Preview

It is easy to check that under this encoding $\langle u_1, u_2\rangle \cdot \mathtt{l} \equiv u_1$, and that a similar equivalence holds for the second projection. We may now define the required representation, $u_p$, of the predecessor function:

$$u'_p \triangleq \lambda\,(x)\, x(\langle \overline{0}, \overline{0}\rangle)(\lambda\,(y)\,\langle y \cdot \mathtt{r}, \mathtt{succ}\,(y \cdot \mathtt{r})\rangle) \tag{17.12}$$

$$u_p \triangleq \lambda\,(x)\, u'_p(x) \cdot \mathtt{l} \tag{17.13}$$

It is easy to check that this gives us the required behavior. Finally, we may define $\mathtt{ifz}(u; u_0; x.u_1)$ to be the untyped term

$$u(u_0)(\lambda\,(_)\,[u_p(u)/x]u_1).$$

This gives us all the apparatus of PCF, apart from general recursion. But this is also definable using a *fixed point combinator*. There are many choices of fixed point combinator, of which the best known is the $\mathtt{Y}$ *combinator*:

$$\mathtt{Y} \triangleq \lambda\,(F)\,(\lambda\,(f)\,F(f(f)))(\lambda\,(f)\,F(f(f))).$$

It is easy to check that

$$\mathtt{Y}(F) \equiv F(\mathtt{Y}(F)).$$

Using the $\mathtt{Y}$ combinator, we may define general recursion by writing $\mathtt{Y}(\lambda\,(x)\,u)$, where $x$ stands for the recursive expression itself.

Although it is clear that $\mathtt{Y}$ as just defined computes a fixed point of its argument, it is probably less clear why it works or how we might have invented it in the first place. The main idea is actually quite simple. If a function is to be recursive, it is given an additional first argument, which is arranged at call sites to be the function itself. Whenever we wish to call a self-referential function with an argument, we apply the function first to itself and then to its argument; this protocol is imposed on both the "external" calls to the function and on the "internal" calls that the function may make to itself. For this reason the first argument is often called `this` or `self`, to remind you that it will be, by convention, bound to the function itself.

With this in mind, it is easy to see how to derive the definition of $\mathtt{Y}$. If $F$ is the function whose fixed point we seek, then the function $F' = \lambda\,(f)\,F(f(f))$ is a variant of $F$ in which the self-application convention has been imposed internally by substituting for each ocurrence of $f$ in $F(f)$ the self-application $f(f)$. Now simply observe that $F'(F') \equiv F(F'(F'))$, so that $F'(F')$ is the desired fixed point of $F$. Expanding the definition of $F'$, we have derived that the desired fixed point of $F$ is

$$\lambda\,(f)\,F(f(f))(\lambda\,(f)\,F(f(f))).$$

To finish the derivation, we need only observe that nothing depends on the particular choice of $F$, which means that we can compute a fixed point for $F$ uniformly in $F$. That is, we may define a *single* function, namely $\mathtt{Y}$ as defined above, that computes the fixed point of *any* $F$.

## 17.3 Scott's Theorem

Scott's Theorem states that definitional equality for the untyped $\lambda$-calculus is undecidable: there is no algorithm to determine whether or not two untyped terms are definitionally equal. The proof



Preview

uses the concept of *inseparability*. Any two properties, $\mathcal{A}_0$ and $\mathcal{A}_1$, of $\lambda$-terms are *inseparable* if there is no decidable property, $\mathcal{B}$, such that $\mathcal{A}_0\ u$ implies that $\mathcal{B}\ u$ holds, and $\mathcal{A}_1\ u$ implies that $\mathcal{B}\ u$ does *not* hold. We say that a property, $\mathcal{A}$, of untyped terms is *behavioral* iff whenever $u \equiv u'$, then $\mathcal{A}\ u$ iff $\mathcal{A}\ u'$.

The proof of Scott's Theorem decomposes into two parts:

1. For any untyped $\lambda$-term $u$, we may find an untyped term $v$ such that $u(\overline{\ulcorner v \urcorner}) \equiv v$, where $\ulcorner v \urcorner$ is the Gödel number of $v$, and $\overline{\ulcorner v \urcorner}$ is its representation as a Church numeral. (See Chapter 9 for a discussion of Gödel-numbering.)

2. Any two non-trivial[2] behavioral properties $\mathcal{A}_0$ and $\mathcal{A}_1$ of untyped terms are inseparable.

**Lemma 17.1.** *For any $u$ there exists $v$ such that $u(\overline{\ulcorner v \urcorner}) \equiv v$.*

*Proof Sketch.* The proof relies on the definability of the following two operations in the untyped $\lambda$-calculus:

1. $\mathtt{ap}(\overline{\ulcorner u_1 \urcorner})(\overline{\ulcorner u_2 \urcorner}) \equiv \overline{\ulcorner u_1(u_2) \urcorner}$.

2. $\mathtt{nm}(\overline{n}) \equiv \overline{\ulcorner \overline{n} \urcorner}$.

Intuitively, the first takes the representations of two untyped terms, and builds the representation of the application of one to the other. The second takes a numeral for $n$, and yields the representation of the Church numeral $\overline{n}$. Given these, we may find the required term $v$ by defining $v \triangleq w(\overline{\ulcorner w \urcorner})$, where $w \triangleq \lambda\,(x)\,u(\mathtt{ap}(x)(\mathtt{nm}(x)))$. We have

$$\begin{aligned} v &= w(\overline{\ulcorner w \urcorner}) \\ &\equiv u(\mathtt{ap}(\overline{\ulcorner w \urcorner})(\mathtt{nm}(\overline{\ulcorner w \urcorner}))) \\ &\equiv u(\overline{\ulcorner w(\overline{\ulcorner w \urcorner}) \urcorner}) \\ &\equiv u(\overline{\ulcorner v \urcorner}). \end{aligned}$$

The definition is very similar to that of $\mathtt{Y}(u)$, except that $u$ takes as input the representation of a term, and we find a $v$ such that, when applied to the representation of $v$, the term $u$ yields $v$ itself. □

**Lemma 17.2.** *Suppose that $\mathcal{A}_0$ and $\mathcal{A}_1$ are two non-trivial behavioral properties of untyped terms. Then there is no untyped term $w$ such that*

1. *For every $u$ either $w(\overline{\ulcorner u \urcorner}) \equiv \overline{0}$ or $w(\overline{\ulcorner u \urcorner}) \equiv \overline{1}$.*

2. *If $\mathcal{A}_0\ u$, then $w(\overline{\ulcorner u \urcorner}) \equiv \overline{0}$.*

3. *If $\mathcal{A}_1\ u$, then $w(\overline{\ulcorner u \urcorner}) \equiv \overline{1}$.*

[2] A property of untyped terms is said to be *trivial* if it either holds for all untyped terms or never holds for any untyped term.



Preview

*Proof.* Suppose there is such an untyped term $w$. Let $v$ be the untyped term

$$\lambda\,(x)\,\mathtt{ifz}(w(x); u_1; _.u_0),$$

where $u_0$ and $u_1$ may be chosen such that $\mathcal{A}_0\ u_0$ and $\mathcal{A}_1\ u_1$. (Such a choice must exist by non-triviality of the properties.) By Lemma 17.1 there is an untyped term $t$ such that $v(\overline{\ulcorner t \urcorner}) \equiv t$. If $w(\overline{\ulcorner t \urcorner}) \equiv \overline{0}$, then $t \equiv v(\overline{\ulcorner t \urcorner}) \equiv u_1$, and so $\mathcal{A}_1\ t$, because $\mathcal{A}_1$ is behavioral and $\mathcal{A}_1\ u_1$. But then $w(\overline{\ulcorner t \urcorner}) \equiv \overline{1}$ by the defining properties of $w$, which is a contradiction. Similarly, if $w(\overline{\ulcorner t \urcorner}) \equiv \overline{1}$, then $\mathcal{A}_0\ t$, and hence $w(\overline{\ulcorner t \urcorner}) \equiv \overline{0}$, again a contradiction. □

**Corollary 17.3.** *There is no algorithm to decide whether or not $u \equiv u'$.*

*Proof.* For fixed $u$, the property $\mathcal{E}_u\ u'$ defined by $u' \equiv u$ is a non-trivial behavioral property of untyped terms. It is therefore inseparable from its negation, and hence is undecidable. □

## 17.4 Untyped Means Uni-Typed

The untyped $\lambda$-calculus may be faithfully embedded in a typed language with recursive types. This means that every untyped $\lambda$-term has a representation as a typed expression in such a way that execution of the representation of a $\lambda$-term corresponds to execution of the term itself. This embedding is *not* a matter of writing an interpreter for the $\lambda$-calculus in **FPC** (which we could surely do), but rather a direct representation of untyped $\lambda$-terms as typed expressions in a language with recursive types.

The key observation is that the *untyped* $\lambda$-calculus is really the *uni-typed* $\lambda$-calculus. It is not the *absence* of types that gives it its power, but rather that it has *only one* type, namely the recursive type

$$D \triangleq \mathtt{rec}\, t\, \mathtt{is}\, t \rightharpoonup t.$$

A value of type $D$ is of the form $\mathtt{fold}(e)$ where $e$ is a value of type $D \rightharpoonup D$ — a function whose domain and range are both $D$. Any such function can be regarded as a value of type $D$ by "folding", and any value of type $D$ can be turned into a function by "unfolding". As usual, a recursive type may be seen as a solution to a type equation, which in the present case is the equation

$$D \cong D \rightharpoonup D.$$

This specifies that $D$ is a type that is isomorphic to the space of partial functions on $D$ itself, which, as was discussed in Chapter 16, is impossible if types are thought of as mere sets.

This isomorphism leads to the following translation, of **Λ** into **FPC**:

$$x^\dagger \triangleq x \tag{17.14a}$$

$$\lambda\,(x)\,u^\dagger \triangleq \mathtt{fold}(\lambda\,(x{:}D)\,u^\dagger) \tag{17.14b}$$

$$u_1(u_2)^\dagger \triangleq \mathtt{unfold}(u_1^\dagger)(u_2^\dagger) \tag{17.14c}$$



Preview

Observe that the embedding of a $\lambda$-abstraction is a value, and that the embedding of an application exposes the function being applied by unfolding the recursive type. Consequently,

$$\begin{aligned}\lambda\,(x)\,u_1(u_2)^\dagger &= \mathtt{unfold}(\mathtt{fold}(\lambda\,(x{:}D)\,u_1^\dagger))(u_2^\dagger)\\ &\equiv \lambda\,(x{:}D)\,u_1^\dagger(u_2^\dagger)\\ &\equiv [u_2^\dagger/x]u_1^\dagger\\ &= ([u_2/x]u_1)^\dagger.\end{aligned}$$

The last step, stating that the embedding commutes with substitution, is easily proved by induction on the structure of $u_1$. Thus $\beta$-reduction is faithfully implemented by evaluation of the embedded terms.

Thus we see that the canonical *untyped* language, $\mathbf{\Lambda}$, which by dint of terminology stands in opposition to *typed* languages, turns out to be but a typed language after all. Rather than eliminating types, an untyped language consolidates an infinite collection of types into a single recursive type. Doing so renders static type checking trivial, at the expense of incurring substantial dynamic overhead to coerce values to and from the recursive type. In Chapter 18 we will take this a step further by admitting many different types of data values (not just functions), each of which is a component of a "master" recursive type. This shows that so-called *dynamically typed* languages are, in fact, *statically typed*. Thus this traditional distinction can hardly be considered an opposition, because dynamic languages are but particular forms of static languages in which undue emphasis is placed on a single recursive type.

## 17.5 Notes

The untyped $\lambda$-calculus was introduced by Church (1941) as a codification of the informal concept of a computable function. Unlike the well-known machine models, such as the Turing machine or the random access machine, the $\lambda$-calculus directly codifies mathematical and programming practice. Barendregt (1984) is the definitive reference for all aspects of the untyped $\lambda$-calculus; the proof of Scott's theorem is adapted from Barendregt's account. Scott (1980) gave the first model of the untyped $\lambda$-calculus in terms of an elegant theory of recursive types. This construction underlies Scott's apt description of the $\lambda$-calculus as "unityped", rather than "untyped."

## Exercises

**17.1**. Define an encoding of finite products as defined in Chapter 10 in $\mathbf{\Lambda}$.

**17.2**. Define the factorial function in $\mathbf{\Lambda}$ two ways, one without using `Y`, and one using `Y`. In both cases show that your solution, $u$, has the property that $u(\overline{n}) \equiv \overline{n!}$.

**17.3**. Define the "Church booleans" in $\mathbf{\Lambda}$ by defining terms `true` and `false` such that

(a) $\mathtt{true}(u_1)(u_2) \equiv u_1$.

Preview

(b) $\mathtt{false}(u_1)(u_2) \equiv u_2$.

What is the encoding of $\mathtt{if}\ u\ \mathtt{then}\ u_1\ \mathtt{else}\ u_2$?

**17.4**. Define an encoding of finite sums as defined in Chapter 11 in $\Lambda$.

**17.5**. Define an encoding of finite lists of natural numbers as defined in Chapter 14 in $\Lambda$.

**17.6**. Define an encoding of the infinite streams of natural numbers as defined in Chapter 14 in $\Lambda$.

**17.7**. Show that $\Lambda$ may be "compiled" to sk-combinators using bracket abstraction (see Exercises **3.4** and **3.5**. Define a translation $u^*$ from $\Lambda$ into sk combinators such that

$$\text{if } u_1 \equiv u_2, \text{ then } u_1^* \equiv u_2^*.$$

*Hint*: Define $u^*$ by induction on the structure of $u$, using the compositional form of bracket abstraction considered in Exercise **3.5**. Show that the translation is itself compositional in that it commutes with substitution:

$$([u_2/x]u_1)^* = [u_2^*/x]u^*.$$

Then proceed by rule induction on rules (17.2) to show the required correctness condition.

Preview

# Chapter 18

# Dynamic Typing

We saw in Chapter 17 that an untyped language may be viewed as a unityped language in which the so-called untyped terms are terms of a distinguished recursive type. In the case of the untyped $\lambda$-calculus this recursive type has a particularly simple form, expressing that every term is isomorphic to a function. Consequently, no run-time errors can occur due to the misuse of a value—the only elimination form is application, and its first argument can only be a function. This property breaks down once more than one class of value is permitted into the language. For example, if we add natural numbers as a primitive concept to the untyped $\lambda$-calculus (rather than defining them via Church encodings), then it is possible to incur a run-time error arising from attempting to apply a number to an argument, or to add a function to a number. One school of thought in language design is to turn this vice into a virtue by embracing a model of computation that has multiple classes of value of a single type. Such languages are said to be *dynamically typed*, in purported opposition to *statically typed* languages. But the supposed opposition is illusory: just as the so-called untyped $\lambda$-calculus turns out to be uni-typed, so dynamic languages turn out to be but restricted forms of static language. This remark is so important it bears repeating: every dynamic language is inherently a static language in which we confine ourselves to a (needlessly) restricted type discipline to ensure safety.

Preview

## 18.1 Dynamically Typed PCF

To illustrate dynamic typing we formulate a dynamically typed version of **PCF**, called **DPCF**. The abstract syntax of **DPCF** is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Exp | $d$ | $::=$ | $x$ | $x$ | variable |
| | | | $\mathtt{num}[n]$ | $\overline{n}$ | numeral |
| | | | $\mathtt{zero}$ | $\mathtt{zero}$ | zero |
| | | | $\mathtt{succ}(d)$ | $\mathtt{succ}(d)$ | successor |
| | | | $\mathtt{ifz}(d; d_0; x.d_1)$ | $\mathtt{ifz}\, d\, \{\mathtt{zero} \hookrightarrow d_0 \mid \mathtt{succ}(x) \hookrightarrow d_1\}$ | zero test |
| | | | $\mathtt{fun}(x.d)$ | $\lambda\,(x)\,d$ | abstraction |
| | | | $\mathtt{ap}(d_1; d_2)$ | $d_1(d_2)$ | application |
| | | | $\mathtt{fix}(x.d)$ | $\mathtt{fix}\, x\, \mathtt{is}\, d$ | recursion |

There are two classes of values in **DPCF**, the *numbers*, which have the form $\mathtt{num}[n]$, and the *functions*, which have the form $\mathtt{fun}(x.d)$. The expressions $\mathtt{zero}$ and $\mathtt{succ}(d)$ are not themselves values, but rather are *constructors* that evaluate to values. General recursion is definable using a fixed point combinator, but is taken as primitive here to simplify the analysis of the dynamics in Section 18.3.

As usual, the abstract syntax of **DPCF** is what matters, but we use the concrete syntax to write examples in a convenient manner. However, it is often the case for dynamic languages, including **DPCF**, that the concrete syntax is deceptive in that it obscures an important detail of the abstract syntax, namely that every value is tagged with a classifier that plays a significant role at run-time (as we shall see shortly). So although the concrete syntax for a number, $\overline{n}$, suggests a "bare" representation, the abstract syntax reveals that the number is labelled with the class $\mathtt{num}$ to indicate that the value is of the numeric class. This is done to distinguish it from a function value, which concretely has the form $\lambda\,(x)\,d$, but whose abstract syntax, $\mathtt{fun}(x.d)$, indicates that it is to be classified with the tag $\mathtt{fun}$ to distinguish it from a number. As we shall see shortly, this tagging is fundamental to any dynamic language, so it is important to pay close attention to the abstract form in what follows.

The statics of **DPCF** is essentially the same as that of **Λ** given in Chapter 17; it merely checks that there are no free variables in the expression. The judgment

$$x_1\ \mathsf{ok}, \ldots x_n\ \mathsf{ok} \vdash d\ \mathsf{ok}$$

states that $d$ is a well-formed expression with free variables among those in the hypotheses. If the assumptions are empty, then we write just $d\ \mathsf{ok}$ to indicate that $d$ is a closed expression of **DPCF**.

The dynamics of **DPCF** must check for errors that would never arise in a language such as **PCF**. For example, evaluation of a function application must ensure that the value being applied is indeed a function, signaling an error if it is not. Similarly the conditional branch must ensure that its principal argument is a number, signaling an error if it is not. To account for these possibilities,

Preview

the dynamics is given by several judgment forms, as summarized in the following chart:

| | |
|---|---|
| $d$ val | $d$ is a (closed) value |
| $d \longmapsto d'$ | $d$ evaluates in one step to $d'$ |
| $d$ err | $d$ incurs a run-time error |
| $d$ is_num $n$ | $d$ is of class `num` with value $n$ |
| $d$ isnt_num | $d$ is not of class `num` |
| $d$ is_fun $x.d$ | $d$ is of class `fun` with body $x.d$ |
| $d$ isnt_fun | $d$ is not of class `fun` |

The last four judgment forms implement dynamic class checking. They are only relevant when $d$ has already been determined to be a value. The affirmative class-checking judgments have a second argument that represents the underlying structure of a value; this argument is *not* itself an expression of **DPCF**.

The value judgment, $d$ val, states that $d$ is a fully evaluated (closed) expression:

$$\frac{}{\texttt{num}[n] \text{ val}} \tag{18.1a}$$

$$\frac{}{\texttt{fun}(x.d) \text{ val}} \tag{18.1b}$$

The affirmative class-checking judgments are defined by the following rules:

$$\frac{}{\texttt{num}[n] \text{ is_num } n} \tag{18.2a}$$

$$\frac{}{\texttt{fun}(x.d) \text{ is_fun } x.d} \tag{18.2b}$$

The negative class-checking judgments are correspondingly defined by these rules:

$$\frac{}{\texttt{num}[n] \text{ isnt_fun}} \tag{18.3a}$$

$$\frac{}{\texttt{fun}(x.d) \text{ isnt_num}} \tag{18.3b}$$

The transition judgment, $d \longmapsto d'$, and the error judgment, $d$ err, are defined simultaneously by the following rules:

$$\frac{}{\texttt{zero} \longmapsto \texttt{num}[\texttt{z}]} \tag{18.4a}$$

$$\frac{d \longmapsto d'}{\texttt{succ}(d) \longmapsto \texttt{succ}(d')} \tag{18.4b}$$

$$\frac{d \text{ err}}{\texttt{succ}(d) \text{ err}} \tag{18.4c}$$

$$\frac{d \text{ is_num } n}{\texttt{succ}(d) \longmapsto \texttt{num}[\texttt{s}(n)]} \tag{18.4d}$$

$$\frac{d \text{ isnt_num}}{\texttt{succ}(d) \text{ err}} \tag{18.4e}$$

Preview

$$\frac{d \longmapsto d'}{\mathtt{ifz}(d; d_0; x.d_1) \longmapsto \mathtt{ifz}(d'; d_0; x.d_1)} \tag{18.4f}$$

$$\frac{d\ \mathsf{err}}{\mathtt{ifz}(d; d_0; x.d_1)\ \mathsf{err}} \tag{18.4g}$$

$$\frac{d\ \mathsf{is_num}\ 0}{\mathtt{ifz}(d; d_0; x.d_1) \longmapsto d_0} \tag{18.4h}$$

$$\frac{d\ \mathsf{is_num}\ n+1}{\mathtt{ifz}(d; d_0; x.d_1) \longmapsto [\mathtt{num}[n]/x]d_1} \tag{18.4i}$$

$$\frac{d\ \mathsf{isnt_num}}{\mathtt{ifz}(d; d_0; x.d_1)\ \mathsf{err}} \tag{18.4j}$$

$$\frac{d_1 \longmapsto d_1'}{\mathtt{ap}(d_1; d_2) \longmapsto \mathtt{ap}(d_1'; d_2)} \tag{18.4k}$$

$$\frac{d_1\ \mathsf{err}}{\mathtt{ap}(d_1; d_2)\ \mathsf{err}} \tag{18.4l}$$

$$\frac{d_1\ \mathsf{is_fun}\ x.d}{\mathtt{ap}(d_1; d_2) \longmapsto [d_2/x]d} \tag{18.4m}$$

$$\frac{d_1\ \mathsf{isnt_fun}}{\mathtt{ap}(d_1; d_2)\ \mathsf{err}} \tag{18.4n}$$

$$\frac{}{\mathtt{fix}(x.d) \longmapsto [\mathtt{fix}(x.d)/x]d} \tag{18.4o}$$

Rule (18.4i) labels the predecessor with the class `num` to maintain the invariant that variables are bound to expressions of **DPCF**.

**Lemma 18.1** (Class Checking)**.** *If $d$ val, then*

1. *either $d$ is_num $n$ for some $n$, or $d$ isnt_num;*
2. *either $d$ is_fun $x.d'$ for some $x$ and $d'$, or $d$ isnt_fun.*

*Proof.* By inspection of the rules defining the class-checking judgments. □

**Theorem 18.2** (Progress)**.** *If $d$ ok, then either $d$ val, or $d$ err, or there exists $d'$ such that $d \longmapsto d'$.*

*Proof.* By induction on the structure of $d$. For example, if $d = \mathtt{succ}(d')$, then we have by induction either $d'$ val, $d'$ err, or $d' \longmapsto d''$ for some $d''$. In the last case we have by rule (18.4b) that $\mathtt{succ}(d') \longmapsto \mathtt{succ}(d'')$, and in the second-to-last case we have by rule (18.4c) that $\mathtt{succ}(d')$ err. If $d'$ val, then by Lemma 18.1, either $d'$ is_num $n$ or $d'$ isnt_num. In the former case $\mathtt{succ}(d') \longmapsto \mathtt{num}[\mathtt{s}(n)]$, and in the latter $\mathtt{succ}(d')$ err. The other cases are handled similarly. □

**Lemma 18.3** (Exclusivity)**.** *For any $d$ in* **DPCF**, *exactly one of the following holds: $d$ val, or $d$ err, or $d \longmapsto d'$ for some $d'$.*

*Proof.* By induction on the structure of $d$, making reference to rules (18.4). □

Preview

## 18.2 Variations and Extensions

The dynamic language **DPCF** defined in Section 18.1 closely parallels the static language **PCF** defined in Chapter 15. One discrepancy, however, is in the treatment of natural numbers. Whereas in **PCF** the zero and successor operations are introductory forms for the type `nat`, in **DPCF** they are elimination forms that act on separately-defined numerals. This is done to ensure that there is a single class of numbers, rather than a separate class for zero and successor.

An alternative is to treat `zero` and $\texttt{succ}(d)$ as values of two separate classes, and to introduce the obvious class checking judgments for them. This complicates the error checking rules, and admits problematic values such as $\texttt{succ}(\lambda\,(x)\,d)$, but it allows us to avoid having a class of numbers. When written in this style, the dynamics of the conditional branch is given as follows:

$$\frac{d \longmapsto d'}{\texttt{ifz}(d; d_0; x.d_1) \longmapsto \texttt{ifz}(d'; d_0; x.d_1)} \tag{18.5a}$$

$$\frac{d\ \mathsf{is_zero}}{\texttt{ifz}(d; d_0; x.d_1) \longmapsto d_0} \tag{18.5b}$$

$$\frac{d\ \mathsf{is_succ}\ d'}{\texttt{ifz}(d; d_0; x.d_1) \longmapsto [d'/x]d_1} \tag{18.5c}$$

$$\frac{d\ \mathsf{isnt_zero} \quad d\ \mathsf{isnt_succ}}{\texttt{ifz}(d; d_0; x.d_1)\ \mathsf{err}} \tag{18.5d}$$

Notice that the predecessor of a value of the successor class need not be a number, whereas in the previous formulation this possibility does not arise.

Structured data may be added to **DPCF** using similar techniques. The classic example is to introduce a class `nil` consisting of a "null" value, and a class `cons` classifying pairs of values.

| | | | | | |
|---|---|---|---|---|---|
| Exp | $d$ | ::= | $\texttt{nil}$ | $\texttt{nil}$ | null |
| | | | $\texttt{cons}(d_1; d_2)$ | $\texttt{cons}(d_1; d_2)$ | pair |
| | | | $\texttt{ifnil}(d; d_0; x, y.d_1)$ | $\texttt{ifnil}\,d\,\{\texttt{nil} \hookrightarrow d_0 \mid \texttt{cons}(x; y) \hookrightarrow d_1\}$ | conditional |

The expression $\texttt{ifnil}(d; d_0; x, y.d_1)$ distinguishes the null value from a pair, and signals an error on any other class of value.

Lists may be represented using null and pairing. For example, the list consisting of three zeroes is represented by the value

$$\texttt{cons}(\texttt{zero}; \texttt{cons}(\texttt{zero}; \texttt{cons}(\texttt{zero}; \texttt{nil}))).$$

But what to make of this beast?

$$\texttt{cons}(\texttt{zero}; \texttt{cons}(\texttt{zero}; \texttt{cons}(\texttt{zero}; \lambda\,(x)\,x)))$$

This value is not a list, because it does not end with `nil`.



Preview

The difficulty with encoding lists using null and pair becomes apparent when defining functions that operate on them. For example, here is a definition of the function `append` that concatenates two lists:

$$\texttt{fix}\, a\, \texttt{is}\, \lambda\,(x)\, \lambda\,(y)\, \texttt{ifnil}(x; y; x_1, x_2.\texttt{cons}(x_1; a(x_2)(y)))$$

Nothing prevents us from applying this function to any two values, regardless of whether they are lists. If the first argument is not a list, then execution aborts with an error. But because the function does not traverse its second argument, it can be any value at all. For example, we may apply `append` with a list and a function to obtain the "list" that ends with a $\lambda$ given above.

It might be argued that the conditional branch that distinguishes null from a pair is inappropriate in **DPCF**, because there are more than just these two classes in the language. One approach that avoids this criticism is to abandon the idea of pattern matching on the class of data entirely, replacing it by a general conditional branch that distinguishes null from all other values, and adding to the language *predicates*[1] that test the class of a value and *destructors* that invert the constructors of each class.

In the present case we would reformulate the extension of **DPCF** with null and pairing as follows:

| | | | | | |
|---|---|---|---|---|---|
| Exp | $d$ | $::=$ | $\texttt{cond}(d; d_0; d_1)$ | $\texttt{cond}(d; d_0; d_1)$ | conditional |
| | | | $\texttt{nil?}(d)$ | $\texttt{nil?}(d)$ | nil test |
| | | | $\texttt{cons?}(d)$ | $\texttt{cons?}(d)$ | pair test |
| | | | $\texttt{car}(d)$ | $\texttt{car}(d)$ | first projection |
| | | | $\texttt{cdr}(d)$ | $\texttt{cdr}(d)$ | second projection |

The conditional $\texttt{cond}(d; d_0; d_1)$ distinguishes $d$ between `nil` and *all other values*. If $d$ is not `nil`, the conditional evaluates to $d_0$, and otherwise evaluates to $d_1$. In other words the value `nil` represents boolean falsehood, and all other values represent boolean truth. The predicates $\texttt{nil?}(d)$ and $\texttt{cons?}(d)$ test the class of their argument, yielding `nil` if the argument is not of the specified class, and yielding some non-`nil` if so. The destructors $\texttt{car}(d)$ and $\texttt{cdr}(d)$ decompose $\texttt{cons}(d_1; d_2)$ into $d_1$ and $d_2$, respectively.[2]

Written in this form, the function `append` is given by the expression

$$\texttt{fix}\, a\, \texttt{is}\, \lambda\,(x)\, \lambda\,(y)\, \texttt{cond}(x; \texttt{cons}(\texttt{car}(x); a(\texttt{cdr}(x))(y)); y).$$

The behavior of this formulation of `append` is no different from the earlier one; the only difference is that instead of dispatching on whether a value is either null or a pair, we instead allow discrimination on any predicate of the value, which includes such checks as special cases.

An alternative, which is not widely used, is to enhance, rather than restrict, the conditional branch so that it includes cases for each possible class of value in the language. So, for example, in a language with numbers, functions, null, and pairing, the conditional would have four branches. The fourth branch, for pairing, would deconstruct the pair into its constituent parts. The difficulty

[1] Predicates evaluate to the null value to indicate that a condition is false, and some non-null value to indicate that it is true.

[2] The terminology for the projections is archaic, but firmly established in the literature. It is said that `car` originally stood for "contents of the address register" and that `cdr` stood for "contents of the data register", referring to the details of the original implementation of Lisp.



Preview

with this approach is that in realistic languages there are many classes of data, and such a conditional would be rather unwieldy. Moreover, even once we have dispatched on the class of a value, it is nevertheless necessary for the primitive operations associated with that class to perform run-time checks. For example, we may determine that a value, $d$, is of the numeric class, but there is no way to propagate this information into the branch of the conditional that then adds $d$ to some other number. The addition operation must still check the class of $d$, recover the underlying number, and create a new value of numeric class. This is an inherent limitation of dynamic languages, which do not permit handling anything other than classified values.

## 18.3 Critique of Dynamic Typing

The safety theorem for **DPCF** is often promoted as an advantage of dynamic over static typing. Unlike static languages, which rule out some candidate programs as ill-typed, essentially every piece of abstract syntax in **DPCF** is well-formed, and hence, by Theorem 18.2, has a well-defined dynamics. But this can also be seen as a disadvantage, because errors that could be ruled out at compile time by type checking are not signalled until run time in **DPCF**. To make this possible, the dynamics of **DPCF** must enforce at run-time conditions that need not be checked at all in a statically typed language.

Consider, for example, the addition function in **DPCF**, whose specification is that, when passed two values of class `num`, returns their sum, which is also of class `num`:[3]

$$\texttt{fun}(x.\texttt{fix}(p.\texttt{fun}(y.\texttt{ifz}(y; x; y'.\texttt{succ}(p(y')))))).$$

The addition function may, deceptively, be written in concrete syntax as follows:

$$\lambda\,(x)\,\texttt{fix}\,p\,\texttt{is}\,\lambda\,(y)\,\texttt{ifz}\,y\,\{\texttt{zero} \hookrightarrow x \mid \texttt{succ}(y') \hookrightarrow \texttt{succ}(p(y'))\}.$$

It is deceptive, because it obscures the class tags on values, and the operations that check the validity of those tags. Let us now examine the costs of these operations in a bit more detail.

First, observe that the body of the fixed point expression is labeled with class `fun`. The dynamics of the fixed point construct binds $p$ to this function. Consequently, the dynamic class check incurred by the application of $p$ in the recursive call is guaranteed to succeed. But **DPCF** offers no means of suppressing the redundant check, because it cannot express the invariant that $p$ is always bound to a value of class `fun`.

Second, observe that the result of applying the inner $\lambda$-abstraction is either $x$, the argument of the outer $\lambda$-abstraction, or the successor of a recursive call to the function itself. The successor operation checks that its argument is of class `num`, even though this condition is guaranteed to hold for all but the base case, which returns the given $x$, which can be of any class at all. In principle we can check that $x$ is of class `num` once, and observe that it is otherwise a loop invariant that the result of applying the inner function is of this class. However, **DPCF** gives us no way to express this invariant; the repeated, redundant tag checks imposed by the successor operation cannot be avoided.

[3] This specification imposes no restrictions on the behavior of addition on arguments that are not classified as numbers, but we could make the further demand that the function abort when applied to arguments that are not classified by `num`.



PREVIEW

Third, the argument, $y$, to the inner function is either the original argument to the addition function, or is the predecessor of some earlier recursive call. But as long as the original call is to a value of class `num`, then the dynamics of the conditional will ensure that all recursive calls have this class. And again there is no way to express this invariant in **DPCF**, and hence there is no way to avoid the class check imposed by the conditional branch.

Classification is not free—storage is required for the class label, and it takes time to detach the class from a value each time it is used and to attach a class to a value whenever it is created. Although the overhead of classification is not asymptotically significant (it slows down the program only by a constant factor), it is nevertheless non-negligible, and should be eliminated whenever possible. But this is impossible within **DPCF**, because it cannot enforce the restrictions required to express the required invariants. For that we need a static type system.

## 18.4 Notes

The earliest dynamically typed language is Lisp (McCarthy, 1965), which continues to influence language design a half century after its invention. Dynamic PCF is essentially the core of Lisp, but with a proper treatment of variable binding, correcting what McCarthy himself has described as an error in the original design. Informal discussions of dynamic languages are often confused by the ellision of the dynamic checks that are made explicit here. Although the surface syntax of dynamic PCF is essentially the same as that for PCF, minus the type annotations, the underlying dynamics is fundamentally different. It is for this reason that static PCF cannot be properly seen as a restriction of dynamic PCF by the imposition of a type system, contrary to what seems to be a widely held belief.

## Exercises

**18.1**. Surface syntax can be deceiving; even simple arithmetic expressions do not have the same meaning in **DPCF** that they do in **PCF**. To see why, define the addition function, `plus`, in **DPCF**, and examine the dynamics of evaluating expressions such as $\texttt{plus}(\overline{5})(\overline{7})$. Even though this expression might be written as "$5 + 7$" in both static and dynamic languages, they have *very* different meanings.

**18.2**. Give a precise dynamics to the data structuring primitives described informally in Section 18.2. What class restrictions should `cons` impose on its arguments? Examine carefully the dynamics of the `append` function when called with two lists as arguments.

**18.3**. To avoid the difficulties with the representation of lists using `cons` and `nil`, introduce a class of lists that are constructed using revised versions of `nil` and `cons` that operate on the class of lists. Revisit the dynamics of the `append` function when redefined using the class of lists.

**18.4**. Allowing multiple arguments to, and multiple results from, functions is a notorious source of trouble in dynamic languages. The restriction to a single type makes it impossible even to distinguish $n$ things from $m$ things, let alone express more subtle properties of a program.



Preview

Numerous work-arounds have been proposed. Explore the problem yourself by enriching **DPCF** with multi-argument and multi-result functions. Be sure to consider these questions:

(a) If a function is defined with $n$ parameters, what should happen if it is called with more or fewer than $n$ arguments?

(b) The standard syntax for multi-argument functions is essentially $\lambda\ (x_1, \ldots, x_n)\ d$ for some fixed $n \geq 0$. Relate this notation to pattern matching as described in Chapter 12.

(c) What happens if one were to admit functions with a *varying* number of arguments? How would you refer to these arguments within the body of such a function? How does this relate to pattern matching?

(d) What if one wished to admit keyword parameter passing by giving names to the arguments, and allowing them to be passed in any order by associating them with their names? How does this relate to pattern matching?

(e) What notation would you suggest for functions returning multiple results? For example, a division function might return the quotient and the remainder. How might one notate this in the function body? How would a caller access the results individually or collectively?

(f) How would one define the composition of two functions when either or both can take multiple arguments or return multiple results?

Preview

Preview

# Chapter 19

# Hybrid Typing

A *hybrid* language is one that combines static and dynamic typing by enriching a statically typed language with a distinguished type, `dyn`, of dynamic values. The dynamically typed language considered in Chapter 18 may be embedded into the hybrid language by regarding a dynamically typed program as a statically typed program of type `dyn`. Static and dynamic types are not opposed to one another, but may coexist harmoniously.

But the notion of a hybrid language is, in fact, illusory: the type `dyn` is just a particular recursive type. There is no need to augment a static language with any special mechanisms to support dynamic typing. On the contrary, *dynamic typing is but a mode of use of static typing*. The supposed opposition between dynamic and static typing is, therefore, a fallacy: dynamic typing can hardly be opposed to that of which it is but a special case.

## 19.1 A Hybrid Language

Consider the language **HPCF**, which extends **PCF** with the following additional constructs:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | ::= | `dyn` | `dyn` | dynamic |
| Exp | $e$ | ::= | $\texttt{new}[l](e)$ | $l\,!\,e$ | construct |
| | | | $\texttt{cast}[l](e)$ | $e\,@\,l$ | destruct |
| | | | $\texttt{inst}[l](e)$ | $l\,?\,e$ | discriminate |
| Cls | $l$ | ::= | `num` | `num` | number |
| | | | `fun` | `fun` | function |

The type `dyn` is the type of dynamically classified values. The constructor attaches a classifier to a value of the appropriate type, the destructor recovers the value classified with the given classifier, and the discriminator tests the class of a classified value.

The statics of **HPCF** extends that of **PCF** with the following additional rules:

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{new}[\texttt{num}](e) : \texttt{dyn}} \tag{19.1a}$$

Preview

$$\frac{\Gamma \vdash e : \mathtt{dyn} \rightharpoonup \mathtt{dyn}}{\Gamma \vdash \mathtt{new}[\mathtt{fun}](e) : \mathtt{dyn}} \tag{19.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{num}](e) : \mathtt{nat}} \tag{19.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{fun}](e) : \mathtt{dyn} \rightharpoonup \mathtt{dyn}} \tag{19.1d}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{inst}[\mathtt{num}](e) : \mathtt{bool}} \tag{19.1e}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{inst}[\mathtt{fun}](e) : \mathtt{bool}} \tag{19.1f}$$

The statics ensures that classifiers are attached to values of the appropriate type, namely natural numbers for `num`, and functions on classified values for `fun`.

The dynamics of **HPCF** extends that of **PCF** with the following rules:

$$\frac{e \ \mathsf{val}}{\mathtt{new}[l](e) \ \mathsf{val}} \tag{19.2a}$$

$$\frac{e \longmapsto e'}{\mathtt{new}[l](e) \longmapsto \mathtt{new}[l](e')} \tag{19.2b}$$

$$\frac{e \longmapsto e'}{\mathtt{cast}[l](e) \longmapsto \mathtt{cast}[l](e')} \tag{19.2c}$$

$$\frac{\mathtt{new}[l](e) \ \mathsf{val}}{\mathtt{cast}[l](\mathtt{new}[l](e)) \longmapsto e} \tag{19.2d}$$

$$\frac{\mathtt{new}[l'](e) \ \mathsf{val} \quad l \neq l'}{\mathtt{cast}[l](\mathtt{new}[l'](e)) \ \mathsf{err}} \tag{19.2e}$$

$$\frac{e \longmapsto e'}{\mathtt{inst}[l](e) \longmapsto \mathtt{inst}[l](e')} \tag{19.2f}$$

$$\frac{\mathtt{new}[l](e) \ \mathsf{val}}{\mathtt{inst}[l](\mathtt{new}[l](e)) \longmapsto \mathtt{true}} \tag{19.2g}$$

$$\frac{\mathtt{new}[l](e) \ \mathsf{val} \quad l \neq l'}{\mathtt{inst}[l'](\mathtt{new}[l](e)) \longmapsto \mathtt{false}} \tag{19.2h}$$

Casting compares the class of the object to the required class, returning the underlying object if these coincide, and signalling an error otherwise.[1]

**Lemma 19.1** (Canonical Forms)**.** *If $e : \mathtt{dyn}$ and $e$ val, then $e = \mathtt{new}[l](e')$ for some class $l$ and some $e'$ val. If $l = \mathtt{num}$, then $e' : \mathtt{nat}$, and if $l = \mathtt{fun}$, then $e' : \mathtt{dyn} \rightharpoonup \mathtt{dyn}$.*

[1] The judgment $e$ err signals a checked error that is to be treated as described in Section 6.3.



Preview

*Proof.* By rule induction on the statics of **HPCF**. □

**Theorem 19.2** (Safety)**.** *The language* **HPCF** *is safe:*

1. *If* $e : \tau$ *and* $e \longmapsto e'$*, then* $e' : \tau$*.*
2. *If* $e : \tau$*, then either* $e$ *val, or* $e$ *err, or* $e \longmapsto e'$ *for some* $e'$*.*

*Proof.* Preservation is proved by rule induction on the dynamics, and progress is proved by rule induction on the statics, making use of the canonical forms lemma. The opportunities for run-time errors are the same as those for **HPCF**—a well-typed cast might fail at run-time if the class of the cast does not match the class of the value. □

Any self-respecting language (such as **FPC**, defined in Chapter 16) has recursive types, and hence need not take the type `dyn` as primtiive. Instead it may be defined to be the recursive type

$$\mathtt{rec}\, t\, \mathtt{is}\, [\mathtt{num} \hookrightarrow \mathtt{nat}, \mathtt{fun} \hookrightarrow t \rightharpoonup t]. \tag{19.3}$$

The introduction and elimination forms are definable as follows:[2]

$$\mathtt{new}[\mathtt{num}](e) \triangleq \mathtt{fold}(\mathtt{num} \cdot e) \tag{19.4}$$

$$\mathtt{new}[\mathtt{fun}](e) \triangleq \mathtt{fold}(\mathtt{fun} \cdot e) \tag{19.5}$$

$$\mathtt{cast}[\mathtt{num}](e) \triangleq \mathtt{case}\, \mathtt{unfold}(e)\, \{\mathtt{num} \cdot x \hookrightarrow x \mid \mathtt{fun} \cdot x \hookrightarrow \mathtt{error}\} \tag{19.6}$$

$$\mathtt{cast}[\mathtt{fun}](e) \triangleq \mathtt{case}\, \mathtt{unfold}(e)\, \{\mathtt{num} \cdot x \hookrightarrow \mathtt{error} \mid \mathtt{fun} \cdot x \hookrightarrow x\}. \tag{19.7}$$

We may readily check that the statics and dynamics of the type `dyn` in **HPCF** are derivable in **FPC** from these definitions.

## 19.2 Dynamic as Static Typing

The language **DPCF** of Chapter 18 may be embedded into **HPCF** by a simple translation that makes explicit the class checking in the dynamics of **DPCF**. Specifically, we may define a translation $d^\dagger$ of expressions of **DPCF** into expressions of **HPCF** according to the following static correctness criterion:

**Theorem 19.3.** *If* $x_1$ *ok*$, \ldots, x_n$ *ok* $\vdash d$ *ok according to the statics of* **DPCF***, then* $x_1 : \mathtt{dyn}, \ldots, x_n : \mathtt{dyn} \vdash d^\dagger : \mathtt{dyn}$ *in* **HPCF***.*

[2] The expression `error` aborts the computation with an error; this can be accomplished using exceptions, which are described in Chapter 29.



Preview

The proof of Theorem 19.3 is given by induction on the structure of $d$ based on the following translation:

$$\begin{aligned}
x^\dagger &\triangleq x \\
\mathtt{num}[n]^\dagger &\triangleq \mathtt{new}[\mathtt{num}](\overline{n}) \\
\mathtt{zero}^\dagger &\triangleq \mathtt{new}[\mathtt{num}](\mathtt{z}) \\
\mathtt{succ}(d)^\dagger &\triangleq \mathtt{new}[\mathtt{num}](\mathtt{s}(\mathtt{cast}[\mathtt{num}](d^\dagger))) \\
\mathtt{ifz}(d; d_0; x.d_1) &\triangleq \mathtt{ifz}(\mathtt{cast}[\mathtt{num}](d^\dagger); d_0^\dagger; x.[\mathtt{new}[\mathtt{num}](x)/x]d_1^\dagger) \\
(\mathtt{fun}(x.d))^\dagger &\triangleq \mathtt{new}[\mathtt{fun}](\lambda\,(x{:}\mathtt{dyn})\,d^\dagger) \\
(\mathtt{ap}(d_1; d_2))^\dagger &\triangleq \mathtt{cast}[\mathtt{fun}](d_1^\dagger)(d_2^\dagger) \\
\mathtt{fix}(x.d) &\triangleq \mathtt{fix}[\mathtt{dyn}](x.d^\dagger)
\end{aligned}$$

A rigorous proof of correctness of this translation requires methods similar to those to be developed in Chapter 49. It should be at least plausible that the translation is correct, since it merely expands the primitive operations of **DPCF** in terms of the operations of the type dyn.

## 19.3 Optimization of Dynamic Typing

The language **HPCF** combines static and dynamic typing by enriching **PCF** with the type, dyn, of classified values. It is, for this reason, called a *hybrid* language. Unlike a purely dynamic type system, a hybrid type system can express invariants that are crucial to the optimization of programs in **HPCF**.

Consider the addition function in **DPCF** given in Section 18.3, which we transcribe here for easy reference:

$$\lambda\,(x)\,\mathtt{fix}\,p\,\mathtt{is}\,\lambda\,(y)\,\mathtt{ifz}\,y\,\{\mathtt{zero} \hookrightarrow x \mid \mathtt{succ}(y') \hookrightarrow \mathtt{succ}(p(y'))\}.$$

This function may be regarded as a value of type dyn in **HPCF** given as follows:

$$\mathtt{fun}\,!\,\lambda\,(x{:}\mathtt{dyn})\,\mathtt{fix}\,p : \mathtt{dyn}\,\mathtt{is}\,\mathtt{fun}\,!\,\lambda\,(y{:}\mathtt{dyn})\,e_{x,p,y}, \tag{19.8}$$

where

$$x : \mathtt{dyn}, p : \mathtt{dyn}, y : \mathtt{dyn} \vdash e_{x,p,y} : \mathtt{dyn}$$

is the expression

$$\mathtt{ifz}\,(y\,@\,\mathtt{num})\,\{\mathtt{zero} \hookrightarrow x \mid \mathtt{succ}(y') \hookrightarrow \mathtt{num}\,!\,(\mathtt{s}((p\,@\,\mathtt{fun})(\mathtt{num}\,!\,y')\,@\,\mathtt{num}))\}.$$

The embedding into **HPCF** makes explicit the run-time checks that are implicit in the dynamics of **DPCF**.

Careful examination of the embedded formulation of addition reveals a great deal of redundancy and overhead that can be eliminated in the statically typed version. Eliminating this redundancy requires a static type discipline, because the intermediate computations involve values of a



Preview

type other than dyn. Because a dynamic language can only express values of one type, it is impossible to express the optimized form within a dynamic language. This shows that the superficial freedoms offered by dynamic languages supposedly accruing from the omission of types are, in fact, severe restrictions on the expressiveness of the language compared to a static language with a type of dynamic values.

The first redundancy arises from the use of recursion in a dynamic language. In the above example we use recursion to define the inner loop, $p$, of the computation. The value $p$ is, by definition, a $\lambda$-abstraction, which is explicitly tagged as a function. Yet the call to $p$ within the loop checks at run-time whether $p$ is in fact a function before applying it. Because $p$ is an internally defined function, all of its call sites are under the control of the addition function, which means that there is no need for such pessimism at calls to $p$, provided that we change its type to $\texttt{dyn} \rightharpoonup \texttt{dyn}$, which directly expresses the invariant that $p$ is a function acting on dynamic values.

Performing this transformation, we obtain the following reformulation of the addition function that eliminates this redundancy:

$$\texttt{fun}\,!\,\lambda\,(x{:}\texttt{dyn})\,\texttt{fun}\,!\,\texttt{fix}\,p:\texttt{dyn} \rightharpoonup \texttt{dyn}\,\texttt{is}\,\lambda\,(y{:}\texttt{dyn})\,e'_{x,p,y},$$

where $e'_{x,p,y}$ is the expression

$$\texttt{ifz}\,(y\,@\,\texttt{num})\,\{\texttt{zero} \hookrightarrow x \mid \texttt{succ}(y') \hookrightarrow \texttt{num}\,!\,(\texttt{s}(p(\texttt{num}\,!\,y')\,@\,\texttt{num}))\}.$$

We have "hoisted" the function class label out of the loop, and suppressed the cast inside the loop. Correspondingly, the type of $p$ has changed to $\texttt{dyn} \rightharpoonup \texttt{dyn}$.

Next, observe that the parameter $y$ of type dyn is cast to a number on each iteration of the loop before it is tested for zero. Because this function is recursive, the bindings of $y$ arise in one of two ways: at the initial call to the addition function, and on each recursive call. But the recursive call is made on the predecessor of $y$, which is a true natural number that is labeled with num at the call site, only to be removed by the class check at the conditional on the next iteration. This suggests that we hoist the check on $y$ outside of the loop, and avoid labeling the argument to the recursive call. Doing so changes the type of the function, however, from $\texttt{dyn} \rightharpoonup \texttt{dyn}$ to $\texttt{nat} \rightharpoonup \texttt{dyn}$. Consequently, further changes are required to ensure that the entire function remains well-typed.

Before doing so, let us make another observation. The result of the recursive call is checked to ensure that it has class num, and, if so, the underlying value is incremented and labeled with class num. If the result of the recursive call came from an earlier use of this branch of the conditional, then obviously the class check is redundant, because we know that it must have class num. But what if the result came from the other branch of the conditional? In that case the function returns $x$, which need not be of class num because it is provided by the caller of the function. However, we may reasonably insist that it is an error to call addition with a non-numeric argument. This can be enforced by replacing $x$ in the zero branch of the conditional by $x\,@\,\texttt{num}$.

Combining these optimizations we obtain the inner loop $e''_x$ defined as follows:

$$\texttt{fix}\,p:\texttt{nat} \rightharpoonup \texttt{nat}\,\texttt{is}\,\lambda\,(y{:}\texttt{nat})\,\texttt{ifz}\,y\,\{\texttt{zero} \hookrightarrow x\,@\,\texttt{num} \mid \texttt{succ}(y') \hookrightarrow \texttt{s}(p(y'))\}.$$

This function has type $\texttt{nat} \rightharpoonup \texttt{nat}$, and runs at full speed when applied to a natural number—all checks have been hoisted out of the inner loop.

Preview

Finally, recall that the overall goal is to define a version of addition that works on values of type dyn. Thus we require a value of type dyn ⇀ dyn, but what we have at hand is a function of type nat ⇀ nat. This can be converted to the required form by pre-composing with a cast to num and post-composing with a coercion to num:

$$\texttt{fun}\,!\,\lambda\,(x{:}\texttt{dyn})\,\texttt{fun}\,!\,\lambda\,(y{:}\texttt{dyn})\,\texttt{num}\,!\,(e_x''(y\,@\,\texttt{num})).$$

The innermost $\lambda$-abstraction converts the function $e_x''$ from type nat ⇀ nat to type dyn ⇀ dyn by composing it with a class check that ensures that $y$ is a natural number at the initial call site, and applies a label to the result to restore it to type dyn.

The outcome of these transformations is that the inner loop of the computation runs at "full speed", without any manipulation of tags on functions or numbers. But the outermost form of addition has been retained as a value of type dyn encapsulating a curried function that takes two arguments of type dyn. This preserves the correctness of all calls to addition, which pass and return values of type dyn, while optimizing its execution during the computation. Of course, we could strip the class tags from the addition function, changing its type from dyn to the more descriptive dyn ⇀ dyn ⇀ dyn, but this imposes the requirement on the caller to treat addition not as a value of type dyn, but rather as a function that must be applied to two successive values of type dyn whose class is num. As long as the call sites to addition are under programmer control, there is no obstacle to effecting this transformation. It is only when there may be external call sites, not directly under programmer control, that there is any need to package addition as a value of type dyn. Applying this principle generally, we see that dynamic typing is only of marginal utility—that is, it is used only at the margins of a system where uncontrolled calls arise. Internally to a system there is no benefit, and considerable drawback, to restricting attention to the type dyn.

## 19.4 Static Versus Dynamic Typing

There have been many attempts by advocates of dynamic typing to distinguish dynamic from static languages. It is useful to review the supposed distinctions from the present viewpoint.

1. *Dynamic languages associate types with values, whereas static languages associate types to variables.* But this is nonsense arising from the confusion of types with classes. Dynamic languages associate *classes*, not *types*, to values by tagging them with identifiers such as num and fun. This form of classification amounts to a use of recursive sum types within a statically typed language, and hence cannot be seen as a distinguishing feature of dynamic languages. Morever, static languages assign types to expressions, not just variables. Because dynamic languages are just particular static languages (with a single type), the same can be said of dynamic languages.

2. *Dynamic languages check types at run-time, whereas static language check types at compile time.* This, too, is erroneous. Dynamic languages are just as surely statically typed as static languages, albeit for a degenerate type system with only one type. As we have seen, dynamic languages do perform class checks at run-time, but so too do static languages that admit sum types. The difference is only the extent to which we must use classification: always in a dynamic language, only as necessary in a static language.

Preview

3. *Dynamic languages support heterogeneous collections, whereas static languages support homogeneous collections*. But this, too, is in error. Sum types exist to support heterogeneity, and any static language with sums admits heterogenous data structures. A typical example is a list such as

$$\texttt{cons}(\texttt{num}[1]; \texttt{cons}(\texttt{fun}(x.x); \texttt{nil}))$$

(written in abstract syntax for emphasis). It is sometimes said that such a list is not representable in a static language, because of the disparate nature of its components. Whether in a static *or* a dynamic language, lists are *type* homogeneous, but may be *class* heterogeneous. All elements of the above list are of *type* `dyn`; the first is of *class* `num`, and the second is of *class* `fun`.

What, then, are we to make of the supposed distinction between dynamic and static languages? Rather than being in opposition to each other, it is more accurate to say that *dynamic languages are a mode of use of static languages*. Every dynamic language *is* a static language, albeit one with a pitiable paucity of types (only one!).

## 19.5 Notes

The concept of a hybrid type system is wholly artificial, serving only as an explanatory bridge between dynamic and static languages. Viewing dynamic languages as static languages with recursive types was first proposed by Scott (1980), who also suggested the term "unityped" as a more descriptive alternative to "untyped."

## Exercises

**19.1**. Consider the extensions to **DPCF** described in Section 18.2 to admit null and pairing and their associated operations. Extend the statics and dynamics of **HPCF** to account for these extensions, and give a translation of the null and pairing operations described informally in Chapter 18 in terms of this extension to **HPCF**.

**19.2**. Continue the interpretation of the null and pairing operations in **HPCF** given in Exercise **19.1** to provide an interpretation in **FPC**. Specifically, define the expanded `dyn` as a recursive type, and give a direct implementation of the null and pairing primitives in terms of this recursive type.

**19.3**. Consider the `append` function defined in Chapter 18 using `nil` and `cons` to represent lists:

$$\texttt{fix}\, a\, \texttt{is}\, \lambda\,(x)\, \lambda\,(y)\, \texttt{cond}(x; \texttt{cons}(\texttt{car}(x); a(\texttt{cdr}(x))(y)); y).$$

Rewrite `append` in **HPCF** using the definitions given in Exercise **19.1**. Then optimize the implementation to eliminate unnecessary overhead while ensuring that `append` still has type `dyn`.



Preview

Preview

# Part VIII

# Variable Types

Preview

Preview

# Chapter 20

# Girard's System F

The languages we have considered so far are all *monomorphic* in that every expression has a unique type, given the types of its free variables, if it has a type at all. Yet it is often the case that essentially the same behavior is required, albeit at several different types. For example, in **T** there is a *distinct* identity function for each type $\tau$, namely $\lambda\,(x{:}\tau)\,x$, even though the behavior is the same for each choice of $\tau$. Similarly, there is a distinct composition operator for each triple of types, namely

$$\circ_{\tau_1,\tau_2,\tau_3} = \lambda\,(f{:}\tau_2 \to \tau_3)\,\lambda\,(g{:}\tau_1 \to \tau_2)\,\lambda\,(x{:}\tau_1)\,f(g(x)).$$

Each choice of the three types requires a *different* program, even though they all exhibit the same behavior when executed.

Obviously it would be useful to capture the general pattern once and for all, and to instantiate this pattern each time we need it. The expression patterns codify generic (type-independent) behaviors that are shared by all instances of the pattern. Such generic expressions are said to be *polymorphic*. In this chapter we will study the language **F**, which was introduced by Girard under the name *System F* and by Reynolds under the name *polymorphic typed $\lambda$-calculus*. Although motivated by a simple practical problem (how to avoid writing redundant code), the concept of polymorphism is central to an impressive variety of seemingly disparate concepts, including the concept of data abstraction (the subject of Chapter 21), and the definability of product, sum, inductive, and coinductive types considered in the preceding chapters. (Only general recursive types extend the expressive power of the language.)

Preview

## 20.1 Polymorphic Abstraction

The language **F** is a variant of **T** in which we eliminate the type of natural numbers, but add, in compensation, polymorphic types:[1]

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | ::= | $t$ | $t$ | variable |
| | | | $\mathtt{arr}(\tau_1;\tau_2)$ | $\tau_1 \to \tau_2$ | function |
| | | | $\mathtt{all}(t.\tau)$ | $\forall(t.\tau)$ | polymorphic |
| Exp | $e$ | ::= | $x$ | $x$ | |
| | | | $\mathtt{lam}[\tau](x.e)$ | $\lambda\,(x{:}\tau)\,e$ | abstraction |
| | | | $\mathtt{ap}(e_1;e_2)$ | $e_1(e_2)$ | application |
| | | | $\mathtt{Lam}(t.e)$ | $\Lambda(t.e)$ | type abstraction |
| | | | $\mathtt{App}[\tau](e)$ | $e[\tau]$ | type application |

A *type abstraction*, $\mathtt{Lam}(t.e)$, defines a *generic*, or *polymorphic*, function with *type parameter* $t$ standing for an unspecified type within $e$. A *type application*, or *instantiation*, $\mathtt{App}[\tau](e)$, applies a polymorphic function to a specified type, which is then plugged in for the type parameter to obtain the result. Polymorphic functions are classified by the *universal type*, $\mathtt{all}(t.\tau)$, that determines the type, $\tau$, of the result as a function of the argument, $t$.

The statics of **F** consists of two judgment forms, the *type formation* judgment,

$$\Delta \vdash \tau \ \mathsf{type},$$

and the *typing judgment*,

$$\Delta\ \Gamma \vdash e : \tau.$$

The hypotheses $\Delta$ have the form $t$ type, where $t$ is a variable of sort Typ, and the hypotheses $\Gamma$ have the form $x : \tau$, where $x$ is a variable of sort Exp.

The rules defining the type formation judgment are as follows:

$$\frac{}{\Delta, t \ \mathsf{type} \vdash t \ \mathsf{type}} \tag{20.1a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type} \quad \Delta \vdash \tau_2 \ \mathsf{type}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2) \ \mathsf{type}} \tag{20.1b}$$

$$\frac{\Delta, t \ \mathsf{type} \vdash \tau \ \mathsf{type}}{\Delta \vdash \mathtt{all}(t.\tau) \ \mathsf{type}} \tag{20.1c}$$

The rules defining the typing judgment are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{20.2a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type} \quad \Delta\ \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam}[\tau_1](x.e) : \mathtt{arr}(\tau_1;\tau_2)} \tag{20.2b}$$

[1] Girard's original version of System F included the natural numbers as a basic type.

Preview

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{20.2c}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{20.2d}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau') \quad \Delta \vdash \tau\ \mathsf{type}}{\Delta\ \Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\tau'} \tag{20.2e}$$

**Lemma 20.1** (Regularity)**.** *If $\Delta\ \Gamma \vdash e : \tau$, and if $\Delta \vdash \tau_i\ \mathsf{type}$ for each assumption $x_i : \tau_i$ in $\Gamma$, then $\Delta \vdash \tau\ \mathsf{type}$.*

*Proof.* By induction on rules (20.2). □

The statics admits the structural rules for a general hypothetical judgment. In particular, we have the following critical substitution property for type formation and expression typing.

**Lemma 20.2** (Substitution)**.** 1. *If $\Delta, t\ \mathsf{type} \vdash \tau'\ \mathsf{type}$ and $\Delta \vdash \tau\ \mathsf{type}$, then $\Delta \vdash [\tau/t]\tau'\ \mathsf{type}$.*

2. *If $\Delta, t\ \mathsf{type}\ \Gamma \vdash e' : \tau'$ and $\Delta \vdash \tau\ \mathsf{type}$, then $\Delta\ [\tau/t]\Gamma \vdash [\tau/t]e' : [\tau/t]\tau'$.*

3. *If $\Delta\ \Gamma, x : \tau \vdash e' : \tau'$ and $\Delta\ \Gamma \vdash e : \tau$, then $\Delta\ \Gamma \vdash [e/x]e' : \tau'$.*

The second part of the lemma requires substitution into the context, $\Gamma$, as well as into the term and its type, because the type variable $t$ may occur freely in any of these positions.

Returning to the motivating examples from the introduction, the polymorphic identity function, $I$, is written

$$\Lambda(t.\lambda\ (x{:}t)\ x);$$

it has the polymorphic type

$$\forall(t.t \to t).$$

Instances of the polymorphic identity are written $I[\tau]$, where $\tau$ is some type, and have the type $\tau \to \tau$.

Similarly, the polymorphic composition function, $C$, is written

$$\Lambda(t_1.\Lambda(t_2.\Lambda(t_3.\lambda\ (f{:}t_2 \to t_3)\ \lambda\ (g{:}t_1 \to t_2)\ \lambda\ (x{:}t_1)\ f(g(x))))).$$

The function $C$ has the polymorphic type

$$\forall(t_1.\forall(t_2.\forall(t_3.(t_2 \to t_3) \to (t_1 \to t_2) \to (t_1 \to t_3)))).$$

Instances of $C$ are obtained by applying it to a triple of types, written $C[\tau_1][\tau_2][\tau_3]$. Each such instance has the type

$$(\tau_2 \to \tau_3) \to (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_3).$$

Preview

## Dynamics

The dynamics of **F** is given as follows:

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{20.3a}$$

$$\frac{}{\mathtt{Lam}(t.e)\ \mathsf{val}} \tag{20.3b}$$

$$\frac{[e_2\ \mathsf{val}]}{\mathtt{ap}(\mathtt{lam}[\tau_1](x.e);e_2) \longmapsto [e_2/x]e} \tag{20.3c}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{ap}(e_1;e_2) \longmapsto \mathtt{ap}(e_1';e_2)} \tag{20.3d}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathtt{ap}(e_1;e_2) \longmapsto \mathtt{ap}(e_1;e_2')}\right] \tag{20.3e}$$

$$\frac{}{\mathtt{App}[\tau](\mathtt{Lam}(t.e)) \longmapsto [\tau/t]e} \tag{20.3f}$$

$$\frac{e \longmapsto e'}{\mathtt{App}[\tau](e) \longmapsto \mathtt{App}[\tau](e')} \tag{20.3g}$$

The bracketed premises and rule are to be included for a call-by-value interpretation, and omitted for a call-by-name interpretation of **F**.

It is a simple matter to prove safety for **F**, using familiar methods.

**Lemma 20.3** (Canonical Forms)**.** *Suppose that $e : \tau$ and $e$ val, then*

1. *If $\tau = \mathtt{arr}(\tau_1;\tau_2)$, then $e = \mathtt{lam}[\tau_1](x.e_2)$ with $x : \tau_1 \vdash e_2 : \tau_2$.*
2. *If $\tau = \mathtt{all}(t.\tau')$, then $e = \mathtt{Lam}(t.e')$ with $t$ type $\vdash e' : \tau'$.*

*Proof.* By rule induction on the statics. □

**Theorem 20.4** (Preservation)**.** *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

*Proof.* By rule induction on the dynamics. □

**Theorem 20.5** (Progress)**.** *If $e : \tau$, then either $e$ val or there exists $e'$ such that $e \longmapsto e'$.*

*Proof.* By rule induction on the statics. □

Preview

## 20.2 Polymorphic Definability

The language **F** is astonishingly expressive. Not only are all finite products and sums definable in the language, but so are all inductive and coinductive types. This is most naturally expressed using definitional equality, which is defined to be the least congruence containing the following two axioms:

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \quad \Delta\ \Gamma \vdash e_1 : \tau_1}{\Delta\ \Gamma \vdash (\lambda\,(x{:}\tau)\,e_2)(e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{20.4a}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau \quad \Delta \vdash \rho\ \mathsf{type}}{\Delta\ \Gamma \vdash \Lambda(t.e)[\rho] \equiv [\rho/t]e : [\rho/t]\tau} \tag{20.4b}$$

In addition there are rules omitted here specifying that definitional equality is a congruence relation (that is, an equivalence relation respected by all expression-forming operations).

### 20.2.1 Products and Sums

The nullary product, or unit, type is definable in **F** as follows:

$$\begin{aligned}\mathtt{unit} &\triangleq \forall(r.r \to r)\\ \langle\rangle &\triangleq \Lambda(r.\lambda\,(x{:}r)\,x)\end{aligned}$$

The identity function plays the role of the null tuple, because it is the only closed value of this type.

Binary products are definable in **F** by using encoding tricks similar to those described in Chapter 17 for the untyped $\lambda$-calculus:

$$\begin{aligned}\tau_1 \times \tau_2 &\triangleq \forall(r.(\tau_1 \to \tau_2 \to r) \to r)\\ \langle e_1, e_2\rangle &\triangleq \Lambda(r.\lambda\,(x{:}\tau_1 \to \tau_2 \to r)\,x(e_1)(e_2))\\ e \cdot \mathtt{l} &\triangleq e[\tau_1](\lambda\,(x{:}\tau_1)\,\lambda\,(y{:}\tau_2)\,x)\\ e \cdot \mathtt{r} &\triangleq e[\tau_2](\lambda\,(x{:}\tau_1)\,\lambda\,(y{:}\tau_2)\,y)\end{aligned}$$

The statics given in Chapter 10 is derivable according to these definitions. Moreover, the following definitional equalities are derivable in **F** from these definitions:

$$\langle e_1, e_2\rangle \cdot \mathtt{l} \equiv e_1 : \tau_1$$

and

$$\langle e_1, e_2\rangle \cdot \mathtt{r} \equiv e_2 : \tau_2.$$

The nullary sum, or void, type is definable in **F**:

$$\begin{aligned}\mathtt{void} &\triangleq \forall(r.r)\\ \mathtt{abort}[\rho](e) &\triangleq e[\rho]\end{aligned}$$

Preview

There is no definitional equality to be checked, there being no introductory rule for the void type.

Binary sums are also definable in **F**:

$$\begin{aligned}
\tau_1 + \tau_2 &\triangleq \forall(r.(\tau_1 \to r) \to (\tau_2 \to r) \to r)\\
\mathtt{l} \cdot e &\triangleq \Lambda(r.\lambda\,(x{:}\tau_1 \to r)\,\lambda\,(y{:}\tau_2 \to r)\,x(e))\\
\mathtt{r} \cdot e &\triangleq \Lambda(r.\lambda\,(x{:}\tau_1 \to r)\,\lambda\,(y{:}\tau_2 \to r)\,y(e))\\
\mathtt{case}\,e\,\{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} &\triangleq\\
e[\rho](\lambda\,(x_1{:}\tau_1)\,e_1)(\lambda\,(x_2{:}\tau_2)\,e_2)&
\end{aligned}$$

provided that the types make sense. It is easy to check that the following equivalences are derivable in **F**:

$$\mathtt{case}\,\mathtt{l} \cdot d_1\,\{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \equiv [d_1/x_1]e_1 : \rho$$

and

$$\mathtt{case}\,\mathtt{r} \cdot d_2\,\{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \equiv [d_2/x_2]e_2 : \rho.$$

Thus the dynamic behavior specified in Chapter 11 is correctly implemented by these definitions.

### 20.2.2 Natural Numbers

As we remarked above, the natural numbers (under a lazy interpretation) are also definable in **F**. The key is the representation of the iterator, whose typing rule we recall here for reference:

$$\frac{e_0 : \mathtt{nat} \quad e_1 : \tau \quad x : \tau \vdash e_2 : \tau}{\mathtt{iter}(e_1; x.e_2; e_0) : \tau}\,.$$

Because the result type $\tau$ is arbitrary, this means that if we have an iterator, then it can be used to define a function of type

$$\mathtt{nat} \to \forall(t.t \to (t \to t) \to t).$$

This function, when applied to an argument $n$, yields a polymorphic function that, for any result type, $t$, given the initial result for $\mathtt{z}$ and a transformation from the result for $x$ into the result for $\mathtt{s}(x)$, yields the result of iterating the transformation $n$ times, starting with the initial result.

Because the *only* operation we can perform on a natural number is to iterate up to it in this manner, we may simply *identify* a natural number, $n$, with the polymorphic iterate-up-to-$n$ function just described. This means that we may define the type of natural numbers in **F** by the following equations:

$$\begin{aligned}
\mathtt{nat} &\triangleq \forall(t.t \to (t \to t) \to t)\\
\mathtt{z} &\triangleq \Lambda(t.\lambda\,(z{:}t)\,\lambda\,(s{:}t \to t)\,z)\\
\mathtt{s}(e) &\triangleq \Lambda(t.\lambda\,(z{:}t)\,\lambda\,(s{:}t \to t)\,s(e[t](z)(s)))\\
\mathtt{iter}(e_1; x.e_2; e_0) &\triangleq e_0[\tau](e_1)(\lambda\,(x{:}\tau)\,e_2)
\end{aligned}$$

Preview

It is easy to check that the statics and dynamics of the natural numbers type given in Chapter 9 are derivable in **F** under these definitions. The representations of the numerals in **F** given by this chart are called the *polymorphic Church numerals*.

This shows that **F** is *at least as expressive* as **T**. But is it *more* expressive? Yes! It is possible to show that the evaluation function for **T** is definable in **F**, even though it is not definable in **T** itself. However, the same diagonal argument given in Chapter 9 applies here, showing that the evaluation function for **F** is not definable in **F**. We may enrich **F** a bit more to define the evaluator for **F**, but as long as all programs in the enriched language terminate, we will once again have an undefinable function, the evaluation function for that extension.

## 20.3 Parametricity Overview

A remarkable property of **F** is that polymorphic types severely constrain the behavior of their elements. We may prove useful theorems about an expression knowing *only* its type—that is, without ever looking at the code. For example, if $i$ is any expression of type $\forall(t.t \to t)$, then it must be the identity function. Informally, when $i$ is applied to a type, $\tau$, and an argument of type $\tau$, it must return a value of type $\tau$. But because $\tau$ is not specified until $i$ is called, the function has no choice but to return its argument, which is to say that it is essentially the identity function. Similarly, if $b$ is any expression of type $\forall(t.t \to t \to t)$, then $b$ must be equivalent to either $\Lambda(t.\lambda\,(x{:}t)\,\lambda\,(y{:}t)\,x)$ or $\Lambda(t.\lambda\,(x{:}t)\,\lambda\,(y{:}t)\,y)$. Intuitively, when $b$ is applied to two arguments of a given type, the only value it can return is one of the givens.

Properties of programs in **F** that may be proved without knowing the code of a program, but only its types, are called *parametricity properties*. The facts about the functions $i$ and $b$ stated above are examples of parametricity properties. Such properties are sometimes called "free theorems," because they come "for free" as a consequence of typing, without any knowledge of the code itself. It bears repeating that in **F** we move prove non-trivial behavioral properties of programs without ever examining the program text. The key to this incredible fact is that we are able to prove a deep property, called *parametricity*, about the language **F**, that then applies to every program written in **F**. One may say that the type system "pre-verifies" programs with respect to a broad range of useful properties, eliminating the need to prove those properties about every program separately. The parametricity theorem for **F** explains the remarkable experience that if a piece of code type checks, then it "just works." Parametricity narrows the space of well-typed programs sufficiently that the opportunities for programmer error are reduced to almost nothing.

So how does the parametricity theorem work? Without getting into too many technical details (but see Chapter 50 for a full treatment), we can give a brief summary of the main idea. Any function $i : \forall(t.t \to t)$ in **F** enjoys the following property:

> *For any type $\tau$ and any property $\mathcal{P}$ of the type $\tau$, then if $\mathcal{P}$ holds of $x : \tau$, then $\mathcal{P}$ holds of $i[\tau](x)$.*

To show that for any type $\tau$, and any $x$ of type $\tau$, the expression $i[\tau](x)$ is equivalent to $x$, it suffices to fix $x_0 : \tau$, and consider the property $\mathcal{P}_{x_0}$ that holds of $y : \tau$ iff $y$ is equivalent to $x_0$. Obviously $\mathcal{P}$ holds of $x_0$ itself, and hence by the above-displayed property of $i$, it must send any argument satisfying $\mathcal{P}_{x_0}$ to a result satisfying $\mathcal{P}_{x_0}$, which is to say that it must send $x_0$ to $x_0$. Because $x_0$ is an

Preview

arbitrary element of $\tau$, it follows that $i[\tau]$ must be the identity function, $\lambda\,(x{:}\tau)\,x$, on the type $\tau$, and because $\tau$ is itself arbitrary, $i$ must be the polymorphic identity function, $\Lambda(t.\lambda\,(x{:}t)\,x)$.

A similar argument suffices to show that the function $b$, defined above, must be either $\Lambda(t.\lambda\,(x{:}t)\,\lambda\,(y{:}t)\,x)$ or $\Lambda(t.\lambda\,(x{:}t)\,\lambda\,(y{:}t)\,y)$. By virtue of its type the function $b$ enjoys the parametricity property

> *For any type $\tau$ and any property $\mathcal{P}$ of $\tau$, if $\mathcal{P}$ holds of $x : \tau$ and of $y : \tau$, then $\mathcal{P}$ holds of $b[\tau](x)(y)$.*

Choose an arbitrary type $\tau$ and two arbitrary elements $x_0$ and $y_0$ of type $\tau$. Define $\mathcal{Q}_{x_0,y_0}$ to hold of $z : \tau$ iff either $z$ is equivalent to $x_0$ or $z$ is equivalent to $y_0$. Clearly $\mathcal{Q}_{x_0,y_0}$ holds of both $x_0$ and $y_0$ themselves, so by the quoted parametricity property of $b$, it follows that $\mathcal{Q}_{x_0,y_0}$ holds of $b[\tau](x_0)(y_0)$, which is to say that it is equivalent to either $x_0$ or $y_0$. Since $\tau$, $x_0$, and $y_0$ are arbitrary, it follows that $b$ is equivalent to either $\Lambda(t.\lambda\,(x{:}t)\,\lambda\,(y{:}t)\,x)$ or $\Lambda(t.\lambda\,(x{:}t)\,\lambda\,(y{:}t)\,y)$.

The parametricity theorem for **F** implies even stronger properties of functions such as $i$ and $b$ considered above. For example, the function $i$ of type $\forall(t.t \to t)$ also satisfies the following condition:

> *If $\tau$ and $\tau'$ are any two types, and $\mathcal{R}$ is a binary relation between $\tau$ and $\tau'$, then for any $x : \tau$ and $x' : \tau'$, if $\mathcal{R}$ relates $x$ to $x'$, then $\mathcal{R}$ relates $i[\tau](x)$ to $i[\tau'](x')$.*

Using this property we may again prove that $i$ is equivalent to the polymorphic identity function by another means. Specifically, we will argue that, for any $g : \tau \to \tau$ and for any $x : \tau$, the application $i[\tau](g(x_0))$ is equivalent to $g(i[\tau](x_0))$. Then we may choose $g$ to be the constantly $x_0$ function to conclude that $i[\tau]$ sends $x_0$ to itself, and hence is the polymorphic identity function.

To this end fix $g : \tau \to \tau$ arbitrarily, and define the relation $\mathcal{R}_g$ to be the graph of $g$, which relates $u : \tau$ to $g(u) : \tau$. Let $x_0 : \tau$ be arbitrary, let $g_0$ be $\lambda\,(_{:}\tau)\,x_0$, and let $y_0$ be $g(x_0)$. Since $R_g$ relates $x_0$ to $y_0$, it follows from the above parametricity property of $i$ that $R_g$ relates $i[\tau](x_0)$ to $i[\tau](y_0)$, which is to say that $g_0(i[\tau](x_0))$ is equivalent to $i[\tau](g_0(x_0))$. But by the choice of $g_0$ this means that $x_0$ is equivalent to $i[\tau](x_0)$. Since $\tau$ and $x_0$ were chosen arbitrarily, $i$ is thereby equivalent to $\Lambda(t.\lambda\,(x{:}t)\,x)$.

## 20.4 Restricted Forms of Polymorphism

In this section we briefly examine some restricted forms of polymorphism with less than the full expressive power of **F**. These are obtained in one of two ways:

1. Restricting type instantiation to unquantified types.
2. Restricting the occurrence of quantifiers within types.

### 20.4.1 Predicative Fragment

The remarkable expressive power of the language **F** may be traced to the ability to instantiate a polymorphic type with another polymorphic type. For example, if we let $\tau$ be the type $\forall(t.t \to t)$,



Preview

and, assuming that $e : \tau$, we may apply $e$ to its own type, obtaining the expression $e[\tau]$ of type $\tau \to \tau$. Written out in full, this is the type

$$\forall(t.t \to t) \to \forall(t.t \to t),$$

which is larger (both textually, and when measured by the number of occurrences of quantified types) than the type of $e$ itself. In fact, this type is large enough that we can go ahead and apply $e[\tau]$ to $e$ again, obtaining the expression $e[\tau](e)$, which is again of type $\tau$ — the very type of $e$.

This property of **F** is called *impredicativity*[2]; the language **F** is said to permit *impredicative (type) quantification*. The distinguishing characteristic of impredicative polymorphism is that it involves a kind of circularity in that the meaning of a quantified type is given in terms of its instances, including the quantified type itself. This circularity is responsible for the surprising expressive power of **F**, and is correspondingly the prime source of complexity when reasoning about it (as detailed in Chapter 50).

Contrast this with function types, in which the type of an application of a function is evidently smaller than the type of the function itself. For if $e : \tau_1 \to \tau_2$, and $e_1 : \tau_1$, then we have $e(e_1) : \tau_2$, a smaller type than the type of $e$. This situation extends to polymorphism, provided that we impose the restriction that a quantified type can only be instantiated by an un-quantified type. For in that case passage from $\forall(t.\tau)$ to $[\rho/t]\tau$ decreases the number of quantifiers (even if the size of the type expression viewed as a tree grows). For example, the type $\forall(t.t \to t)$ may be instantiated with the type $u \to u$ to obtain the type $(u \to u) \to (u \to u)$. This type has more symbols in it than $\tau$, but is smaller in that it has fewer quantifiers. The restriction to quantification to unquantified types is called *predicative*[3] *polymorphism*. The predicative fragment is significantly less expressive than the full impredicative language. In particular, the definitions of types given in Section 20.2 are not valid in the predicative sublanguage of **F**.

### 20.4.2 Prenex Fragment

A rather more restricted form of polymorphism, called the *prenex fragment*, further restricts polymorphism to occur only at the outermost level — not only is quantification predicative, but quantifiers are not permitted to occur within the arguments to any other type constructors. This restriction, called *prenex quantification*, is often imposed for the sake of type inference, which permits type annotations to be omitted entirely in the knowledge that they can be recovered from the way the expression is used. We will not discuss type inference here, but we will give a formulation of the prenex fragment of **F**, because it plays an important role in the design of practical polymorphic languages.

The prenex fragment of **F** is designated $\mathbf{P}_1$, for reasons that will become clear in the next subsection. It is defined by *stratifying* types into two sorts, the *monotypes* (or *rank*-0 types) and the *polytypes* (or *rank*-1 types). The monotypes are those that do not involve any quantification, and may be used to instantiate the polymorphic quantifier. The polytypes include the monotypes, but also permit quantification over monotypes. These classifications are expressed by the judgments

[2] pronounced *im-PRED-ic-a-tiv-it-y*
[3] pronounced *PRED-i-ca-tive*



Preview

$\Delta \vdash \tau$ mono and $\Delta \vdash \tau$ poly, where $\Delta$ is a finite set of hypotheses of the form $t$ mono, where $t$ is a type variable not otherwise declared in $\Delta$. The rules for deriving these judgments are as follows:

$$\frac{}{\Delta, t \text{ mono} \vdash t \text{ mono}} \tag{20.5a}$$

$$\frac{\Delta \vdash \tau_1 \text{ mono} \quad \Delta \vdash \tau_2 \text{ mono}}{\Delta \vdash \mathtt{arr}(\tau_1; \tau_2) \text{ mono}} \tag{20.5b}$$

$$\frac{\Delta \vdash \tau \text{ mono}}{\Delta \vdash \tau \text{ poly}} \tag{20.5c}$$

$$\frac{\Delta, t \text{ mono} \vdash \tau \text{ poly}}{\Delta \vdash \mathtt{all}(t.\tau) \text{ poly}} \tag{20.5d}$$

Base types, such as `nat` (as a primitive), or other type constructors, such as sums and products, would be added to the language as monotypes.

The statics of $\mathbf{P}_1$ is given by rules for deriving hypothetical judgments of the form $\Delta\ \Gamma \vdash e : \rho$, where $\Delta$ consists of hypotheses of the form $t$ mono, and $\Gamma$ consists of hypotheses of the form $x : \rho$, where $\Delta \vdash \rho$ poly. The rules defining this judgment are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{20.6a}$$

$$\frac{\Delta \vdash \tau_1 \text{ mono} \quad \Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) : \mathtt{arr}(\tau_1; \tau_2)} \tag{20.6b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2; \tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1; e_2) : \tau} \tag{20.6c}$$

$$\frac{\Delta, t \text{ mono}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{20.6d}$$

$$\frac{\Delta \vdash \tau \text{ mono} \quad \Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau')}{\Delta\ \Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\tau'} \tag{20.6e}$$

We tacitly exploit the inclusion of monotypes as polytypes so that all typing judgments have the form $e : \rho$ for some expression $e$ and polytype $\rho$.

The restriction on the domain of a $\lambda$-abstraction to be a monotype means that a fully general `let` construct is no longer definable—there is no means of binding an expression of polymorphic type to a variable. For this reason it is usual to augment $\mathbf{P}_1$ with a primitive `let` construct whose statics is as follows:

$$\frac{\Delta \vdash \tau_1 \text{ poly} \quad \Delta\ \Gamma \vdash e_1 : \tau_1 \quad \Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{let}[\tau_1](e_1; x.e_2) : \tau_2}\ . \tag{20.7}$$

For example, the expression

$$\mathtt{let}\ I{:}\forall(t.t \to t)\ \mathtt{be}\ \Lambda(t.\lambda\ (x{:}t)\ x)\ \mathtt{in}\ I[\tau \to \tau](I[\tau])$$

has type $\tau \to \tau$ for any polytype $\tau$.



Preview

### 20.4.3 Rank-Restricted Fragments

The binary distinction between monomorphic and polymorphic types in $\mathbf{P}_1$ may be generalized to form a hierarchy of languages in which the occurrences of polymorphic types are restricted in relation to function types. The key feature of the prenex fragment is that quantified types are not permitted to occur in the domain of a function type. The prenex fragment also prohibits polymorphic types from the range of a function type, but it would be harmless to admit it, there being no difference between the type $\rho \to \forall(t.\tau)$ and the type $\forall(t.\rho \to \tau)$ (where $t \notin \rho$). This motivates the definition of a hierarchy of fragments of **F** that includes the prenex fragment as a special case.

We will define a judgment of the form $\tau\ \mathsf{type}\,[k]$, where $k \geq 0$, to mean that $\tau$ is a type of *rank* $k$. Informally, types of rank 0 have no quantification, and types of rank $k+1$ may involve quantification, but the domains of function types are restricted to be of rank $k$. Thus, in the terminology of Section 20.4.2, a monotype is a type of rank 0 and a polytype is a type of rank 1.

The definition of the types of rank $k$ is defined simultaneously for all $k$ by the following rules. These rules involve hypothetical judgments of the form $\Delta \vdash \tau\ \mathsf{type}\,[k]$, where $\Delta$ is a finite set of hypotheses of the form $t_i\ \mathsf{type}\,[k_i]$ for some pairwise distinct set of type variables $t_i$. The rules defining these judgments are as follows:

$$\frac{}{\Delta, t\ \mathsf{type}\,[k] \vdash t\ \mathsf{type}\,[k]} \tag{20.8a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type}\,[0] \quad \Delta \vdash \tau_2\ \mathsf{type}\,[0]}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2)\ \mathsf{type}\,[0]} \tag{20.8b}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type}\,[k] \quad \Delta \vdash \tau_2\ \mathsf{type}\,[k+1]}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2)\ \mathsf{type}\,[k+1]} \tag{20.8c}$$

$$\frac{\Delta \vdash \tau\ \mathsf{type}\,[k]}{\Delta \vdash \tau\ \mathsf{type}\,[k+1]} \tag{20.8d}$$

$$\frac{\Delta, t\ \mathsf{type}\,[k] \vdash \tau\ \mathsf{type}\,[k+1]}{\Delta \vdash \mathtt{all}(t.\tau)\ \mathsf{type}\,[k+1]} \tag{20.8e}$$

With these restrictions in mind, it is a good exercise to define the statics of $\mathbf{P}_k$, the restriction of **F** to types of rank $k$ (or less). It is most convenient to consider judgments of the form $e : \tau\,[k]$ specifying simultaneously that $e : \tau$ and $\tau\ \mathsf{type}\,[k]$. For example, the rank-limited rules for $\lambda$-abstractions is phrased as follows:

$$\frac{\Delta \vdash \tau_1\ \mathsf{type}\,[0] \quad \Delta\ \Gamma, x : \tau_1\,[0] \vdash e_2 : \tau_2\,[0]}{\Delta\ \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) : \mathtt{arr}(\tau_1;\tau_2)\,[0]} \tag{20.9a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type}\,[k] \quad \Delta\ \Gamma, x : \tau_1\,[k] \vdash e_2 : \tau_2\,[k+1]}{\Delta\ \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) : \mathtt{arr}(\tau_1;\tau_2)\,[k+1]} \tag{20.9b}$$

The remaining rules follow a similar pattern.



Preview

## 20.5 Notes

System F was introduced by Girard (1972) in the context of proof theory and by Reynolds (1974) in the context of programming languages. The concept of parametricity was originally isolated by Strachey, but was not fully developed until the work of Reynolds (1983). The description of parametricity as providing "theorems for free" was popularized by Wadler (1989).

## Exercises

**20.1**. Define in **F** the type `bool` of *Church booleans*. Give the definition of the type `bool`, and define `true` and `false` of this type, and the conditional `if` $e$ `then` $e_0$ `else` $e_1$, where $e$ is of this type.

**20.2**. Define in **F** the inductive type of lists of natural numbers as defined in Chapter 14. *Hint*: Define the representation in terms of the recursor (elimination form) for lists, following the pattern for defining the type of natural numbers.

**20.3**. Define in **F** an arbitrary inductive type, $\mu(t.\tau)$. *Hint*: generalize your answer to Exercise **20.2**.

**20.4**. Define the type $t$ `list` as in Exercise **20.2**, with the element type, $t$, unspecified. Define the finite set of *elements* of a list $l$ to be those $x$ given by the head of some number of tails of $l$. Now suppose that $f : \forall(t.t\,\texttt{list} \to t\,\texttt{list})$ is an arbitrary function of the stated type. Show that the elements of $f[\tau](l)$ are a subset of those of $l$. This means that $f$ may only permute, replicate, or drop elements from its input list to obtain its output list.

**20.5**. The type of natural numbers given in Section 20.2.2 is a well-formed polymorphic type of rank 0. And it is easy to check that the Church numerals have this rank-restricted type. But what goes wrong if we try to consider the rank 0 version of the representation of the natural numbers to be the type of natural numbers?

Preview

# Chapter 21

# Abstract Types

Data abstraction is perhaps the most important technique for structuring programs. The main idea is to introduce an *interface* that serves as a contract between the *client* and the *implementor* of an abstract type. The interface specifies what the client may rely on for its own work, and, simultaneously, what the implementor must provide to satisfy the contract. The interface serves to isolate the client from the implementor so that each may be developed in isolation from the other. In particular one implementation may be replaced by another without affecting the behavior of the client, provided that the two implementations meet the same interface and are, in a sense to be made precise below, suitably related to one another. (Roughly, each simulates the other with respect to the operations in the interface.) This property is called *representation independence* for an abstract type.

Data abstraction may be formalized by extending the language **F** with *existential types*. Interfaces are modelled as existential types that provide a collection of operations acting on an unspecified, or abstract, type. Implementations are modelled as packages, the introductory form for existentials, and clients are modelled as uses of the corresponding elimination form. It is remarkable that the programming concept of data abstraction is modelled so naturally and directly by the logical concept of existential type quantification. Existential types are closely connected with universal types, and hence are often treated together. The superficial reason is that both are forms of type quantification, and hence both require the machinery of type variables. The deeper reason is that existentials are *definable* from universals — surprisingly, data abstraction is actually just a form of polymorphism! One consequence of this observation is that representation independence is just a use of the parametricity properties of polymorphic functions discussed in Chapter 20.

## 21.1 Existential Types

The syntax of **FE** is the extension of **F** with the following constructs:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | ::= | $\mathtt{some}(t.\tau)$ | $\exists(t.\tau)$ | interface |
| Exp | $e$ | ::= | $\mathtt{pack}[t.\tau][\rho](e)$ | $\mathtt{pack}\,\rho\,\mathtt{with}\,e\,\mathtt{as}\,\exists(t.\tau)$ | implementation |
| | | | $\mathtt{open}[t.\tau][\rho](e_1; t, x.e_2)$ | $\mathtt{open}\,e_1\,\mathtt{as}\,t\,\mathtt{with}\,x{:}\tau\,\mathtt{in}\,e_2$ | client |

Preview

The introductory form $\exists(t.\tau)$ is a *package* of the form $\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau)$, where $\rho$ is a type and $e$ is an expression of type $[\rho/t]\tau$. The type $\rho$ is called the *representation type* of the package, and the expression $e$ is called the *implementation* of the package. The eliminatory form is the expression $\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x{:}\tau\,\texttt{in}\,e_2$, which *opens* the package $e_1$ for use within the *client* $e_2$ by binding its representation type to $t$ and its implementation to $x$ for use within $e_2$. Crucially, the typing rules ensure that the client is type-correct independently of the actual representation type used by the implementor, so that it may be varied without affecting the type correctness of the client.

The abstract syntax of the open construct specifies that the type variable, $t$, and the expression variable, $x$, are bound within the client. They may be renamed at will by $\alpha$-equivalence without affecting the meaning of the construct, provided, of course, that the names are chosen so as not to conflict with any others that may be in scope. In other words the type, $t$, may be thought of as a "new" type, one that is distinct from all other types, when it is introduced. This is sometimes called *generativity* of abstract types: the use of an abstract type by a client "generates" a "new" type within that client. This behavior is simply a consequence of identifying terms up to $\alpha$-equivalence, and is not particularly tied to data abstraction.

### 21.1.1 Statics

The statics of **FE** is given by these rules:

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \texttt{some}(t.\tau)\ \mathsf{type}} \tag{21.1a}$$

$$\frac{\Delta \vdash \rho\ \mathsf{type} \quad \Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \quad \Delta\ \Gamma \vdash e : [\rho/t]\tau}{\Delta\ \Gamma \vdash \texttt{pack}[t.\tau][\rho](e) : \texttt{some}(t.\tau)} \tag{21.1b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \texttt{some}(t.\tau) \quad \Delta, t\ \mathsf{type}\ \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta\ \Gamma \vdash \texttt{open}[t.\tau][\tau_2](e_1; t, x.e_2) : \tau_2} \tag{21.1c}$$

Rule (21.1c) is complex, so study it carefully! There are two important things to notice:

1. The type of the client, $\tau_2$, must not involve the abstract type $t$. This restriction prevents the client from attempting to export a value of the abstract type outside of the scope of its definition.
2. The body of the client, $e_2$, is type checked without knowledge of the representation type, $t$. The client is, in effect, polymorphic in the type variable $t$.

**Lemma 21.1** (Regularity)**.** *Suppose that* $\Delta\ \Gamma \vdash e : \tau$*. If* $\Delta \vdash \tau_i\ \mathsf{type}$ *for each* $x_i : \tau_i$ *in* $\Gamma$*, then* $\Delta \vdash \tau\ \mathsf{type}$*.*

*Proof.* By induction on rules (21.1). □

### 21.1.2 Dynamics

The dynamics of **FE** is specified by the following rules (including the bracketed material for an eager interpretation, and omitting it for a lazy interpretation):

$$\frac{[e\ \mathsf{val}]}{\texttt{pack}[t.\tau][\rho](e)\ \mathsf{val}} \tag{21.2a}$$



Preview

$$\left[\frac{e \longmapsto e'}{\mathtt{pack}[t.\tau][\rho](e) \longmapsto \mathtt{pack}[t.\tau][\rho](e')}\right] \tag{21.2b}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{open}[t.\tau][\tau_2](e_1; t, x.e_2) \longmapsto \mathtt{open}[t.\tau][\tau_2](e_1'; t, x.e_2)} \tag{21.2c}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{open}[t.\tau][\tau_2](\mathtt{pack}[t.\tau][\rho](e); t, x.e_2) \longmapsto [\rho, e/t, x]e_2} \tag{21.2d}$$

It is important to observe that, according to these rules, *there are no abstract types at run time*! The representation type is propagated to the client by substitution when the package is opened, thereby eliminating the abstraction boundary between the client and the implementor. Thus, data abstraction is a *compile-time discipline* that leaves no traces of its presence at execution time.

### 21.1.3 Safety

The safety of **FE** is stated and proved as usual, extending the proof of safety for **F**.

**Theorem 21.2** (Preservation)**.** *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

*Proof.* By rule induction on $e \longmapsto e'$, making use of substitution for both expression- and type variables. □

**Lemma 21.3** (Canonical Forms)**.** *If $e : \mathtt{some}(t.\tau)$ and $e$ val, then $e = \mathtt{pack}[t.\tau][\rho](e')$ for some type $\rho$ and some $e'$ such that $e' : [\rho/t]\tau$.*

*Proof.* By rule induction on the statics, making use of the definition of closed values. □

**Theorem 21.4** (Progress)**.** *If $e : \tau$ then either $e$ val or there exists $e'$ such that $e \longmapsto e'$.*

*Proof.* By rule induction on $e : \tau$, making use of the canonical forms lemma. □

## 21.2 Data Abstraction

To illustrate the use of **FE**, we consider an abstract type of queues of natural numbers supporting three operations:

1. Forming the empty queue.
2. Inserting an element at the tail of the queue.
3. Removing the head of the queue, which is assumed to be non-empty.

This is clearly a bare-bones interface, but is sufficient to illustrate the main ideas of data abstraction. Queue elements are here taken to be natural numbers, but nothing depends on this choice.



Preview

The crucial property of this description is that nowhere do we specify what queues actually *are*, only what we can *do* with them. This is captured by the following existential type, $\exists(t.\tau)$, which serves as the interface of the queue abstraction:

$$\exists(t.\langle \texttt{emp} \hookrightarrow t, \texttt{ins} \hookrightarrow \texttt{nat} \times t \to t, \texttt{rem} \hookrightarrow t \to (\texttt{nat} \times t)\,\texttt{opt}\rangle).$$

The representation type, $t$, of queues is *abstract* — all that is specified about it is that it supports the operations `emp`, `ins`, and `rem`, with the specified types.

An implementation of queues consists of a package specifying the representation type, together with the implementation of the associated operations in terms of that representation. Internally to the implementation, the representation of queues is known and relied upon by the operations. Here is a very simple implementation, $e_l$, in which queues are represented as lists:

$$\texttt{pack}\,\texttt{natlist}\,\texttt{with}\,\langle \texttt{emp} \hookrightarrow \texttt{nil}, \texttt{ins} \hookrightarrow e_i, \texttt{rem} \hookrightarrow e_r\rangle\,\texttt{as}\,\exists(t.\tau),$$

where

$$e_i : \texttt{nat} \times \texttt{natlist} \to \texttt{natlist} = \lambda\,(x{:}\texttt{nat} \times \texttt{natlist})\ldots,$$

and

$$e_r : \texttt{natlist} \to \texttt{nat} \times \texttt{natlist} = \lambda\,(x{:}\texttt{natlist})\ldots.$$

The elided body of $e_i$ conses the first component of $x$, the element, onto the second component of $x$, the queue, and the elided body of $e_r$ reverses its argument, and returns the head element paired with the reversal of the tail. Both of these operations "know" that queues are represented as values of type `natlist`, and are programmed accordingly.

It is also possible to give another implementation, $e_p$, of the same interface, $\exists(t.\tau)$, but in which queues are represented as pairs of lists, consisting of the "back half" of the queue paired with the reversal of the "front half". This representation avoids the need for reversals on each call, and, as a result, achieves amortized constant-time behavior:

$$\texttt{pack}\,\texttt{natlist} \times \texttt{natlist}\,\texttt{with}\,\langle \texttt{emp} \hookrightarrow \langle \texttt{nil}, \texttt{nil}\rangle, \texttt{ins} \hookrightarrow e_i, \texttt{rem} \hookrightarrow e_r\rangle\,\texttt{as}\,\exists(t.\tau).$$

In this case $e_i$ has type

$$\texttt{nat} \times (\texttt{natlist} \times \texttt{natlist}) \to (\texttt{natlist} \times \texttt{natlist}),$$

and $e_r$ has type

$$(\texttt{natlist} \times \texttt{natlist}) \to \texttt{nat} \times (\texttt{natlist} \times \texttt{natlist}).$$

These operations "know" that queues are represented as values of type $\texttt{natlist} \times \texttt{natlist}$, and are implemented accordingly.

The important point is that the *same* client type checks regardless of which implementation of queues we choose. This is because the representation type is hidden, or *held abstract*, from the client during type checking. Consequently, it cannot rely on whether it is `natlist` or $\texttt{natlist} \times \texttt{natlist}$ or some other type. That is, the client is *independent* of the representation of the abstract type.

Preview

## 21.3 Definability of Existentials

The language **FE** is not a proper extension of **F**, because existential types are definable in terms of universal types. Why should this be possible? The key is to observe that the client of an abstract type is *polymorphic* in the representation type. The typing rule for

$$\texttt{open}\, e_1\, \texttt{as}\, t\, \texttt{with}\, x{:}\tau\, \texttt{in}\, e_2 : \tau_2,$$

where $e_1 : \exists(t.\tau)$, specifies that $e_2 : \tau_2$ under the assumptions $t$ type and $x : \tau$. In essence, the client is a polymorphic function of type

$$\forall(t.\tau \to \tau_2),$$

where $t$ may occur in $\tau$ (the type of the operations), but not in $\tau_2$ (the type of the result).

This suggests the following encoding of existential types:

$$\begin{aligned}
\exists(t.\tau) &\triangleq \forall(u.\forall(t.\tau \to u) \to u)\\
\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau) &\triangleq \Lambda(u.\lambda\,(x{:}\forall(t.\tau \to u))\,x[\rho](e))\\
\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x{:}\tau\,\texttt{in}\,e_2 &\triangleq e_1[\tau_2](\Lambda(t.\lambda\,(x{:}\tau)\,e_2))
\end{aligned}$$

An existential is encoded as a polymorphic function taking the overall result type, $u$, as argument, followed by a polymorphic function representing the client with result type $u$, and yielding a value of type $u$ as overall result. Consequently, the `open` construct simply packages the client as such a polymorphic function, instantiates the existential at the result type, $\tau_2$, and applies it to the polymorphic client. (The translation therefore depends on knowing the overall result type, $\tau_2$, of the `open` construct.) Finally, a package consisting of a representation type $\rho$ and an implementation $e$ is a polymorphic function that, when given the result type, $u$, and the client, $x$, instantiates $x$ with $\rho$ and passes to it the implementation $e$.

It is a good exercise to show that this translation correctly reflects the statics and dynamics of existential types.

## 21.4 Representation Independence

An important consequence of parametricity is that it ensures that clients are insensitive to the representations of abstract types. More precisely, there is a criterion, called *bisimilarity*, for relating two implementations of an abstract type such that the behavior of a client is unaffected by swapping one implementation by another that is bisimilar to it. This leads to a simple methodology for proving the correctness of *candidate* implementation of an abstract type, which is to show that it is bisimilar to an obviously correct *reference* implementation of it. Because the candidate and the reference implementations are bisimilar, no client may distinguish them from one another, and hence if the client behaves properly with the reference implementation, then it must also behave properly with the candidate.

To derive the definition of bisimilarity of implementations, it is helpful to examine the definition of existentials in terms of universals given in Section 21.3. It is an immediate consequence of the definition that the client of an abstract type is polymorphic in the representation of the abstract

Preview

type. A client, $c$, of an abstract type $\exists(t.\tau)$ has type $\forall(t.\tau \to \tau_2)$, where $t$ does not occur free in $\tau_2$ (but may, of course, occur in $\tau$). Applying the parametricity property described informally in Chapter 20 (and developed rigorously in Chapter 50), this says that if $R$ is a bisimulation relation between any two implementations of the abstract type, then the client behaves identically on both of them. The fact that $t$ does not occur in the result type ensures that the behavior of the client is independent of the choice of relation between the implementations, provided that this relation is preserved by the operations that implement it.

To see what this means requires that we specify what is meant by a bisimulation. This is best done by example. Consider the existential type $\exists(t.\tau)$, where $\tau$ is the labeled tuple type

$$\langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow \mathtt{nat} \times t \to t, \mathtt{rem} \hookrightarrow t \to (\mathtt{nat} \times t)\,\mathtt{opt} \rangle.$$

This specifies an abstract type of queues. The operations `emp`, `ins`, and `rem` specify, respectively, the empty queue, an insert operation, and a remove operation. For the sake of simplicity the element type is chosen to be the natural numbers. The result of removal is an optional pair, according to whether the queue is empty or not.

Theorem 50.12 ensures that if $\rho$ and $\rho'$ are any two closed types, and if $R$ is a relation between expressions of these two types, then if the implementations $e : [\rho/x]\tau$ and $e' : [\rho'/x]\tau$ respect $R$, then $c[\rho]e$ behaves the same as $c[\rho']e'$. It remains to define when two implementations respect the relation $R$. Let

$$e \triangleq \langle \mathtt{emp} \hookrightarrow e_{\mathsf{m}}, \mathtt{ins} \hookrightarrow e_{\mathsf{i}}, \mathtt{rem} \hookrightarrow e_{\mathsf{r}} \rangle$$

and

$$e' \triangleq \langle \mathtt{emp} \hookrightarrow e'_{\mathsf{m}}, \mathtt{ins} \hookrightarrow e'_{\mathsf{i}}, \mathtt{rem} \hookrightarrow e'_{\mathsf{r}} \rangle.$$

For these implementations to respect $R$ means that the following three conditions hold:

1. The empty queues are related: $R(e_{\mathsf{m}}, e'_{\mathsf{m}})$.
2. Inserting the same element on each of two related queues yields related queues: if $d : \tau$ and $R(q, q')$, then $R(e_{\mathsf{i}}(d)(q), e'_{\mathsf{i}}(d)(q'))$.
3. If two queues are related, then either they are both empty, or their front elements are the same and their back elements are related: if $R(q, q')$, then either
   (a) $e_{\mathsf{r}}(q) \cong \mathtt{null} \cong e'_{\mathsf{r}}(q')$, or
   (b) $e_{\mathsf{r}}(q) \cong \mathtt{just}(\langle d, r \rangle)$ and $e'_{\mathsf{r}}(q') \cong \mathtt{just}(\langle d', r' \rangle)$, with $d \cong d'$ and $R(r, r')$.

If such a relation $R$ exists, then the implementations $e$ and $e'$ are said to be *bisimilar*. The terminology stems from the requirement that the operations of the abstract type preserve the relation: if it holds before an operation is performed, then it must also hold afterwards, and the relation must hold for the initial state of the queue. Thus each implementation *simulates* the other up to the relationship specified by $R$.

To see how this works in practice, let us consider informally two implementations of the abstract type of queues specified above. For the reference implementation we choose $\rho$ to be the type `natlist`, and define the empty queue to be the empty list, define insert to add the specified



Preview

element to the head of the list, and define remove to remove the last element of the list. The code is as follows:

$$
\begin{aligned}
t &\triangleq \texttt{natlist} \\
\texttt{emp} &\triangleq \texttt{nil} \\
\texttt{ins} &\triangleq \lambda\,(x{:}\texttt{nat})\,\lambda\,(q{:}t)\,\texttt{cons}(x;q) \\
\texttt{rem} &\triangleq \lambda\,(q{:}t)\,\texttt{case}\,\texttt{rev}(q)\,\{\texttt{nil} \hookrightarrow \texttt{null} \mid \texttt{cons}(f;qr) \hookrightarrow \texttt{just}(\langle f, \texttt{rev}(qr)\rangle)\}.
\end{aligned}
$$

Removing an element takes time linear in the length of the list, because of the reversal.

For the candidate implementation we choose $\rho'$ to be the type $\texttt{natlist} \times \texttt{natlist}$ of pairs of lists $\langle b, f\rangle$ in which $b$ is the "back half" of the queue, and $f$ is the *reversal* of the "front half" of the queue. For this representation we define the empty queue to be a pair of empty lists, define insert to extend the back with that element at the head, and define remove based on whether the front is empty or not. If it is non-empty, the head element is removed from it, and returned along with the pair consisting of the back and the tail of the front. If it is empty, and the back is not, then we reverse the back, remove the head element, and return the pair consisting of the empty list and the tail of the now-reversed back. The code is as follows:

$$
\begin{aligned}
t &\triangleq \texttt{natlist} \times \texttt{natlist} \\
\texttt{emp} &\triangleq \langle \texttt{nil}, \texttt{nil}\rangle \\
\texttt{ins} &\triangleq \lambda\,(x{:}\texttt{nat})\,\lambda\,(\langle bs, fs\rangle{:}t)\,\langle \texttt{cons}(x;bs), fs\rangle \\
\texttt{rem} &\triangleq \lambda\,(\langle bs, fs\rangle{:}t)\,\texttt{case}\,fs\,\{\texttt{nil} \hookrightarrow e \mid \texttt{cons}(f;fs') \hookrightarrow \langle bs, fs'\rangle\},\ \text{where} \\
e &\triangleq \texttt{case}\,\texttt{rev}(bs)\,\{\texttt{nil} \hookrightarrow \texttt{null} \mid \texttt{cons}(b;bs') \hookrightarrow \texttt{just}(\langle b, \langle \texttt{nil}, bs'\rangle\rangle)\}.
\end{aligned}
$$

The cost of the occasional reversal may be amortized across the sequence of inserts and removes to show that each operation in a sequence costs unit time overall.

To show that the candidate implementation is correct, we show that it is bisimilar to the reference implementation. This reduces to specifying a relation, $R$, between the types $\texttt{natlist}$ and $\texttt{natlist} \times \texttt{natlist}$ such that the three simulation conditions given above are satisfied by the two implementations just described. The relation in question states that $R(l, \langle b, f\rangle)$ iff the list $l$ is the list $\texttt{app}(b)(\texttt{rev}(f))$, where app is the evident append function on lists. That is, thinking of $l$ as the reference representation of the queue, the candidate must maintain that the elements of $b$ followed by the elements of $f$ in reverse order form precisely the list $l$. It is easy to check that the implementations just described preserve this relation. Having done so, we are assured that the client, $c$, behaves the same regardless of whether we use the reference or the candidate. Because the reference implementation is obviously correct (albeit inefficient), the candidate must also be correct in that the behavior of any client is unaffected by using it instead of the reference.

## 21.5 Notes

The connection between abstract types in programming languages and existential types in logic was made by Mitchell and Plotkin (1988). Closely related ideas were already present in Reynolds

Preview

(1974), but the connection with existential types was not explicitly drawn there. The account of representation independence given here is derived from Mitchell (1986).

## Exercises

**21.1**. Define in **FE** of the coinductive type of streams of natural numbers as defined in Chapter 14. *Hint*: Define the representation in terms of the generator (introduction form) for streams.

**21.2**. Define in **FE** an arbitrary coinductive type, $\nu(t.\tau)$. *Hint*: generalize your answer to Exercise **21.1**.

**21.3**. Representation independence for abstract types may be seen, via the interpretation of **FE** in **F** given in Section 21.3, as a corollary of the parametricity property of polymorphic types. Recast the proof of equivalence of the two implementations of queues given in Section 21.4 as an instance of parametricity as defined informally in Chapter 20.

Preview

# Chapter 22

# Constructors and Kinds

The types $\texttt{nat} \to \texttt{nat}$ and $\texttt{nat}\,\texttt{list}$ may be thought of as being built from other types by the application of a *type constructor*, or *type operator*. These two examples differ from each other in that the function space type constructor takes two arguments, whereas the list type constructor takes only one. We may, for the sake of uniformity, think of types such as $\texttt{nat}$ as being built by a type constructor of *no* arguments. More subtly, we may even think of the types $\forall(t.\tau)$ and $\exists(t.\tau)$ as being built up in the same way by regarding the quantifiers as *higher-order* type operators.

These seemingly disparate cases may be treated uniformly by enriching the syntactic structure of a language with a new layer of *(type) constructors*. To ensure that constructors are used properly (for example, that the list constructor is given only one argument, and that the function constructor is given two), we classify constructors by *kinds*. Constructors of a distinguished kind, $\texttt{T}$, are types, which may be used to classify expressions. To allow for multi-argument and higher-order constructors, we will also consider finite product and function kinds. (Later we shall consider even richer kinds.)

The distinction between constructors and kinds on one hand and types and expressions on the other reflects a fundamental separation between the static and dynamic *phase* of processing of a programming language, called the *phase distinction*. The static phase implements the statics and the dynamic phase implements the dynamics. Constructors may be seen as a form of *static data* that is manipulated during the static phase of processing. Expressions are a form of *dynamic data* that is manipulated at run-time. Because the dynamic phase follows the static phase (we only execute well-typed programs), we may also manipulate constructors at run-time.

We will study the extension of the language **F** of Chapter 20 with constructors and kinds. This involves more complications than might at first be apparent. The main difficulty is that we shall have to consider a notion of definitional equality for constructors that determine when two constructors are the same to within some simplifications arising from the cancellation of introductory by eliminatory forms. The standard way to handle these simplifications is to formulate definitional equivalence using inference rules that ensure that it is a congruence, and require that typing respect this equivalence in the sense that if an expression has a type, then it also has every other type definitionally equal to the first-mentioned type. This approach seems natural, but leaves open the question of how to decide whether two constructors are equivalent, and hence leaves open the

Preview

question of how to decide whether a given expression has a given type in a given context.

The usual solution to this problem is to formulate an algorithm for checking definitional equivalence of well-formed constructors, from which decidability of type checking follows directly. An alternative, which we consider here, is to maintain constructors in fully-simplified (so-called "canonical") form, so that two fully simplified constructors are equivalent iff they are $\alpha$-equivalent. The difficulty here is that the property of being in canonical form is not preserved by (ordinary) substitution. For example, a constructor of the form $u[c]$, where $u$ is a variable and $c$ is fully simplified, is itself fully simplified, but if we plug in $\lambda\, v.c'$, then we obtain $(\lambda\, v.c')[c]$, which can be simplified to $[c/v]c'$, which may then lead to further opportunities for simplification. The ensuing cascade of simplifications following a substitution can be shown to be finite, which suggests that we can maintain constructors in fully simplified form by employing a special form of substitution, called *canonizing substitution*, that does the simplifications "on the fly" during substitution.

## 22.1 Constructors and Kinds

The syntax of kinds of $\mathbf{F}_\omega$ is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Kind | $\kappa$ | ::= | $\mathtt{Type}$ | $\mathtt{T}$ | types |
| | | | $\mathtt{Unit}$ | $1$ | nullary product |
| | | | $\mathtt{Prod}(\kappa_1;\kappa_2)$ | $\kappa_1 \times \kappa_2$ | binary product |
| | | | $\mathtt{Arr}(\kappa_1;\kappa_2)$ | $\kappa_1 \to \kappa_2$ | function |

The kinds consist of the kind of types, `T`, and the unit kind, `Unit`, and are closed under formation of product and function kinds.

The syntax of constructors of $\mathbf{F}_\omega$ is divided into two syntactic sorts, the *neutral* and the *canonical*, according to the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| NCon | $a$ | ::= | $u$ | $u$ | variable |
| | | | $\mathtt{proj}[\mathtt{l}](a)$ | $a \cdot \mathtt{l}$ | first projection |
| | | | $\mathtt{proj}[\mathtt{r}](a)$ | $a \cdot \mathtt{r}$ | second projection |
| | | | $\mathtt{app}(a_1;c_2)$ | $a_1[c_2]$ | application |
| CCon | $c$ | ::= | $\mathtt{atom}(a)$ | $\widehat{a}$ | neutral |
| | | | $\mathtt{unit}$ | $\langle\rangle$ | null tuple |
| | | | $\mathtt{pair}(c_1;c_2)$ | $\langle c_1,c_2\rangle$ | pair |
| | | | $\mathtt{lam}(u.c)$ | $\lambda\, u.c$ | abstraction |

The reason to distinguish neutral from canonical constructors is to ensure that it is impossible to apply an elimination form to an introduction form, which demands an equation to capture the inversion principle. For example, the putative constructor $\langle c_1,c_2\rangle \cdot \mathtt{l}$, which would be definitionally equal to $c_1$, is ill-formed according to the above syntax chart. This is because the argument to a projection must be neutral, but a pair is only canonical, not neutral.

The canonical constructor $\widehat{a}$ is the inclusion of neutral constructors into canonical constructors. However, the grammar does not capture a crucial property of the statics that ensures that only neutral constructors of kind `T` may be treated as canonical. This requirement is imposed to limit



Preview

the forms of canonical constructors of the other kinds. In particular, variables of function, product, or unit kind will turn out not to be canonical, but only neutral.

The statics of constructors and kinds in $\mathbf{F}_\omega$ is specified by the judgments

| | |
|---|---|
| $\Delta \vdash a \Uparrow \kappa$ | neutral constructor formation |
| $\Delta \vdash c \Downarrow \kappa$ | canonical constructor formation |

In each of these judgments $\Delta$ is a finite set of hypotheses of the form

$$u_1 \Uparrow \kappa_1, \ldots, u_n \Uparrow \kappa_n$$

for some $n \geq 0$. The form of the hypotheses expresses the principle that variables are neutral constructors. The formation judgments are to be understood as generic hypothetical judgments with parameters $u_1, \ldots, u_n$ that are determined by the forms of the hypotheses.

The rules for constructor formation are as follows:

$$\frac{}{\Delta, u \Uparrow \kappa \vdash u \Uparrow \kappa} \tag{22.1a}$$

$$\frac{\Delta \vdash a \Uparrow \kappa_1 \times \kappa_2}{\Delta \vdash a \cdot \mathtt{l} \Uparrow \kappa_1} \tag{22.1b}$$

$$\frac{\Delta \vdash a \Uparrow \kappa_1 \times \kappa_2}{\Delta \vdash a \cdot \mathtt{r} \Uparrow \kappa_2} \tag{22.1c}$$

$$\frac{\Delta \vdash a_1 \Uparrow \kappa_2 \to \kappa \quad \Delta \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash a_1[c_2] \Uparrow \kappa} \tag{22.1d}$$

$$\frac{\Delta \vdash a \Uparrow \mathtt{T}}{\Delta \vdash \widehat{a} \Downarrow \mathtt{T}} \tag{22.1e}$$

$$\frac{}{\Delta \vdash \langle\rangle \Downarrow 1} \tag{22.1f}$$

$$\frac{\Delta \vdash c_1 \Downarrow \kappa_1 \quad \Delta \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle \Downarrow \kappa_1 \times \kappa_2} \tag{22.1g}$$

$$\frac{\Delta, u \Uparrow \kappa_1 \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash \lambda\, u.c_2 \Downarrow \kappa_1 \to \kappa_2} \tag{22.1h}$$

Rule (22.1e) specifies that the only neutral constructors that are canonical are those with kind $\mathtt{T}$. This ensures that the language enjoys the following canonical forms property, which is easily proved by inspection of rules (22.1).

**Lemma 22.1.** *Suppose that $\Delta \vdash c \Downarrow \kappa$.*

1. *If $\kappa = 1$, then $c = \langle\rangle$.*
2. *If $\kappa = \kappa_1 \times \kappa_2$, then $c = \langle c_1, c_2 \rangle$ for some $c_1$ and $c_2$ such that $\Delta \vdash c_i \Downarrow \kappa_i$ for $i = 1, 2$.*
3. *If $\kappa = \kappa_1 \to \kappa_2$, then $c = \lambda\, u.c_2$ for some $u$ and $c_2$ such that $\Delta, u \Uparrow \kappa_1 \vdash c_2 \Downarrow \kappa_2$.*

*Proof.* By induction on rules (22.1). □

Preview

## 22.2 Expressions and Types

The statics of expressions of $\mathbf{F}_\omega$ is defined using two judgment forms:

| | |
|---|---|
| $\Delta \vdash \tau \ \mathsf{type}$ | type formation |
| $\Delta\ \Gamma \vdash e : \tau$ | expression formation |

Here, as before, $\Gamma$ is a finite set of hypotheses of the form

$$x_1 : \tau_1, \ldots, x_k : \tau_k$$

such that $\Delta \vdash \tau_i \ \mathsf{type}$ for each $1 \leq i \leq k$.

Every constructor of kind $\mathtt{T}$ is considered to be a classifier:

$$\frac{\Delta \vdash \tau \Uparrow \mathtt{T}}{\Delta \vdash \tau \ \mathsf{type}}\ . \tag{22.2}$$

In $\mathbf{F}_\omega$ this is the sole rule of type formation: every classifier arises as a constructor of kind $\mathtt{T}$. In particular, polymorphic types are constructors, and these quantify over the kind $\mathtt{T}$; they are impredicative. The predicative fragment, $\mathbf{F}^{\mathsf{p}}_\omega$, avoids this circularity by regarding the polymorphic type as a classifier, and not as a constructor. That way quantification is only over a separately defined domain, the kind of unquantified types, which does not include itself.

More precisely, $\mathbf{F}_\omega$ includes the following two constructor constants:

$$\Delta \vdash \to \Uparrow \mathtt{T} \to \mathtt{T} \to \mathtt{T} \tag{22.3a}$$

$$\Delta \vdash \forall_\kappa \Uparrow (\kappa \to \mathtt{T}) \to \mathtt{T} \tag{22.3b}$$

Officially, the classifier $\tau_1 \to \tau_2$ is the application $\to[\tau_1][\tau_2]$ of the constant given by Rule (22.3a) to two constructors, $\tau_1$ and $\tau_2$, of kind $\mathtt{T}$. Similarly, $\forall\, u :: \kappa.\tau$ is the application $\forall_\kappa[\lambda\, u.\tau]$ of the constant given by rule (22.3b) to the constructor $\lambda\, u.\tau$ of kind $\kappa \to \mathtt{T}$.

The predicative fragment, $\mathbf{F}^{\mathsf{p}}_\omega$, of $\mathbf{F}_\omega$ excludes rule (22.3b), but adds the following rule instead:

$$\frac{\Delta, u \Uparrow \kappa \vdash \tau \ \mathsf{type}}{\Delta \vdash \forall\, u :: \kappa.\tau \ \mathsf{type}}\ . \tag{22.4}$$

The type $\forall\, u :: \kappa.\tau$ is a classifier, but is not a constructor. The significance of this distinction becomes apparent when we consider the introduction and elimination forms for the generalized quantifier, which are the same for both fragments:

$$\frac{\Delta, u \Uparrow \kappa\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \Lambda(u{::}\kappa.e) : \forall\, u :: \kappa.\tau} \tag{22.5a}$$

$$\frac{\Delta\ \Gamma \vdash e : \forall\, u :: \kappa.\tau \quad \Delta \vdash c \Downarrow \kappa}{\Delta\ \Gamma \vdash e[c] : [c/u]\tau} \tag{22.5b}$$

Rule (22.5b) makes use of substitution, whose definition we will consider shortly.



Preview

Rule (22.5b) makes clear that a polymorphic abstraction quantifies over the constructors of kind $\kappa$. When $\kappa$ is $\mathtt{T}$ this kind may or may not include all of the classifiers of the language, according to whether we are working with the impredicative formulation of quantification (in which the quantifiers are distinguished constants for building constructors of kind $\mathtt{T}$) or the predicative formulation (in which quantifiers arise only as classifiers and not as constructors).

The idea is that *constructors are static data*, so that a constructor abstraction $\Lambda(u{::}\kappa.e)$ of type $\forall\, u :: \kappa.\tau$ is a mapping from static data $c$ of kind $\kappa$ to dynamic data $[c/u]e$ of type $[c/u]\tau$. Rule (22.2) tells us that every constructor of kind $\mathtt{T}$ determines a classifier, but it may or may not be the case that every classifier arises in this manner. If so, then the quantification is said to be *impredicative*, and is otherwise *predicative*.

## 22.3 Canonizing Substitution

Rule (22.5b) involves substitution of a canonical constructor, $c$, of kind $\kappa$ into a family of types $u \Uparrow \kappa \vdash \tau$ type. This operation is written $[c/u]\tau$, as usual. Although the intended meaning is clear, it is in fact impossible to interpret $[c/u]\tau$ as the standard concept of substitution defined in Chapter 1. The reason is that to do so would risk violating the distinction between neutral and canonical constructors. Consider, for example, the case of the family of types

$$u \Uparrow \mathtt{T} \to \mathtt{T} \vdash u[d] \Uparrow \mathtt{T},$$

where $d \Uparrow \mathtt{T}$. (It is not important what we choose for $d$, so we leave it abstract.) Now if $c \Downarrow \mathtt{T} \to \mathtt{T}$, then by Lemma 22.1 we have that $c$ is $\lambda\, u'.c'$. Thus, if interpreted conventionally, substitution of $c$ for $u$ in the given family yields the "constructor" $(\lambda\, u'.c')[d]$, which is not well-formed.

The solution is to define a form of *canonizing substitution* that simplifies such "illegal" combinations as it performs the replacement of a variable by a constructor of the same kind. In the case just sketched this means that we must ensure that

$$[\lambda\, u'.c'/u](u[d]) = [d/u']c'.$$

If viewed as a definition this equation is problematic because it switches from substituting for $u$ in the constructor $u[d]$ to substituting for $u'$ in the unrelated constructor $c'$. Why should such a process terminate? The answer lies in the observation that the kind of $u'$ is smaller than the kind of $u$, because the kind of the former is the domain kind of the latter. In all other cases of substitution (as we shall see shortly) the size of the target of the substitution becomes smaller; in the case just cited the size may increase, but the type of the target variable decreases. Therefore by a lexicographic induction on the type of the target variable and the structure of the target constructor, we may prove that canonizing substitution is properly defined.

We now turn to the task of making this precise. We will define simultaneously two principal forms of substitution, one of which divides into two cases:

| | |
|---|---|
| $[c/u:\kappa]a = a'$ | canonical into neutral yielding neutral |
| $[c/u:\kappa]a = c' \Downarrow \kappa'$ | canonical into neutral yielding canonical and kind |
| $[c/u:\kappa]c' = c''$ | canonical into canonical yielding canonical |

Preview

Substitution into a neutral constructor divides into two cases according to whether the substituted variable $u$ occurs in *critical position* in a sense to be made precise below.

These forms of substitution are simultaneously inductively defined by the following rules, which are broken into groups for clarity.

The first set of rules defines substitution of a canonical constructor into a canonical constructor; the result is always canonical.

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa]\widehat{a'} = \widehat{a''}} \tag{22.6a}$$

$$\frac{[c/u:\kappa]a' = c'' \Downarrow \kappa''}{[c/u:\kappa]\widehat{a'} = c''} \tag{22.6b}$$

$$\frac{}{[u/\langle\rangle:\kappa]=\langle\rangle} \tag{22.6c}$$

$$\frac{[c/u:\kappa]c_1' = c_1'' \quad [c/u:\kappa]c_2' = c_2''}{[c/u:\kappa]\langle c_1',c_2'\rangle = \langle c_1'',c_2''\rangle} \tag{22.6d}$$

$$\frac{[c/u:\kappa]c' = c'' \quad (u \neq u') \quad (u' \notin c)}{[c/u:\kappa](\lambda\, u'.c') = \lambda\, u'.c''} \tag{22.6e}$$

The conditions on variables in rule (22.6e) may always be met by renaming the bound variable, $u'$, of the abstraction.

The second set of rules defines substitution of a canonical constructor into a neutral constructor, yielding another neutral constructor.

$$\frac{(u \neq u')}{[c/u:\kappa]u' = u'} \tag{22.7a}$$

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa](a' \cdot \mathtt{l}) = a'' \cdot \mathtt{l}} \tag{22.7b}$$

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa](a' \cdot \mathtt{r}) = a'' \cdot \mathtt{r}} \tag{22.7c}$$

$$\frac{[c/u:\kappa]a_1 = a_1' \quad [c/u:\kappa]c_2 = c_2'}{[c/u:\kappa](a_1[c_2]) = a_1'(c_2')} \tag{22.7d}$$

Rule (22.7a) pertains to a *non-critical* variable, which is not the target of substitution. The remaining rules pertain to situations in which the recursive call on a neutral constructor yields a neutral constructor.

The third set of rules defines substitution of a canonical constructor into a neutral constructor, yielding a canonical constructor and its kind.

$$\frac{}{[c/u:\kappa]u = c \Downarrow \kappa} \tag{22.8a}$$

$$\frac{[c/u:\kappa]a' = \langle c_1',c_2'\rangle \Downarrow \kappa_1' \times \kappa_2'}{[c/u:\kappa](a' \cdot \mathtt{l}) = c_1' \Downarrow \kappa_1'} \tag{22.8b}$$

Preview

$$\frac{[c/u:\kappa]a' = \langle c_1', c_2' \rangle \Downarrow \kappa_1' \times \kappa_2'}{[c/u:\kappa](a' \cdot \mathtt{r}) = c_2' \Downarrow \kappa_2'} \tag{22.8c}$$

$$\frac{[c/u:\kappa]a_1' = \lambda\, u'.c' \Downarrow \kappa_2' \to \kappa' \quad [c/u:\kappa]c_2' = c_2'' \quad [c_2''/u':\kappa_2']c' = c''}{[c/u:\kappa](a_1'[c_2']) = c'' \Downarrow \kappa'} \tag{22.8d}$$

Rule (22.8a) governs a *critical* variable, which is the target of substitution. The substitution transforms it from a neutral constructor to a canonical constructor. This has a knock-on effect in the remaining rules of the group, which analyze the canonical form of the result of the recursive call to determine how to proceed. Rule (22.8d) is the most interesting rule. In the third premise, all three arguments to substitution change as we substitute the (substituted) argument of the application for the parameter of the (substituted) function into the body of that function. Here we require the type of the function in order to determine the type of its parameter.

**Theorem 22.2.** *Suppose that $\Delta \vdash c \Downarrow \kappa$, and $\Delta, u \Uparrow \kappa \vdash c' \Downarrow \kappa'$, and $\Delta, u \Uparrow \kappa \vdash a' \Uparrow \kappa'$. There exists a unique $\Delta \vdash c'' \Downarrow \kappa'$ such that $[c/u:\kappa]c' = c''$. Either there exists a unique $\Delta \vdash a'' \Uparrow \kappa'$ such that $[c/u:\kappa]a' = a''$, or there exists a unique $\Delta \vdash c'' \Downarrow \kappa'$ such that $[c/u:\kappa]a' = c''$, but not both.*

*Proof.* Simultaneously by a lexicographic induction with major index being the structure of the kind $\kappa$, and with minor index determined by the formation rules for $c'$ and $a'$. For all rules except rule (22.8d) the inductive hypothesis applies to the premise(s) of the relevant formation rules. For rule (22.8d) we appeal to the major inductive hypothesis applied to $\kappa_2'$, which is a component of the kind $\kappa_2' \to \kappa'$. □

## 22.4 Canonization

With canonizing substitution in hand, it is possible to confine our attention solely to constructors in canonical form. However, for some purposes it can be useful to admit a more relaxed syntax with non-canonical constructors that may be mechanically transformed into canonical form. The prototypical example is the constructor $(\lambda\, u.c_2)[c_1]$, which is malformed according to rules (22.1), because the first argument of an application is required to be in neutral form, whereas the $\lambda$-abstraction is in canonical form. However, if $c_1$ and $c_2$ are already canonical, then the malformed application may be transformed into canonical form using canonizing substitution of $c_1$ for $u$ in $c_2$.

A constructor in *general form* is one that is well-formed with respect to rules (22.1), but disregarding the distinction between neutral and canonical forms. We write $\Delta \vdash c :: \kappa$ to mean that $c$ is a well-formed constructor of kind $\kappa$ in general form. The difficulty with admitting general form constructors is that they introduce non-trivial equivalences between constructors. For example, we must ensure that $\langle \mathtt{int}, \mathtt{bool} \rangle \cdot \mathtt{l}$ is equivalent to $\mathtt{int}$. With this in mind we will introduce a *canonization* procedure that allows us to define equivalence of general form constructors, $\Delta \vdash c_1 \equiv c_2 :: \kappa$, to mean that $c_1$ and $c_2$ have identical canonical forms (up to $\alpha$-equivalence).

Canonization of general-form constructors is defined by these three judgments:

1. Canonization: $\Delta \vdash c :: \kappa \Downarrow \overline{c}$. Transform general-form constructor $c$ of kind $\kappa$ to canonical form $\overline{c}$.



Preview

2. Neutralization: $\Delta \vdash c \Uparrow \underline{c} :: \kappa$. Transform general-form constructor $c$ to obtain neutral form $\underline{c}$ of kind $\kappa$.

3. Head simplification: $\Delta \vdash c \twoheadrightarrow c'$. Simplify general-form constructor $c$ at the root, if possible.

The canonization judgment is used to determine the canonical form of a general-form constructor; the neutralization judgment is an auxiliary to the first that transforms constructors into neutral form. The canonization judgment is to be thought of as having mode $(\forall, \forall, \exists)$, whereas the neutralization judgment is to be thought of as having mode $(\forall, \exists, \exists)$.

The head simplification judgment makes no use of kinds, and is to be thought of as having mode $(\forall, \exists)$. It makes use of ordinary, rather than canonizing, substitution, and is defined by the following three axioms:

$$\frac{}{\Delta \vdash \langle c_1, c_2 \rangle \cdot \mathtt{l} \twoheadrightarrow c_1} \tag{22.9a}$$

$$\frac{}{\Delta \vdash \langle c_1, c_2 \rangle \cdot \mathtt{r} \twoheadrightarrow c_2} \tag{22.9b}$$

$$\frac{}{\Delta \vdash (\lambda\, u.c_1)[c_2] \twoheadrightarrow [c_2/u]c_1} \tag{22.9c}$$

It is easy to see that simplification preserves the kind of a constructor. By an argument similar to that used to justify the definition of canonizing substitution one may show that every well-kinded constructor has a unique simplified form, which is atomic if the kind is $\mathtt{T}$.

Canonization and neutralization are defined by the following rules:

$$\frac{\Delta \vdash a \twoheadrightarrow a' \quad \Delta \vdash a' \Uparrow \underline{a'} :: \mathtt{T}}{\Delta \vdash a \Uparrow \underline{a'} :: \mathtt{T}} \tag{22.10a}$$

$$\frac{\Delta \vdash a \Uparrow \underline{a} :: \mathtt{T}}{\Delta \vdash a :: \mathtt{T} \Downarrow \widehat{\underline{a}}} \tag{22.10b}$$

$$\frac{}{\Delta \vdash c :: 1 \Downarrow \langle\rangle} \tag{22.10c}$$

$$\frac{\Delta \vdash c \cdot \mathtt{l} :: \kappa_1 \Downarrow \overline{c_1} \quad \Delta \vdash c \cdot \mathtt{r} :: \kappa_2 \Downarrow \overline{c_2}}{\Delta \vdash c :: \kappa_1 \times \kappa_2 \Downarrow \langle \overline{c_1}, \overline{c_2} \rangle} \tag{22.10d}$$

$$\frac{\Delta, u :: \kappa_1 \vdash c[u] :: \kappa_2 \Downarrow \overline{c_2}}{\Delta \vdash c :: \kappa_1 \to \kappa_2 \Downarrow \lambda\, u.\overline{c_2}} \tag{22.10e}$$

$$\frac{}{\Delta, u :: \kappa \vdash u \Uparrow u :: \kappa} \tag{22.10f}$$

$$\frac{\Delta \vdash c \Uparrow \underline{c} :: \kappa_1 \times \kappa_2}{\Delta \vdash c \cdot \mathtt{l} \Uparrow \underline{c} \cdot \mathtt{l} :: \kappa_1} \tag{22.10g}$$

$$\frac{\Delta \vdash c \Uparrow \underline{c} :: \kappa_1 \times \kappa_2}{\Delta \vdash c \cdot \mathtt{r} \Uparrow \underline{c} \cdot \mathtt{r} :: \kappa_2} \tag{22.10h}$$

Preview

$$\frac{\Delta \vdash c_1 \Uparrow \underline{c_1} :: \kappa_1 \to \kappa_2 \quad \Delta \vdash c_2 :: \kappa_1 \Downarrow \overline{c_2}}{\Delta \vdash c_1[c_2] \Uparrow \underline{c_1}[\overline{c_2}] :: \kappa_2} \tag{22.10i}$$

The canonization judgment produces canonical forms, the neutralization judgment produces neutral forms, and the head normalization

**Lemma 22.3.** *1. If $\Delta \vdash c :: \kappa \Downarrow \overline{c}$, then $\Delta \vdash \overline{c} \Downarrow \kappa$.*

*2. If $\Delta \vdash c \Uparrow \underline{c} :: \kappa$, then $\Delta \vdash \underline{c} \Uparrow \kappa$.*

*Proof.* By induction on rules (22.10). □

**Theorem 22.4.** *If $\Gamma \vdash c :: \kappa$, then there exists $\overline{c}$ such that $\Delta \vdash c :: \kappa \Downarrow \overline{c}$.*

*Proof.* By induction on the formation rules for general-form constructors, making use of the fact that a general-form constructor of kind $\mathtt{T}$ must be in neutral form with components that are themselves general-form constructors. □

## 22.5 Notes

The classical approach is to consider general-form constructors at the outset, for which substitution is readily defined, and then to test equivalence of general-form constructors by reduction to a common irreducible form. Two main lemmas are required for this approach. First, every constructor must reduce in a finite number of steps to an irreducible form; this is called *normalization*. Second, the relation "has a common irreducible form" must be shown to be transitive; this is called *confluence*. Here we have turned the development on its head by considering only canonical constructors in the first place, then defining canonizing substitution as introduced by Watkins et al. (2008). It is then relatively easy to decide equivalence of general-form constructors by canonization of both sides of an equation.

Preview

Preview

# Part IX

# Subtyping

Preview

Preview

# Chapter 23

# Subtyping

A *subtype* relation is a pre-order (reflexive and transitive relation) on types that validates the *subsumption principle*:

> *if $\tau'$ is a subtype of $\tau$, then a value of type $\tau'$ may be provided whenever a value of type $\tau$ is required.*

The subsumption principle relaxes the strictures of a type system to permit values of one type to be treated as values of another.

Experience shows that the subsumption principle, although useful as a general guide, can be tricky to apply correctly in practice. The key to getting it right is the principle of introduction and elimination. To determine whether a candidate subtyping relationship is sensible, it suffices to consider whether every *introductory* form of the subtype can be safely manipulated by every *eliminatory* form of the supertype. A subtyping principle makes sense only if it passes this test; the proof of the type safety theorem for a given subtyping relation ensures that this is the case.

A good way to get a subtyping principle wrong is to think of a type merely as a set of values (generated by introductory forms), and to consider whether every value of the subtype can also be considered to be a value of the supertype. The intuition behind this approach is to think of subtyping as akin to the subset relation in ordinary mathematics. But, as we shall see, this can lead to serious errors, because it fails to take account of the eliminatory forms that are applicable to the supertype. It is not enough to think only of the introductory forms; subtyping is a matter of *behavior*, rather than *containment*.

## 23.1 Subsumption

A *subtyping judgment* has the form $\tau' <: \tau$, and states that $\tau'$ is a subtype of $\tau$. At a minimum we demand that the following *structural rules* of subtyping be admissible:

$$\frac{}{\tau <: \tau} \tag{23.1a}$$

Preview

$$\frac{\tau'' <: \tau' \quad \tau' <: \tau}{\tau'' <: \tau} \tag{23.1b}$$

In practice we either tacitly include these rules as primitive, or prove that they are admissible for a given set of subtyping rules.

The point of a subtyping relation is to enlarge the set of well-typed programs, which is achieved by the *subsumption rule*:

$$\frac{\Gamma \vdash e : \tau' \quad \tau' <: \tau}{\Gamma \vdash e : \tau} \tag{23.2}$$

In contrast to most other typing rules, the rule of subsumption is *not* syntax-directed, because it does not constrain the form of $e$. That is, the subsumption rule may be applied to *any* form of expression. In particular, to show that $e : \tau$, we have two choices: either apply the rule appropriate to the particular form of $e$, or apply the subsumption rule, checking that $e : \tau'$ and $\tau' <: \tau$.

## 23.2 Varieties of Subtyping

In this section we will informally explore several different forms of subtyping in the context of extensions of the language **FPC** introduced in Chapter 16.

### 23.2.1 Numeric Types

We begin with an informal discusson of numeric types such as are common in many programming languages. Our mathematical experience suggests subtyping relationships among numeric types. For example, in a language with types `int`, `rat`, and `real`, representing, respectively, the integers, the rationals, and the reals, it is tempting to postulate the subtyping relationships

$$\texttt{int} <: \texttt{rat} <: \texttt{real}$$

by analogy with the set containments

$$\mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}.$$

But are these subtyping relationships sensible? The answer depends on the representations and interpretations of these types. Even in mathematics, the containments just mentioned are usually not quite true—or are true only in an approximate sense. For example, the set of rational numbers may be considered to consist of ordered pairs $(m, n)$, with $n \neq 0$ and $\gcd(m, n) = 1$, representing the ratio $m/n$. The set $\mathbb{Z}$ of integers may be isomorphically embedded within $\mathbb{Q}$ by identifying $n \in \mathbb{Z}$ with the ratio $n/1$. Similarly, the real numbers are often represented as convergent sequences of rationals, so that strictly speaking the rationals are not a subset of the reals, but rather may be embedded in them by choosing a canonical representative (a particular convergent sequence) of each rational.

For mathematical purposes it is entirely reasonable to overlook fine distinctions such as that between $\mathbb{Z}$ and its embedding within $\mathbb{Q}$. This is justified because the operations on rationals restrict to the embedding in the expected manner: if we add two integers thought of as rationals in the canonical way, then the result is the rational associated with their sum. And similarly for the other operations, provided that we take some care in defining them to ensure that it all works

Preview

out properly. For the purposes of computing, however, we cannot be quite so cavalier, because we must also take account of algorithmic efficiency and the finiteness of machine representations. For example, what are often called "real numbers" in a programming language are, of course, finite precision floating point numbers, a finite subset of the rational numbers. Not every rational can be exactly represented as a floating point number, nor does floating point arithmetic restrict to rational arithmetic, even when its arguments are exactly represented as floating point numbers.

### 23.2.2 Product Types

Product types give rise to a form of subtyping based on the subsumption principle. The only elimination form applicable to a value of product type is a projection. Under mild assumptions about the dynamics of projections, we may consider one product type to be a subtype of another by considering whether the projections applicable to the supertype may be validly applied to values of the subtype.

Consider a context in which a value of type $\tau = \langle \tau_j \rangle_{j \in J}$ is required. The statics of finite products (rules (10.3)) ensures that the only operation we may perform on a value of type $\tau$, other than to bind it to a variable, is to take the $j$th projection from it for some $j \in J$ to obtain a value of type $\tau_j$. Now suppose that $e$ is of type $\tau'$. If the projection $e \cdot j$ is to be well-formed, then $\tau'$ must be a finite product type $\langle \tau'_i \rangle_{i \in I}$ such that $j \in I$. Moreover, for this to be of type $\tau_j$, it is enough to require that $\tau'_j = \tau_j$. Because $j \in J$ is arbitrary, we arrive at the following subtyping rule for finite product types:

$$\frac{J \subseteq I}{\prod_{i \in I} \tau_i <: \prod_{j \in J} \tau_j}. \tag{23.3}$$

This rule is sufficient for the required subtyping, but not necessary; we will consider a more liberal form of this rule in Section 23.3. The justification for rule (23.3) is that we may evaluate $e \cdot i$ regardless of the actual form of $e$, provided only that it has a field indexed by $i \in I$.

### 23.2.3 Sum Types

By an argument dual to the one given for finite product types we may derive a related subtyping rule for finite sum types. If a value of type $\sum_{j \in J} \tau_j$ is required, the statics of sums (rules (11.3)) ensures that the only non-trivial operation that we may perform on that value is a $J$-indexed case analysis. If we provide a value of type $\sum_{i \in I} \tau'_i$ instead, no difficulty will arise so long as $I \subseteq J$ and each $\tau'_i$ is equal to $\tau_i$. If the containment is strict, some cases cannot arise, but this does not disrupt safety. This leads to the following subtyping rule for finite sums:

$$\frac{I \subseteq J}{\sum_{i \in I} \tau_i <: \sum_{j \in J} \tau_j}. \tag{23.4}$$

Note well the reversal of the containment as compared to rule (23.3).



Preview

### 23.2.4 Dynamic Types

A popular form of subtyping arises in connection with the type `dyn` introduced in Chapter 19. The type `dyn` provides no information about the class of a value of this type. One might argue that it is precisely the point of dynamic typing to suppress this information statically (at compile time), making it available only dynamically (at execution time). On the other hand, it is not much trouble to introduce subtypes of `dyn` that specify the class of a value, relying on subsumption to "forget" the class when it cannot be accurately determined statically.

Working in the context of Chapter 19 this amounts to introduce two new types, `dyn[num]` and `dyn[fun]`, governed by the following two subtyping axioms:

$$\frac{}{\texttt{dyn[num]} <: \texttt{dyn}} \tag{23.5a}$$

$$\frac{}{\texttt{dyn[fun]} <: \texttt{dyn}} \tag{23.5b}$$

Of course, in a richer language with more classes of dynamic values one would correspondingly introduce more such subtypes of `dyn`, one for each additional class. As a matter of notation, the type `dyn` is frequently spelled `object`, and its class-specific subtypes `dyn[num]` and `dyn[fun]`, are often written as `num` and `fun`, respectively. But doing so invites confusion between the separate concepts of *class* and *type*, as discussed in detail in Chapters 18 and 19.

The class-specific subtypes of `dyn` come into play by reformulating the typing rules for introducing values of type `dyn` to note the class of the created value:

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{new[num]}(e) : \texttt{dyn[num]}} \tag{23.6a}$$

$$\frac{\Gamma \vdash e : \texttt{dyn} \rightharpoonup \texttt{dyn}}{\Gamma \vdash \texttt{new[fun]}(e) : \texttt{dyn[fun]}} \tag{23.6b}$$

Thus, in this formulation, classified values "start life" with class-specific types, because in those cases it is statically apparent what is the class of the introduced value. Subsumption is used to weaken the type to `dyn` in those cases where no accurate static prediction can be made—for example, when the branches of a conditional evaluate to dynamic values of different classes it is necessary to weaken the type of the branches to `dyn`.

The advantage of such a subtyping mechanism is that we can express more precise types, such as the type $\texttt{dyn[num]} \rightharpoonup \texttt{dyn[num]}$ of functions mapping a value of type `dyn` with class `num` to another such value. This is a more precise typing than, say, $\texttt{dyn} \rightharpoonup \texttt{dyn}$, which merely classifies functions that act on dynamically typed values. In this manner weak invariants can be expressed and enforced, but only insofar as it is possible to track the classes of the values involved in a computation. Subtyping is not nearly a powerful enough mechanism for practical situations, rendering the additional specificity not worth the effort of including it. (A more useful approach is discussed in Chapter 24.)

Preview

## 23.3 Variance

In addition to basic subtyping principles such as those considered in Section 23.2, it is also important to consider the effect of subtyping on type constructors. A type constructor is said to be *covariant* in an argument if subtyping in that argument is preserved by the constructor. It is said to be *contravariant* if subtyping in that argument is reversed by the constructor. It is said to be *invariant* in an argument if subtyping for the constructed type is not affected by subtyping in that argument.

### 23.3.1 Product and Sum Types

Finite product types are *covariant* in each field. For if $e$ is of type $\prod_{i \in I} \tau_i'$, and the projection $e \cdot j$ is expected to be of type $\tau_j$, then it is sufficient to require that $j \in I$ and $\tau_j' <: \tau_j$. This is summarized by the following rule:

$$\frac{(\forall i \in I)\ \tau_i' <: \tau_i}{\prod_{i \in I} \tau_i' <: \prod_{i \in I} \tau_i} \tag{23.7}$$

It is implicit in this rule that the dynamics of projection must not be sensitive to the precise type of any of the fields of a value of finite product type.

Finite sum types are also covariant, because each branch of a case analysis on a value of the supertype expects a value of the corresponding summand, for which it is sufficient to provide a value of the corresponding subtype summand:

$$\frac{(\forall i \in I)\ \tau_i' <: \tau_i}{\sum_{i \in I} \tau_i' <: \sum_{i \in I} \tau_i} \tag{23.8}$$

### 23.3.2 Partial Function Types

The variance of the function type constructors is a bit more subtle. We will work here with the total function types, but the same considerations apply to the partial function type. Let us consider first the variance of the function type in its range. Suppose that $e : \tau_1 \rightharpoonup \tau_2'$. This means that if $e_1 : \tau_1$, then $e(e_1) : \tau_2'$. If $\tau_2' <: \tau_2$, then $e(e_1) : \tau_2$ as well. This suggests the following covariance principle for function types:

$$\frac{\tau_2' <: \tau_2}{\tau_1 \rightharpoonup \tau_2' <: \tau_1 \rightharpoonup \tau_2} \tag{23.9}$$

Every function that delivers a value of type $\tau_2'$ also delivers a value of type $\tau_2$, provided that $\tau_2' <: \tau_2$. Thus the function type constructor is covariant in its range.

Now let us consider the variance of the function type in its domain. Suppose again that $e : \tau_1 \rightharpoonup \tau_2$. This means that $e$ may be applied to any value of type $\tau_1$ to obtain a value of type $\tau_2$. Hence, by the subsumption principle, it may be applied to any value of a subtype, $\tau_1'$, of $\tau_1$, and it will still deliver a value of type $\tau_2$. Consequently, we may just as well think of $e$ as having type $\tau_1' \rightharpoonup \tau_2$.

$$\frac{\tau_1' <: \tau_1}{\tau_1 \rightharpoonup \tau_2 <: \tau_1' \rightharpoonup \tau_2} \tag{23.10}$$



Preview

The function type is contravariant in its domain position. Note well the reversal of the subtyping relation in the premise as compared to the conclusion of the rule!

Combining these rules we obtain the following general principle of contra- and co-variance for function types:

$$\frac{\tau_1' <: \tau_1 \quad \tau_2' <: \tau_2}{\tau_1 \rightharpoonup \tau_2' <: \tau_1' \rightharpoonup \tau_2} \tag{23.11}$$

Beware of the reversal of the ordering in the domain!

### 23.3.3 Recursive Types

The language **FPC** has a partial function types, which behave the same under subtyping as total function types, sums and products, which behave as described above, and recursive types, which introduce some subtleties that have been the source of error in language design. To gain some intuition, consider the type of labeled binary trees with natural numbers at each node,

$$\mathtt{rec}\, t\, \mathtt{is}\, [\mathtt{empty} \hookrightarrow \mathtt{unit}, \mathtt{binode} \hookrightarrow \langle \mathtt{data} \hookrightarrow \mathtt{nat}, \mathtt{lft} \hookrightarrow t, \mathtt{rht} \hookrightarrow t \rangle],$$

and the type of "bare" binary trees, without data attached to the nodes,

$$\mathtt{rec}\, t\, \mathtt{is}\, [\mathtt{empty} \hookrightarrow \mathtt{unit}, \mathtt{binode} \hookrightarrow \langle \mathtt{lft} \hookrightarrow t, \mathtt{rht} \hookrightarrow t \rangle].$$

Is either a subtype of the other? Intuitively, we might expect the type of labeled binary trees to be a *subtype* of the type of bare binary trees, because any use of a bare binary tree can simply ignore the presence of the label.

Now consider the type of bare "two-three" trees with two sorts of nodes, those with two children, and those with three:

$$\mathtt{rec}\, t\, \mathtt{is}\, [\mathtt{empty} \hookrightarrow \mathtt{unit}, \mathtt{binode} \hookrightarrow \tau_2, \mathtt{trinode} \hookrightarrow \tau_3],$$

where

$$\tau_2 \triangleq \langle \mathtt{lft} \hookrightarrow t, \mathtt{rht} \hookrightarrow t \rangle, \text{and}$$
$$\tau_3 \triangleq \langle \mathtt{lft} \hookrightarrow t, \mathtt{mid} \hookrightarrow t, \mathtt{rht} \hookrightarrow t \rangle.$$

What subtype relationships should hold between this type and the preceding two tree types? Intuitively the type of bare two-three trees should be a *supertype* of the type of bare binary trees, because any use of a two-three tree must proceed by three-way case analysis, which covers both forms of binary tree.

To capture the pattern illustrated by these examples, we must formulate a subtyping rule for recursive types. It is tempting to consider the following rule:

$$\frac{t\ \mathsf{type} \vdash \tau' <: \tau}{\mathtt{rec}\, t\, \mathtt{is}\, \tau' <: \mathtt{rec}\, t\, \mathtt{is}\, \tau}\ \textbf{??} \tag{23.12}$$



Preview

That is, to determine whether one recursive type is a subtype of the other, we simply compare their bodies, with the bound variable treated as a parameter. Notice that by reflexivity of subtyping, we have $t <: t$, and hence we may use this fact in the derivation of $\tau' <: \tau$.

Rule (23.12) validates the intuitively plausible subtyping between labeled binary tree and bare binary trees just described. Deriving this requires checking that the subtyping relationship

$$\langle \mathtt{data} \hookrightarrow \mathtt{nat}, \mathtt{lft} \hookrightarrow t, \mathtt{rht} \hookrightarrow t \rangle <: \langle \mathtt{lft} \hookrightarrow t, \mathtt{rht} \hookrightarrow t \rangle,$$

holds generically in $t$, which is evidently the case.

Unfortunately, Rule (23.12) also underwrites *incorrect* subtyping relationships, as well as some correct ones. As an example of what goes wrong, consider the recursive types

$$\tau' = \mathtt{rec}\, t\, \mathtt{is}\, \langle \mathtt{a} \hookrightarrow t \rightharpoonup \mathtt{nat}, \mathtt{b} \hookrightarrow t \rightharpoonup \mathtt{int} \rangle$$

and

$$\tau = \mathtt{rec}\, t\, \mathtt{is}\, \langle \mathtt{a} \hookrightarrow t \rightharpoonup \mathtt{int}, \mathtt{b} \hookrightarrow t \rightharpoonup \mathtt{int} \rangle.$$

We assume for the sake of the example that $\mathtt{nat} <: \mathtt{int}$, so that by using rule (23.12) we may derive $\tau' <: \tau$, which we will show to be incorrect. Let $e : \tau'$ be the expression

$$\mathtt{fold}(\langle \mathtt{a} \hookrightarrow \lambda\,(x{:}\tau')\, 4, \mathtt{b} \hookrightarrow \lambda\,(x{:}\tau')\, q((\mathtt{unfold}(x) \cdot \mathtt{a})(x)) \rangle),$$

where $q : \mathtt{nat} \rightharpoonup \mathtt{nat}$ is the discrete square root function. Because $\tau' <: \tau$, it follows that $e : \tau$ as well, and hence

$$\mathtt{unfold}(e) : \langle \mathtt{a} \hookrightarrow \tau \rightharpoonup \mathtt{int}, \mathtt{b} \hookrightarrow \tau \rightharpoonup \mathtt{int} \rangle.$$

Now let $e' : \tau$ be the expression

$$\mathtt{fold}(\langle \mathtt{a} \hookrightarrow \lambda\,(x{:}\tau)\, \mathtt{-4}, \mathtt{b} \hookrightarrow \lambda\,(x{:}\tau)\, 0 \rangle).$$

(The important point about $e'$ is that the `a` method returns a negative number; the `b` method is of no significance.) To finish the proof, observe that

$$(\mathtt{unfold}(e) \cdot \mathtt{b})(e') \longmapsto^* q(\mathtt{-4}),$$

which is a stuck state. We have derived a well-typed program that "gets stuck", refuting type safety!

Rule (23.12) is therefore incorrect. But what has gone wrong? The error lies in the choice of a single parameter to stand for both recursive types, which does not correctly model self-reference. In effect we are regarding two distinct recursive types as equal while checking their bodies for a subtyping relationship. But this is clearly wrong! It fails to take account of the self-referential nature of recursive types. On the left side the bound variable stands for the subtype, whereas on the right the bound variable stands for the super-type. Confusing them leads to the unsoundness just illustrated.

As is often the case with self-reference, the solution is to *assume* what we are trying to prove, and check that this assumption can be maintained by examining the bodies of the recursive types. To do so we make use of hypothetical judgments of the form $\Delta \vdash \tau' <: \tau$, where $\Delta$ consists of

Preview

hypotheses $t$ type and $t <: \tau$ that declate a fresh type variable $t$ that is not otherwise declared in $\Delta$. Using such hypothetical judgments we may state the correct rule for subtyping recursive types as follows:

$$\frac{\Delta, t \text{ type}, t' \text{ type}, t' <: t \vdash \tau' <: \tau \quad \Delta, t' \text{ type} \vdash \tau' \text{ type} \quad \Delta, t \text{ type} \vdash \tau \text{ type}}{\Delta \vdash \mathtt{rec}\, t' \,\mathtt{is}\, \tau' <: \mathtt{rec}\, t \,\mathtt{is}\, \tau} . \tag{23.13}$$

That is, to check whether $\mathtt{rec}\, t' \,\mathtt{is}\, \tau' <: \mathtt{rec}\, t \,\mathtt{is}\, \tau$, we assume that $t' <: t$, because $t'$ and $t$ stand for the respective recursive types, and check that $\tau' <: \tau$ under this assumption. It is instructive to check that the unsound subtyping example given above is not derivable using this rule: the subtyping assumption is at odds with the contravariance of the function type in its domain.

### 23.3.4 Quantified Types

Consider the extension of **FPC** with the universal and existential quantified types discussed in Chapters 20 and 21. The variance principles for the quantifiers state that they are uniformly covariant in the quantified types:

$$\frac{\Delta, t \text{ type} \vdash \tau' <: \tau}{\Delta \vdash \forall(t.\tau') <: \forall(t.\tau)} \tag{23.14a}$$

$$\frac{\Delta, t \text{ type} \vdash \tau' <: \tau}{\Delta \vdash \exists(t.\tau') <: \exists(t.\tau)} \tag{23.14b}$$

Consequently, we may derive the principle of substitution:

**Lemma 23.1.** *If $\Delta, t$ type $\vdash \tau_1 <: \tau_2$, and $\Delta \vdash \tau$ type, then $\Delta \vdash [\tau/t]\tau_1 <: [\tau/t]\tau_2$.*

*Proof.* By induction on the subtyping derivation. □

It is easy to check that the above variance principles for the quantifiers are consistent with the principle of subsumption. For example, a package of the subtype $\exists(t.\tau')$ consists of a representation type, $\rho$, and an implementation, $e$, of type $[\rho/t]\tau'$. But if $t$ type $\vdash \tau' <: \tau$, we have by substitution that $[\rho/t]\tau' <: [\rho/t]\tau$, and hence $e$ is also an implementation of type $[\rho/t]\tau$. This is sufficient to ensure that the package is also of the supertype.

It is natural to extend subtyping to the quantifiers by allowing quantification over all subtypes of a specified type; this is called *bounded quantification*.

$$\frac{}{\Delta, t \text{ type}, t <: \tau \vdash t <: \tau} \tag{23.15a}$$

$$\frac{\Delta \vdash \tau :: \mathtt{T}}{\Delta \vdash \tau <: \tau} \tag{23.15b}$$

$$\frac{\Delta \vdash \tau'' <: \tau' \quad \Delta \vdash \tau' <: \tau}{\Delta \vdash \tau'' <: \tau} \tag{23.15c}$$

$$\frac{\Delta \vdash \tau_1' <: \tau_1 \quad \Delta, t \text{ type}, t <: \tau_1' \vdash \tau_2 <: \tau_2'}{\Delta \vdash \forall\, t <: \tau_1.\tau_2 <: \forall\, t <: \tau_1'.\tau_2'} \tag{23.15d}$$



Preview

$$\frac{\Delta \vdash \tau_1 <: \tau_1' \quad \Delta, t \text{ type}, t <: \tau_1 \vdash \tau_2 <: \tau_2'}{\Delta \vdash \exists\, t <: \tau_1.\tau_2 <: \exists\, t <: \tau_1'.\tau_2'} \tag{23.15e}$$

Rule (23.15d) states that the universal quantifier is contravariant in its bound, whereas rule (23.15e) states that the existential quantifier is covariant in its bound.

## 23.4 Dynamics and Safety

There is a subtle assumption in the definition of product subtyping in Section 23.2.2, namely that the *same* projection operation from an $I$-tuple applies also to a $J$-tuple, provided $J \supseteq I$. But this need not be the case. One could, for example, represent $I$-tuples differently from $J$-tuples at will, so that the meaning of the projection at position $i \in I \subseteq J$ is different in the two cases. Nothing rules out this possibility, yet product subtyping relies on it not being the case. From this point of view product subtyping is not well-justified, but one may instead consider that subtyping limits possible implementations to ensure that it makes sense.

Similar considerations apply to sum types. An $J$-way case analysis need not be applicable to an $I$-way value of sum type, even when $I \subseteq J$ and all the types in common agree. For example, one might represent values of a sum type with a "small" index set to be represented in a way that is not applicable for a "large" index set. In that case the "large" case analysis would not make sense on a value of "small" sum type. Here again we may consider either that subtyping is not justified, or that it imposes limitations on the implementation that are not otherwise forced.

These considerations merit careful consideration of the safety of languages with subtyping. As an illustrative case we consider the safety of **FPC** enriched with product subtyping. The main concern is that the subsumption rule obscures the "true" type of a value, complicating the canonical forms lemma. Moreover, we assume that the same projection makes sense for a wider tuple than a narrower one, provided that it is within range.

**Lemma 23.2** (Structurality)**.**

1. *The tuple subtyping relation is reflexive and transitive.*
2. *The typing judgment $\Gamma \vdash e : \tau$ is closed under weakening and substitution.*

*Proof.*

1. Reflexivity is proved by induction on the structure of types. Transitivity is proved by induction on the derivations of the judgments $\tau'' <: \tau'$ and $\tau' <: \tau$ to obtain a derivation of $\tau'' <: \tau$.
2. By induction on rules (10.3), augmented by rule (23.2).

□

**Lemma 23.3** (Inversion)**.**

1. *If $e \cdot j : \tau$, then $e : \prod_{i \in I} \tau_i$, $j \in I$, and $\tau_j <: \tau$.*



Preview

2. *If* $\langle e_i \rangle_{i \in I} : \tau$, *then* $\prod_{i \in I} \tau'_i <: \tau$ *where* $e_i : \tau'_i$ *for each* $i \in I$.

3. *If* $\tau' <: \prod_{j \in J} \tau_j$, *then* $\tau' = \prod_{i \in I} \tau'_i$ *for some* $I$ *and some types* $\tau'_i$ *for* $i \in I$.

4. *If* $\prod_{i \in I} \tau'_i <: \prod_{j \in J} \tau_j$, *then* $J \subseteq I$ *and* $\tau'_j <: \tau_j$ *for each* $j \in J$.

*Proof.* By induction on the subtyping and typing rules, paying special attention to rule (23.2). □

**Theorem 23.4** (Preservation)**.** *If* $e : \tau$ *and* $e \longmapsto e'$, *then* $e' : \tau$.

*Proof.* By induction on rules (10.4). For example, consider rule (10.4d), so that $e = \langle e_i \rangle_{i \in I} \cdot k$ and $e' = e_k$. By Lemma 23.3 we have $\langle e_i \rangle_{i \in I} : \prod_{j \in J} \tau_j$, with $k \in J$ and $\tau_k <: \tau$. By another application of Lemma 23.3 for each $i \in I$ there exists $\tau'_i$ such that $e_i : \tau'_i$ and $\prod_{i \in I} \tau'_i <: \prod_{j \in J} \tau_j$. By Lemma 23.3 again, we have $J \subseteq I$ and $\tau'_j <: \tau_j$ for each $j \in J$. But then $e_k : \tau_k$, as desired. The remaining cases are similar. □

**Lemma 23.5** (Canonical Forms)**.** *If* $e$ *val and* $e : \prod_{j \in J} \tau_j$, *then* $e$ *is of the form* $\langle e_i \rangle_{i \in I}$, *where* $J \subseteq I$, *and* $e_j : \tau_j$ *for each* $j \in J$.

*Proof.* By induction on rules (10.3) augmented by rule (23.2). □

**Theorem 23.6** (Progress)**.** *If* $e : \tau$, *then either* $e$ *val or there exists* $e'$ *such that* $e \longmapsto e'$.

*Proof.* By induction on rules (10.3) augmented by rule (23.2). The rule of subsumption is handled by appeal to the inductive hypothesis on the premise of the rule. rule (10.4d) follows from Lemma 23.5. □

## 23.5 Notes

Subtyping is perhaps the most widely misunderstood concept in programming languages. Subtyping is principally a convenience, akin to type inference, that makes some programs simpler to write. But the subsumption rule cuts both ways. Inasmuch as it allows the implicit passage from $\tau'$ to $\tau$ whenever $\tau'$ is a subtype of $\tau$, it also weakens the meaning of a type assertion $e : \tau$ to mean that $e$ has some type contained in the type $\tau$. This precludes expressing the requirement that $e$ has *exactly* the type $\tau$, or that two expressions jointly have the *same* type. And it is precisely this weakness that creates so many of the difficulties with subtyping.

Much has been written about subtyping, often in connection with object-oriented programming. The functional language Standard ML (Milner et al., 1997) is one of the earliest full-scale languages to make use of subtyping, in two forms, called *enrichment* and *realization*. The former corresponds to product subtyping, and the latter to the "forgetful" subtyping associated with type definitions (see Chapter 25). The first systematic studies of subtyping include those by Mitchell (1984); Reynolds (1980), and Cardelli (1988). Pierce (2002) give a thorough account of subtyping, especially of recursive and polymorphic types, and proves that subtyping for bounded impredicative universal quantification is undecidable.

Preview

## Exercises

**23.1**. Examine the variance of the type

$$(\mathtt{unit} \rightharpoonup \tau) \times (\tau \rightharpoonup \mathtt{unit}).$$

When viewed as a constructor with argument $\tau$, is it covariant or contravariant? Give a precise proof or counterexample in each case.

**23.2**. Consider the two recursive types,

$$\rho_1 \triangleq \mathtt{rec}\, t\, \mathtt{is}\, \langle \mathtt{eq} \hookrightarrow (t \rightharpoonup \mathtt{bool}) \rangle,$$

and

$$\rho_2 \triangleq \mathtt{rec}\, t\, \mathtt{is}\, \langle \mathtt{eq} \hookrightarrow (t \rightharpoonup \mathtt{bool}), \mathtt{f} \hookrightarrow \mathtt{bool} \rangle.$$

It is clear that $\rho_1$ could not be a subtype of $\rho_2$, because, viewed as a product after unrolling, a value of the former type lacks a component that a value of the latter has. But is $\rho_2$ a subtype of $\rho_1$? If so, demonstrate it by exhibiting a derivation of this fact using the rules given in Section 23.3.3. If not, give a counterexample showing that the suggested subtyping is erroneous in that it would violate type safety.

**23.3**. Another approach to the dynamics of subtyping that ensures safety, but gives subsumption dynamic significance, associates a witness, called a *coercion*, to each subtyping relation, and inserts a coercion wherever subsumption is used. More precisely,

(a) Assign to each valid subtyping $\tau <: \tau'$ a coercion function $\chi : \tau \rightharpoonup \tau'$ that transforms a value of type $\tau$ into a value of type $\tau'$.

(b) Interpret the subsumption rule as implicit coercion. Specifically, whenever $\tau <: \tau'$ is witnessed by $\chi : \tau \rightharpoonup \tau'$, applying subsumption to $e : \tau$ inserts an application of $\chi$ to obtain $\chi(e) : \tau'$.

Formulate this idea precisely for the case of a subtype relation generated by "width" subtyping for products, and the variance principles for product, sum and function types. Your solution should make clear that it evades the tacit projection assumption mentioned above.

A fly in the ointment is the possibity of there being more than one coercion $\chi : \tau \rightharpoonup \tau'$ corresponding to the subtyping $\tau <: \tau'$. The meaning of a program would then depend on which coercion is chosen when subsumption is used. The property of there being exactly one coercion for each subtyping relation is called *coherence of subtyping*. Is your coercion interpretation of product subtyping coherent? (A proper treatment of this issue requires a full treatment of expression equivalence, which is discussed in Chapter 49.)

Preview

Preview

# Chapter 24

# Type Refinements

In Chapter 19 we demonstrated that dynamic typing is but a mode of use of static typing, one in which dynamically typed values are of type `dyn`, a particular recursive sum type. A value of type `dyn` is always of the form $\texttt{new}[c](e)$, where $c$ is its *class* and $e$ is its underlying value. Importantly, the class, $c$, determines the type of the underlying value of a dynamic value. The type system of the hybrid language is rather weak in that every dynamically classified value has the same type, and there is no indication in the type of the class of the value. To rectify this it is common to enrich the type system of the hybrid language so as to capture such information, for example as described in Section 23.2.4.

In such a situation subtyping may be seen as an attempt to resolve a fundamental tension between *structure* and *behavior* in the design of type systems. On the one hand types determine the structure of a programming language, and on the other serve a behavioral specifications of expressions written in that language. Subtyping attempts to resolve this tension, unsuccessfully, by allowing certain forms of retyping. Although subtyping works reasonably well for small examples, things get far more complicated when we wish to, for example, specify the deep structure of a value, say that it is of a class $c$ and its underlying value is of another class $d$ whose underlying value is a natural number. There is no limit to the degree of specificity one may desire in such descriptions, which gives rise to endless variations on type systems to accommodate various special situations.

Another resolution of the tension between structure and behavior in typing is to separate, rather than consolidate, these aspects by distinguishing *types* from *type refinements*. Type refinements specify the execution behavior of an expression of a particular type using specifications that capture whatever properties may be of interest, limited only by the difficulty of proving that a program satisfies the specification given by a refinement.

Certain limited forms of behavioral specifications can express many useful properties of programs while remaining mechanically checkable. These include the fundamental behavioral properties determined by the type itself, but may be extended to include sharper conditions than just these structural properties. In this chapter we will consider a particular notion of refinement tailored to the hybrid language of Chapter 19. It is based on two basic principles:

PREVIEW

1. Type constructors, such as product, sum, and function space, act on refinements of their component types to induce a refinement on the compound type formed by those constructors.
2. It is useful to track the class of a value of type `dyn`, and to assign multiple refinements to specify distinct behaviors of expressions involving values of dynamic type.

We will formulate a system of refinements based on these principles that ensures that a well-refined program cannot incur a run-time error arising from the attempt to cast a value to a class other than its own.

## 24.1 Statics

We will develop a system of refinements for the extension with sums and products of the language **HPCF** defined in Chapter 19 in which there are but two classes, $c$, of values of type `dyn`, namely `num` and `fun`.[1] The syntax of refinements, $\phi$, is given by the following grammar:

| Ref | $\phi$ | ::= | | | |
|---|---|---|---|---|---|
| | | | $\mathtt{true}[\tau]$ | $\top_\tau$ | truth |
| | | | $\mathtt{and}[\tau](\phi_1;\phi_2)$ | $\phi_1 \wedge_\tau \phi_2$ | conjunction |
| | | | $\mathtt{new}[\mathtt{num}](\phi)$ | $\mathtt{num}\,!\,\phi$ | dynamic number |
| | | | $\mathtt{new}[\mathtt{fun}](\phi)$ | $\mathtt{fun}\,!\,\phi$ | dynamic function |
| | | | $\mathtt{prod}(\phi_1;\phi_2)$ | $\phi_1 \times \phi_2$ | product |
| | | | $\mathtt{sum}(\phi_1;\phi_2)$ | $\phi_1 + \phi_2$ | sum |
| | | | $\mathtt{parr}(\phi_1;\phi_2)$ | $\phi_1 \rightharpoonup \phi_2$ | function |

Informally, a refinement is a *predicate* specifying a property of the values of some type. Equivalently, one may think of a refinement as a *subset* of the values of a type, those that satisfy the specified property. To make the dependence of refinements on types explicit, the syntax of truth and conjunction is parameterized by the type whose values they govern. In most cases the underlying type is clear from context, in which case it is omitted from the notation for the sake of concision. Note carefully that the syntax of the product, sum, and function refinements is exactly the same as the syntax of the types they govern, but they are refinements, not types.

The judgment $\phi \sqsubseteq \tau$ means that $\phi$ is a refinement of the type $\tau$. It is defined by the following rules:

$$\frac{}{\top \sqsubseteq \tau} \tag{24.1a}$$

$$\frac{\phi_1 \sqsubseteq \tau \quad \phi_2 \sqsubseteq \tau}{\phi_1 \wedge \phi_2 \sqsubseteq \tau} \tag{24.1b}$$

$$\frac{\phi \sqsubseteq \mathtt{nat}}{\mathtt{num}\,!\,\phi \sqsubseteq \mathtt{dyn}} \tag{24.1c}$$

$$\frac{\phi \sqsubseteq \mathtt{dyn} \rightharpoonup \mathtt{dyn}}{\mathtt{fun}\,!\,\phi \sqsubseteq \mathtt{dyn}} \tag{24.1d}$$

[1] Of course, in a richer language there would be more classes than just these two, each with an associated type of the underlying data that it classifies.

Preview

$$\frac{\phi_1 \sqsubseteq \tau_1 \quad \phi_2 \sqsubseteq \tau_2}{\phi_1 \times \phi_2 \sqsubseteq \tau_1 \times \tau_2} \tag{24.1e}$$

$$\frac{\phi_1 \sqsubseteq \tau_1 \quad \phi_2 \sqsubseteq \tau_2}{\phi_1 + \phi_2 \sqsubseteq \tau_1 + \tau_2} \tag{24.1f}$$

$$\frac{\phi_1 \sqsubseteq \tau_1 \quad \phi_2 \sqsubseteq \tau_2}{\phi_1 \rightharpoonup \phi_2 \sqsubseteq \tau_1 \rightharpoonup \tau_2} \tag{24.1g}$$

It is easy to see that each refinement refines a unique type, the *underlying type* of that refinement. The concrete syntax $\texttt{num}\,!\,\phi$ and $\texttt{fun}\,!\,\phi$ is both concise and commonplace, but, beware, it tends to obscure the critical distinctions between types, classes, and refinements.

We will shortly give a definition of the *refinement satisfaction* judgment, $e \in_\tau \phi$, where $e : \tau$ and $\phi \sqsubseteq \tau$, which states that the well-typed expression $e$ exhibits the behavior specified by $\phi$. The hypothetical form,

$$x_1 \in_{\tau_1} \phi_1, \ldots, x_n \in_{\tau_n} \phi_n \vdash e \in_\tau \phi,$$

constrains the expressions that may be substituted for a variable to satisfy its associated refinement type (which may be the trival refinement, $\top$, that imposes no constraints). We write $\Phi$ for such a finite sequence of refinement assumptions on variables, called a *refinement context*. Each such $\Phi$ determines a typing context $\Gamma$ given by $x_1 : \tau_1, \ldots, x_n : \tau_n$, specifying only the types of the variables involved. We often write $\Phi_\Gamma$ to indicate that $\Gamma$ is the unique typing context determined by $\Phi$ in this manner.

The definition of the refinement satisfaction judgment makes use of an auxiliary judgement, $\phi_1 \leq_\tau \phi_2$, where $\phi_1 \sqsubseteq \tau$ and $\phi_1 \sqsubseteq \tau$, which we shall often just write as $\phi_1 \leq \phi_2$ when $\tau$ is clear from context. This judgment is called *refinement entailment*. It states that the refinement $\phi_1$ is *at least as strong as*, or *no weaker than*, the refinement $\phi_2$. Informally, this means that if $e : \tau$ satisfies $\phi_1$, then it must also satisfy $\phi_2$. According to this interpretation, refinement entailment should be reflexive and transitive. The refinement $\top$, which holds of any well-typed expression, should be greater than (entailed by) any other refinement, and the conjunction of two refinements, $\phi_1 \wedge \phi_2$ should be the meet (greatest lower bound) of $\phi_1$ and $\phi_2$. Because no value can be of two different classes, the conjunction of $\texttt{num}\,!\,\phi_1$ and $\texttt{fun}\,!\,\phi_2$ entails any refinement all. Finally, refinement entailment satisfies the same variance principles given for subtyping in Section 23.3.

$$\frac{\phi \sqsubseteq \tau}{\phi \leq_\tau \phi} \tag{24.2a}$$

$$\frac{\phi_1 \leq_\tau \phi_2 \quad \phi_2 \leq_\tau \phi_3}{\phi_1 \leq_\tau \phi_3} \tag{24.2b}$$

$$\frac{\phi \sqsubseteq \tau}{\phi \leq_\tau \top} \tag{24.2c}$$

$$\frac{\phi_1 \sqsubseteq \tau \quad \phi_2 \sqsubseteq \tau}{\phi_1 \wedge \phi_2 \leq_\tau \phi_1} \tag{24.2d}$$

$$\frac{\phi_1 \sqsubseteq \tau \quad \phi_2 \sqsubseteq \tau}{\phi_1 \wedge \phi_2 \leq_\tau \phi_2} \tag{24.2e}$$

Preview

$$\frac{\phi \leq_\tau \phi_1 \quad \phi \leq_\tau \phi_2}{\phi \leq_\tau \phi_1 \wedge \phi_2} \tag{24.2f}$$

$$\frac{}{\mathtt{num}\,!\,\phi_1 \wedge \mathtt{fun}\,!\,\phi_2 \leq_{\mathtt{dyn}} \phi} \tag{24.2g}$$

$$\frac{\phi \leq_{\mathtt{nat}} \phi'}{\mathtt{num}\,!\,\phi \leq_{\mathtt{dyn}} \mathtt{num}\,!\,\phi'} \tag{24.2h}$$

$$\frac{\phi \leq_{\mathtt{dyn}\rightharpoonup\mathtt{dyn}} \phi'}{\mathtt{fun}\,!\,\phi \leq_{\mathtt{dyn}} \mathtt{fun}\,!\,\phi'} \tag{24.2i}$$

$$\frac{\phi_1 \leq_{\tau_1} \phi_1' \quad \phi_2 \leq_{\tau_2} \phi_2'}{\phi_1 \times \phi_2 \leq_{\tau_1 \times \tau_2} \phi_1' \times \phi_2'} \tag{24.2j}$$

$$\frac{\phi_1 \leq_{\tau_1} \phi_1' \quad \phi_2 \leq_{\tau_2} \phi_2'}{\phi_1 + \phi_2 \leq_{\tau_1 + \tau_2} \phi_1' + \phi_2'} \tag{24.2k}$$

$$\frac{\phi_1' \leq_{\tau_1} \phi_1 \quad \phi_2 \leq_{\tau_2} \phi_2'}{\phi_1 \rightharpoonup \phi_2 \leq_{\tau_1 \rightharpoonup \tau_2} \phi_1' \rightharpoonup \phi_2'} \tag{24.2l}$$

For the sake of brevity we usually omit the type subscripts from refinements and the refinement entailment relation.

We are now in a position to define the refinement satisfaction judgment, $\Phi_\Gamma \vdash e \in_\tau \phi$, in which we assume that $\Gamma \vdash e : \tau$. When such a satisfaction judgment holds, we say that $e$ is *well-refined*, a property that may be stated only for expressions that are already *well-typed*. The main objective is to ensure that well-refined expressions do not incur (checked) run-time errors. In the present setting refinements rule out the possibility of casting a value of class $c$ to a class $c' \neq c$. The formulation of satisfaction, though simple-minded, will expose many key ideas. It is to be emphasized, however, that many variations are possible.

To simplify the exposition, it is best to present the rules in groups, rather than all at once. The first group consists of the rules that pertain to expressions independently of their types.

$$\frac{}{\Phi_\Gamma, x \in_\tau \phi \vdash x \in_\tau \phi} \tag{24.3a}$$

$$\frac{\Phi \vdash e \in_\tau \phi' \quad \phi' \leq_\tau \phi}{\Phi \vdash e \in_\tau \phi} \tag{24.3b}$$

$$\frac{\Phi \vdash e \in_\tau \phi_1 \quad \Phi \vdash e \in_\tau \phi_2}{\Phi \vdash e \in_\tau \phi_1 \wedge \phi_2} \tag{24.3c}$$

$$\frac{\Phi, x \in_\tau \phi \vdash e \in_\tau \phi}{\Phi \vdash \mathtt{fix}\, x : \tau \,\mathtt{is}\, e \in_\tau \phi} \tag{24.3d}$$

Rule (24.3a) expresses the obvious principle that if a variable is *assumed* to satisfy a refinement $\phi$, then of course it *does* satisfy that refinement. As usual, the principle of substitution will turn out

Preview

to be admissible. It states that if a variable is assumed to satisfy $\phi$, then we may substitute for it any expression that does satisfy that refinement, and the resulting instance will continue to satisfy the same refinement it had priot to the substitution.

Rule (24.3b) is analogous to the subsumption principle given in Chapter 23, although here it has a subly different meaning. Specifically, if an expression $e$ satisfies a refinement $\phi'$, and $\phi'$ is stronger than some refinement $\phi$ (as determined by rules (24.2)), then $e$ must also satisfy the refinement $\phi$. This is simply a matter of logic, because the judgment $\phi' \leq \phi$ states that $\phi'$ logically entails $\phi$.

Rule (24.3c) expresses the logical meaning of conjunction. If an expression $e$ satisfies both $\phi_1$ and $\phi_2$, then it also satisfies $\phi_1 \wedge \phi_2$. Rule (24.3b) ensures that the converse holds as well, noting that by rules (24.2)and-left and (24.2)and-right a conjunction is stronger than either of its conjuncts. Similarly, the same rule ensures that if $e$ satisfies $\phi$, then it also satisfies $\top$, but we *do not* postulate that every well-typed expression satisfies $\top$, for that would defeat the goal of ensuring that well-refined expressions do not incur a run-time fault.

Rule (24.3d) states that refinements are closed under general recursion (formation of fixed points). To show that $\mathtt{fix}\, x : \tau\, \mathtt{is}\, e$ satisfies a refinement $\phi$, it suffices to show that $e$ satisfies $\phi$, under the assumption that $x$, which stands for the recursive expression itself, satisfies $\phi$. It is thus obvious that non-terminating expressions, such as $\mathtt{fix}\, x : \tau\, \mathtt{is}\, x$, satisfy any refinement at all. In particular, such a divergent expression does not incur a run-time error, the guarantee we are after with the present system of refinements.

The second group concerns the type `dyn` of classified values.

$$\frac{\Phi \vdash e \in_{\mathtt{nat}} \phi}{\Phi \vdash \mathtt{num}\,!\,e \in_{\mathtt{dyn}} \mathtt{num}\,!\,\phi} \tag{24.4a}$$

$$\frac{\Phi \vdash e \in_{\mathtt{dyn}\rightharpoonup\mathtt{dyn}} \phi}{\Phi \vdash \mathtt{fun}\,!\,e \in_{\mathtt{dyn}} \mathtt{fun}\,!\,\phi} \tag{24.4b}$$

$$\frac{\Phi \vdash e \in_{\mathtt{dyn}} \mathtt{num}\,!\,\phi}{\Phi \vdash e\,@\,\mathtt{num} \in_{\mathtt{nat}} \phi} \tag{24.4c}$$

$$\frac{\Phi \vdash e \in_{\mathtt{dyn}\rightharpoonup\mathtt{dyn}} \mathtt{fun}\,!\,\phi}{\Phi \vdash e\,@\,\mathtt{fun} \in_{\mathtt{dyn}\rightharpoonup\mathtt{dyn}} \phi} \tag{24.4d}$$

$$\frac{\Phi \vdash e \in_{\mathtt{dyn}} \top}{\Phi \vdash \mathtt{num}\,?\,e \in_{\mathtt{bool}} \top} \tag{24.4e}$$

$$\frac{\Phi \vdash e \in_{\mathtt{dyn}} \top}{\Phi \vdash \mathtt{fun}\,?\,e \in_{\mathtt{bool}} \top} \tag{24.4f}$$

Rules (24.4a) and (24.4b) state that a newly-created value of class $c$ satisfies the refinement of the type `dyn` stating this fact, provided that the underlying value satisifes the given refinement, $\phi$, of the type associated to that class (`nat` for `num` and $\mathtt{dyn} \rightharpoonup \mathtt{dyn}$ for `fun`).

Rules (24.4c) and (24.4d) state that a value of type `dyn` may only be safely cast to a class $c$ if the value is known statically to be of this class. This requirement is stated by the premises of these rules, which require the class of the cast value to be known and appropriate for the cast. The



Preview

result of a well-refined cast satisfies the refinement given to its underlying value. It is important to realize that in the quest to avoid run-time faults, it is not possible to cast a value whose only known refinement is $\top$, which imposes no restrictions on it. This is a serious limitation, because in many situations it is not possible to determine statically what is the class of a value. We will return to this critical point shortly.

Rules (24.4e) and (24.4f) compute a boolean based on the class of its argument. We shall have more to say about this in Section 24.2.

The third group of rules govern nullary and binary product types.

$$\frac{}{\Phi \vdash \langle\rangle \in_{\mathtt{unit}} \top} \tag{24.5a}$$

$$\frac{\Phi \vdash e_1 \in_{\tau_1} \phi_1 \quad \Phi \vdash e_2 \in_{\tau_2} \phi_2}{\Phi \vdash \langle e_1, e_2\rangle \in_{\tau_1\times\tau_2} \phi_1 \times \phi_2} \tag{24.5b}$$

$$\frac{\Phi \vdash e \in_{\tau_1\times\tau_2} \phi_1 \times \phi_2}{\Phi \vdash e \cdot \mathtt{l} \in_{\tau_1} \phi_1} \tag{24.5c}$$

$$\frac{\Phi \vdash e \in_{\tau_1\times\tau_2} \phi_1 \times \phi_2}{\Phi \vdash e \cdot \mathtt{r} \in_{\tau_1} \phi_1} \tag{24.5d}$$

Rule (24.5a) states the obvious: the null-tuple is well-refined by the trivial refinement. Because `unit` contains only one element, little else can be said about it. Rule (24.5b) states that a pair satisifes a product of refinements if each component satisfies the corresponding refinement. Rules (24.5c) and (24.5d) state the converse.

The fourth group of rules govern nullary and binary sum types.

$$\frac{\Phi \vdash e \in_{\mathtt{void}} \phi'}{\Phi \vdash e \in_{\mathtt{void}} \phi} \tag{24.6a}$$

$$\frac{\Phi \vdash e_1 \in_{\tau_1} \phi_1}{\Phi \vdash \mathtt{l} \cdot e_1 \in_{\tau_1+\tau_2} \phi_1 + \phi_2} \tag{24.6b}$$

$$\frac{\Phi \vdash e_2 \in_{\tau_2} \phi_2}{\Phi \vdash \mathtt{r} \cdot e_2 \in_{\tau_1+\tau_2} \phi_1 + \phi_2} \tag{24.6c}$$

$$\frac{\Phi \vdash e \in_{\tau_1+\tau_2} \phi_1 + \phi_2 \quad \Phi, x_1 \in_{\tau_1} \phi_1 \vdash e_1 \in_{\tau} \phi \quad \Phi, x_2 \in_{\tau_2} \phi_2 \vdash e_2 \in_{\tau} \phi}{\Phi \vdash \mathtt{case}\, e\, \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \in_{\tau} \phi} \tag{24.6d}$$

Rule (24.6a) states that if an expression of type `void` satisfies *some* refinement (and hence is error-free), then it satisfies *every* refinement (of type `void`), because there are no values of this type, and hence, being error-free, must diverge.

Rules (24.6b) and (24.6c) are similarly motivated. If $e_1$ satisfies $\phi_1$, then $\mathtt{l} \cdot e_1$ satifies $\phi_1 + \phi_2$ *for any* refinement $\phi_2$, precisely because the latter refinement is irrelevant to the injection. Similarly, the right injection is independent of the refinement of the left summand.

Rule (24.6d) is in some respects the most interesting rule of all, one that we shall have occasion to revise shortly. The salient feature of this rule is that it propagates refinement information about

Preview

the injected value into the corresponding branch by stating an appropriate assumption about the bound variable of each branch. But it *does not* propagate any information into the branches about what is known about $e$ in each branch, namely that in the first branch $e$ must be of the form $\mathtt{l} \cdot e_1$ and in the second branch $e$ must be of the form $\mathtt{r} \cdot e_2$.

The failure to propagate this information may seem harmless, but it is, in fact, quite restrictive. To see why, consider the special case of the type `bool`, which is defined in Section 11.3.2 to be $\mathtt{unit} + \mathtt{unit}$. The conditional expression $\mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2$ is defined to be a case analysis in which there is no associated data to pass into the branches of the conditional. This means that within the `then` branch, $e_1$, it is not known statically that $e$ is in fact `true`, nor is it known within the `else` branch, $e_2$, that $e$ is in fact `false`.

The fifth group of rules governs the function type.

$$\frac{\Phi, x \in_{\tau_1} \phi_1 \vdash e_2 \in_{\tau_2} \phi_2}{\Phi \vdash \lambda\,(x{:}\tau_1)\, e_2 \in_{\tau_1 \to \tau_2} \phi_1 \rightharpoonup \phi_2} \tag{24.7a}$$

$$\frac{\Phi \vdash e_1 \in_{\tau_2 \to \tau} \phi_2 \rightharpoonup \phi \quad e_2 \in_{\tau_2} \phi_2}{\Phi \vdash e_1(e_2) \in_{\tau} \phi} \tag{24.7b}$$

Rule (24.7a) states that a $\lambda$-abstraction satisfies a function refinement if its body satisfies the range refinement, under the assumption that its argument satisfies the domain refinement. This is only to be expected.

Rule (24.7b) states the converse. If an expression of function type satisfies a function refinement, and it is applied to an argument satisfying the domain refinement, then the application must satisfy the range refinement.

The last group of rules govern the type `nat`:

$$\frac{}{\Phi \vdash \mathtt{z} \in_{\mathtt{nat}} \top} \tag{24.8a}$$

$$\frac{\Phi \vdash e \in_{\mathtt{nat}} \top}{\Phi \vdash \mathtt{s}(e) \in_{\mathtt{nat}} \top} \tag{24.8b}$$

$$\frac{\Phi \vdash e \in_{\mathtt{nat}} \top \quad \Phi \vdash e_0 \in_{\tau} \phi \quad \Phi, x \in_{\mathtt{nat}} \top \vdash e_1 \in_{\tau} \phi}{\Phi \vdash \mathtt{ifz}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\} \in_{\tau} \phi} \tag{24.8c}$$

These rules are completely unambitious: they merely restate the typing rules as refinement rules that impose no requirements and make no guarantees of any properties of the natural numbers. One could envision adding refinements of the natural numbers, for instance stating that a natural number is known to be zero or non-zero, for example.

To get a feel for the foregoing rules, it is useful to consider some simple examples. First, we may combine rules (24.3b) and (24.4a) to derive the judgment

$$\mathtt{num}\,!\,\overline{n} \in_{\mathtt{dyn}} \top$$

for any natural number $n$. That is, we may "forget" the class of a value by applying subsumption, and appealing to rule (24.2c). Second, such reasoning is essential in stating refinement satisfaction for a boolean conditional (or, more generally, any case analysis). For example, the following

Preview

judgment is directly derivable without subsumption:

$$\Phi, x \in_{\mathtt{bool}} \top \vdash \mathtt{if}\, x\, \mathtt{then}\, (\mathtt{num}\,!\,\mathtt{z})\, \mathtt{else}\, (\mathtt{num}\,!\,\mathtt{s}(\mathtt{z})) \in_{\mathtt{dyn}} \mathtt{num}\,!\,\top.$$

But the following judgment is only derivable because we may weaken knowledge of the class of a value in each branch to a common refinement, in this case the weakest refinement of all:

$$\Phi, x \in_{\mathtt{bool}} \top \vdash \mathtt{if}\, x\, \mathtt{then}\, (\mathtt{num}\,!\,\mathtt{z})\, \mathtt{else}\, (\mathtt{fun}\,!\,(\lambda\,(y{:}\mathtt{dyn})\, y)) \in_{\mathtt{dyn}} \top.$$

In general conditionals attenuates the information we possess about a value, except in those cases where the same information is known about both branches. Conditionals are the main source of lossiness in checking refinement satisfaction.

Conjunction refinements are used to express multiple properties of a single expression. For example, the identity function on the type dyn may be shown to satisfy the conjunctive refinement

$$(\mathtt{num}\,!\,\top \rightharpoonup \mathtt{num}\,!\,\top) \wedge (\mathtt{fun}\,!\,\top \rightharpoonup \mathtt{fun}\,!\,\top).$$

The occurrences of $\top$ in the first conjunct refine the type nat, whereas the occurrences in the second conjunct refine the type $\mathtt{dyn} \rightharpoonup \mathtt{dyn}$. It is a good exercise to check that $\lambda\,(x{:}\mathtt{dyn})\, x$ satisfies the above refinement of the type $\mathtt{dyn} \rightharpoonup \mathtt{dyn}$.

## 24.2 Boolean Blindness

Let us consider a very simple example that exposes a serious disease suffered by many programming languages, a condition called *boolean blindness*. Suppose that $x$ is a variable of type dyn with refinement $\top$, and consider the expression

$$\mathtt{if}\, (\mathtt{num}\,?\,x)\, \mathtt{then}\, x\, @\, \mathtt{num}\, \mathtt{else}\, \mathtt{z}.$$

Although it is evident that this expression has type nat, it is nevertheless *ill-refined* (satisfies no refinement), even though it does not incur a run-time error. Specifically, within the then branch, we as programmers know that $x$ is a value of class num, but this fact is not propagated into the then branch by rules (24.6) (of which the boolean conditional is a special case). Consequently, rule (24.4c) does not apply (formally we do not know enough about $x$ to cast it safely), so the expression as a whole is rejected as ill-refined. The branches of the conditional are "blind" both to the outcome and the meaning of the boolean value computed by the test whether $x$ is of class num.

Boolean blindness is endemic among programming languages. The difficulty is that a boolean carries exactly one bit of information, which is not sufficient to capture the *meaning* of that bit. Put in other terms, there is a widespread confusion between a *boolean*, which is (literally) a bit of data, and a *proposition*, which expresses a fact. The two are not even of the same character, let alone synonymous. Taken by itself, a boolean conveys no information other than its value, whereas a proposition expresses the reasoning that goes into ensuring that a piece of code is correct. In terms of the above example knowing that $\mathtt{num}\,?\,x$ evaluates dynamically to the boolean true is not connected statically with the fact that the class of $x$ is num or not. That information lives elsewhere, essentially in the specification of the class test primitive. The question is how to connect

Preview

the boolean value returned by the class test with relevant facts about whether the class of $x$ is `num` or not.

Because the purpose of a type refinement system is precisely to capture such facts in a form that can be written down and verified, one may suspect that the difficulty with the foregoing example is that the system of refinements under consideration is too weak to capture the property that we need to ensure that run-time faults do not occur. The example shows that something more is needed if type refinement is to be useful, so we first consider what might be involved in enriching the definition of refinement satisfaction to ensure that the proper connections are made. The matter boils down to two issues:

1. Propagating the fact that $\texttt{num}\,?\,x$ returned `true` into the `then` branch, and the fact that it returned `false` into the `else` branch.

2. Connecting facts about the return value of a test to facts about the value being tested.

These are, in the present context, distinct aspects of the problem. Let us consider them in turn. The upshot of the discussion will be to uncover a design flaw in the treatment of casting, and to suggest an alternative formulation that does not suffer from boolean blindness.

To address the problem as stated, we first need to enrich the language of refinements to include $\texttt{true} \sqsubseteq \texttt{bool}$ and $\texttt{false} \sqsubseteq \texttt{bool}$, stating that a boolean is known to be either `true` or `false`, respectively. That way we can hope to express facts about boolean-valued expressions as refinement judgments. But what, then, should be the refinement rule for the boolean conditional? As a first guess one might consider something like the following:

$$\frac{\Phi, e \in_{\texttt{bool}} \texttt{true} \vdash e_1 \in_\tau \phi \quad \Phi, e \in_{\texttt{bool}} \texttt{false} \vdash e_2 \in_\tau \phi}{\Phi \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \in_\tau \phi} \tag{24.9}$$

Such a rule is a bit unusual in that it introduces a hypothesis about an expression, rather than a variable, but let us ignore that for the time being and press on.

Having re-formulated the refinement rule for the conditional, we immediately encounter another problem: how to deduce that $x \in_{\texttt{dyn}} \texttt{num}\,!\,\top$, which is required for casting, from the assumption that $\texttt{num}\,?\,x \in_{\texttt{bool}} \texttt{true}$? One way to achieve this is by modifying the refinement rule for the conditional to account for the special case in which the boolean expression is *literally* $\texttt{num}\,?\,e$ for some $e$. If it is, then we propagate into the `then` branch the additional assumption $e \in_{\texttt{dyn}} \texttt{num}\,!\,\top$, but if it is not, then no fact about the class of $e$ is propagated into the `else` branch.[2]

This is enough to ensure that the example under consideration is well-refined. But what if the boolean on which we are conditioning is not literally $\texttt{num}\,?\,e$, but merely *implies* the test would return `true`? How then are we to make the connection between such expressions and relevant facts about the class of $e$? Clearly there is no end to the special cases we could consider in an effort to restore vision to refinements, but there seems to be no canonical solution.

All is not lost, however! The foregoing analysis suggests that the fault lies not in our refinements, but in our language design. The sole source of run-time errors is the "naked cast" that

[2] For the special case of there being exactly two classes, we could propagate that the class of $e$ is `fun` into the `else` branch, but this approach does not generalize.



PREVIEW

attempts to extract the underlying natural number from a value of type dyn—and signals a run-time error if it cannot do so because the class of the value is not num. The boolean-valued test for the class of a value seems to provide a way to avoid these errors. But, as we have just seen, the true cause of the problem is the attempt to separate the test from the cast. We may, instead, consolidate these into a single form, say

$$\texttt{ifofcl}[\texttt{num}](e;x_0.e_0;e_1),$$

that tests whether the class of $e$ is num, and, if so, passes the underlying number to $e_0$ by substituting it for $x_0$, and otherwise evaluates $e_1$. No run-time error is possible, and hence no refinements are needed to ensure that it cannot occur.

But does this not mean that type refinements are pointless? Not at all. It simply means that in some cases verification methods, such as type refinements, are needed solely to remedy a language design flaw, rather than provide a useful tool for programmers to help express and verify the correctness of their programs. We may still wish to express invariants such as the property that a particular function maps values of class $c$ into values of class $c'$, simply for the purpose of stating a programmer's intention. Or we may enrich the system of refinements to track properties such as whether a number is even or odd, and to state conditions such as that a given function maps evens to odds and odds to evens. There is no limit, in principle, to the variations and extensions that we can consider as a means to help ensure that programs behave as expected.

## 24.3 Refinement Safety

The judgment $\Phi_\Gamma \vdash a \in_\tau \phi$ presupposes that $\Gamma \vdash e : \tau$, so by adapting the proofs given earlier, we may type preservation and progress for well-typed terms.

**Theorem 24.1** (Type Safety)**.** *Suppose that $e : \tau$ for closed $e$.*

1. *If $e \longmapsto^* e'$, then $e' : \tau$.*
2. *Either $e$ err, or $e$ val, or there exists $e'$ such that $e \longmapsto e'$.*

The proof of progress requires a canonical forms lemma, which for the type dyn may be stated as follows:

**Lemma 24.2** (Canonical Forms)**.** *Suppose that $e$ : dyn and $e$ val. Then either $e = \texttt{num}\,!\,e'$ or $e = \texttt{fun}\,!\,e'$ for some $e'$.*

The expression $e'$ would also be a value under an eager dynamics, but need not be under a lazy dynamics. The proof of Lemma 24.2 proceeds as usual, by analyzing the typing rules for values.

The goal of the refinement system introduced in Section 24.1 is to ensure that errors cannot arise in a well-refined program. To establish this, we first establish that the dynamics preserves refinements.

**Lemma 24.3.** *Suppose that $e$ val and $e \in_{\texttt{dyn}} \phi$. If $\phi \leq \texttt{num}\,!\,\phi'$, then $e = \texttt{num}\,!\,e'$, where $e' \in_{\mathit{nat}} \phi'$, and if $\phi \leq \texttt{fun}\,!\,\phi'$, then $e = \texttt{fun}\,!\,e'$, where $e' \in_{\texttt{dyn}\to\texttt{dyn}} \phi'$.*



Preview

*Proof.* The proof requires Lemma 24.2 to characterize the possible values of dyn, and an analysis of the refinement satisfaction rules. The statement of the lemma is designed to accommodate rule (24.3b), which appeals to the transitivity of refinment entailment. □

**Theorem 24.4** (Refinement Preservation)**.** *If $e \in_\tau \phi$ and $e \longmapsto e'$, then $e' \in_\tau \phi$.*

*Proof.* We know by the preceding theorem that $e' : \tau$. To show that $e' \in_\tau \phi$ we proceed by induction on the definition of refinment satisfaction given in Section 24.1. The type-independent group, rules (24.3), are all easily handled, apart from the rule for fixed points, which requires an appeal to a substitution lemma, just as in Theorem 15.2. The remaining groups are all handled easily, bearing in mind that an erroneous expression cannot make a transition. □

**Theorem 24.5** (Refinement Error-Freedom)**.** *If $e \in_\tau \phi$, then $\neg(e\ \mathsf{err})$.*

*Proof.* By induction on the definition of refinment satisfaction. The only interesting cases are rules (24.4c) and (24.4d), which are handled by appeal to Lemma 24.3 in the case that the cast expression is a value. □

**Corollary 24.6** (Refinement Safety)**.** *If $e \in_\tau \phi$, then either $e$ val or there exists $e'$ such that $e' \in_\tau \phi$ and $e \longmapsto e'$. In particular, $\neg(e\ \mathsf{err})$.*

*Proof.* By Theorems 24.1, 24.4, and 24.5. □

## 24.4 Notes

The distinction between types and refinements is fundamental, yet the two are often conflated. Types determine the structure of a programming language, including its statics and dynamics; refinements specify the behavior of well-typed programs. In full generality the satisfaction judgment $e \in_\tau \phi$ need not be decidable, whereas it is sensible to insist that the typing judgment $e : \tau$ be decidable. The refinement system presented in this chapter is designed to be easily decidable, but one may consider many notions of refinement that are not. For example, one may postulate that if $e \in_\tau \phi$ and that $e'$ is indistinguishable from $e$ by any program in the language,[3] then $e' \in_\tau \phi$. In contrast such a move is not sensible in a type system, because the dynamics is determined by the inversion principle derived from the statics. Therefore, refinement is of necessity posterior to typing.

The syntactic formulation of type refinements considered in this chapter was originally given by Freeman and Pfenning (1991), and subsequently studied and extended by Davies and Pfenning (2000); Davies (2005), Xi and Pfenning (1998), Dunfield and Pfenning (2003), and Mandelbaum et al. (2003). A more general semantic formulation of type refinement was given explicitly by Denney (1998) in the style of the realizability interpretation of type theory on which NuPRL (Constable, 1986) is based. (See the survey by van Oosten (2002) for the history of the realizability interpretations of constructive logic.)

[3] See Chapter 49 for a precise definition and development of this concept.



Preview

## Exercises

**24.1**. Show that if $\phi_1 \leq \phi_1'$ and $\phi_2 \leq \phi_2'$, then $\phi_1 \wedge \phi_2 \leq \phi_1' \wedge \phi_2'$.

**24.2**. Show that $\phi \leq \phi'$ iff for every $\phi''$, if $\phi'' \leq \phi$, then $\phi'' \leq \phi'$. (This property of entailment is an instance of the more general *Yoneda Lemma* in category theory.)

**24.3**. Extend the system of refinements to recursive types by introducing a refinement $\mathtt{fold}(\phi)$ that classifies values of recursive type $\mathtt{rec}\, t\, \mathtt{is}\, \tau$ in terms of a refinement of the unfolding of that recursive type.

**24.4**. Consider the following two additional forms of refinement for sum types, called *summand refinements*:

$$\frac{\phi_1 \sqsubseteq \tau_1}{\mathtt{l} \cdot \phi_1 \sqsubseteq \tau_1 + \tau_2} \tag{24.10a}$$

$$\frac{\phi_2 \sqsubseteq \tau_2}{\mathtt{r} \cdot \phi_2 \sqsubseteq \tau_1 + \tau_2} \tag{24.10b}$$

Informally, $\mathtt{l} \cdot \phi_1$ classifies expressions of type $\tau_1 + \tau_2$ that lie within the left summand and whose underlying value satisfies $\phi_1$, and similarly for $\mathtt{r} \cdot \phi_2$.

(a) State appropriate entailment rules governing summand refinements.

(b) State appropriate refinement rules assigning summand refinements to the introductory forms of a sum type.

(c) Give additional rules for the case analysis construct using summand refinements to allow unreachable branches to be disregarded during refinement checking.

(d) Modify rule (24.6d) so that the information "learned" by examining the value of $e$ at execution time is propagated into the appropriate branch of the case analysis.

Examine the relevance of this extension to the prospects for a cure for Boolean blindness.

**24.5**. Using the preceding exercise, derive the refinements $\mathtt{num}\,!\,\phi$ and $\mathtt{fun}\,!\,\phi$ from the other refinement rules, including the refinement $\mathtt{fold}(\phi)$ considered in Exercise **24.3**.

**24.6**. Show that the addition function (19.8), a value of type `dyn`, satisfies the refinement

$$\mathtt{fun}\,!\,(\mathtt{num}\,!\,\top \rightharpoonup \mathtt{fun}\,!\,(\mathtt{num}\,!\,\top \rightharpoonup \mathtt{num}\,!\,\top)),$$

stating that

(a) It is itself a value of class `fun`.

(b) The so-classified function maps a value of class `num` to a result, if any, of class `fun`.

(c) The so-classified function maps a value of class `num` to a result, if any, of class `num`.

This description exposes the hidden complexity in the superficial simplicity of a unityped language.

Preview

**24.7**. Revisit the optimization process of the addition function carried out in Section 19.3 in view of your answer to Exercise **24.6**. Show that the validity of the optimizations is guaranteed by the satisfaction of the stated type refinement for addition.

Preview

Preview

# Chapter 25

# Singleton Kinds

The expression $\texttt{let}\ e_1{:}\tau\ \texttt{be}\ x\ \texttt{in}\ e_2$ is a form of abbreviation mechanism by which we may bind $e_1$ to the variable $x$ for use within $e_2$. In the presence of function types this expression is definable as the application $(\lambda\,(x{:}\tau)\,e_2)(e_1)$, which accomplishes the same thing. It is natural to consider an analogous form of `let` expression which permits a type expression to be bound to a type variable within a specified scope. Using $\texttt{def}\ t\ \texttt{is}\ \tau\ \texttt{in}\ e$ to bind the type variable $t$ to $\tau$ within the expression $e$, we may write expressions such as

$$\texttt{def}\ t\ \texttt{is}\ \texttt{nat} \times \texttt{nat}\ \texttt{in}\ \lambda\,(x{:}t)\ \texttt{s}(x \cdot \texttt{l}),$$

which introduces a *type abbreviation* within an expression. To ensure that this expression is well-typed, the type variable $t$ must be *synonymous* with the type $\texttt{nat} \times \texttt{nat}$, for otherwise the body of the $\lambda$-abstraction is not type correct.

Following the pattern of the expression-level `let`, we might guess that $\texttt{def}\ t\ \texttt{is}\ \tau\ \texttt{in}\ e$ abbreviates the polymorphic instantiation $\Lambda(t.e)[\tau]$, which binds $t$ to $\tau$ within $e$. This does, indeed, capture the dynamics of type abbreviation, but it fails to validate the intended statics. The difficulty is that, according to this interpretation of type definitions, the expression $e$ is type-checked in the absence of any knowledge of the binding of $t$, rather than in the knowledge that $t$ is synonymous with $\tau$. Thus, in the above example, the expression $\texttt{s}(x \cdot \texttt{l})$ would fail to type check, unless the binding of $t$ were exposed.

The interpretation of type definition in terms of type abstraction and type application fails. Lacking any other idea, we might argue that type abbreviation ought to be considered as a primitive concept, rather than a derived notion. The expression $\texttt{def}\ t\ \texttt{is}\ \tau\ \texttt{in}\ e$ would be taken as a primitive form of expression whose statics is given by the following rule:

$$\frac{\Gamma \vdash [\tau/t]e : \tau'}{\Gamma \vdash \texttt{def}\ t\ \texttt{is}\ \tau\ \texttt{in}\ e : \tau'} \tag{25.1}$$

This would address the problem of supporting type abbreviations, but it does so in a rather *ad hoc* manner. We seek a more principled solution that arises naturally from the type structure of the language.

PREVIEW

The methodology of identifying language constructs with type structure suggests that we ask not how to support type abbreviations, but rather what form of type structure gives rise to type abbreviations? Thinking along these lines leads to the concept of *singleton kinds*, which not only account for type abbreviations but also play a crucial role in the design of module systems, as will be discussed in detail in Chapters 46 and 47.

## 25.1 Overview

The central organizing principle of type theory is *compositionality*. To ensure that a program may be decomposed into separable parts, we ensure that the composition of a program from constituent parts is mediated by the types of those parts. Put in other terms, the only thing that one portion of a program "knows" about another is its type. For example, the formation rule for addition of natural numbers depends only on the type of its arguments (both must have type `nat`), and not on their specific form or value. But in the case of a type abbreviation of the form $\texttt{def}\ t\ \texttt{is}\ \tau\ \texttt{in}\ e$, the principle of compositionality dictates that the only thing that $e$ "knows" about the type variable $t$ is its kind, namely $\texttt{T}$, and not its binding, namely $\tau$. This is accurately captured by the proposed representation of type abbreviation as the combination of type abstraction and type application, but, as we have just seen, this is not the intended meaning of the construct!

We could, as suggested in the introduction, abandon the core principles of type theory, and introduce type abbreviations as a primitive notion. But there is no need to do so. Instead we can simply note that what is needed is for the kind of $t$ to capture its identity. This may be achieved through the notion of a *singleton kind*. Informally, the kind $\texttt{S}(\tau)$ is the kind of types that are definitionally equal to $\tau$. That is, up to definitional equality, this kind has only one inhabitant, namely $\tau$. Consequently, if $u :: \texttt{S}(\tau)$ is a variable of singleton kind, then within its scope, the variable $u$ is synonymous with $\tau$. Thus we may represent $\texttt{def}\ t\ \texttt{is}\ \tau\ \texttt{in}\ e$ by $\Lambda(t{::}\texttt{S}(\tau).e)[\tau]$, which correctly propagates the identity of $t$, namely $\tau$, to $e$ during type checking.

A proper treatment of singleton kinds requires some additional machinery at the constructor and kind level. First, we must capture the idea that a constructor of singleton kind is *a fortiori* a constructor of kind $\texttt{T}$, and hence is a type. Otherwise, a variable, $u$, of singleton kind cannot be used as a type, even though it is explicitly defined to be one! This may be captured by introducing a *subkinding* relation, $\kappa_1 :<: \kappa_2$, which is analogous to subtyping, except at the kind level. The fundamental axiom of subkinding is $\texttt{S}(\tau) :<: \texttt{T}$, stating that every constructor of singleton kind is a type.

Second, we must account for the occurrence of a constructor of kind $\texttt{T}$ within the singleton kind $\texttt{S}(\tau)$. This intermixing of the constructor and kind level means that singletons are a form of *dependent kind* in that a kind may depend on a constructor. Another way to say the same thing is that $\texttt{S}(\tau)$ represents a *family of kinds* indexed by constructors of kind $\texttt{T}$. This, in turn, implies that we must generalize the product and function kinds to *dependent products* and *dependent functions*. The dependent product kind, $\Sigma\, u :: \kappa_1.\kappa_2$, classifies pairs $\langle c_1, c_2\rangle$ such that $c_1 :: \kappa_1$, as might be expected, and $c_2 :: [c_1/u]\kappa_2$, in which the kind of the second component is sensitive to the first component itself, and not just its kind. The dependent function kind, $\Pi\, u :: \kappa_1.\kappa_2$ classifies functions that, when applied to a constructor $c_1 :: \kappa_1$, results in a constructor of kind $[c_1/u]\kappa_2$. The important point is that the kind of the result is sensitive to the argument, and not just to its kind.

PREVIEW

Third, it is useful to consider singletons not just of kind $\mathtt{T}$, but also of higher kinds. To support this we introduce *higher singletons*, written $\mathtt{S}(c :: \kappa)$, where $\kappa$ is a kind and $c$ is a constructor of kind $k$. These are definable in terms of the basic form of singleton kinds using dependent function and product kinds.

## 25.2 Singletons

A singleton kind has the form $\mathtt{S}(c)$, where $c$ is a constructor. The singleton classifies all constructors that are equivalent, in a sense to be made precise shortly, to the constructor $c$. For the time being we consider singleton kinds in the context of the language $\mathbf{F}_\omega$ described in Chapter 22, which includes a kind of types, and is closed under product and function kinds. In Section 25.3 we will enrich the language of kinds in a manner that will render definable the product and function kinds of $\mathbf{F}_\omega$.

The following judgment forms comprise the statics of singletons:

| | |
|---|---|
| $\Delta \vdash \kappa\ \mathsf{kind}$ | kind formation |
| $\Delta \vdash \kappa_1 \equiv \kappa_2$ | kind equivalence |
| $\Delta \vdash c :: \kappa$ | constructor formation |
| $\Delta \vdash c_1 \equiv c_2 :: \kappa$ | constructor equivalence |
| $\Delta \vdash \kappa_1 :<: \kappa_2$ | subkinding |

These judgments are defined simultaneously by a collection of rules including the following:

$$\frac{\Delta \vdash c :: \mathtt{Type}}{\Delta \vdash \mathtt{S}(c)\ \mathsf{kind}} \tag{25.2a}$$

$$\frac{\Delta \vdash c :: \mathtt{Type}}{\Delta \vdash c :: \mathtt{S}(c)} \tag{25.2b}$$

$$\frac{\Delta \vdash c :: \mathtt{S}(d)}{\Delta \vdash c \equiv d :: \mathtt{Type}} \tag{25.2c}$$

$$\frac{\Delta \vdash c :: \kappa_1 \quad \Delta \vdash \kappa_1 :<: \kappa_2}{\Delta \vdash c :: \kappa_2} \tag{25.2d}$$

$$\frac{\Delta \vdash c :: \mathtt{Type}}{\Delta \vdash \mathtt{S}(c) :<: \mathtt{Type}} \tag{25.2e}$$

$$\frac{\Delta \vdash c \equiv d :: \mathtt{Type}}{\Delta \vdash \mathtt{S}(c) \equiv \mathtt{S}(d)} \tag{25.2f}$$

$$\frac{\Delta \vdash \kappa_1 \equiv \kappa_2}{\Delta \vdash \kappa_1 :<: \kappa_2} \tag{25.2g}$$

$$\frac{\Delta \vdash \kappa_1 :<: \kappa_2 \quad \Delta \vdash \kappa_2 :<: \kappa_3}{\Delta \vdash \kappa_1 :<: \kappa_3} \tag{25.2h}$$

Preview

Omitted for brevity are rules stating that constructor and kind equivalence are reflexive, symmetric, transitive, and preserved by kind and constructor formation.

Rule (25.2b) expresses the principle of "self-recognition," which states that every constructor, $c$, of kind $\mathtt{Type}$ also has the kind $\mathtt{S}(c)$. By rule (25.2c) any constructor of kind $\mathtt{S}(c)$ is definitionally equal to $c$. Consequently, self-recognition is in this sense an expression of the reflexivity of constructor equivalence. Rule (25.2e) is just the subsumption principle re-stated at the level of constructors and kinds. Rule (25.2f) states that the singleton kind respects equivalence of its constructors, so that equivalent constructors determine the same singletons. Rules (25.2g) and (25.2h) state that the sub-kinding relation is a pre-order that respects kind equivalence.

To see these rules in action let us consider a few illustrative examples. First, consider the behavior of variables of singleton kind. Suppose that $\Delta \vdash u :: \mathtt{S}(c)$ is such a variable. Then by rule (25.2c) we may deduce that $\Delta \vdash u \equiv c :: \mathtt{T}$. Thus, the declaration of $u$ with a singleton kind serves to define $u$ to be the constructor (of kind $\mathtt{T}$) specified by its kind. Singletons capture the concept of a type definition discussed in the introduction to this chapter.

Taking this a step further, the existential type $\exists\, u :: \mathtt{S}(c).\tau$ is the type of packages whose representation type is (equivalent to) $c$—it is an abstract type whose identity is revealed by assigning it a singleton kind. By the general principles of equivalence we have that the type $\exists\, u :: \mathtt{S}(c).\tau$ is equivalent to the type $\exists\, _ :: \mathtt{S}(c).[c/u]\tau$, wherein we have propagated the equivalence of $u$ and $c$ into the type $\tau$. On the other hand we may also "forget" the definition of $u$, because the subtyping

$$\exists\, u :: \mathtt{S}(c).\tau <: \exists\, u :: \mathtt{T}.\tau$$

is derivable using the following variance rule for existentials over a kind:

$$\frac{\Delta \vdash \kappa_1 :<: \kappa_2 \quad \Delta, u :: \kappa_1 \vdash \tau_1 <: \tau_2}{\Delta \vdash \exists\, u :: \kappa_1.\tau_1 <: \exists\, u :: \kappa_2.\tau_2} \tag{25.3}$$

Similarly, we may derive the subtyping

$$\forall\, u :: \mathtt{T}.\tau <: \forall\, u :: \mathtt{S}(c).\tau$$

from the following variance rule for universals over a kind:

$$\frac{\Delta \vdash \kappa_2 :<: \kappa_1 \quad \Delta, u :: \kappa_2 \vdash \tau_1 <: \tau_2}{\Delta \vdash \forall\, u :: \kappa_1.\tau_1 <: \forall\, u :: \kappa_2.\tau_2} \tag{25.4}$$

Informally, the displayed subtyping states that a polymorphic function that may be applied to *any* type is one that may only be applied to a particular type, $c$.

These examples show that singleton kinds express the idea of a scoped definition of a type variable in a way that is not tied to an *ad hoc* definition mechanism, but rather arises naturally from general principles of binding and scope. We will see in Chapters 46 and 47 more sophisticated uses of singletons to manage the interaction among program modules.

## 25.3 Dependent Kinds

Although it is perfectly possible to add singleton kinds to the framework of higher kinds introduced in Chapter 22, to do so would be to shortchange the expressiveness of the language. Using



Preview

higher kinds we can express the kind of constructors that, when applied to a type, yield a specific type, say `int`, as result, namely $\mathtt{T} \to \mathtt{S}(\mathtt{int})$. But we cannot express the kind of constructors that, when applied to a type, yield *that very type* as result, for there is no way for the result kind to refer to the argument of the function. Similarly, using product kinds we can express the kind of pairs whose first component is `int` and whose second component is an arbitrary type, namely $\mathtt{S}(\mathtt{int}) \times \mathtt{T}$. But we cannot express the kind of pairs whose second component is equivalent to its first component, for there is no way for the kind of the second component to make reference to the first component itself.

To express such concepts requires a generalization of product and function kinds in which the kind of the second component of a pair may mention the first component of that pair, or the kind of the result of a function may mention the argument to which it is applied. Such kinds are called *dependent kinds* because they involve kinds that mention, or depend upon, constructors (of kind $\mathtt{T}$). The syntax of dependent kinds is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Kind | $\kappa$ | $::=$ | $\mathtt{S}(c)$ | $\mathtt{S}(c)$ | singleton |
| | | | $\Sigma(\kappa_1; u.\kappa_2)$ | $\Sigma\, u :: \kappa_1.\kappa_2$ | dependent product |
| | | | $\Pi(\kappa_1; u.\kappa_2)$ | $\Pi\, u :: \kappa_1.\kappa_2$ | dependent function |
| Con | $c$ | $::=$ | $u$ | $u$ | variable |
| | | | $\mathtt{pair}(c_1; c_2)$ | $\langle c_1, c_2 \rangle$ | pair |
| | | | $\mathtt{proj}[\mathtt{l}](c)$ | $c \cdot \mathtt{l}$ | first projection |
| | | | $\mathtt{proj}[\mathtt{r}](c)$ | $c \cdot \mathtt{r}$ | second projection |
| | | | $\mathtt{lam}[\kappa](u.c)$ | $\lambda\,(u{::}\kappa)\, c$ | abstraction |
| | | | $\mathtt{app}(c_1; c_2)$ | $c_1[c_2]$ | application |

As a notational convenience, when there is no dependency in a kind we write $\kappa_1 \times \kappa_2$ for $\Sigma\, _ :: \kappa_1.\kappa_2$, and $\kappa_1 \to \kappa_2$ for $\Pi\, _ :: \kappa_1.\kappa_2$, where the "blank" stands for an irrelevant variable.

The syntax of dependent kinds differs from that given in Chapter 22 for higher kinds in that we do not draw a distinction between atomic and canonical constructors, and consider that substitution is defined conventionally, rather than hereditarily. This simplifies the syntax, but at the expense of leaving open the decidability of constructor equivalence. The method of hereditary substitution considered in Chapter 22 may be extended to singleton kinds, but we will not develop this extension here. Instead we will simply assert without proof that equivalence of well-formed constructors is decidable.

The dependent product kind $\Sigma\, u :: \kappa_1.\kappa_2$ classifies pairs $\langle c_1, c_2 \rangle$ of constructors in which $c_1$ has kind $\kappa_1$ and $c_2$ has kind $[c_1/u]\kappa_2$. For example, the kind $\Sigma\, u :: \mathtt{T}.\mathtt{S}(u)$ classifies pairs $\langle c, c \rangle$, where $c$ is a constructor of kind $\mathtt{T}$. More generally, this kind classifies pairs of the form $\langle c_1, c_2 \rangle$ where $c_1$ and $c_2$ are equivalent, but not necessarily identical, constructors. The dependent function kind $\Pi\, u :: \kappa_1.\kappa_2$ classifies constructors $c$ that, when applied to a constructor $c_1$ of kind $\kappa_1$ yield a constructor of kind $[c_1/u]\kappa_2$. For example, the kind $\Pi\, u :: \mathtt{T}.\mathtt{S}(u)$ classifies constructors that, when applied to a constructor, $c$, yield a constructor equivalent to $c$; a constructor of this kind is essentially the identity function. We may, of course, combine these to form kinds such as

$$\Pi\, u :: \mathtt{T} \times \mathtt{T}.\mathtt{S}(u \cdot \mathtt{r}) \times \mathtt{S}(u \cdot \mathtt{l}),$$

which classifies functions that swap the components of a pair of types. (Such examples may lead



Preview

us to surmise that the behavior of any constructor may be pinned down precisely using dependent kinds. We shall see in Section 25.4 that this is indeed the case.)

The formation, introduction, and elimination rules for the product kind are as follows:

$$\frac{\Delta \vdash \kappa_1 \text{ kind} \quad \Delta, u :: \kappa_1 \vdash \kappa_2 \text{ kind}}{\Delta \vdash \Sigma\, u :: \kappa_1.\kappa_2 \text{ kind}} \tag{25.5a}$$

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: [c_1/u]\kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle :: \Sigma\, u :: \kappa_1.\kappa_2} \tag{25.5b}$$

$$\frac{\Delta \vdash c :: \Sigma\, u :: \kappa_1.\kappa_2}{\Delta \vdash c \cdot \mathtt{l} :: \kappa_1} \tag{25.5c}$$

$$\frac{\Delta \vdash c :: \Sigma\, u :: \kappa_1.\kappa_2}{\Delta \vdash c \cdot \mathtt{r} :: [c_1/u]\kappa_2} \tag{25.5d}$$

In rule (25.5a), observe that the variable, $u$, may occur in the kind $\kappa_2$ by appearing in a singleton kind. Correspondingly, rules (25.5b), (25.5c), and (25.5d) substitute a constructor for this variable.

Constructor equivalence is defined to be an equivalence relation that is compatible with all forms of constructors and kinds, so that a constructor may always be replaced by an equivalent constructor and the result will be equivalent. The following equivalence axioms govern the constructors associated with the dependent product kind:

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle \cdot \mathtt{l} \equiv c_1 :: \kappa_1} \tag{25.6a}$$

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle \cdot \mathtt{r} \equiv c_2 :: \kappa_2} \tag{25.6b}$$

The subkinding rule for the dependent product kind specifies that it is covariant in both positions:

$$\frac{\Delta \vdash \kappa_1 :<: \kappa_1' \quad \Delta, u :: \kappa_1 \vdash \kappa_2 :<: \kappa_2'}{\Delta \vdash \Sigma\, u :: \kappa_1.\kappa_2 :<: \Sigma\, u :: \kappa_1'.\kappa_2'} \tag{25.7}$$

The congruence rule for equivalence of dependent product kinds is formally similar:

$$\frac{\Delta \vdash \kappa_1 \equiv \kappa_1' \quad \Delta, u :: \kappa_1 \vdash \kappa_2 \equiv \kappa_2'}{\Delta \vdash \Sigma\, u :: \kappa_1.\kappa_2 \equiv \Sigma\, u :: \kappa_1'.\kappa_2'} \tag{25.8}$$

Notable consequences of these rules include the subkindings

$$\Sigma\, u :: \mathtt{S}(\mathtt{int}).\mathtt{S}(u) :<: \Sigma\, u :: \mathtt{T}.\mathtt{S}(u)$$

and

$$\Sigma\, u :: \mathtt{T}.\mathtt{S}(u) :<: \mathtt{T} \times \mathtt{T},$$

and the equivalence

$$\Sigma\, u :: \mathtt{S}(\mathtt{int}).\mathtt{S}(u) \equiv \mathtt{S}(\mathtt{int}) \times \mathtt{S}(\mathtt{int}).$$

Preview

Subkinding is used to "forget" information about the identity of the components of a pair, and equivalence is used to propagate such information within a kind.

The formation, introduction, and elimination rules for dependent function kinds are as follows:

$$\frac{\Delta \vdash \kappa_1 \ \mathsf{kind} \quad \Delta, u :: \kappa_1 \vdash \kappa_2 \ \mathsf{kind}}{\Delta \vdash \prod u :: \kappa_1.\kappa_2 \ \mathsf{kind}} \tag{25.9a}$$

$$\frac{\Delta, u :: \kappa_1 \vdash c :: \kappa_2}{\Delta \vdash \lambda\,(u{::}\kappa_1)\,c :: \prod u :: \kappa_1.\kappa_2} \tag{25.9b}$$

$$\frac{\Delta \vdash c :: \prod u :: \kappa_1.\kappa_2 \quad \Delta \vdash c_1 :: \kappa_1}{\Delta \vdash c[c_1] :: [c_1/u]\kappa_2} \tag{25.9c}$$

Rule (25.9b) specifies that the result kind of a $\lambda$-abstraction depends uniformly on the argument, $u$. Correspondingly, rule (25.9c) specifies that the kind of an application is obtained by substitution of the argument into the result kind of the function itself.

The following rule of equivalence governs the constructors associated with the dependent product kind:

$$\frac{\Delta, u :: \kappa_1 \vdash c :: \kappa_2 \quad \Delta \vdash c_1 :: \kappa_1}{\Delta \vdash (\lambda\,(u{::}\kappa_1)\,c)[c_1] \equiv [c_1/u]c :: \kappa_2} \tag{25.10}$$

The subkinding rule for the dependent function kind specifies that it is contravariant in its domain and covariant in its range:

$$\frac{\Delta \vdash \kappa_1' :<: \kappa_1 \quad \Delta, u :: \kappa_1' \vdash \kappa_2 :<: \kappa_2'}{\Delta \vdash \prod u :: \kappa_1.\kappa_2 :<: \prod u :: \kappa_1'.\kappa_2'} \tag{25.11}$$

The equivalence rule is similar, except that the symmetry of equivalence obviates a choice of variance:

$$\frac{\Delta \vdash \kappa_1 \equiv \kappa_1' \quad \Delta, u :: \kappa_1 \vdash \kappa_2 \equiv \kappa_2'}{\Delta \vdash \prod u :: \kappa_1.\kappa_2 \equiv \prod u :: \kappa_1'.\kappa_2'} \tag{25.12}$$

Rule (25.11) gives rise to the subkinding

$$\prod u :: \mathtt{T}.\mathtt{S}(\mathtt{int}) :<: \prod u :: \mathtt{S}(\mathtt{int}).\mathtt{T},$$

which illustrates the co- and contra-variance of the dependent function kind. In particular a function that takes any type and delivers the type `int` is also a function that takes the type `int` and delivers a type. Rule (25.12) gives rise to the equivalence

$$\prod u :: \mathtt{S}(\mathtt{int}).\mathtt{S}(u) \equiv \mathtt{S}(\mathtt{int}) \to \mathtt{S}(\mathtt{int}),$$

which propagates information about the argument into the range kind. Combining these two rules we may derive the subkinding

$$\prod u :: \mathtt{T}.\mathtt{S}(u) :<: \mathtt{S}(\mathtt{int}) \to \mathtt{S}(\mathtt{int}).$$

Intuitively, a constructor function that yields its argument is, in particular, a constructor function that may only be applied to `int`, and yields `int`. Formally, by contravariance we have the subkinding

$$\prod u :: \mathtt{T}.\mathtt{S}(u) :<: \prod u :: \mathtt{S}(\mathtt{int}).\mathtt{S}(u),$$

and by sharing propagation we may derive the indicated superkind.

Preview

## 25.4 Higher Singletons

Although singletons are restricted to constructors of kind $\mathtt{T}$, we may use dependent product and function kinds to define singletons of every kind. Specifically, we wish to define the kind $\mathtt{S}(c :: \kappa)$, where $c$ is of kind $\kappa$, that classifies constructors equivalent to $c$. When $\kappa = \mathtt{T}$ this is, of course, just $\mathtt{S}(c)$; the problem is to define singletons for the higher kinds $\Sigma\, u :: \kappa_1.\kappa_2$ and $\Pi\, u :: \kappa_1.\kappa_2$.

To see what is involved, suppose that $c :: \kappa_1 \times \kappa_2$. The singleton kind $\mathtt{S}(c :: \kappa_1 \times \kappa_2)$ should classify constructors equivalent to $c$. If we assume, inductively, that singletons have been defined for $\kappa_1$ and $\kappa_2$, then we need only observe that $c$ is equivalent to $\langle c \cdot \mathtt{l}, c \cdot \mathtt{r}\rangle$. For then the singleton $\mathtt{S}(c :: \kappa_1 \times \kappa_2)$ may be defined to be $\mathtt{S}(c \cdot \mathtt{l} :: \kappa_1) \times \mathtt{S}(c \cdot \mathtt{r} :: \kappa_2)$. Similarly, suppose that $c :: \kappa_1 \to \kappa_2$. Using the equivalence of $c$ and $\lambda\,(u{::}\kappa_1 \to \kappa_2)\, c[u]$, we may define $\mathtt{S}(c :: \kappa_1 \to \kappa_2)$ to be $\Pi\, u :: \kappa_1.\mathtt{S}(c[u] :: \kappa_2)$.

In general the kind $\mathtt{S}(c :: \kappa)$ is defined by induction on the structure of $\kappa$ by the following kind equivalences:

$$\frac{\Delta \vdash c :: \mathtt{S}(c')}{\Delta \vdash \mathtt{S}(c :: \mathtt{S}(c')) \equiv \mathtt{S}(c)} \tag{25.13a}$$

$$\frac{\Delta \vdash c :: \Sigma\, u :: \kappa_1.\kappa_2}{\Delta \vdash \mathtt{S}(c :: \Sigma\, u :: \kappa_1.\kappa_2) \equiv \Sigma\, u :: \mathtt{S}(c \cdot \mathtt{l} :: \kappa_1).\mathtt{S}(c \cdot \mathtt{r} :: \kappa_2)} \tag{25.13b}$$

$$\frac{\Delta \vdash c :: \Pi\, u :: \kappa_1.\kappa_2}{\Delta \vdash \mathtt{S}(c :: \Pi\, u :: \kappa_1.\kappa_2) \equiv \Pi\, u :: \kappa_1.\mathtt{S}(c[u] :: \kappa_2)} \tag{25.13c}$$

The sensibility of these equations relies on rule (25.2c) together with the following principles of constructor equivalence, called *extensionality principles*:

$$\frac{\Delta \vdash c :: \Sigma\, u :: \kappa_1.\kappa_2}{\Delta \vdash c \equiv \langle c \cdot \mathtt{l}, c \cdot \mathtt{r}\rangle :: \Sigma\, u :: \kappa_1.\kappa_2} \tag{25.14a}$$

$$\frac{\Delta \vdash c :: \Pi\, u :: \kappa_1.\kappa_2}{\Delta \vdash c \equiv \lambda\,(u{::}\kappa_1)\, c[u] :: \Pi\, u :: \kappa_1.\kappa_2} \tag{25.14b}$$

Rule (25.2c) states that the only constructors of kind $\mathtt{S}(c')$ are those equivalent to $c'$, and rules (25.14a) and (25.14b) state that the only members of the dependent product and function types are, respectively, pairs and $\lambda$-abstractions of the appropriate kinds.

Finally, the following *self-recognition* rules are required to ensure that rule (25.2b) may be extended to higher kinds.

$$\frac{\Delta \vdash c \cdot \mathtt{l} :: \kappa_1 \quad \Delta \vdash c \cdot \mathtt{r} :: [c \cdot \mathtt{l}/u]\kappa_2}{\Delta \vdash c :: \Sigma\, u :: \kappa_1.\kappa_2} \tag{25.15a}$$

$$\frac{\Delta, u :: \kappa_1 \vdash c[u] :: \kappa_2}{\Delta \vdash c :: \Pi\, u :: \kappa_1.\kappa_2} \tag{25.15b}$$

An illustrative case arises when $u$ is a constructor variable of kind $\Sigma\, v :: \mathtt{T}.\mathtt{S}(v)$. We may derive that $u \cdot \mathtt{l} :: \mathtt{S}(u \cdot \mathtt{l})$ using rule (25.2b). We may also derive $u \cdot \mathtt{r} :: \mathtt{S}(u \cdot \mathtt{l})$ using rule (25.5d). Therefore, by rule (25.15a), we may derive $u :: \Sigma\, v :: \mathtt{S}(u \cdot \mathtt{l}).\mathtt{S}(u \cdot \mathtt{l})$, which is a subkind of $\Sigma\, v :: \mathtt{T}.\mathtt{S}(v)$. This more precise kind is a correct kinding for $u$, because the first component of $u$ is indeed $u \cdot \mathtt{l}$, and

Preview

the second component of $u$ is equivalent to the first component, and hence is also $u \cdot \mathtt{1}$. But without rule (25.15a) it is impossible to derive this fact.

The point of introducing higher singletons is to ensure that every constructor may be classified by a kind that determines it up to definitional equality. Viewed as an extension of singleton types, we would expect that higher singletons enjoy similar properties. This is captured by the following lemma:

**Theorem 25.1.** *If* $\Delta \vdash c :: \kappa$, *then* $\Delta \vdash \mathtt{S}(c :: \kappa) :<: \kappa$ *and* $\Delta \vdash c :: \mathtt{S}(c :: \kappa)$.

The proof of this theorem is surprisingly intricate; the reader is referred to the references below for details.

## 25.5 Notes

Singleton kinds were introduced by Stone and Harper (2006) to isolate the concept of type sharing that arises in the ML module system (Milner et al., 1997; Harper and Lillibridge, 1994; Leroy, 1994). Crary (2009) extends the concept of hereditary substitution discussed in Chapter 22 to singleton kinds.

Preview

Preview

# Part X

# Dynamic Dispatch

Preview

Preview

# Chapter 26

# Classes and Methods

It frequently arises that the values of a type are partitioned into a variety of *classes*, each classifying data with distinct internal structure. A simple example is provided by the type of points in the plane, which may be classified according to whether they are represented in cartesian or polar form. Both are represented by a pair of real numbers, but in the cartesian case these are the $x$ and $y$ coordinates of the point, whereas in the polar case these are its distance, $r$, from the origin and its angle, $\theta$, with the polar axis. A classified value is said to be an *object*, or *instance*, of its class. The class determines the type of the classified data, which is called the *instance type* of the class. The classified data itself is called the *instance data* of the object.

Functions that act on classified values are sometimes called *methods*. The behavior of a method is determined by the class of its argument. The method is said to *dispatch* on the class of the argument.[1] Because it happens at run-time, this is called *dynamic dispatch*. For example, the squared distance of a point from the origin is calculated differently according to whether the point is represented in cartesian or polar form. In the former case the required distance is $x^2 + y^2$, whereas in the latter it is simply $r^2$. Similarly, the quadrant of a cartesian point may be determined by examining the sign of its $x$ and $y$ coordinates, and the quadrant of a polar point may be calculated by taking the integral part of the angle $\theta$ divided by $\pi/2$.

Dynamic dispatch is often described in terms of a particular implementation strategy, which we will call the *class-based* organization. In this organization each object is represented by a vector of methods specialized to the class of that object. We may equivalently use a *method-based* organization in which each method branches on the class of an object to determine its behavior. Regardless of the organization used, the fundamental idea is that (a) objects are classified, and (b) methods dispatch on the class of an object. The class-based and method-based organizations are interchangeable, and, in fact, related by a natural duality between sum and product types. We elucidate this symmetry by focusing first on the behavior of each method on each object, which is given by a *dispatch matrix*. From this we derive both a class-based and a method-based organization in such a way that their equivalence is evident.

[1] More generally, we may dispatch on the class of multiple arguments simultaneously. We concentrate on single dispatch for the sake of simplicity.

Preview

## 26.1 The Dispatch Matrix

Because each method acts by dispatch on the class of its argument, we may envision the entire system of classes and methods as a matrix, $e_{\mathsf{dm}}$, called the *dispatch matrix*, whose rows are classes, whose columns are methods, and whose $(c, d)$-entry defines the behavior of method $d$ acting on an argument of class $c$, expressed as a function of the instance data of the object. Thus, the dispatch matrix has a type of the form

$$\prod_{c \in C} \prod_{d \in D} (\tau^c \rightharpoonup \rho_d),$$

where $C$ is the set of class names, $D$ is the set of method names, $\tau^c$ is the instance type associated with class $c$ and $\rho_d$ is the result type of method $d$. The instance type is the same for all methods acting on a given class, and the result type is the same for all classes acted on by a given method.

As an illustrative example, let us consider the type of points in the plane classified into two classes, `cart` and `pol`, corresponding to the cartesian and polar representations. The instance data for a cartesian point has the type

$$\tau^{\mathtt{cart}} = \langle \mathtt{x} \hookrightarrow \mathtt{float}, \mathtt{y} \hookrightarrow \mathtt{float} \rangle,$$

and the instance data for a polar point has the type

$$\tau^{\mathtt{pol}} = \langle \mathtt{r} \hookrightarrow \mathtt{float}, \mathtt{th} \hookrightarrow \mathtt{float} \rangle.$$

Consider two methods acting on points, `dist` and `quad`, which compute, respectively, the squared distance of a point from the origin and the quadrant of a point. The squared distance method is given by the tuple $e_{\mathtt{dist}} = \langle \mathtt{cart} \hookrightarrow e^{\mathtt{cart}}_{\mathtt{dist}}, \mathtt{pol} \hookrightarrow e^{\mathtt{pol}}_{\mathtt{dist}} \rangle$ of type

$$\langle \mathtt{cart} \hookrightarrow \tau^{\mathtt{cart}} \rightharpoonup \rho_{\mathtt{dist}}, \mathtt{pol} \hookrightarrow \tau^{\mathtt{pol}} \rightharpoonup \rho_{\mathtt{dist}} \rangle,$$

where $\rho_{\mathtt{dist}} = \mathtt{float}$ is the result type,

$$e^{\mathtt{cart}}_{\mathtt{dist}} = \lambda\,(u{:}\tau^{\mathtt{cart}})\,(u \cdot \mathtt{x})^2 + (u \cdot \mathtt{y})^2$$

is the squared distance computation for a cartesian point, and

$$e^{\mathtt{pol}}_{\mathtt{dist}} = \lambda\,(v{:}\tau^{\mathtt{pol}})\,(v \cdot \mathtt{r})^2$$

is the squared distance computation for a polar point. Similarly, the quadrant method is given by the tuple $e_{\mathtt{quad}} = \langle \mathtt{cart} \hookrightarrow e^{\mathtt{cart}}_{\mathtt{quad}}, \mathtt{pol} \hookrightarrow e^{\mathtt{pol}}_{\mathtt{quad}} \rangle$ of type

$$\langle \mathtt{cart} \hookrightarrow \tau^{\mathtt{cart}} \rightharpoonup \rho_{\mathtt{quad}}, \mathtt{pol} \hookrightarrow \tau^{\mathtt{pol}} \rightharpoonup \rho_{\mathtt{quad}} \rangle,$$

where $\rho_{\mathtt{quad}} = [\mathtt{I}, \mathtt{II}, \mathtt{III}, \mathtt{IV}]$ is the type of quadrants, and $e^{\mathtt{cart}}_{\mathtt{quad}}$ and $e^{\mathtt{pol}}_{\mathtt{quad}}$ are expressions that compute the quadrant of a point in rectangular and polar forms, respectively.

Now let $C = \{\, \mathtt{cart}, \mathtt{pol} \,\}$ and let $D = \{\, \mathtt{dist}, \mathtt{quad} \,\}$, and define the dispatch matrix, $e_{\mathsf{dm}}$, to be the value of type

$$\prod_{c \in C} \prod_{d \in D} (\tau^c \rightharpoonup \rho_d)$$

Preview

such that, for each class $c$ and method $d$,

$$e_{\mathsf{dm}} \cdot c \cdot d \longmapsto^* e_d^c.$$

That is, the entry in the dispatch matrix, $e_{\mathsf{dm}}$, for class $c$ and method $d$ defines the behavior of that method acting on an object of that class.

Dynamic dispatch is an abstraction given by the following components:

- A type, $\tau_{\mathsf{obj}}$, of *objects*, which are classified by the classes on which the methods act.
- An operation $\mathtt{new}[c](e)$ of type $\tau_{\mathsf{obj}}$ that creates an object of the class $c$ with instance data given by the expression $e$ of type $\tau^c$.
- An operation $e \Leftarrow d$ of type $\rho_d$ that invokes method $d$ on the object given by the expression $e$ of type $\tau_{\mathsf{obj}}$.

These operations are required to satisfy the defining characteristic of dynamic dispatch,

$$(\mathtt{new}[c](e)) \Leftarrow d \longmapsto^* e_d^c(e),$$

which states that invoking method $d$ on an object of class $c$ with instance data $e$ amounts to applying $e_d^c$, the code in the dispatch matrix for class $c$ and method $d$, to the instance data, $e$.

In other words dynamic dispatch is an *abstract type* with interface given by the existential type

$$\exists(t_{\mathtt{obj}}.\langle \mathtt{new} \hookrightarrow \prod_{c \in C} \tau^c \rightharpoonup t_{\mathtt{obj}}, \mathtt{snd} \hookrightarrow \prod_{d \in D} t_{\mathtt{obj}} \rightharpoonup \tau_d \rangle).^{2}$$

There are two main ways to implement this abstract type. One formulation, called the *class-based* organization, defines objects as tuples of methods, and creates objects by specializing the methods to the given instance data. Another formulation, called the *method-based* organization, creates objects by tagging the instance data with the class, and defines methods by examining the class of the object. These two organizations are isomorphic to one another, and hence may be interchanged at will. Nevertheless, many languages favor one representation over the other, asymmetrizing an inherently symmetric situation.

## 26.2 Class-Based Organization

The class-based organization starts with the observation that the dispatch matrix may be reorganized to "factor out" the instance data for each method acting on that class to obtain the *class vector*, $e_{\mathsf{cv}}$, of type

$$\tau_{\mathsf{cv}} \triangleq \prod_{c \in C} (\tau^c \rightharpoonup (\prod_{d \in D} \rho_d)).$$

Each entry of the class vector consists of a *constructor* that determines the result of each of the methods when acting on given instance data.

[2] Thus dynamic dispatch is seen to be an instance of data abstraction, rather than an alternative to it.



Preview

An object has the type $\rho = \prod_{d \in D} \rho_d$ consisting of the product over the methods of the result types of the methods. For example, in the case of points in the plane, the type $\rho$ is the product type

$$\langle \mathtt{dist} \hookrightarrow \rho_{\mathtt{dist}}, \mathtt{quad} \hookrightarrow \rho_{\mathtt{quad}} \rangle.$$

Each component specifies the result of each of the methods acting on that object.

The message send operation, $e \Leftarrow d$, is just the projection $e \cdot d$. So, in the case of points in the plane, $e \Leftarrow \mathtt{dist}$ is the projection $e \cdot \mathtt{dist}$, and similarly $e \Leftarrow \mathtt{quad}$ is the projection $e \cdot \mathtt{quad}$.

The class-based organization consolidates the implementation of each class into a *class vector*, $e_{\mathsf{cv}}$, a tuple of type $\tau_{\mathsf{cv}}$ consisting of the *constructor* of type $\tau^c \rightharpoonup \rho$ for each class $c \in C$. The class vector is defined by $e_{\mathsf{cv}} = \langle c \hookrightarrow e^c \rangle_{c \in C}$, where for each $c \in C$ the expression $e^c$ is

$$\lambda\,(u{:}\tau^c)\,\langle d \hookrightarrow e_{\mathsf{dm}} \cdot c \cdot d(u) \rangle_{d \in D}.$$

For example, the constructor for the class `cart` is the function $e^{\mathtt{cart}}$ given by the expression

$$\lambda\,(u{:}\tau^{\mathtt{cart}})\,\langle \mathtt{dist} \hookrightarrow e_{\mathsf{dm}} \cdot \mathtt{cart} \cdot \mathtt{dist}(u), \mathtt{quad} \hookrightarrow e_{\mathsf{dm}} \cdot \mathtt{cart} \cdot \mathtt{quad}(u) \rangle.$$

Similarly, the constructor for the class `pol` is the function $e^{\mathtt{pol}}$ given by the expression

$$\lambda\,(u{:}\tau^{\mathtt{pol}})\,\langle \mathtt{dist} \hookrightarrow e_{\mathsf{dm}} \cdot \mathtt{pol} \cdot \mathtt{dist}(u), \mathtt{quad} \hookrightarrow e_{\mathsf{dm}} \cdot \mathtt{pol} \cdot \mathtt{quad}(u) \rangle.$$

The class vector, $e_{\mathsf{cv}}$, in this case is the tuple $\langle \mathtt{cart} \hookrightarrow e^{\mathtt{cart}}, \mathtt{pol} \hookrightarrow e^{\mathtt{pol}} \rangle$ of type $\langle \mathtt{cart} \hookrightarrow \tau^{\mathtt{cart}} \rightharpoonup \rho, \mathtt{pol} \hookrightarrow \tau^{\mathtt{pol}} \rightharpoonup \rho \rangle$.

An object of a class is obtained by applying the constructor for that class to the instance data:

$$\mathtt{new}[c](e) \triangleq e_{\mathsf{cv}} \cdot c(e).$$

For example, a cartesian point is obtained by writing $\mathtt{new}[\mathtt{cart}](\langle \mathtt{x} \hookrightarrow x_0, \mathtt{y} \hookrightarrow y_0 \rangle)$, which is defined by the expression

$$e_{\mathsf{cv}} \cdot \mathtt{cart}(\langle \mathtt{x} \hookrightarrow x_0, \mathtt{y} \hookrightarrow y_0 \rangle).$$

Similarly, a polar point is obtained by writing $\mathtt{new}[\mathtt{pol}](\mathtt{r} \hookrightarrow r_0, \mathtt{th} \hookrightarrow \theta_0)$, which is defined by the expression

$$e_{\mathsf{cv}} \cdot \mathtt{pol}(\langle \mathtt{r} \hookrightarrow r_0, \mathtt{th} \hookrightarrow \theta_0 \rangle).$$

It is easy to check for this organization of points that for each class $c$ and method $d$, we may derive

$$\begin{aligned}(\mathtt{new}[c](e)) \Leftarrow d &\longmapsto^* (e_{\mathsf{cv}} \cdot c(e)) \cdot d \\ &\longmapsto^* e_{\mathsf{dm}} \cdot c \cdot d(e).\end{aligned}$$

That is, the message send evokes the behavior of the given method on the instance data of the given object.



Preview

## 26.3 Method-Based Organization

The method-based organization starts with the *transpose* of the dispatch matrix, which has the type

$$\prod_{d \in D} \prod_{c \in C} (\tau^c \rightharpoonup \rho_d).$$

By observing that each row of the transposed dispatch matrix determines a method, we obtain the *method vector*, $e_{\mathsf{mv}}$, of type

$$\tau_{\mathsf{mv}} \triangleq \prod_{d \in D} (\sum_{c \in C} \tau^c) \rightharpoonup \rho_d.$$

Each entry of the method vector consists of a *dispatcher* that determines the result as a function of the instance data associated with a given object.

An object is a value of type $\tau = \sum_{c \in C} \tau^c$, the sum over the classes of the instance types. For example, the type of points in the plane is the sum type

$$[\mathtt{cart} \hookrightarrow \tau^{\mathtt{cart}}, \mathtt{pol} \hookrightarrow \tau^{\mathtt{pol}}].$$

Each point is labeled with its class, specifying its representation as having either cartesian or polar form.

An object of a class $c$ is just the instance data labeled with its class to form an element of the object type:

$$\mathtt{new}[c](e) \triangleq c \cdot e.$$

For example, a cartesian point with coordinates $x_0$ and $y_0$ is given by the expression

$$\mathtt{new}[\mathtt{cart}](\langle \mathtt{x} \hookrightarrow x_0, \mathtt{y} \hookrightarrow y_0 \rangle) \triangleq \mathtt{cart} \cdot \langle \mathtt{x} \hookrightarrow x_0, \mathtt{y} \hookrightarrow y_0 \rangle.$$

Similarly, a polar point with distance $r_0$ and angle $\theta_0$ is given by the expression

$$\mathtt{new}[\mathtt{pol}](\langle \mathtt{r} \hookrightarrow r_0, \mathtt{th} \hookrightarrow \theta_0 \rangle) \triangleq \mathtt{pol} \cdot \langle \mathtt{r} \hookrightarrow r_0, \mathtt{th} \hookrightarrow \theta_0 \rangle.$$

The method-based organization consolidates the implementation of each method into the *method vector*, $e_{\mathsf{mv}}$ of type $\tau_{\mathsf{mv}}$, defined by $\langle d \hookrightarrow e_d \rangle_{d \in D}$, where for each $d \in D$ the expression $e_d : \tau \rightharpoonup \rho_d$ is

$$\lambda\,(\mathit{this}{:}\tau)\ \mathtt{case}\ \mathit{this}\ \{c \cdot u \hookrightarrow e_{\mathsf{dm}} \cdot c \cdot d(u)\}_{c \in C}.$$

Each entry in the method vector may be thought of as a *dispatch function* that determines the action of that method on each class of object.

In the case of points in the plane, the method vector has the product type

$$\langle \mathtt{dist} \hookrightarrow \tau \rightharpoonup \rho_{\mathtt{dist}}, \mathtt{quad} \hookrightarrow \tau \rightharpoonup \rho_{\mathtt{quad}} \rangle.$$

The dispatch function for the `dist` method has the form

$$\lambda\,(\mathit{this}{:}\tau)\ \mathtt{case}\ \mathit{this}\ \{\mathtt{cart} \cdot u \hookrightarrow e_{\mathsf{dm}} \cdot \mathtt{cart} \cdot \mathtt{dist}(u) \mid \mathtt{pol} \cdot v \hookrightarrow e_{\mathsf{dm}} \cdot \mathtt{pol} \cdot \mathtt{dist}(v)\},$$



Preview

and the dispatch function for the quad method has the similar form

$$\lambda\,(\mathit{this}{:}\tau)\,\mathtt{case}\,\mathit{this}\,\{\mathtt{cart}\cdot u \hookrightarrow e_{\mathsf{dm}}\cdot\mathtt{cart}\cdot\mathtt{quad}(u)\mid\mathtt{pol}\cdot v\hookrightarrow e_{\mathsf{dm}}\cdot\mathtt{pol}\cdot\mathtt{quad}(v)\}.$$

The *message send* operation, $e \Leftarrow d$, applies the dispatch function for method $d$ to the object $e$:

$$e \Leftarrow d \triangleq e_{\mathsf{mv}}\cdot d(e).$$

Thus we have, for each class, $c$, and method, $d$,

$$\begin{aligned}(\mathtt{new}[c](e)) \Leftarrow d &\longmapsto^* e_{\mathsf{mv}}\cdot d(c\cdot e)\\ &\longmapsto^* e_{\mathsf{dm}}\cdot c\cdot d(e)\end{aligned}$$

The result is, of course, the same as for the class-based organization.

## 26.4 Self-Reference

It is often useful to allow methods to create new objects or to send messages to objects. This is not possible using the simple dispatch matrix described in Section 26.1, for the simple reason that there is no provision for self-reference within its entries. This deficiency may be remedied by changing the type of the entries of the dispatch matrix to account for sending messages and creating objects, as follows:

$$\prod_{c\in C}\prod_{d\in D}\forall(t_{\mathsf{obj}}.\tau_{\mathsf{cv}}\rightharpoonup\tau_{\mathsf{mv}}\rightharpoonup\tau^c\rightharpoonup\rho_d).$$

The type variable, $t_{\mathsf{obj}}$, represents the abstract object type.[3] The types $\tau_{\mathsf{cv}}$ and $\tau_{\mathsf{mv}}$, are, respectively, the type of the class and method vectors, defined in terms of the abstract type of objects, $t_{\mathsf{obj}}$. They are defined by the equations

$$\tau_{\mathsf{cv}} \triangleq \prod_{c\in C}(\tau^c\rightharpoonup t_{\mathsf{obj}})$$

and

$$\tau_{\mathsf{mv}} \triangleq \prod_{d\in D}(t_{\mathsf{obj}}\rightharpoonup\rho_d).$$

The component of the class vector corresponding to a class, $c$, is a constructor that builds a value of the abstract object type, $t_{\mathsf{obj}}$, from the instance data for $c$. The component of the method vector corresponding to a method, $d$, is a dispatcher that yields a result of type $\rho_d$ when applied to a value of the abstract object type, $t_{\mathsf{obj}}$.

In accordance with the revised type of the dispatch matrix the behavior associated to class $c$ and method $d$ has the form

$$\Lambda(t_{\mathsf{obj}}.\lambda\,(\mathit{cv}{:}\tau_{\mathsf{cv}})\,\lambda\,(\mathit{mv}{:}\tau_{\mathsf{mv}})\,\lambda\,(u{:}\tau^c)\,e_d^c).$$

The arguments *cv* and *mv* are used to create new objects and to send messages to objects. Within the expression $e_d^c$ an object of class $c'$ with instance data $e'$ is created by writing $\mathit{cv}\cdot c'(e')$, which

[3] The variable $t_{\mathsf{obj}}$ is to be chosen so as not to occur in any $\tau^c$ or $\rho_d$. This restriction may be relaxed; see Exercise 26.4.



Preview

selects the appropriate constructor from the class vector, $cv$, and applies it to the given instance data. The class $c'$ may well be the class $c$ itself; this is one form of self-reference within $e_d^c$. Similarly, within $e$ a method $d'$ is invoked on $e'$ by writing $mv \cdot d'(e')$. The method $d'$ may well be the method $d$ itself; this is another aspect of self-reference within $e_d^c$.

To account for self-reference in the method-based organization, the method vector, $e_{\mathsf{mv}}$, will be defined to have the self-referential type $[\tau/t_{\mathsf{obj}}]\tau_{\mathsf{mv}}$ `self` in which the object type, $\tau$, is, as before, the sum of the instance types of the classes, $\sum_{c\in C}\tau^c$. The method vector, $e_{\mathsf{mv}}$, is defined by the following expression:

$$\texttt{self}\ mv\ \texttt{is}\ \langle d \hookrightarrow \lambda\,(\mathit{this}{:}\tau)\ \texttt{case}\ \mathit{this}\ \{c\cdot u \hookrightarrow e_{\mathsf{dm}}\cdot c\cdot d[\tau](e'_{\mathsf{cv}})(e'_{\mathsf{mv}})(u)\}_{c\in\mathbb{C}}\rangle_{d\in D},$$

where

$$e'_{\mathsf{cv}} \triangleq \langle c \hookrightarrow \lambda\,(u{:}\tau^c)\,c\cdot u\rangle_{c\in C} : [\tau/t_{\mathsf{obj}}]\tau_{\mathsf{cv}}.$$

and

$$e'_{\mathsf{mv}} \triangleq \texttt{unroll}(mv) : [\tau/t_{\mathsf{obj}}]\tau_{\mathsf{mv}}.$$

Object creation is defined by the equation

$$\texttt{new}[c](e) \triangleq c\cdot e : \tau$$

and message send is defined by the equation

$$e \Leftarrow d \triangleq \texttt{unroll}(e_{\mathsf{mv}})\cdot d(e) : \rho_d.$$

To account for self-reference in the class-based organization, the class vector, $e_{\mathsf{cv}}$, will be defined to have the type $[\rho/t_{\mathsf{obj}}]\tau_{\mathsf{cv}}$ `self` in which the object type, $\rho$, is, as before, the product of the result types of the methods, $\prod_{d\in D}\rho_d$. The class vector, $e_{\mathsf{cv}}$, is defined by the following expression:

$$\texttt{self}\ cv\ \texttt{is}\ \langle c \hookrightarrow \lambda\,(u{:}\tau^c)\,\langle d \hookrightarrow e_{\mathsf{dm}}\cdot c\cdot d[\rho](e''_{\mathsf{cv}})(e''_{\mathsf{mv}})(u)\rangle_{d\in D}\rangle_{c\in C},$$

where

$$e''_{\mathsf{cv}} \triangleq \texttt{unroll}(cv) : [\rho/t_{\mathsf{obj}}]\tau_{\mathsf{cv}}$$

and

$$e''_{\mathsf{mv}} \triangleq \langle d \hookrightarrow \lambda\,(\mathit{this}{:}\rho)\,\mathit{this}\cdot d\rangle_{d\in D} : [\rho/t_{\mathsf{obj}}]\tau_{\mathsf{mv}}.$$

Object creation is defined by the equation

$$\texttt{new}[c](e) \triangleq \texttt{unroll}(e_{\mathsf{cv}})\cdot c(e) : \rho,$$

and message send is defined by the equation

$$e \Leftarrow d \triangleq e\cdot d : \rho_d.$$

The symmetries between the two organizations are striking. They are a reflection of the fundamental symmetries between sum and product types.

Preview

## 26.5 Notes

The term "object-oriented" means many things to many people, but certainly dynamic dispatch, the association of "methods" with "classes," is one of its central concepts. According to the present development these characteristic features emerge as instances of the more general concepts of sum-, product-, and function types, which are useful, alone and in combination, in a wide variety of circumstances. A bias towards either a class- or method-based organization seems misplaced in view of the inherent symmetries of the situation. The literature on object-oriented programming, of which dynamic dispatch is a major aspect, is extensive. Abadi and Cardelli (1996) and Pierce (2002) provide a thorough account of much of this work.

## Exercises

**26.1**. Consider the possibility that some methods may only be defined on instances of some classes, so that a message send operation may result in a "not understood" error at run-time. Use the type $\tau\,\mathtt{opt}$ defined in Section 11.3.4 to rework the dispatch matrix to account for "not understood" errors. Reformulate the class- and method-based implementations of dynamic dispatch using the revised dispatch matrix representation. Proceed in two stages. In the first stage ignore the possibility of self-reference so that the behavior associated to a method on an instance of a particular class cannot incur a "not understood" error. In the second stage use your solution to the first stage to further rework the dispatch matrix and the implementations of dynamic dispatch to account for the behavior of a method to include incurring a "not understood" error.

**26.2**. Type refinements may be used to ensure the absence of specified "not understood" errors that may otherwise arise in the context of Exercise **26.1**. To do so begin by specifying, for each $c \in C$, a subset $D_c \subseteq D$ of methods that are required to be well-defined on instances of class $c$. This definition determines, for each $d \in D$, the set $C_d \triangleq \{\, c \in C \mid d \in D_c \,\}$ of classes on which method $d$ is required to be well-defined. Using summand refinements for the type $\tau\,\mathtt{opt}$, define the type refinement

$$\phi_{\mathsf{dm}} \triangleq \prod_{c \in C} \Big( \prod_{d \in D_c} \mathtt{just}(\top_{\tau^c} \rightharpoonup \top_{\rho_d}) \Big) \times \Big( \prod_{d \in D \setminus D_c} \top_{(\tau^c \rightharpoonup \rho_d)\,\mathtt{opt}} \Big),$$

which refines the type of dispatch matrix to within a permutation of its columns.[4] It specifies that if $d \in D_c$, then the dispatch matrix entry for class $c$ and method $d$ must be present, and imposes no restriction on any other entry. Assume that $e_{\mathsf{dm}} \in_{\tau_{\mathsf{dm}}} \phi_{\mathsf{dm}}$, in accordance with expectations. For the sake of definiteness, assume a method-based organization in which the object type, $\tau_{\mathsf{obj}}$, is the sum over all classes of their instance types.

[4] Working up to such a permutation is a notational convenience, and can be avoided at the expense of some clarity in the presentation.



Preview

(a) Define the refinements $\texttt{inst}[c]$ and $\texttt{admits}[d]$ of $\tau_{\mathsf{obj}}$ stating that $e \in \tau_{\mathsf{obj}}$ is an instance of class $c \in C$ and that $e$ admits method $d \in D$, respectively. Show that if $d \in D_c$, then $\texttt{inst}[c] \leq \texttt{admits}[d]$, which is to say that any instance of class $c$ admits any method $d$ for that class.

(b) Define $\phi_{\mathsf{cv}} \sqsubseteq \tau_{\mathsf{cv}}$ and $\phi_{\mathsf{mv}} \sqsubseteq \tau_{\mathsf{mv}}$ in terms of $\texttt{inst}[c]$ and $\texttt{admits}[d]$ so that $e_{\mathsf{cv}} \in_{\tau_{\mathsf{cv}}} \phi_{\mathsf{cv}}$ and $e_{\mathsf{mv}} \in_{\tau_{\mathsf{mv}}} \phi_{\mathsf{mv}}$. Remember to use the class- and method vectors derived in Exercise **26.1**.

(c) Referring to the definitions of object creation and message send and of the class- and method vectors derived in Exercise **26.1**, conclude that if message $d \in D_c$ is sent to an instance of $c \in C$, then no "not understood" error can arise at run-time in a well-refined program.

**26.3**. Using self-reference set up a dispatch matrix in which two methods may call one another mutually recursively when invoked on an instance of a class. Specifically, let `num` be a class of numbers with instance type $\tau^{\texttt{num}} = \texttt{nat}$, and let `ev` and `od` be two methods with result type $\rho_{\texttt{ev}} = \rho_{\texttt{od}} = \texttt{bool}$. Define the dispatch entries for methods `ev` and `od` for the class `num` so that they determine, by laborious mutual recursion, whether the instance datum is an even or an odd natural number.

**26.4**. Generalize the account of self-reference to admit constructors whose arguments may involve objects and methods whose results may involve objects. Specifically, allow the abstract object type, $t_{\mathsf{obj}}$, to occur in the instance type $\tau^c$ of a class $c$ or in the result type $\rho_d$ of a method $d$. Rework the development in Section 26.4 to account for this generalization. *Hint*: make use of recursive types as described in Chapter 16.

Preview

Preview

# Chapter 27

# Inheritance

In this chapter we build on Chapter 26 and consider the process of defining the dispatch matrix that determines the behavior of each method on each class. A common strategy is to build the dispatch matrix incrementally by adding new classes or methods to an existing dispatch matrix. To add a class requires that we define the behavior of each method on objects of that class, and to define a method requires that we define the behavior of that method on objects of each of the classes. The definition of these behaviors may be given by any means available in the language. However, it is often suggested that a useful means of defining a new class is to *inherit* the behavior of another class on some methods, and to *override* its behavior on others, resulting in an amalgam of the old and new behaviors. The new class is often called a *subclass* of the old class, which is then called the *superclass*. Similarly, a new method may be defined by inheriting the behavior of another method on some classes, and overriding the behavior on others. By analogy we may call the new method a *submethod* of a given *supermethod*. For the sake of clarity we restrict attention to the non-self-referential case in the following development.

## 27.1 Class and Method Extension

We begin by considering the extension of a given dispatch matrix, $e_{\mathsf{dm}}$, of type

$$\prod_{c \in C} \prod_{d \in D} (\tau^c \to \rho_d)$$

with a new class, $c^* \notin C$, and a new method, $d^* \notin D$, to obtain a new dispatch matrix, $e^*_{\mathsf{dm}}$, of type

$$\prod_{c \in C^*} \prod_{d \in D^*} (\tau^c \to \rho_d),$$

where $C^* = C \cup \{ c^* \}$ and $D^* = D \cup \{ d^* \}$.

To add a new class, $c^*$, to the dispatch matrix, we must specify the following information:[1]

[1] The extension with a new method will be considered separately for the sake of clarity.

Preview

1. The instance type $\tau^{c^*}$ of the new class, $c^*$.
2. The behavior, $e_d^{c^*}$, of each method $d \in D$ on an object of the new class $c^*$, a function of type $\tau^{c^*} \to \rho_d$.

This determines a new dispatch matrix, $e^*_{\mathsf{dm}}$, such that the following conditions are satisfied:

1. For each $c \in C$ and $d \in D$, the behavior $e^*_{\mathsf{dm}} \cdot c \cdot d$ is the same as the behavior $e_{\mathsf{dm}} \cdot c \cdot d$.
2. For each $d \in D$, the behavior $e^*_{\mathsf{dm}} \cdot c^* \cdot d$ is given by $e_d^{c^*}$.

To define $c^*$ as a subclass of some class $c \in C$ means to define the behavior $e_d^{c^*}$ to be $e_d^c$ for some (perhaps many) $d \in D$. It is sensible to inherit a method $d$ in this manner only if the subtype relationship

$$\tau^c \to \rho_d <: \tau^{c^*} \to \rho_d$$

is valid, which will be the case if $\tau^{c^*} <: \tau^c$. This ensures that the inherited behavior may be invoked on the instance data of the new class.

Similarly, to add a new method, $d^*$, to the dispatch matrix, we must specify the following information:

1. The result type, $\rho_{d^*}$, of the new method, $d^*$.
2. The behavior, $e^c_{d^*}$, of the new method, $d^*$, on an object of each class $c \in C$, a function of type $\tau^c \to \rho_{d^*}$.

This determines a new dispatch matrix, $e^*_{\mathsf{dm}}$, such that the following conditions are satisfied:

1. For each $c \in C$ and $d \in D$, the behavior $e^*_{\mathsf{dm}} \cdot c \cdot d$ is the same as $e_{\mathsf{dm}} \cdot c \cdot d$.
2. The behavior $e^*_{\mathsf{dm}} \cdot c \cdot d^*$ is given by $e^c_{d^*}$.

To define $d^*$ as a submethod of some $d \in D$ means to define the behavior $e^c_{d^*}$ to be $e^c_d$ for some (perhaps many) classes $c \in C$. This is only sensible if the subtype relationship

$$\tau^c \to \rho_d <: \tau^c \to \rho_{d^*}$$

holds, which is the case if $\rho_d <: \rho_{d^*}$. This ensures that the result of the old behavior is sufficient for the new behavior.

We will now consider how inheritance relates to the method- and class-based organizations of dynamic dispatch considered in Chapter 26.

## 27.2 Class-Based Inheritance

Recall that the class-based organization given in Chapter 26 consists of a class vector, $e_{\mathsf{cv}}$, of type

$$\tau_{\mathsf{cv}} \triangleq \prod_{c \in C} (\tau^c \to \rho),$$



Preview

where the object type, $\rho$, is the finite product type $\prod_{d \in D} \rho_d$. The class vector consists of a tuple of constructors that specialize the methods to a given object of each class.

Let us consider the effect of adding a new class, $c^*$, as described in Section 27.1. The new class vector, $e^*_{\mathsf{cv}}$, has type

$$\tau^*_{\mathsf{cv}} \triangleq \prod_{c \in C^*} (\tau^c \to \rho).$$

There is an isomorphism, written $(\,)^\dagger$, between $\tau^*_{\mathsf{cv}}$ and the type

$$\tau_{\mathsf{cv}} \times (\tau^{c^*} \to \rho),$$

which may be used to define the new class vector, $e^*_{\mathsf{cv}}$, as follows:

$$\langle e_{\mathsf{cv}}, \lambda\,(u{:}\tau^{c^*})\,\langle d \hookrightarrow e_d^{c^*}(u)\rangle_{d \in D}\rangle^\dagger.$$

This definition makes clear that the old class vector, $e_{\mathsf{cv}}$, is reused intact in the new class vector, which is just an extension of the old class vector with a new constructor.

Although the object type, $\rho$, is the same both before and after the extension with the new class, the behavior of an object of class $c^*$ may differ arbitrarily from that of any other object, even that of the superclass from which it inherits its behavior. So, knowing that $c^*$ inherits from $c$ tells us nothing about the behavior of its objects, but only about the means by which the class is defined. Inheritance carries no semantic significance, but is only a record of the history of how a class is defined.

Now let us consider the effect of adding a new method, $d^*$, as described in Section 27.1. The new class vector, $e^*_{\mathsf{cv}}$, has type

$$\tau^*_{\mathsf{cv}} \triangleq \prod_{c \in C} (\tau^c \to \rho^*),$$

where $\rho^*$ is the product type $\prod_{d \in D^*} \rho_d$. There is an isomorphism, written $(\,)^\ddagger$, between $\rho^*$ and the type $\rho \times \rho_{d^*}$, where $\rho$ is the old object type. Using this the new class vector, $e^*_{\mathsf{cv}}$, may be defined by

$$\langle c \hookrightarrow \lambda\,(u{:}\tau^c)\,\langle\langle d \hookrightarrow ((e_{\mathsf{cv}} \cdot c)(u)) \cdot d\rangle_{d \in D}, e^c_{d^*}(u)\rangle^\ddagger\rangle_{c \in C}.$$

Observe that each constructor must be re-defined to account for the new method, but the definition makes use of the old class vector for the definitions of the old methods.

By this construction the new object type, $\rho^*$, is a subtype of the old object type, $\rho$. This means that any objects with the new method may be used in situations expecting an object without the new method, as might be expected. To avoid the redefinition of old classes when a new method is introduced, we may restrict inheritance so that new methods are only added to new subclasses. Subclasses may then have more methods than superclasses, and objects of the subclass may be provided when an object of the superclass is required.

## 27.3 Method-Based Inheritance

The situation with the method-based organization is dual to that of the class-based organization. Recall that the method-based organization given in Chapter 26 consists of a method vector, $e_{\mathsf{mv}}$, of



Preview

type

$$\tau_{\mathsf{mv}} \triangleq \prod_{d \in D} \tau \to \rho_d,$$

where the instance type, $\tau$, is the sum type $\sum_{c \in C} \tau^c$. The method vector consists of a tuple of functions that dispatch on the class of the object to determine their behavior.

Let us consider the effect of adding a new method, $d^*$, as described in Section 27.1. The new method vector, $e^*_{\mathsf{mv}}$, has type

$$\tau^*_{\mathsf{mv}} \triangleq \prod_{d \in D^*} \tau \to \rho_d.$$

There is an isomorphism, written $(\ )^\ddagger$, between $\tau^*_{\mathsf{mv}}$ and the type

$$\tau_{\mathsf{mv}} \times (\tau \to \rho_{d^*}).$$

Using this isomorphism, the new method vector, $e^*_{\mathsf{mv}}$, may be defined as

$$\langle e_{\mathsf{mv}}, \lambda\,(\mathit{this}{:}\tau)\ \mathtt{case}\ \mathit{this}\ \{c \cdot u \hookrightarrow e^c_{d^*}(u)\}_{c \in C} \rangle^\ddagger.$$

The old method vector is re-used intact, extended with an additional dispatch function for the new method.

The object type does not change under the extension with a new method, but because $\rho^* <: \rho$, there is no difficulty using a new object in a context expecting an old object—the additional method is ignored.

Finally, let us consider the effect of adding a new class, $c^*$, as described in Section 27.1. The new method vector, $e^*_{\mathsf{mv}}$, has the type

$$\tau^*_{\mathsf{mv}} \triangleq \prod_{d \in D} \tau^* \to \rho_d,$$

where $\tau^*$ is the new object type $\sum_{c \in C^*} \tau^c$, which is a supertype of the old object type $\tau$. There is an isomorphism, written $(\ )^\dagger$, between $\tau^*$ and the sum type $\tau + \tau^{c^*}$, which we may use to define the new method vector, $e^*_{\mathsf{mv}}$, as follows:

$$\langle d \hookrightarrow \lambda\,(\mathit{this}{:}\tau^*)\ \mathtt{case}\ \mathit{this}^\dagger\ \{\mathtt{l} \cdot u \hookrightarrow (e_{\mathsf{mv}} \cdot d)(u) \mid \mathtt{r} \cdot u \hookrightarrow e^{c^*}_d(u)\}\rangle_{d \in D}.$$

Every method must be redefined to account for the new class, but the old method vector is reused in this definition.

## 27.4 Notes

Advocates of object-oriented programming differ on the importance of inheritance. Philosophers tend to stress inheritance, but practitioners commonly avoid it, or reduce it to vestigial form. The most common restrictions amount to a reformulation of some of the modularity mechanisms that are discussed in Chapters 46 and 47.

Abadi and Cardelli (1996) and Pierce (2002) provide thorough accounts of the interaction of inheritance and subtyping. Liskov and Wing (1994) discuss it from a behavioral perspective. They propose the methodological requirement that subclasses respect the behavior of the superclass whenever inheritance is used.

Preview

## Exercises

**27.1**. Consider the case of extending a dispatch matrix with self-reference by a new class, $c^*$, in which a method, $d$, is to be inherited from an existing class, $c$. What requirements must be made to ensure that such an inheritance is properly defined? What happens if we extend a self-referential dispatch matrix with a new method, $d^*$, that inherits its behavior on class $c$ from another method $d$?

**27.2**. Consider the example of two mutually recursive methods given in Exercise **26.3**. Suppose that `num*` is to be a new class with instance type $\tau^{\texttt{num*}} <: \tau^{\texttt{num}}$ that inherits the `ev` method from `num`, but defines its own version of the `od` method. What happens when message `ev` is sent to an instance of `num*`? Will the revised `od` method ever be invoked?

**27.3**. *Method specialization* consists of defining a new class by inheriting methods from another class or classes, while redefining some of the methods that the inherited methods might invoke. The behavior of the inherited methods on instances of the new class is altered to the extent that they invoke a method that is specialized to the new class. Reconsider Exercise **26.3** in light of Exercise **27.2**, seeking to ensure that the specialization of `od` is invoked whenever the inherited method `ev` is invoked on instances of the new class.

(a) Redefine the class `num` along the following lines. The instance data of `num` is an object admitting methods `ev` and `od`. The class `num` admits these methods, and simply hands them off to the instance object.

(b) The classes `zero` or of `succ` admit both the `ev` and `od` methods, and are defined using message send to effect mutual recursion as necessary.

(c) Define a subclass `succ*` of `succ` that overrides the `od` method. Show that `ev` on an instance of `succ*` correctly invokes the overridden `od` method.

Preview

Preview

# Part XI

# Exceptions and Continuations

Preview

Preview

# Chapter 28

# Control Stacks

The technique of structural dynamics is very useful for theoretical purposes, such as proving type safety, but is too high level to be directly usable in an implementation. One reason is that the use of "search rules" requires the traversal and reconstruction of an expression in order to simplify one small part of it. In an implementation we would prefer to use some mechanism to record "where we are" in the expression so that we may resume from that point after a simplification. This can be achieved by introducing an explicit mechanism, called a *control stack*, that keeps track of the context of an instruction step for just this purpose. By making the control stack explicit, the transition rules avoid the need for any premises—every rule is an axiom. This is the formal expression of the informal idea that no traversals or reconstructions are required to implement it. In this chapter we introduce an abstract machine for **PCF**. The purpose of this machine is to make control flow explicit by introducing a control stack that maintains a record of the pending sub-computations of a computation. We then prove the equivalence of the structural dynamics and machine dynamics of **PCF**.

## 28.1 Machine Definition

A state, $s$, of the **PCF** machine consists of a *control stack*, $k$, and a closed expression, $e$. States may take one of two forms:

1. An *evaluation* state of the form $k \triangleright e$ corresponds to the evaluation of a closed expression, $e$, relative to a control stack, $k$.

2. A *return* state of the form $k \triangleleft e$, where $e$ $\mathsf{val}$, corresponds to the evaluation of a stack, $k$, relative to a closed value, $e$.

As an aid to memory, note that the separator "points to" the focal entity of the state, the expression in an evaluation state and the stack in a return state.

The control stack represents the context of evaluation. It records the "current location" of evaluation, the context into which the value of the current expression is to be returned. Formally, a

Preview

control stack is a list of *frames*:

$$\frac{}{\epsilon\ \mathsf{stack}} \tag{28.1a}$$

$$\frac{f\ \mathsf{frame} \quad k\ \mathsf{stack}}{k;f\ \mathsf{stack}} \tag{28.1b}$$

The definition of frame depends on the language we are evaluating. The frames of the **PCF** machine are inductively defined by the following rules:

$$\frac{}{\mathtt{s}(-)\ \mathsf{frame}} \tag{28.2a}$$

$$\frac{}{\mathtt{ifz}(-;e_0;x.e_1)\ \mathsf{frame}} \tag{28.2b}$$

$$\frac{}{\mathtt{ap}(-;e_2)\ \mathsf{frame}} \tag{28.2c}$$

The frames correspond to search rules in the dynamics of **PCF**. Thus, instead of relying on the structure of the transition derivation to maintain a record of pending computations, we make an explicit record of them in the form of a frame on the control stack.

The transition judgment between states of the **PCF** machine is inductively defined by a set of inference rules. We begin with the rules for natural numbers, using an eager semantics for the successor.

$$\frac{}{k \rhd \mathtt{z} \longmapsto k \lhd \mathtt{z}} \tag{28.3a}$$

$$\frac{}{k \rhd \mathtt{s}(e) \longmapsto k;\mathtt{s}(-) \rhd e} \tag{28.3b}$$

$$\frac{}{k;\mathtt{s}(-) \lhd e \longmapsto k \lhd \mathtt{s}(e)} \tag{28.3c}$$

To evaluate $\mathtt{z}$ we simply return it. To evaluate $\mathtt{s}(e)$, we push a frame on the stack to record the pending successor, and evaluate $e$; when that returns with $e'$, we return $\mathtt{s}(e')$ to the stack.

Next, we consider the rules for case analysis.

$$\frac{}{k \rhd \mathtt{ifz}(e;e_0;x.e_1) \longmapsto k;\mathtt{ifz}(-;e_0;x.e_1) \rhd e} \tag{28.4a}$$

$$\frac{}{k;\mathtt{ifz}(-;e_0;x.e_1) \lhd \mathtt{z} \longmapsto k \rhd e_0} \tag{28.4b}$$

$$\frac{}{k;\mathtt{ifz}(-;e_0;x.e_1) \lhd \mathtt{s}(e) \longmapsto k \rhd [e/x]e_1} \tag{28.4c}$$

Preview

First, the test expression is evaluated, recording the pending case analysis on the stack. Once the value of the test expression has been determined, we branch to the appropriate arm of the conditional, substituting the predecessor in the case of a positive number.

Finally, we consider the rules for functions, which we evaluate by-name, and for general recursion.

$$\overline{k \rhd \mathtt{lam}[\tau](x.e) \longmapsto k \lhd \mathtt{lam}[\tau](x.e)} \tag{28.5a}$$

$$\overline{k \rhd \mathtt{ap}(e_1; e_2) \longmapsto k;\mathtt{ap}(-; e_2) \rhd e_1} \tag{28.5b}$$

$$\overline{k;\mathtt{ap}(-; e_2) \lhd \mathtt{lam}[\tau](x.e) \longmapsto k \rhd [e_2/x]e} \tag{28.5c}$$

$$\overline{k \rhd \mathtt{fix}[\tau](x.e) \longmapsto k \rhd [\mathtt{fix}[\tau](x.e)/x]e} \tag{28.5d}$$

It is important to note that general recursion requires no stack space to evaluate!

The initial and final states of the **PCF** machine are defined by the following rules:

$$\frac{}{\epsilon \rhd e \text{ initial}} \tag{28.6a}$$

$$\frac{e \text{ val}}{\epsilon \lhd e \text{ final}} \tag{28.6b}$$

## 28.2 Safety

To define and prove safety for the **PCF** machine requires that we introduce a new typing judgment, $k \lhd: \tau$, which states that the stack $k$ expects a value of type $\tau$. This judgment is inductively defined by the following rules:

$$\frac{}{\epsilon \lhd: \tau} \tag{28.7a}$$

$$\frac{k \lhd: \tau' \quad f : \tau \rightsquigarrow \tau'}{k;f \lhd: \tau} \tag{28.7b}$$

This definition makes use of an auxiliary judgment, $f : \tau \rightsquigarrow \tau'$, stating that a frame $f$ transforms a value of type $\tau$ to a value of type $\tau'$.

$$\overline{\mathtt{s}(-) : \mathtt{nat} \rightsquigarrow \mathtt{nat}} \tag{28.8a}$$

$$\frac{e_0 : \tau \quad x : \mathtt{nat} \vdash e_1 : \tau}{\mathtt{ifz}(-; e_0; x.e_1) : \mathtt{nat} \rightsquigarrow \tau} \tag{28.8b}$$

$$\frac{e_2 : \tau_2}{\mathtt{ap}(-; e_2) : \mathtt{parr}(\tau_2; \tau) \rightsquigarrow \tau} \tag{28.8c}$$

Preview

The states of the **PCF** machine are well-formed if their stack and expression components match:

$$\frac{k \lhd : \tau \quad e : \tau}{k \rhd e \ \mathsf{ok}} \tag{28.9a}$$

$$\frac{k \lhd : \tau \quad e : \tau \quad e \ \mathsf{val}}{k \lhd e \ \mathsf{ok}} \tag{28.9b}$$

We leave the proof of safety of the **PCF** machine as an exercise.

**Theorem 28.1** (Safety)**.** *1. If $s$ ok and $s \longmapsto s'$, then $s'$ ok.*

*2. If $s$ ok, then either $s$ final or there exists $s'$ such that $s \longmapsto s'$.*

## 28.3 Correctness of the Control Machine

If we evaluate a given expression, $e$, using the **PCF** machine, do we get the same result as would be given by the structural dynamics of **PCF**, and *vice versa*?

Answering this question decomposes into two conditions relating the machine to the structural dynamics for **PCF**:

**Completeness** If $e \longmapsto^* e'$, where $e'$ val, then $\epsilon \rhd e \longmapsto^* \epsilon \lhd e'$.

**Soundness** If $\epsilon \rhd e \longmapsto^* \epsilon \lhd e'$, then $e \longmapsto^* e'$ with $e'$ val.

Let us consider, in turn, what is involved in the proof of each part.

For completeness a plausible first step is to consider a proof by induction on the definition of multistep transition, which reduces the theorem to the following two lemmas:

1. If $e$ val, then $\epsilon \rhd e \longmapsto^* \epsilon \lhd e$.
2. If $e \longmapsto e'$, then, for every $v$ val, if $\epsilon \rhd e' \longmapsto^* \epsilon \lhd v$, then $\epsilon \rhd e \longmapsto^* \epsilon \lhd v$.

The first can be proved easily by induction on the structure of $e$. The second requires an inductive analysis of the derivation of $e \longmapsto e'$, giving rise to two complications that must be accounted for in the proof. The first complication is that we cannot restrict attention to the empty stack, for if $e$ is, say, $\mathtt{ap}(e_1; e_2)$, then the first step of the machine is

$$\epsilon \rhd \mathtt{ap}(e_1; e_2) \longmapsto \epsilon; \mathtt{ap}(-; e_2) \rhd e_1,$$

and so we must consider evaluation of $e_1$ on a non-empty stack.

A generalization is to prove that if $e \longmapsto e'$ and $k \rhd e' \longmapsto^* k \lhd v$, then $k \rhd e \longmapsto^* k \lhd v$. Consider again the case $e = \mathtt{ap}(e_1; e_2)$, $e' = \mathtt{ap}(e_1'; e_2)$, with $e_1 \longmapsto e_1'$. We are given that $k \rhd \mathtt{ap}(e_1'; e_2) \longmapsto^* k \lhd v$, and we are to show that $k \rhd \mathtt{ap}(e_1; e_2) \longmapsto^* k \lhd v$. It is easy to show that the first step of the former derivation is

$$k \rhd \mathtt{ap}(e_1'; e_2) \longmapsto k; \mathtt{ap}(-; e_2) \rhd e_1'.$$

We would like to apply induction to the derivation of $e_1 \longmapsto e_1'$, but to do so we must have a $v_1$ such that $e_1' \longmapsto^* v_1$, which is not immediately at hand.



Preview

This means that we must consider the ultimate value of each sub-expression of an expression in order to complete the proof. This information is provided by the evaluation dynamics described in Chapter 7, which has the property that $e \Downarrow e'$ iff $e \longmapsto^* e'$ and $e'$ val.

**Lemma 28.2.** *If $e \Downarrow v$, then for every $k$ stack, $k \triangleright e \longmapsto^* k \triangleleft v$.*

The desired result follows by the analogue of Theorem 7.2 for **PCF**, which states that $e \Downarrow v$ iff $e \longmapsto^* v$.

For the proof of soundness, it is awkward to reason inductively about the multistep transition from $\epsilon \triangleright e \longmapsto^* \epsilon \triangleleft v$, because the intervening steps may involve alternations of evaluation and return states. Instead we regard each **PCF** machine state as encoding an expression, and show that machine transitions are simulated by transitions of the structural dynamics of **PCF** under this encoding.

Specifically, we define a judgment, $s \leftarrow e$, stating that state $s$ "unravels to" expression $e$. It will turn out that for initial states, $s = \epsilon \triangleright e$, and final states, $s = \epsilon \triangleleft e$, we have $s \leftarrow e$. Then we show that if $s \longmapsto^* s'$, where $s'$ final, $s \leftarrow e$, and $s' \leftarrow e'$, then $e'$ val and $e \longmapsto^* e'$. For this it is enough to show the following two facts:

1. If $s \leftarrow e$ and $s$ final, then $e$ val.
2. If $s \longmapsto s'$, $s \leftarrow e$, $s' \leftarrow e'$, and $e' \longmapsto^* v$, where $v$ val, then $e \longmapsto^* v$.

The first is quite simple, we need only observe that the unravelling of a final state is a value. For the second, it is enough to show the following lemma.

**Lemma 28.3.** *If $s \longmapsto s'$, $s \leftarrow e$, and $s' \leftarrow e'$, then $e \longmapsto^* e'$.*

**Corollary 28.4.** *$e \longmapsto^* \overline{n}$ iff $\epsilon \triangleright e \longmapsto^* \epsilon \triangleleft \overline{n}$.*

The remainder of this section is devoted to the proofs of the soundness and completeness lemmas.

### 28.3.1 Completeness

*Proof of Lemma 28.2.* The proof is by induction on an evaluation dynamics for **PCF**.

Consider the evaluation rule

$$\frac{e_1 \Downarrow \mathtt{lam}[\tau_2](x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1; e_2) \Downarrow v} \tag{28.10}$$

For an arbitrary control stack, $k$, we are to show that $k \triangleright \mathtt{ap}(e_1; e_2) \longmapsto^* k \triangleleft v$. Applying both of the inductive hypotheses in succession, interleaved with steps of the abstract machine, we obtain

$$\begin{aligned} k \triangleright \mathtt{ap}(e_1; e_2) &\longmapsto k;\mathtt{ap}(-; e_2) \triangleright e_1 \\ &\longmapsto^* k;\mathtt{ap}(-; e_2) \triangleleft \mathtt{lam}[\tau_2](x.e) \\ &\longmapsto k \triangleright [e_2/x]e \\ &\longmapsto^* k \triangleleft v. \end{aligned}$$

The other cases of the proof are handled similarly. □

Preview

### 28.3.2 Soundness

The judgment $s \mapsto e'$, where $s$ is either $k \triangleright e$ or $k \triangleleft e$, is defined in terms of the auxiliary judgment $k \bowtie e = e'$ by the following rules:

$$\frac{k \bowtie e = e'}{k \triangleright e \mapsto e'} \tag{28.11a}$$

$$\frac{k \bowtie e = e'}{k \triangleleft e \mapsto e'} \tag{28.11b}$$

In words, to unravel a state we wrap the stack around the expression. The latter relation is inductively defined by the following rules:

$$\frac{}{\epsilon \bowtie e = e} \tag{28.12a}$$

$$\frac{k \bowtie \mathtt{s}(e) = e'}{k;\mathtt{s}(-) \bowtie e = e'} \tag{28.12b}$$

$$\frac{k \bowtie \mathtt{ifz}(e; e_0; x.e_1) = e'}{k;\mathtt{ifz}(-; e_0; x.e_1) \bowtie e = e'} \tag{28.12c}$$

$$\frac{k \bowtie \mathtt{ap}(e_1; e_2) = e}{k;\mathtt{ap}(-; e_2) \bowtie e_1 = e} \tag{28.12d}$$

These judgments both define total functions.

**Lemma 28.5.** *The judgment $s \mapsto e$ has mode $(\forall, \exists!)$, and the judgment $k \bowtie e = e'$ has mode $(\forall, \forall, \exists!)$.*

That is, each state unravels to a unique expression, and the result of wrapping a stack around an expression is uniquely determined. We are therefore justified in writing $k \bowtie e$ for the unique $e'$ such that $k \bowtie e = e'$.

The following lemma is crucial. It states that unravelling preserves the transition relation.

**Lemma 28.6.** *If $e \longmapsto e'$, $k \bowtie e = d$, $k \bowtie e' = d'$, then $d \longmapsto d'$.*

*Proof.* The proof is by rule induction on the transition $e \longmapsto e'$. The inductive cases, in which the transition rule has a premise, follow easily by induction. The base cases, in which the transition is an axiom, are proved by an inductive analysis of the stack, $k$.

For an example of an inductive case, suppose that $e = \mathtt{ap}(e_1; e_2)$, $e' = \mathtt{ap}(e_1'; e_2)$, and $e_1 \longmapsto e_1'$. We have $k \bowtie e = d$ and $k \bowtie e' = d'$. It follows from rules (28.12) that $k;\mathtt{ap}(-; e_2) \bowtie e_1 = d$ and $k;\mathtt{ap}(-; e_2) \bowtie e_1' = d'$. So by induction $d \longmapsto d'$, as desired.

For an example of a base case, suppose that $e = \mathtt{ap}(\mathtt{lam}[\tau_2](x.e); e_2)$ and $e' = [e_2/x]e$ with $e \longmapsto e'$ directly. Assume that $k \bowtie e = d$ and $k \bowtie e' = d'$; we are to show that $d \longmapsto d'$. We proceed by an inner induction on the structure of $k$. If $k = \epsilon$, the result follows immediately. Consider, say, the stack $k = k';\mathtt{ap}(-; c_2)$. It follows from rules (28.12) that $k' \bowtie \mathtt{ap}(e; c_2) = d$ and $k' \bowtie \mathtt{ap}(e'; c_2) = d'$. But by the structural dynamics $\mathtt{ap}(e; c_2) \longmapsto \mathtt{ap}(e'; c_2)$, so by the inner inductive hypothesis we have $d \longmapsto d'$, as desired. □

We are now in a position to complete the proof of Lemma 28.3.



Preview

*Proof of Lemma 28.3.* The proof is by case analysis on the transitions of the **PCF** machine. In each case, after unravelling, the transition will correspond to zero or one transitions of **PCF**.

Suppose that $s = k \triangleright \mathtt{s}(e)$ and $s' = k;\mathtt{s}(-) \triangleright e$. Note that $k \bowtie \mathtt{s}(e) = e'$ iff $k;\mathtt{s}(-) \bowtie e = e'$, from which the result follows immediately.

Suppose that $s = k;\mathtt{ap}(\mathtt{lam}[\tau](x.e_1);-) \triangleleft e_2$ and $s' = k \triangleright [e_2/x]e_1$. Let $e'$ be such that $k;\mathtt{ap}(\mathtt{lam}[\tau](x.e_1);-) \bowtie e_2 = e'$ and let $e''$ be such that $k \bowtie [e_2/x]e_1 = e''$. Observe that $k \bowtie \mathtt{ap}(\mathtt{lam}[\tau](x.e_1);e_2) = e'$. The result follows from Lemma 28.6.

□

## 28.4 Notes

The abstract machine considered here is typical of a wide class of machines that make control flow explicit in the state. The prototype is the SECD machine (Landin, 1965), which may be seen as a linearization of a structural operational semantics (Plotkin, 1981). An advantage of a machine model is that the explicit treatment of control is required for languages that allow the control state to be explicitly manipulated (see Chapter 30 for a prime example). A disadvantage is that we are required to make explicit the control state of the computation, rather than leave it implicit as in structural operational semantics. Which is better depends wholly on the situation at hand, though historically there has been greater emphasis on abstract machines than on structural semantics.

## Exercises

**28.1**. Give the proof of Theorem 28.1 for conditional expressions.

**28.2**. Formulate a call-by-value variant of the **PCF** machine by giving the necessary modifications to the dynamics.

**28.3**. Examine the dynamics of the **PCF** machine and determine the asymptotic worst-case complexity of executiong each abstract machine instruction.

**28.4**. Refine the proof of Lemma 28.2 by bounding the number of machine steps taken for each step of the **PCF** dynamics.

Preview

Preview

# Chapter 29

# Exceptions

Exceptions effect a non-local transfer of control from the point at which the exception is *raised* to an enclosing *handler* for that exception. This transfer interrupts the normal flow of control in a program in response to unusual conditions. For example, exceptions can be used to signal an error condition, or to indicate the need for special handling in certain circumstances that arise only rarely. To be sure, we could use conditionals to check for and process errors or unusual conditions, but using exceptions is often more convenient, particularly because the transfer to the handler is direct and immediate, rather than indirect via a series of explicit checks.

In this chapter we will consider three extensions of **PCF** with exceptions. The first, **FPCF**, enriches **PCF** with the simplest form of exception, called a *failure*, with no associated data. A failure may be intercepted, and turned into a success (or another failure!) by transferring control to another expression. The second, **XPCF**, enriches **PCF** with *exceptions*, with associated data that is passed to an exception handler that intercepts it. The handler may analyze the associated data to determine how to recover from the exceptional condition. A key choice is to decide on the type of the data associated to an exception. The third, **MPCF**, is a variant of **FPCF** in which we draw a *modal distinction* between *expressions*, which may not raise an exception, and *commands*, which might do so. Execution of commands is explicitly sequentialized to ensure that there is no ambiguity about the order in which exceptions are raised.

## 29.1 Simple Failures

The syntax of **FPCF** is defined by the following extension of the grammar of **PCF**:

| | | | | | |
|---|---|---|---|---|---|
| Exp | $e$ | ::= | `fail` | `fail` | signal a failure |
| | | | `catch`$(e_1;e_2)$ | `catch` $e_1$ `ow` $e_2$ | catch a failure |

The expression `fail` aborts the current evaluation, and the expression `catch`$(e_1;e_2)$ catches any failure in $e_1$ by evaluating $e_2$ instead. Either $e_1$ or $e_2$ may themselves abort, or they may diverge or return a value as usual in **PCF**.

Preview

The statics of **FPCF** is given by these rules:

$$\frac{}{\Gamma \vdash \mathtt{fail} : \tau} \tag{29.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{catch}(e_1; e_2) : \tau} \tag{29.1b}$$

A failure can have any type, because it never returns. The two expressions in a `catch` expression must have the same type, because either might determine the value of that expression.

The dynamics of **FPCF** is given using a technique called *stack unwinding*. Evaluation of a `catch` pushes a frame of the form $\mathtt{catch}(-; e)$ onto the control stack that awaits the arrival of a failure. Evaluation of a `fail` expression pops frames from the control stack until it reaches a frame of the form $\mathtt{catch}(-; e)$, at which point the frame is removed from the stack and the expression $e$ is evaluated. Failure propagation is expressed by a state of the form $k \blacktriangleleft$, which extends the two forms of state considered in Chapter 28 to express failure propagation.

The **FPCF** machine extends the **PCF** machine with the following additional rules:

$$\frac{}{k \triangleright \mathtt{fail} \longmapsto k \blacktriangleleft} \tag{29.2a}$$

$$\frac{}{k \triangleright \mathtt{catch}(e_1; e_2) \longmapsto k; \mathtt{catch}(-; e_2) \triangleright e_1} \tag{29.2b}$$

$$\frac{}{k; \mathtt{catch}(-; e_2) \triangleleft v \longmapsto k \triangleleft v} \tag{29.2c}$$

$$\frac{}{k; \mathtt{catch}(-; e_2) \blacktriangleleft \longmapsto k \triangleright e_2} \tag{29.2d}$$

$$\frac{(f \neq \mathtt{catch}(-; e))}{k; f \blacktriangleleft \longmapsto k \blacktriangleleft} \tag{29.2e}$$

Evaluating `fail` propagates a failure up the stack. The act of failing itself, `fail`, will, of course, give rise to a failure. Evaluating $\mathtt{catch}(e_1; e_2)$ consists of pushing the handler onto the control stack and evaluating $e_1$. If a value is propagated to the handler, the handler is removed and the value continues to propagate upwards. If a failure is propagated to the handler, the stored expression is evaluated with the handler removed from the control stack. Failures propagate through all frames other than the `catch` frame.

The initial and final states of the **FPCF** machine are defined by the following rules:

$$\frac{}{\epsilon \ \mathsf{initial} e} \tag{29.3a}$$

$$\frac{e \ \mathsf{val}}{\epsilon \triangleleft e \ \mathsf{final}} \tag{29.3b}$$



Preview

$$\frac{}{\epsilon \blacktriangleleft \mathsf{final}} \tag{29.3c}$$

It is easy to extend the definition of stack typing given in Chapter 28 to account for the new forms of frame, and then to prove safety in the same manner. The only difference is that the statement of progress must be weakened to take account of failure: a well-typed expression is either a value, or may take a step, or may signal failure.

**Theorem 29.1** (Safety for **FPCF**). *1. If $s$ ok and $s \longmapsto s'$, then $s'$ ok.*

*2. If $s$ ok, then either $s$ final or there exists $s'$ such that $s \longmapsto s'$.*

## 29.2 Value-Carrying Exceptions

The language **XPCF** enriches **FPCF** with *exceptions*, failures to which a value is attached. The syntax of **XPCF** extends that of **PCF** with the following forms of expression:

| | | | | | |
|---|---|---|---|---|---|
| Exp | $e$ | ::= | $\mathtt{raise}[\tau](e)$ | $\mathtt{raise}(e)$ | raise an exception |
| | | | $\mathtt{handle}(e_1; x.e_2)$ | $\mathtt{handle}\, e_1\, \mathtt{ow}\, x \hookrightarrow e_2$ | handler an exception |

The argument to `raise` is evaluated to determine the value passed to the handler. The expression $\mathtt{handle}(e_1; x.e_2)$ binds a variable, $x$, in the handler, $e_2$. The associated value of the exception is bound to that variable within $e_2$, should an exception be raised during the execution of $e_1$.

The statics of exceptions extends the statics of failures to account for the type of the value carried with the exception:

$$\frac{\Gamma \vdash e : \tau_{\mathsf{exn}}}{\Gamma \vdash \mathtt{raise}[\tau](e) : \tau} \tag{29.4a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{\mathsf{exn}} \vdash e_2 : \tau}{\Gamma \vdash \mathtt{handle}(e_1; x.e_2) : \tau} \tag{29.4b}$$

The type $\tau_{\mathsf{exn}}$ is some fixed, but as yet unspecified, type of exception values. (The choice of $\tau_{\mathsf{exn}}$ is considered in Section 29.3 below.)

The dynamics of **XPCF** is similar to that of **FPCF**, except that the failure state, $k \blacktriangleleft$ , is replaced by the exception state, $k \blacktriangleleft e$, which passes an exception value $e$ to the stack $k$. There is only one notion of exception, but the associated value may be used to differentiate among its various sources. (See Section 29.3 for further discussion of this point.)

As long as there is more than one value of type $\tau_{\mathsf{exn}}$, it is possible to distinguish the order of evaluation of expressions within a program. (It is impossible to do so with simple failures, because there is no data associated with the failure, making it impossible to identify the source of the failure from the mere fact that a failure occurred.) The problem is particularly acute when using a by-name evaluation order for function applications, for in that case unevaluated expressions are replicated by substitution, and may be left unevaluated, or evaluated at unpredictable times. When this is the case, exceptions are said to be *imprecise*, making it difficult to understand programs that use exceptions. However, under by-value evaluation exceptions are *precise*, and there are no such complications. In this section we will build on the by-value variant of the **PCF**

Preview

machine discussed in Exercise 28.2 so as to avoid the difficulties of imprecise exceptions. (But see Section 29.4 for an alternative approach.)

The stack frames of the **PCF** machine are extended to include $\mathtt{raise}[\tau](-)$ and $\mathtt{handle}(-;x.e_2)$, and the following additional rules govern the exception raising and handling constructs of **XPCF**:

$$\frac{}{k \rhd \mathtt{raise}[\tau](e) \longmapsto k;\mathtt{raise}[\tau](-) \rhd e} \tag{29.5a}$$

$$\frac{}{k;\mathtt{raise}[\tau](-) \lhd e \longmapsto k \blacktriangleleft e} \tag{29.5b}$$

$$\frac{}{k \rhd \mathtt{handle}(e_1;x.e_2) \longmapsto k;\mathtt{handle}(-;x.e_2) \rhd e_1} \tag{29.5c}$$

$$\frac{}{k;\mathtt{handle}(-;x.e_2) \lhd e \longmapsto k \lhd e} \tag{29.5d}$$

$$\frac{}{k;\mathtt{handle}(-;x.e_2) \blacktriangleleft e \longmapsto k \rhd [e/x]e_2} \tag{29.5e}$$

$$\frac{(f \neq \mathtt{handle}(-;x.e_2))}{k;f \blacktriangleleft e \longmapsto k \blacktriangleleft e} \tag{29.5f}$$

The main difference compared to Rules (29.2) is that an exception passes a values to the stack, whereas a failure does not.

The initial and final states of the **XPCF** machine are defined by the following rules:

$$\frac{}{\epsilon \rhd e \ \mathsf{initial}} \tag{29.6a}$$

$$\frac{e \ \mathsf{val}}{\epsilon \lhd e \ \mathsf{final}} \tag{29.6b}$$

$$\frac{}{\epsilon \blacktriangleleft e \ \mathsf{final}} \tag{29.6c}$$

**Theorem 29.2** (Safety for **XPCF**). *1. If $s$ ok and $s \longmapsto s'$, then $s'$ ok.*

*2. If $s$ ok, then either $s$ final or there exists $s'$ such that $s \longmapsto s'$.*

## 29.3 Exception Values

The statics of **XPCF** is parameterized by the type, $\tau_{\mathsf{exn}}$, of values associated to exceptions. The choice of $\tau_{\mathsf{exn}}$ determines the extent to which the source of an exception can be identified in a program. We first consider two limiting cases, before discussing more useful alternatives. If $\tau_{\mathsf{exn}}$ is chosen to be the empty type `void`, then the exception mechanism is rendered inoperative, because



Preview

there are no exception values that can be raised. If $\tau_{\mathsf{exn}}$ is chosen to be the one-element type `unit`, then we exceptions are degenerated to simple failures, because the exception value is always the sole element of that type.

Other choices of the type of exception values have been considered in various languages. A hopelessly naïve, but nevertheless extant, choice is to take $\tau_{\mathsf{exn}}$ to be the type of character strings, `str`. This allows one to write, for example,

```
raise "Division by zero error."
```

to signal the obvious arithmetic fault. This is well and good for exceptions that terminate the program, but if they are to be handled within the program, then nothing but trouble can come from such a choice. The exception handler would be reduced to parsing strings to distinguish the sources of an exception!

A slightly better choice that makes life easier for the handler is to choose $\tau_{\mathsf{exn}}$ to be the type of natural numbers, `nat`. Exceptional conditions are encoded as *error numbers* that can be distinguished by a handler. The trouble with this approach, however, is that it requires an up-front assignment of numbers to sources of exceptions, a hopelessly non-modular and unmaintainable state of affairs. Moreover, short of using obtuse encodings, there is no practical way to associate meaningful data with the exception, another shortcoming of this approach.

A more sensible choice of $\tau_{\mathsf{exn}}$ is a finite sum type whose classes are the forms of exception, and whose associated type determines the data associated with that exception. For example, the type $\tau_{\mathsf{exn}}$ might be chosen to be a sum type of the form

$$[\mathtt{div} \hookrightarrow \mathtt{unit}, \mathtt{fnf} \hookrightarrow \mathtt{string}, \ldots].$$

In this example the class `div` might represent an arithmetic fault, with no associated data, and the class `fnf` might represent a "file not found" error, with associated data being the name of the file that was not found.

When $\tau_{\mathsf{exn}}$ is chosen to be a sum type, the classes of the sum are called *exceptions*, so that one may speak of "the `fnf` exception" and the "`div` exception" in the above example. This terminology is harmless, and all but unavoidable, but it invites confusion between two separate concepts:

1. Exceptions as a *control mechanism* that allows the course of evaluation to be altered from its default course by raising and handling exceptions.
2. Exceptions as a *data value* associated with such a deviation of control that allows the source of the deviation to be identified.

One symptom of conflating these two usages of the word "exception" is the misapprehension that exceptions are related to fluid binding (which is discussed in Chapter 34).

The advantage $\tau_{\mathsf{exn}}$ to be a sum type is that it makes it convenient for the handler to dispatch on the source of an exception by case analysis. Moreover, it allows data to be associated with the exception on a per-case basis, without compromising type safety. For example, we might write

```
handle e1 ow x ↪
  match x {
    div ⟨⟩ ↪ ediv
  | fnf s ↪ efnf }
```

PREVIEW

to handle the exceptions specified by the sum type given in the preceding paragraph. Because the exception and its associated data are coupled in a sum type, there is no possibility of misinterpreting the data associated to one exception as being that of another.

The disadvantage of choosing a finite sum for $\tau_{\mathsf{exn}}$ is that it imposes a *closed world* restriction on exceptions in the sense that the possible exceptions that may be raised in a program must be fixed in advance. This interferes with modular development and evolution, and precludes separate compilation of program components (see Chapter 45 for more on separate compilation). In the interest of enhancing modularity it is preferable to adopt an *open world* view of exceptions in which no assumption is made about the possible classes of exception that may arise at run-time. To do so requires a generalization of the finite sum type, called *dynamic classification*, that is discused in Chapter 35. Using dynamic classification one may create fresh exceptions without fear of conflict with any other part of the program or any future extension of it.

## 29.4 Modal Separation of Exceptions

We noted in Section 29.2 that, as long as $\tau_{\mathsf{exn}}$ has more than one element, the order of evaluation of expressions is visible to the programmer. This is particularly problematic with laziness, which gives rise to the problem of imprecise exceptions. Rather than avoid laziness, an alternative is to regard exceptions as *computational effects* that require explicit sequencing to avoid imprecision (and, in some cases, utter chaos). This may be achieved by making a *modal separation* between *pure expressions*, which may not raise exceptions, and *impure commands*, which may. The evaluation order of pure expressions is immaterial to their meaning, but the same cannot be said for impure commands. To avoid imprecision commands are composed using a construct that evaluates a command to completion, and proceeds based on whether the outcome was to return a value, or to raise an exception. The modal distinction between expressions and commands is mediated by a *modality*, called the *lax modality*, or a *monad*, whose introductory forms are unevaluated commands, and whose eliminatory form is the aforementioned sequentialization construct that acts on such an unevaluated command.

The language **MPCF** is a modal extension of **PCF** in which just such a distinction is enforced. Its syntax is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{cmd}(\tau)$ | $\tau\ \mathtt{cmd}$ | exception modality |
| Exp | $e$ | $::=$ | $\mathtt{cmd}(m)$ | $\mathtt{cmd}\ m$ | unevaluated command |
| Cmd | $m$ | $::=$ | $\mathtt{ret}(e)$ | $\mathtt{ret}\ e$ | return value |
| | | | $\mathtt{raise}[\tau](e)$ | $\mathtt{raise}(e)$ | raise exception |
| | | | $\mathtt{bnd}(e; x_1.m_1; x_2.m_2)$ | $\mathtt{bnd}\ x \hookleftarrow e\ \mathtt{in}\ m_1\ \mathtt{ow}\ y \hookleftarrow m_2$ | sequentialize |

The types and expressions of **PCF** are implicitly included as expressions of **MPCF**. It is immaterial whether these are given an eager or a lazy dynamics. The expression $\mathtt{cmd}(m)$ is the introductory form of type $\mathtt{cmd}(\tau)$; it is an unevaluated command of type $\tau$ thought of as a value. The command $\mathtt{ret}(e)$ is the command that merely returns the value of the expression $e$; it cannot raise an exception. The command $\mathtt{raise}[\tau](e)$ raises an exception with associated value given by the expression $e$. The sequencing command $\mathtt{bnd}(e; x_1.m_1; x_2.m_2)$ is the elimination form for the type $\mathtt{cmd}(\tau)$. It evaluates $e$ to a value $\mathtt{cmd}(m)$, then executes $m$ to determine its outcome. If it returns a value, $v_1$,



Preview

evaluation proceeds with $[v_1/x_1]m_1$, the *ordinary return*. If it raises an exception with value $v_2$, evaluation proceeds with $[v_2/x_2]m_2$, the *exceptional return*.

The statics of **MPCF** consists of two judgment forms:

| | |
|---|---|
| $\Gamma \vdash e : \tau$ | $e$ is a pure expression of type $\tau$ |
| $\Gamma \vdash m \div \tau$ | $m$ is an impure command returning type $\tau$ |

The typing judgment for commands implicitly allows for a command to raise an exception, but does not otherwise indicate anything about the exception value that may be associated with it. (See Exercise **29.3** for more on how to track this information using type refinements.)

$$\frac{\Gamma \vdash m \div \tau}{\Gamma \vdash \mathtt{cmd}(m) : \mathtt{cmd}(\tau)} \tag{29.7a}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{ret}(e) \div \tau} \tag{29.7b}$$

$$\frac{\Gamma \vdash e : \tau_{\mathsf{exn}}}{\Gamma \vdash \mathtt{raise}[\tau](e) \div \tau} \tag{29.7c}$$

$$\frac{\Gamma \vdash e : \mathtt{cmd}(\tau) \quad \Gamma, x_1 : \tau \vdash m_1 \div \tau' \quad \Gamma, x_2 : \tau_{\mathsf{exn}} \vdash m_2 \div \tau'}{\Gamma \vdash \mathtt{bnd}(e; x_1.m_1; x_2.m_2) \div \tau'} \tag{29.7d}$$

Observe that in rule (29.7d) the result type, $\tau'$, is required to be the same for either outcome of evaluation of the command given by the expression $e$.

The dynamics of **MPCF** is conveniently given by a stack machine defined by the following four judgment forms:

1. $k \triangleright e$, evaluate an expression $e$ on stack $k$.
2. $k \triangleleft e$, return a value $e$ to the stack $k$.
3. $k \blacktriangleright m$, evaluate a command $m$ on stack $k$.
4. $k \blacktriangleleft e$, raise an exception value $e$ on the stack $k$.

These judgments are defined by the following rules, in addition to those given in Chapter 28 for the expressions of **PCF**:

$$\overline{k \triangleright \mathtt{cmd}(m) \longmapsto k \triangleleft \mathtt{cmd}(m)} \tag{29.8a}$$

$$\overline{k \blacktriangleright \mathtt{ret}(e) \longmapsto k;\mathtt{ret}(-) \triangleright e} \tag{29.8b}$$

$$\overline{k;\mathtt{ret}(-) \triangleleft e \longmapsto k \triangleleft e} \tag{29.8c}$$

Preview

$$\frac{}{k \blacktriangleright \mathtt{raise}[\tau](e) \longmapsto k;\mathtt{raise}[\tau](-) \triangleright e} \tag{29.8d}$$

$$\frac{}{k;\mathtt{raise}[\tau](-) \triangleleft e \longmapsto k \blacktriangleleft e} \tag{29.8e}$$

$$\frac{}{k \blacktriangleright \mathtt{bnd}(e; x_1.m_1; x_2.m_2) \longmapsto k;\mathtt{bnd}(-; x_1.m_1; x_2.m_2) \triangleright e} \tag{29.8f}$$

$$\frac{}{k;\mathtt{bnd}(-; x_1.m_1; x_2.m_2) \triangleleft \mathtt{cmd}(m) \longmapsto k;\mathtt{bnd}(-; x_1.m_1; x_2.m_2) \blacktriangleright m} \tag{29.8g}$$

$$\frac{}{k;\mathtt{bnd}(-; x_1.m_1; x_2.m_2) \triangleleft e \longmapsto k \blacktriangleright [e/x_1]m_1} \tag{29.8h}$$

$$\frac{}{k;\mathtt{bnd}(-; x_1.m_1; x_2.m_2) \blacktriangleleft e \longmapsto k \blacktriangleright [e/x_2]m_2} \tag{29.8i}$$

$$\frac{(f \neq \mathtt{bnd}(-; x_1.e_1; x_2.e_2))}{k; f \blacktriangleleft e \longmapsto k \blacktriangleleft e} \tag{29.8j}$$

The initial and final states of the **MPCF** machine are defined as follows:

$$\frac{}{\epsilon \blacktriangleright m \text{ initial}} \tag{29.9a}$$

$$\frac{e \text{ val}}{\epsilon \triangleleft e \text{ final}} \tag{29.9b}$$

$$\frac{}{\epsilon \blacktriangleleft e \text{ final}} \tag{29.9c}$$

## 29.5 Notes

Various forms of exceptions were explored Lisp (for example, Steele (1990)). The original formulation of ML (Gordon et al., 1979) as a metalanguage for mechanized logic made extensive use of exceptions, called "failures," to implement tactics and tacticals. These days most languages include an exception mechanism of the kind considered here.

The essential distinction between the exception *mechanism* and exception *values* is often misunderstood. The choice of type $\tau_{\mathsf{exn}}$ to be a (static or dynamic) sum type determines the possible "exceptions" that can be raised by determining which exception values can be constructed and deconstructed by pattern matching.

The use of a monad to classify computational effects was introduced by Moggi (1989), and developed further by Wadler (1992). The modal reformulation considered here was introduced by Pfenning and Davies (2001) as an instance of the lax modality. The consolidation of exception handling with sequentialization was proposed by (Benton and Kennedy, 2001).

Preview

## Exercises

**29.1**. Prove Theorem 29.2. Are any properties of $\tau_{\mathsf{exn}}$ required for the proof?

**29.2**. In Section 29.4 it was noted that exceptions are distinctive among effects in that they may be encapsulated within a pure expression. Develop this idea by introducing a form of exception handler at the expression level of **MPCF** that ensures that no exception may escape from evaluation of a given command.

**29.3**. The closed world assumption on exceptions amounts to choosing the type of exception values to be a finite sum type shared by the entire program. Under such an assumption it is feasible to perform *exception tracking* by placing an upper bound on the possible classes of an exception value.

Type refinements (defined in Chapter 24) may be used to perform exception tracking in a closed-world setting. Define *finite sum refinements* by the rule

$$\frac{X' \subseteq X \quad (\forall x \in X')\ \phi_x \sqsubseteq \tau_x}{[\phi_x]_{x \in X'} \sqsubseteq [\tau_x]_{x \in X}}.$$

In particular, the refinement $\emptyset$ is the vacuous sum refinement $[]$ satisfied by no value. Entailment of finite sum refinements is defined by the rule

$$\frac{X' \subseteq X'' \quad (\forall x \in X')\ \phi_x \leq \phi'_x}{[\phi_x]_{x \in X'} \leq [\phi'_x]_{x \in X''}}$$

So, in particular, $\emptyset \leq \phi$ for all sum refinements $\phi$ of $\tau_{\mathsf{exn}}$. Entailment weakens knowledge of the class of a value of sum type, which is crucial to their application to exception tracking.

The goal of this exercise is to develop a system of type refinements for the modal formulation of exceptions in **MPCF** using sum refinements to perform exception tracking.

(a) Define the command refinement judgment $m \in_\tau \phi\,\mathtt{ow}\,\chi$, where $m \div \tau$, $\phi \sqsubseteq \tau$, and $\chi \sqsubseteq \tau_{\mathsf{exn}}$, to mean that if $m$ returns $e$, then $e \in_\tau \phi$, and if $m$ raises $e$, then $e \in_{\tau_{\mathsf{exn}}} \chi$.

(b) Define satisfaction and entailment for the expression refinement $\mathtt{cmd}(\phi; \chi) \sqsubseteq \mathtt{cmd}(\tau)$, where $\phi \sqsubseteq \tau$ and $\chi \sqsubseteq \tau_{\mathsf{exn}}$. This refinement classifies encapsulated commands that satisfy the stated value and exception refinements in the sense of the preceding problem.

**29.4**. Show that exceptions in **MPCF** may be systematically eliminated by a translation into **PCF** enriched with sum types by what is called the *exception-passing style* transformation. Each command $m \div \tau$ of **MPCF** is translated to a pure expression $\widehat{m}$ of type $\widehat{\tau} + \tau_{\mathsf{exn}}$ whose value is either $\mathtt{l} \cdot e$, where $e : \tau$, for normal return, or $\mathtt{r} \cdot e$, where $e : \tau_{\mathsf{exn}}$, for an exceptional return. The command translation is extended to an expression translation $\widehat{e}$ that replaces occurrences of $\mathtt{cmd}(m)$ by $\widehat{m}$. The corresponding type translation, $\widehat{\tau}$, replaces $\mathtt{cmd}(\tau)$ by $\widehat{\tau} + \tau_{\mathsf{exn}}$. Define the command translation from **MPCF** to **PCF** enriched with sums, and show that it has the required type and correctly simulates the behavior of exceptions.

Preview

Preview

# Chapter 30

# Continuations

The semantics of many control constructs (such as exceptions and co-routines) can be expressed in terms of *reified* control stacks, a representation of a control stack as an ordinary value. This is achieved by allowing a stack to be passed as a value within a program and to be restored at a later point, *even if* control has long since returned past the point of reification. Reified control stacks of this kind are called *continuations*; they are values that can be passed and returned at will in a computation. Continuations never "expire", and it is always sensible to reinstate a continuation without compromising safety. Thus continuations support unlimited "time travel" — we can go back to a previous point in the computation and then return to some point in its future, at will.

Why are continuations useful? Fundamentally, they are representations of the control state of a computation at a given point in time. Using continuations we can "checkpoint" the control state of a program, save it in a data structure, and return to it later. In fact this is precisely what is necessary to implement *threads* (concurrently executing programs) — the thread scheduler must be able to checkpoint a program and save it for later execution, perhaps after a pending event occurs or another thread yields the processor.

## 30.1 Overview

We will consider the extension, **KPCF**, of **PCF** with the type $\texttt{cont}(\tau)$ of continuations accepting values of type $\tau$. The introduction form for $\texttt{cont}(\tau)$ is $\texttt{letcc}[\tau](x.e)$, which binds the *current continuation* (that is, the current control stack) to the variable $x$, and evaluates the expression $e$. The corresponding elimination form is $\texttt{throw}[\tau](e_1; e_2)$, which restores the value given by $e_1$ to the control stack given by $e_2$.

To illustrate the use of these primitives, consider the problem of multiplying the first $n$ elements of an infinite sequence $q$ of natural numbers, where $q$ is represented by a function of type $\texttt{nat} \rightharpoonup \texttt{nat}$. If zero occurs among the first $n$ elements, we would like to effect an "early return" with the value zero, rather than perform the remaining multiplications. This problem can be solved using exceptions (we leave this as an exercise), but we will give a solution that uses continuations in preparation for what follows.

Preview

Here is the solution in **PCF**, without short-cutting:

```
fix ms is
  λ q : nat ⇀ nat.
    λ n : nat.
      case n {
        z ↪ s(z)
      | s(n') ↪ (q z) × (ms (q ∘ succ) n')
      }
```

The recursive call composes $q$ with the successor function to shift the sequence by one step.

Here is the solution in **KPCF**, with short-cutting:

```
λ q : nat ⇀ nat.
  λ n : nat.
    letcc ret : nat cont in
      let ms be
        fix ms is
          λ q : nat ⇀ nat.
            λ n : nat.
              case n {
                z ↪ s(z)
              | s(n') ↪
                case q z {
                  z ↪ throw z to ret
                | s(n'') ↪ (q z) × (ms (q ∘ succ) n')
                }
              }
      in
        ms q n
```

The `letcc` binds the return point of the function to the variable `ret` for use within the main loop of the computation. If zero is encountered, control is thrown to `ret`, effecting an early return with the value zero.

Let's look at another example: given a continuation $k$ of type $\tau$ `cont` and a function $f$ of type $\tau' \rightharpoonup \tau$, return a continuation $k'$ of type $\tau'$ `cont` with the following behavior: throwing a value $v'$ of type $\tau'$ to $k'$ throws the value $f(v')$ to $k$. This is called *composition of a function with a continuation*. We wish to fill in the following template:

```
fun compose(f:τ' ⇀ τ,k:τ cont):τ' cont = ....
```

The first problem is to obtain the continuation we wish to return. The second problem is how to return it. The continuation we seek is the one in effect at the point of the ellipsis in the expression `throw` $f$`(...) to` $k$. This is the continuation that, when given a value $v'$, applies $f$ to it, and throws the result to $k$. We can seize this continuation using `letcc`, writing

```
throw f(letcc x:τ' cont in ...) to k
```



Preview

At the point of the ellipsis the variable $x$ is bound to the continuation we wish to return. How can we return it? By using the same trick as we used for short-circuiting evaluation above! We don't want to actually throw a value to this continuation (yet), instead we wish to abort it and return it as the result. Here's the final code:

```
fun compose (f:τ′ ⇀ τ, k:τ cont):τ′ cont =
    letcc ret:τ′ cont cont in
      throw (f (letcc r in throw r to ret)) to k
```

The type of `ret` is that of a continuation-expecting continuation.

## 30.2 Continuation Dynamics

The syntax of **KPCF** is as follows:

| | | | | | |
|---|---|---|---|---|---|
| Type | $\tau$ | ::= | $\texttt{cont}(\tau)$ | $\tau\ \texttt{cont}$ | continuation |
| Expr | $e$ | ::= | $\texttt{letcc}[\tau](x.e)$ | $\texttt{letcc}\ x\ \texttt{in}\ e$ | mark |
| | | | $\texttt{throw}[\tau](e_1;e_2)$ | $\texttt{throw}\ e_1\ \texttt{to}\ e_2$ | goto |
| | | | $\texttt{cont}(k)$ | $\texttt{cont}(k)$ | continuation |

The expression $\texttt{cont}(k)$ is a reified control stack, which arises during evaluation.

The statics of **KPCF** is defined by the following rules:

$$\frac{\Gamma, x : \texttt{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{letcc}[\tau](x.e) : \tau} \tag{30.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \texttt{cont}(\tau_1)}{\Gamma \vdash \texttt{throw}[\tau](e_1;e_2) : \tau} \tag{30.1b}$$

The result type of a `throw` expression is arbitrary because it does not return to the point of the call.

The statics of continuation values is given by the following rule:

$$\frac{k : \tau}{\Gamma \vdash \texttt{cont}(k) : \texttt{cont}(\tau)} \tag{30.2}$$

A continuation value $\texttt{cont}(k)$ has type $\texttt{cont}(\tau)$ exactly if it is a stack accepting values of type $\tau$.

To define the dynamics of **KPCF** we extend the **PCF** machine with two forms of stack frame:

$$\frac{}{\texttt{throw}[\tau](-;e_2)\ \mathsf{frame}} \tag{30.3a}$$

$$\frac{e_1\ \mathsf{val}}{\texttt{throw}[\tau](e_1;-)\ \mathsf{frame}} \tag{30.3b}$$

Every reified control stack is a value:

$$\frac{k\ \mathsf{stack}}{\texttt{cont}(k)\ \mathsf{val}} \tag{30.4}$$

Preview

The transition rules of the **PCF** machine governing continuations are as follows:

$$\overline{k \triangleright \mathtt{cont}(k) \longmapsto k \triangleleft \mathtt{cont}(k')} \tag{30.5a}$$

$$\overline{k \triangleright \mathtt{letcc}[\tau](x.e) \longmapsto k \triangleright [\mathtt{cont}(k)/x]e} \tag{30.5b}$$

$$\overline{k \triangleright \mathtt{throw}[\tau](e_1; e_2) \longmapsto k; \mathtt{throw}[\tau](-; e_2) \triangleright e_1} \tag{30.5c}$$

$$\frac{e_1 \ \mathsf{val}}{k; \mathtt{throw}[\tau](-; e_2) \triangleleft e_1 \longmapsto k; \mathtt{throw}[\tau](e_1; -) \triangleright e_2} \tag{30.5d}$$

$$\frac{e \ \mathsf{val}}{k; \mathtt{throw}[\tau](e; -) \triangleleft \mathtt{cont}(k') \longmapsto k' \triangleleft e} \tag{30.5e}$$

Evaluation of a `letcc` expression duplicates the control stack; evaluation of a `throw` expression destroys the current control stack.

The safety of **KPCF** may be established by extending the safety proof for the **PCF** machine given in Chapter 28.

We need only add typing rules for the two new forms of frame, which are as follows:

$$\frac{e_2 : \mathtt{cont}(\tau)}{\mathtt{throw}[\tau'](-; e_2) : \tau \rightsquigarrow \tau'} \tag{30.6a}$$

$$\frac{e_1 : \tau \quad e_1 \ \mathsf{val}}{\mathtt{throw}[\tau'](e_1; -) : \mathtt{cont}(\tau) \rightsquigarrow \tau'} \tag{30.6b}$$

The rest of the definitions remain as in Chapter 28.

**Lemma 30.1** (Canonical Forms)**.** *If $e : \mathtt{cont}(\tau)$ and $e$ val, then $e = \mathtt{cont}(k)$ for some $k$ such that $k : \tau$.*

**Theorem 30.2** (Safety)**.** *1. If $s$ ok and $s \longmapsto s'$, then $s'$ ok.*

*2. If $s$ ok, then either $s$ final or there exists $s'$ such that $s \longmapsto s'$.*

## 30.3 Coroutines from Continuations

The distinction between a routine and a subroutine is the distinction between a manager and a worker. The routine calls upon the subroutine to accomplish a piece of work, and the subroutine returns to the routine when its work is done. The relationship is asymmetric in that there is a clear distinction between the *caller*, the main routine, and the *callee*, the subroutine. Often it is useful to consider a symmetric situation in which two routines each call the other to help accomplish a task. Such a pair of routines are called *coroutines*; their relationship to one another is symmetric rather than hierarchical.

The key to implementing a subroutine is for the caller to pass to the callee a continuation representing the return point of the subroutine call. When the subroutine is finished, it calls the

Preview

continuation passed to it by the calling routine. Because the subroutine is finished at that point, there is no need for the callee to pass a continuation back to the caller. The key to implementing coroutines is to have each routine treat the other as a subroutine of itself. In particular, whenever a coroutine cedes control to its caller, it provides a continuation that the caller may use to cede control back to the callee, in the process providing a continuation for itself. (This raises an interesting question of how the whole process gets started. We'll return to this shortly.)

To see how a pair of coroutines is implemented, let us consider the type of each routine in the pair. A routine is a continuation accepting two arguments, a datum to be passed to that routine when it is resumed, and a continuation to be resumed when the routine has finished its task. The datum represents the state of the computation, and the continuation is a coroutine that accepts arguments of the same form. Thus, the type of a coroutine must satisfy the type isomorphism

$$\tau\,\mathtt{coro} \cong (\tau \times \tau\,\mathtt{coro})\,\mathtt{cont}.$$

So we may take $\tau\,\mathtt{coro}$ to be the recursive type

$$\tau\,\mathtt{coro} \triangleq \mathtt{rec}\,t\,\mathtt{is}\,(\tau \times t)\,\mathtt{cont}.$$

Up to isomorphism, the type $\tau\,\mathtt{coro}$ is the type of continuations that accept a value of type $\tau$, representing the state of the coroutine, and the partner coroutine, a value of the same type.

A coroutine, $r$, passes control to another coroutine, $r'$, by evaluating the expression $\mathtt{resume}(\langle s, r'\rangle)$, where $s$ is the current state of the computation. Doing so creates a new coroutine whose entry point is the return point (calling site) of the application of `resume`. Therefore the type of `resume` is

$$\tau \times \tau\,\mathtt{coro} \rightharpoonup \tau \times \tau\,\mathtt{coro}.$$

The definition of `resume` is as follows:

$$\lambda\,(\langle s, r'\rangle{:}\tau \times \tau\,\mathtt{coro})\,\mathtt{letcc}\,k\,\mathtt{in}\,\mathtt{throw}\,\langle s, \mathtt{fold}(k)\rangle\,\mathtt{to}\,\mathtt{unfold}(r')$$

When applied, `resume` seizes the current continuation, and passes the state, $s$, and the seized continuation (packaged as a coroutine) to the called coroutine.

But how do we create a system of coroutines in the first place? Because the state is explicitly passed from one routine to the other, a coroutine may be defined as a state transformation function that, when activated with the current state, determines the next state of the computation. A system of coroutines is created by establishing a joint exit point to which the result of the system is thrown, and creating a pair of coroutines that iteratively transform the state and pass control to the partner routine. If either routine wishes to terminate the computation, it does so by throwing a result value to their common exit point. Thus, a coroutine may be specified by a function of type

$$(\tau', \tau)\,\mathtt{rout} \triangleq \tau'\,\mathtt{cont} \rightharpoonup \tau \rightharpoonup \tau,$$

where $\tau'$ is the result type and $\tau$ is the state type of the system of coroutines.

To set up a system of coroutines we define a function `run` that, given two routines, creates a function of type $\tau \rightharpoonup \tau'$ that, when applied to the initial state, computes a result of type $\tau'$.

Preview

The computation consists of a cooperating pair of routines that share a common exit point. The definition of `run` begins as follows:

$$\lambda\,(\langle r_1, r_2\rangle)\,\lambda\,(s_0)\,\texttt{letcc}\,x_0\,\texttt{in}\,\texttt{let}\,r_1'\,\texttt{be}\,r_1(x_0)\,\texttt{in}\,\texttt{let}\,r_2'\,\texttt{be}\,r_2(x_0)\,\texttt{in}\ldots$$

Given two routines, `run` establishes their common exit point, and passes this continuation to both routines. By throwing to this continuation either routine may terminate the computation with a result of type $\tau'$. The body of the `run` function continues as follows:

$$\texttt{rep}(r_2')(\texttt{letcc}\,k\,\texttt{in}\,\texttt{rep}(r_1')(\langle s_0, \texttt{fold}(k)\rangle))$$

The auxiliary function `rep` creates an infinite loop that transforms the state and passes control to the other routine:

$$\lambda\,(t)\,\texttt{fix}\,l\,\texttt{is}\,\lambda\,(\langle s, r\rangle)\,l(\texttt{resume}(\langle t(s), r\rangle)).$$

The system is initialized by starting routine $r_1$ with the initial state, and arranging that, when it cedes control to its partner, it starts routine $r_2$ with the resulting state. At that point the system is bootstrapped: each routine will resume the other on each iteration of the loop.

A good example of coroutining arises whenever we wish to interleave input and output in a computation. We may achieve this using a coroutine between a *producer* routine and a *consumer* routine. The producer emits the next element of the input, if any, and passes control to the consumer with that element removed from the input. The consumer processes the next data item, and returns control to the producer, with the result of processing attached to the output. The input and output are modeled as lists of type $\tau_i\,\texttt{list}$ and $\tau_o\,\texttt{list}$, respectively, which are passed back and forth between the routines.[1] The routines exchange messages according to the following protocol. The message $\texttt{OK}(\langle i, o\rangle)$ is sent from the consumer to producer to acknowledge receipt of the previous message, and to pass back the current state of the input and output channels. The message $\texttt{EMIT}(\langle e, \langle i, o\rangle\rangle)$, where $e$ is a value of type $\tau_i\,\texttt{opt}$, is sent from the producer to the consumer to emit the next value (if any) from the input, and to pass the current state of the input and output channels to the consumer.

This leads to the following implementation of the producer/consumer model. The type $\tau$ of the state maintained by the routines is the labeled sum type

$$[\texttt{OK} \hookrightarrow \tau_i\,\texttt{list} \times \tau_o\,\texttt{list}, \texttt{EMIT} \hookrightarrow \tau_i\,\texttt{opt} \times (\tau_i\,\texttt{list} \times \tau_o\,\texttt{list})].$$

This type specifies the message protocol between the producer and the consumer described in the preceding paragraph.

The producer, $P$, is defined by the expression

$$\lambda\,(x_0)\,\lambda\,(msg)\,\texttt{case}\,msg\,\{b_1 \mid b_2 \mid b_3\},$$

where the first branch, $b_1$, is

$$\texttt{OK} \cdot \langle \texttt{nil}, os\rangle \hookrightarrow \texttt{EMIT} \cdot \langle \texttt{null}, \langle \texttt{nil}, os\rangle\rangle$$

[1] In practice the input and output state are implicit, but we prefer to make them explicit for the sake of clarity.



Preview

and the second branch, $b_2$, is

$$\texttt{OK} \cdot \langle \texttt{cons}(i; is), os \rangle \hookrightarrow \texttt{EMIT} \cdot \langle \texttt{just}(i), \langle is, os \rangle \rangle,$$

and the third branch, $b_3$, is

$$\texttt{EMIT} \cdot _ \hookrightarrow \texttt{error}.$$

In words, if the input is exhausted, the producer emits the value `null`, along with the current channel state. Otherwise, it emits $\texttt{just}(i)$, where $i$ is the first remaining input, and removes that element from the passed channel state. The producer cannot see an `EMIT` message, and signals an error if it should occur.

The consumer, $C$, is defined by the expression

$$\lambda\,(x_0)\,\lambda\,(msg)\,\texttt{case}\,msg\,\{b'_1 \mid b'_2 \mid b'_3\},$$

where the first branch, $b'_1$, is

$$\texttt{EMIT} \cdot \langle \texttt{null}, \langle _, os \rangle \rangle \hookrightarrow \texttt{throw}\,os\,\texttt{to}\,x_0,$$

the second branch, $b'_2$, is

$$\texttt{EMIT} \cdot \langle \texttt{just}(i), \langle is, os \rangle \rangle \hookrightarrow \texttt{OK} \cdot \langle is, \texttt{cons}(f(i); os) \rangle,$$

and the third branch, $b'_3$, is

$$\texttt{OK} \cdot _ \hookrightarrow \texttt{error}.$$

The consumer dispatches on the emitted datum. If it is absent, the output channel state is passed to $x_0$ as the ultimate value of the computation. If it is present, the function $f$ (unspecified here) of type $\tau_i \rightharpoonup \tau_o$ is applied to transform the input to the output, and the result is added to the output channel. If the message `OK` is received, the consumer signals an error, as the producer never produces such a message.

The initial state, $s_0$, has the form $\texttt{OK} \cdot \langle is, os \rangle$, where *is* and *os* are the initial input and output channel state, respectively. The computation is created by the expression

$$\texttt{run}(\langle P, C \rangle)(s_0),$$

which sets up the coroutines as described earlier.

Although it is relatively easy to visualize and implement coroutines involving only two partners, it is more complex, and less useful, to consider a similar pattern of control among $n \geq 2$ participants. In such cases it is more common to structure the interaction as a collection of $n$ routines, each of which is a coroutine of a central *scheduler*. When a routine resumes its partner, it passes control to the scheduler, which determines which routine to execute next, again as a coroutine of itself. When structured as coroutines of a scheduler, the individual routines are called *threads*. A thread *yields* control by resuming its partner, the scheduler, which then determines which thread to execute next as a coroutine of itself. This pattern of control is called *cooperative multi-threading*, because it is based on explicit yields, rather than implicit yields imposed by asynchronous events such as timer interrupts.

Preview

## 30.4 Notes

Continuations are a ubiquitous notion in programming languages. Reynolds (1993) provides an excellent account of the multiple discoveries of continuations. The formulation given here is inspired by Felleisen and Hieb (1992), who pioneered the development of linguistic theories of control and state.

## Exercises

**30.1**. Type safety for **KPCF** follows almost directly from Theorem 28.1. Isolate the key observations required to extend the proof to include continuation types.

**30.2**. Exhibit a closed **KPCF** expression of each of the following types:

(a) $\tau + (\tau\ \mathtt{cont})$.

(b) $\tau\ \mathtt{cont}\ \mathtt{cont} \to \tau$.

(c) $(\tau_2\ \mathtt{cont} \to \tau_1\ \mathtt{cont}) \to (\tau_1 \to \tau_2)$.

(d) $(\tau_1 + \tau_2)\ \mathtt{cont} \to (\tau_1\ \mathtt{cont} \times \tau_2\ \mathtt{cont})$.

*Hint*: you will need to use `letcc` and `throw`.

**30.3**. The type `stream` of infinite streams of natural numbers defined in Chapter 14 may be implemented using continuations. Define `stream` to be the recursive type satisfying the isomorphism

$$\mathtt{stream} \cong (\mathtt{nat} \times \mathtt{stream})\ \mathtt{cont}\ \mathtt{cont}.$$

To examine the front of the stream, throw to it a continuation expecting a natural number and another stream. When passed such a continuation, the stream throws to it the next number in the stream, paired with another stream (that is, another continuation) representing the stream of numbers following that number. Define the introductory and eliminatory forms for streams defined in Chapter 14 using this representation.

Preview

# Part XII

# Types and Propositions

Preview

Preview

# Chapter 31

# Constructive Logic

Constructive logic codifies the principles of mathematical reasoning as it is actually practiced. In mathematics a proposition may be judged to be true exactly when it has a proof, and may be judged to be false exactly when it has a refutation. Because there are, and always will be, unsolved problems, we cannot expect in general that a proposition is either true or false, for in most cases we have neither a proof nor a refutation of it. Constructive logic may be described as *logic as if people matter*, as distinct from classical logic, which may be described as *the logic of the mind of god*. From a constructive viewpoint the judgment "$\phi$ true" means that "there is a proof of $\phi$."

What constitutes a proof is a social construct, an agreement among people as to what is a valid argument. The rules of logic codify a set of principles of reasoning that may be used in a valid proof. The valid forms of proof are determined by the outermost structure of the proposition whose truth is asserted. For example, a proof of a conjunction consists of a proof of each of its conjuncts, and a proof of an implication consists of a transformation of a proof of its antecedent to a proof of its consequent. When spelled out in full, the forms of proof are seen to correspond exactly to the forms of expression of a programming language. To each proposition is associated the type of its proofs; a proof is an expression of the associated type. This association between programs and proofs induces a dynamics on proofs, for they are but programs of some type. In this way proofs in constructive logic have *computational content*, which is to say that they may be interpreted as executable programs of the associated type. Conversely, programs have *mathematical content* as proofs of the proposition associated to their type.

This unification of logic and programming is called the *propositions as types* principle. It is the central organizing principle of the theory of programming languages. Propositions are identified with types, and proofs are identified with programs. A programming technique corresponds to a method of proof; a proof technique corresponds to a method of programming. Viewing types as behavioral specifications of programs, propositions may be seen as problem statements whose proofs are solutions that implement the specification.

Preview

## 31.1 Constructive Semantics

Constructive logic is concerned with two judgments, $\phi$ prop, stating that $\phi$ expresses a proposition, and $\phi$ true, stating that $\phi$ is a true proposition. What distinguishes constructive from non-constructive logic is that a proposition is not conceived of as merely a truth value, but instead as a *problem statement* whose solution, if it has one, is given by a proof. A proposition is said to be *true* exactly when it has a proof, in keeping with ordinary mathematical practice. In practice there is no other criterion of truth than the existence of a proof.

This principle has important, possibly surprising, consequences, the most important of which is that we cannot say, in general, that a proposition is either true or false. If for a proposition to be true means to have a proof of it, what does it mean for a proposition to be false? It means that we have a *refutation* of it, showing that it cannot be proved. That is, a proposition is false if we can show that the assumption that it is true (has a proof) contradicts known facts. In this sense constructive logic is a logic of *positive*, or *affirmative*, *information* — we must have explicit evidence in the form of a proof in order to affirm the truth or falsity of a proposition.

In light of this it should be clear that not every proposition is either true or false. For if $\phi$ expresses an unsolved problem, such as the famous $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ problem, then we have neither a proof nor a refutation of it (the mere absence of a proof not being a refutation). Such a problem is *undecided*, precisely because it is unsolved. Because there will always be unsolved problems (there being infinitely many propositions, but only finitely many proofs at a given point in the evolution of our knowledge), we cannot say that every proposition is *decidable*, that is, either true or false.

Of course, some propositions are decidable, and hence may be considered to be either true or false. For example, if $\phi$ expresses an inequality between natural numbers, then $\phi$ is decidable, because we can always work out, for given natural numbers $m$ and $n$, whether $m \leq n$ or $m \not\leq n$ — we can either prove or refute the given inequality. This argument does not extend to the real numbers. To get an idea of why not, consider the presentation of a real number by its decimal expansion. At any finite time we will have explored only a finite initial segment of the expansion, which is not enough to determine if it is, say, less than 1. For if we have determined the expansion to be $0.99\ldots9$, we cannot decide at any time, short of infinity, whether or not the number is 1. (This argument is not a proof, because we may wonder whether there is some other representation of real numbers that admits such a decision to be made finitely, but it turns out that this is not the case.)

The constructive attitude is simply to accept the situation as inevitable, and make our peace with that. When faced with a problem we have no choice but to roll up our sleeves and try to prove it or refute it. There is no guarantee of success! Life is hard, but we muddle through somehow.

## 31.2 Constructive Logic

The judgments $\phi$ prop and $\phi$ true of constructive logic are rarely of interest by themselves, but rather in the context of a hypothetical judgment of the form

$$\phi_1 \text{ true}, \ldots, \phi_n \text{ true} \vdash \phi \text{ true}.$$



Preview

This judgment expresses that the proposition $\phi$ is true (has a proof), under the assumptions that each of $\phi_1, \ldots, \phi_n$ are also true (have proofs). Of course, when $n = 0$ this is just the same as the judgment $\phi$ true.

The structural properties of the hypothetical judgment, when specialized to constructive logic, define what we mean by reasoning under hypotheses:

$$\frac{}{\Gamma, \phi \text{ true} \vdash \phi \text{ true}} \tag{31.1a}$$

$$\frac{\Gamma \vdash \phi_1 \text{ true} \quad \Gamma, \phi_1 \text{ true} \vdash \phi_2 \text{ true}}{\Gamma \vdash \phi_2 \text{ true}} \tag{31.1b}$$

$$\frac{\Gamma \vdash \phi_2 \text{ true}}{\Gamma, \phi_1 \text{ true} \vdash \phi_2 \text{ true}} \tag{31.1c}$$

$$\frac{\Gamma, \phi_1 \text{ true}, \phi_1 \text{ true} \vdash \phi_2 \text{ true}}{\Gamma, \phi_1 \text{ true} \vdash \phi_2 \text{ true}} \tag{31.1d}$$

$$\frac{\Gamma_1, \phi_2 \text{ true}, \phi_1 \text{ true}, \Gamma_2 \vdash \phi \text{ true}}{\Gamma_1, \phi_1 \text{ true}, \phi_2 \text{ true}, \Gamma_2 \vdash \phi \text{ true}} \tag{31.1e}$$

The last two rules are implicit in that we regard $\Gamma$ as a set of hypotheses, so that two "copies" are as good as one, and the order of hypotheses does not matter.

### 31.2.1 Provability

The syntax of propositional logic is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Prop | $\phi$ | ::= | $\top$ | $\top$ | truth |
| | | | $\bot$ | $\bot$ | falsity |
| | | | $\wedge(\phi_1;\phi_2)$ | $\phi_1 \wedge \phi_2$ | conjunction |
| | | | $\vee(\phi_1;\phi_2)$ | $\phi_1 \vee \phi_2$ | disjunction |
| | | | $\supset(\phi_1;\phi_2)$ | $\phi_1 \supset \phi_2$ | implication |

The connectives of propositional logic are given meaning by rules that determine (a) what constitutes a "direct" proof of a proposition formed from a given connective, and (b) how to exploit the existence of such a proof in an "indirect" proof of another proposition. These are called the *introduction* and *elimination* rules for the connective. The principle of *conservation of proof* states that these rules are inverse to one another — the elimination rule cannot extract more information (in the form of a proof) than was put into it by the introduction rule, and the introduction rules can be used to reconstruct a proof from the information extracted from it by the elimination rules.

**Truth** Our first proposition is trivially true. No information goes into proving it, and so no information can be obtained from it.

$$\frac{}{\Gamma \vdash \top \text{ true}} \tag{31.2a}$$

*(no elimination rule)*

(31.2b)



Preview

**Conjunction** Conjunction expresses the truth of both of its conjuncts.

$$\frac{\Gamma \vdash \phi_1 \ \mathsf{true} \quad \Gamma \vdash \phi_2 \ \mathsf{true}}{\Gamma \vdash \phi_1 \wedge \phi_2 \ \mathsf{true}} \tag{31.3a}$$

$$\frac{\Gamma \vdash \phi_1 \wedge \phi_2 \ \mathsf{true}}{\Gamma \vdash \phi_1 \ \mathsf{true}} \tag{31.3b}$$

$$\frac{\Gamma \vdash \phi_1 \wedge \phi_2 \ \mathsf{true}}{\Gamma \vdash \phi_2 \ \mathsf{true}} \tag{31.3c}$$

**Implication** Implication states the truth of a proposition under an assumption.

$$\frac{\Gamma, \phi_1 \ \mathsf{true} \vdash \phi_2 \ \mathsf{true}}{\Gamma \vdash \phi_1 \supset \phi_2 \ \mathsf{true}} \tag{31.4a}$$

$$\frac{\Gamma \vdash \phi_1 \supset \phi_2 \ \mathsf{true} \quad \Gamma \vdash \phi_1 \ \mathsf{true}}{\Gamma \vdash \phi_2 \ \mathsf{true}} \tag{31.4b}$$

**Falsehood** Falsehood expresses the trivially false (refutable) proposition.

*(no introduction rule)*

(31.5a)

$$\frac{\Gamma \vdash \bot \ \mathsf{true}}{\Gamma \vdash \phi \ \mathsf{true}} \tag{31.5b}$$

**Disjunction** Disjunction expresses the truth of either (or both) of two propositions.

$$\frac{\Gamma \vdash \phi_1 \ \mathsf{true}}{\Gamma \vdash \phi_1 \vee \phi_2 \ \mathsf{true}} \tag{31.6a}$$

$$\frac{\Gamma \vdash \phi_2 \ \mathsf{true}}{\Gamma \vdash \phi_1 \vee \phi_2 \ \mathsf{true}} \tag{31.6b}$$

$$\frac{\Gamma \vdash \phi_1 \vee \phi_2 \ \mathsf{true} \quad \Gamma, \phi_1 \ \mathsf{true} \vdash \phi \ \mathsf{true} \quad \Gamma, \phi_2 \ \mathsf{true} \vdash \phi \ \mathsf{true}}{\Gamma \vdash \phi \ \mathsf{true}} \tag{31.6c}$$

**Negation** The negation, $\neg\phi$, of a proposition, $\phi$, may be defined as the implication $\phi \supset \bot$. This means that $\neg\phi \ \mathsf{true}$ if $\phi \ \mathsf{true} \vdash \bot \ \mathsf{true}$, which is to say that the truth of $\phi$ is *refutable* in that we may derive a proof of falsehood from any purported proof of $\phi$. Because constructive truth is identified with the existence of a proof, the implied semantics of negation is rather strong. In particular, a problem, $\phi$, is *open* exactly when we can neither affirm nor refute it. This is in contrast to the classical conception of truth, which assigns a fixed truth value to each proposition, so that every proposition is either true or false.

Preview

### 31.2.2 Proof Terms

The key to the propositions-as-types principle is to make explict the forms of proof. The basic judgment $\phi$ true, which states that $\phi$ has a proof, is replaced by the judgment $p : \phi$, stating that $p$ is a proof of $\phi$. (Sometimes $p$ is called a "proof term", but we will simply call $p$ a "proof.") The hypothetical judgment is modified correspondingly, with variables standing for the presumed, but unknown, proofs:

$$x_1 : \phi_1, \ldots, x_n : \phi_n \vdash p : \phi.$$

We again let $\Gamma$ range over such hypothesis lists, subject to the restriction that no variable occurs more than once.

The syntax of proof terms is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Prf | $p$ | ::= | $\mathtt{true\text{-}I}$ | $\langle\rangle$ | truth intro |
| | | | $\mathtt{and\text{-}I}(p_1; p_2)$ | $\langle p_1, p_2\rangle$ | conj. intro |
| | | | $\mathtt{and\text{-}E}[\mathtt{l}](p)$ | $p \cdot \mathtt{l}$ | conj. elim |
| | | | $\mathtt{and\text{-}E}[\mathtt{r}](p)$ | $p \cdot \mathtt{r}$ | conj. elim |
| | | | $\mathtt{imp\text{-}I}(x.p)$ | $\lambda\,(x)\,p$ | impl. intro |
| | | | $\mathtt{imp\text{-}E}(p_1; p_2)$ | $p_1(p_2)$ | impl. elim |
| | | | $\mathtt{false\text{-}E}(p)$ | $\mathtt{abort}(p)$ | false elim |
| | | | $\mathtt{or\text{-}I}[\mathtt{l}](p)$ | $\mathtt{l} \cdot p$ | disj. intro |
| | | | $\mathtt{or\text{-}I}[\mathtt{r}](p)$ | $\mathtt{r} \cdot p$ | disj. intro |
| | | | $\mathtt{or\text{-}E}(p; x_1.p_1; x_2.p_2)$ | $\mathtt{case}\, p\, \{\mathtt{l} \cdot x_1 \hookrightarrow p_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow p_2\}$ | disj. elim |

The concrete syntax of proof terms is chosen to stress the correspondence between propositions and types discussed in Section 31.4 below.

The rules of constructive propositional logic may be restated using proof terms as follows.

$$\frac{}{\Gamma \vdash \langle\rangle : \top} \tag{31.7a}$$

$$\frac{\Gamma \vdash p_1 : \phi_1 \quad \Gamma \vdash p_2 : \phi_2}{\Gamma \vdash \langle p_1, p_2\rangle : \phi_1 \wedge \phi_2} \tag{31.7b}$$

$$\frac{\Gamma \vdash p_1 : \phi_1 \wedge \phi_2}{\Gamma \vdash p_1 \cdot \mathtt{l} : \phi_1} \tag{31.7c}$$

$$\frac{\Gamma \vdash p_1 : \phi_1 \wedge \phi_2}{\Gamma \vdash p_1 \cdot \mathtt{r} : \phi_2} \tag{31.7d}$$

$$\frac{\Gamma, x : \phi_1 \vdash p_2 : \phi_2}{\Gamma \vdash \lambda\,(x)\,p_2 : \phi_1 \supset \phi_2} \tag{31.7e}$$

$$\frac{\Gamma \vdash p : \phi_1 \supset \phi_2 \quad \Gamma \vdash p_1 : \phi_1}{\Gamma \vdash p(p_1) : \phi_2} \tag{31.7f}$$

Preview

$$\frac{\Gamma \vdash p : \bot}{\Gamma \vdash \mathtt{abort}(p) : \phi} \tag{31.7g}$$

$$\frac{\Gamma \vdash p_1 : \phi_1}{\Gamma \vdash \mathtt{l} \cdot p_1 : \phi_1 \vee \phi_2} \tag{31.7h}$$

$$\frac{\Gamma \vdash p_2 : \phi_2}{\Gamma \vdash \mathtt{r} \cdot p_2 : \phi_1 \vee \phi_2} \tag{31.7i}$$

$$\frac{\Gamma \vdash p : \phi_1 \vee \phi_2 \quad \Gamma, x_1 : \phi_1 \vdash p_1 : \phi \quad \Gamma, x_2 : \phi_2 \vdash p_2 : \phi}{\Gamma \vdash \mathtt{case}\, p\, \{\mathtt{l} \cdot x_1 \hookrightarrow p_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow p_2\} : \phi} \tag{31.7j}$$

## 31.3 Proof Dynamics

Proof terms in constructive logic are equipped with a dynamics by *Gentzen's Principle*, which states that the eliminatory forms are to be thought of as inverse to the introductory forms. One aspect of Gentzen's Principle is the principle of *conservation of proof*, which states that the information introduced into a proof of a proposition may be extracted without loss by elimination. So, for example, we may state that conjunction elimination is post-inverse to conjunction introduction by the definitional equivalences

$$\frac{\Gamma \vdash p_1 : \phi_1 \quad \Gamma \vdash p_2 : \phi_2}{\Gamma \vdash \langle p_1, p_2 \rangle \cdot \mathtt{l} \equiv p_1 : \phi_1} \tag{31.8a}$$

$$\frac{\Gamma \vdash p_1 : \phi_1 \quad \Gamma \vdash p_2 : \phi_2}{\Gamma \vdash \langle p_1, p_2 \rangle \cdot \mathtt{r} \equiv p_2 : \phi_2} \tag{31.8b}$$

Another aspect of Gentzen's Principle is that principle of *reversability of proof*, which states that every proof may be reconstructed from the information that may be extracted from it by elimination. In the case of conjunction this may be stated by the definitional equivalence

$$\frac{\Gamma \vdash p_1 : \phi_1 \quad \Gamma \vdash p_2 : \phi_2}{\Gamma \vdash \langle p \cdot \mathtt{l}, p \cdot \mathtt{r} \rangle \equiv p : \phi_1 \wedge \phi_2} \tag{31.9}$$

Similar equivalences may be stated for the other connectives. For example, the conservation and reversability principles for implication are given by these rules:

$$\frac{\Gamma, x : \phi_1 \vdash p_2 : \phi_2 \quad \Gamma \vdash p_2 : \phi_2}{\Gamma \vdash (\lambda\,(x)\, p_2)(p_1) \equiv [p_1/x]p_2 : \phi_2} \tag{31.10a}$$

$$\frac{\Gamma \vdash p : \phi_1 \supset \phi_2}{\Gamma \vdash \lambda\,(x)\,(p(x)) \equiv p : \phi_1 \supset \phi_2} \tag{31.10b}$$

The corresponding rules for disjunction and falsehood are given as follows:

$$\frac{\Gamma \vdash p : \phi_1 \vee \phi_2 \quad \Gamma, x_1 : \phi_1 \vdash p_1 : \psi \quad \Gamma, x_2 : \phi_2 \vdash p_2 : \psi}{\Gamma \vdash \mathtt{case}\, \mathtt{l} \cdot p\, \{\mathtt{l} \cdot x_1 \hookrightarrow p_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow p_2\} \equiv [p/x_1]p_1 : \psi} \tag{31.11a}$$

Preview

$$\frac{\Gamma \vdash p : \phi_1 \vee \phi_2 \quad \Gamma, x_1 : \phi_1 \vdash p_1 : \psi \quad \Gamma, x_2 : \phi_2 \vdash p_2 : \psi}{\Gamma \vdash \mathtt{case}\,\mathtt{r} \cdot p\,\{\mathtt{l} \cdot x_1 \hookrightarrow p_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow p_2\} \equiv [p/x_2]p_2 : \psi} \tag{31.11b}$$

$$\frac{\Gamma \vdash p : \phi_1 \vee \phi_2 \quad \Gamma, x : \phi_1 \vee \phi_2 \vdash q : \psi}{\Gamma \vdash [p/x]q \equiv \mathtt{case}\,p\,\{\mathtt{l} \cdot x_1 \hookrightarrow [\mathtt{l} \cdot x_1/x]q \mid \mathtt{r} \cdot x_2 \hookrightarrow [\mathtt{r} \cdot x_2/x]q\} : \psi} \tag{31.11c}$$

$$\frac{\Gamma \vdash p : \bot \quad \Gamma, x : \bot \vdash q : \psi}{\Gamma \vdash [p/x]q \equiv \mathtt{abort}(p) : \psi} \tag{31.11d}$$

## 31.4 Propositions as Types

Reviewing the statics and dynamics of proofs in constructive logic reveals a striking similarity to the statics and dynamics of expressions of various types. For example, the introduction rule for conjunction specifies that a proof of a conjunction consists of a pair of proofs, one for each conjunct, and the elimination rule inverts this, allowing us to extract a proof of each conjunct from any proof of a conjunction. There is an obvious analogy with the static semantics of product types, whose introductory form is a pair and whose eliminatory forms are projections. Gentzen's Principle extends the analogy to the dynamics as well, so that, for example, the elimination forms for conjunction amount to projections that extract the appropriate components from an ordered pair.

The correspondence between propositions and types and between proofs and programs, is summarized by the following chart:

| Prop | Type |
|---|---|
| $\top$ | $\mathtt{unit}$ |
| $\bot$ | $\mathtt{void}$ |
| $\phi_1 \wedge \phi_2$ | $\tau_1 \times \tau_2$ |
| $\phi_1 \supset \phi_2$ | $\tau_1 \to \tau_2$ |
| $\phi_1 \vee \phi_2$ | $\tau_1 + \tau_2$ |

The correspondence between propositions and types is a cornerstone of the theory of programming languages. It exposes a deep connection between computation and deduction, and serves as a framework for the analysis of language constructs and reasoning principles by relating them to one another.

## 31.5 Notes

The propositions as types principle has its origins in the semantics of intuitionistic logic developed by Brouwer, according to which the truth of a proposition is witnessed by a construction providing computable evidence for it. The forms of evidence are determined by the form of the proposition, so that, for example, evidence for an implication is a computable function transforming evidence for the hypothesis into evidence for the conclusion. An explicit formulation of this semantics was introduced by Heyting, and further developed by a number of authors, including de Bruijn, Curry, Gentzen, Girard, Howard, Kolmogorov, Martin-Löf, and Tait. The propositions-as-types

Preview

correspondence is sometimes called the *Curry-Howard Isomorphism*, but this terminology neglects the crucial contributions of the others just mentioned. Moreover, the correspondence is not, in general, an isomorphism; rather, it is an expression of Brouwer's Dictum that the concept of proof is best explained by the more general concept of construction (program).

## Exercises

**31.1**. The *law of the excluded middle (LEM)* is the statement that every proposition $\phi$ is *decidable* in the sense that $\phi \vee \neg\phi$ true. Constructively the law of the excluded middle states that, for every proposition $\phi$, we either have a proof of $\phi$ or a refutation of $\phi$ (proof of its negation). But this is manifestly not the case, so one may suspect that the law of the excluded middle is not constructively valid. This is so, but not in the sense that the law is *refuted*, but rather in the sense that it is *not affirmed*. First, any proposition, $\phi$, for which we have a proof or a refutation is already decided, and so is decidable. Second, there are broad classes of propositions for which we can, on demand, produce a proof or a refutation. For example, it is decidable whether or not two integers are equal. Third, and most importantly, there are, and always will be, propositions, $\phi$, whose status is unresolved: it may turn out that $\phi$ is true, or it may turn out that $\phi$ is false. For all these reasons constructive logic *does not refute* the decidability propositions: $\neg\neg(\phi \vee \neg\phi)$ true for any proposition $\phi$. Prove it using the rules given in this chapter.

**31.2**. The proposition $\neg\neg\phi$ is no stronger than $\phi$: prove $\phi \supset \neg\neg\phi$ true. The *law of double-negation elimination (DNE)* states that $(\neg\neg\phi) \supset \phi$ true for every proposition $\phi$. It follows immediately from Exercise **31.1** that DNE entails LEM; prove the converse.

**31.3**. Define the relation $\phi \leq \psi$ to mean that $\phi$ true $\vdash \psi$ true according to the rules of constructive logic given above. With respect to this relation, show the following facts:

(a) It is a *pre-order*, which is say that it is reflexive and transitive.

(b) $\phi \wedge \psi$ is the *meet*, or *greatest lower bound*, of $\phi$ and $\psi$, and $\top$ is the *top*, or *greatest*, element.

(c) Show that $\phi \vee \psi$ is the *join*, or *least upper bound*, of $\phi$ and $\psi$, and that $\bot$ is the *bottom*, or *least*, element.

(d) Show that $\phi \supset \psi$ is an *exponential*, or *pseudo-complement*, in the sense that it is the *largest* $\rho$ such that $\phi \wedge \rho \leq \psi$. (The exponential $\phi \supset \psi$ is sometimes written $\psi^\phi$.)

Altogether these facts state that entailment in constructive propositional forms a *Heyting algebra*. Show that a general Heyting algebra (that is, an ordering with the above structure) is *distributive* in the sense that

$$\phi \wedge (\psi_1 \vee \psi_2) \equiv (\phi \wedge \psi_1) \vee (\phi \wedge \psi_2)$$
$$\phi \vee (\psi_1 \wedge \psi_2) \equiv (\phi \vee \psi_1) \wedge (\phi \vee \psi_2),$$

where $\phi \equiv \psi$ means $\phi \leq \psi$ and $\psi \leq \phi$.

Preview

**31.4**. In any Heying algebra we have $\phi \wedge \neg\phi \leq \bot$, which is to say that the negation is inconsistent with the negated. But $\neg\phi$ is not necessarily the *complement* of $\phi$ in the sense that $\phi \vee \neg\phi \leq \top$. A *Boolean algebra* is a Heyting algebra in which negation is always the complement of the negated: $\top \leq \phi \vee \neg\phi$ for every $\phi$. Check that the two-element Boolean algebra for which meets, joins, and exponentials are given by the classical truth tables (defining $\phi \supset \psi$ as $(\neg\phi) \vee \psi$ is Boolean algebra. Conclude that it is consistent to adjoin LEM to constructive logic, which is to say that classical logic is a special case of constructive logic in which we assume that every proposition is decidable. Being a Heyting algebra, every Boolean algebra is clearly distributive. Show that every Boolean algebra also satisfies the *de Morgan duality laws*:

$$\neg(\phi \vee \psi) \equiv \neg\phi \wedge \neg\psi$$
$$\neg(\phi \wedge \psi) \equiv \neg\phi \vee \neg\psi.$$

The first of these is valid in any Heyting algebra; the second only in a Boolean algebra.

Preview

Preview

# Chapter 32

# Classical Logic

In constructive logic a proposition is true exactly when it has a proof, a derivation of it from axioms and assumptions, and is false exactly when it has a refutation, a derivation of a contradiction from the assumption that it is true. Constructive logic is a logic of positive evidence. To affirm or deny a proposition requires a proof, either of the proposition itself, or of a contradiction, under the assumption that it has a proof. We are not always in a position to affirm or deny a proposition. An open problem is one for which we have neither a proof nor a refutation—so that, constructively speaking, it is neither true nor false.

In contrast classical logic (the one we learned in school) is a logic of perfect information in which every proposition is either true or false. We may say that classical logic corresponds to "god's view" of the world—there are no open problems, rather all propositions are either true or false. Put another way, to assert that every proposition is either true or false is to weaken the notion of truth to encompass all that is not false, dually to the constructively (and classically) valid interpretation of falsity as all that is not true. The symmetry between truth and falsity is appealing, but there is a price to pay for this: the meanings of the logical connectives are weaker in the classical case than in the constructive.

A prime example is provided by the law of the excluded middle. Constructively, this principle is not universally valid, as we have seen in Exercise **31.1**. Classically, however, it is valid, because every proposition is either considered to be either false or not false (which is identified with being true in classical logic, in contrast to constructive logic). Nevertheless, classical logic is consistent with constructive logic in that constructive logic does not refute classical logic. As we have seen, constructive logic proves that the law of the excluded middle is positively not refuted (its double negation is constructively true). This shows that constructive logic is stronger (more expressive) than classical logic, because it can express more distinctions (namely, between affirmation and irrefutability), and because it is consistent with classical logic (the law of the excluded middle can be added without contradiction).

Proofs in constructive logic have computational content: they can be executed as programs, and their behavior is constrained by their type. Proofs in classical logic also have computational content, but in a weaker sense than in classical logic. Rather than positively affirm a proposition, a proof in classical logic is a computation that cannot be refuted. Computationally, a refutation

Preview

consists of a continuation, or control stack, that takes a proof of a proposition and derives a contradiction from it. So a proof of a proposition in classical logic is a computation that, when given a refutation of that proposition derives a contradiction, witnessing the impossibility of refuting it. In this sense the law of the excluded middle has a proof, precisely because it is irrefutable.

## 32.1 Classical Logic

In constructive logic a connective is defined by giving its introduction and elimination rules. In classical logic a connective is defined by giving its truth and falsity conditions. Its truth rules correspond to introduction, and its falsity rules to elimination. The symmetry between truth and falsity is expressed by the principle of indirect proof. To show that $\phi$ true it is enough to show that $\phi$ false entails a contradiction, and, conversely, to show that $\phi$ false it is enough to show that $\phi$ true leads to a contradiction. Although the second of these is constructively valid, the first is fundamentally classical, expressing the principle of indirect proof.

### 32.1.1 Provability and Refutability

There are three basic judgment forms in classical logic:

1. $\phi$ true, stating that the proposition $\phi$ is provable;
2. $\phi$ false, stating that the proposition $\phi$ is refutable;
3. #, stating that a contradiction has been derived.

These are extended to hypothetical judgments in which we admit both provability and refutability assumptions:

$$\phi_1 \text{ false}, \ldots, \phi_m \text{ false } \psi_1 \text{ true}, \ldots, \psi_n \text{ true} \vdash J.$$

The hypotheses are divided into two zones, one for falsity assumptions, $\Delta$, and one for truth assumptions, $\Gamma$.

The rules of classical logic are organized around the symmetry between truth and falsity, which is mediated by the contradiction judgment.

The hypothetical judgment is reflexive:

$$\frac{}{\Delta, \phi \text{ false } \Gamma \vdash \phi \text{ false}} \tag{32.1a}$$

$$\frac{}{\Delta\ \Gamma, \phi \text{ true} \vdash \phi \text{ true}} \tag{32.1b}$$

The remaining rules are stated so that the structural properties of weakening, contraction, and transitivity are admissible.

A contradiction arises when a proposition is judged to be both true and false. A proposition is true if its falsity is absurd, and is false if its truth is absurd.

$$\frac{\Delta\ \Gamma \vdash \phi \text{ false} \quad \Delta\ \Gamma \vdash \phi \text{ true}}{\Delta\ \Gamma \vdash \#} \tag{32.1c}$$



Preview

$$\frac{\Delta, \phi \text{ false } \Gamma \vdash \#}{\Delta\ \Gamma \vdash \phi \text{ true}} \tag{32.1d}$$

$$\frac{\Delta\ \Gamma, \phi \text{ true} \vdash \#}{\Delta\ \Gamma \vdash \phi \text{ false}} \tag{32.1e}$$

Truth is trivially true, and cannot be refuted.

$$\frac{}{\Delta\ \Gamma \vdash \top \text{ true}} \tag{32.1f}$$

A conjunction is true if both conjuncts are true, and is false if either conjunct is false.

$$\frac{\Delta\ \Gamma \vdash \phi_1 \text{ true} \quad \Delta\ \Gamma \vdash \phi_2 \text{ true}}{\Delta\ \Gamma \vdash \phi_1 \wedge \phi_2 \text{ true}} \tag{32.1g}$$

$$\frac{\Delta\ \Gamma \vdash \phi_1 \text{ false}}{\Delta\ \Gamma \vdash \phi_1 \wedge \phi_2 \text{ false}} \tag{32.1h}$$

$$\frac{\Delta\ \Gamma \vdash \phi_2 \text{ false}}{\Delta\ \Gamma \vdash \phi_1 \wedge \phi_2 \text{ false}} \tag{32.1i}$$

Falsity is trivially false, and cannot be proved.

$$\frac{}{\Delta\ \Gamma \vdash \bot \text{ false}} \tag{32.1j}$$

A disjunction is true if either disjunct is true, and is false if both disjuncts are false.

$$\frac{\Delta\ \Gamma \vdash \phi_1 \text{ true}}{\Delta\ \Gamma \vdash \phi_1 \vee \phi_2 \text{ true}} \tag{32.1k}$$

$$\frac{\Delta\ \Gamma \vdash \phi_2 \text{ true}}{\Delta\ \Gamma \vdash \phi_1 \vee \phi_2 \text{ true}} \tag{32.1l}$$

$$\frac{\Delta\ \Gamma \vdash \phi_1 \text{ false} \quad \Delta\ \Gamma \vdash \phi_2 \text{ false}}{\Delta\ \Gamma \vdash \phi_1 \vee \phi_2 \text{ false}} \tag{32.1m}$$

Negation inverts the sense of each judgment:

$$\frac{\Delta\ \Gamma \vdash \phi \text{ false}}{\Delta\ \Gamma \vdash \neg\phi \text{ true}} \tag{32.1n}$$

$$\frac{\Delta\ \Gamma \vdash \phi \text{ true}}{\Delta\ \Gamma \vdash \neg\phi \text{ false}} \tag{32.1o}$$

An implication is true if its conclusion is true whenever the assumption is true, and is false if its conclusion is false yet its assumption is true.

$$\frac{\Delta\ \Gamma, \phi_1 \text{ true} \vdash \phi_2 \text{ true}}{\Delta\ \Gamma \vdash \phi_1 \supset \phi_2 \text{ true}} \tag{32.1p}$$

$$\frac{\Delta\ \Gamma \vdash \phi_1 \text{ true} \quad \Delta\ \Gamma \vdash \phi_2 \text{ false}}{\Delta\ \Gamma \vdash \phi_1 \supset \phi_2 \text{ false}} \tag{32.1q}$$

Preview

### 32.1.2 Proofs and Refutations

To explain the dynamics of classical proofs we first introduce an explicit syntax for proofs and refutations. We will define three hypothetical judgments for classical logic with explicit derivations:

1. $\Delta\ \Gamma \vdash p : \phi$, stating that $p$ is a proof of $\phi$;
2. $\Delta\ \Gamma \vdash k \div \phi$, stating that $k$ is a refutation of $\phi$;
3. $\Delta\ \Gamma \vdash k \mathbin{\#} p$, stating that $k$ and $p$ are contradictory.

The falsity assumptions, $\Delta$, are represented by a context of the form

$$u_1 \div \phi_1, \ldots, u_m \div \phi_m,$$

where $m \geq 0$, in which the variables $u_1, \ldots, u_n$ stand for refutations. The truth assumptions, $\Gamma$, are represented by a context of the form

$$x_1 : \psi_1, \ldots, x_n : \psi_n,$$

where $n \geq 0$, in which the variables $x_1, \ldots, x_n$ stand for proofs.

The syntax of proofs and refutations is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Prf | $p$ | $::=$ | $\texttt{true-T}$ | $\langle\rangle$ | truth |
| | | | $\texttt{and-T}(p_1; p_2)$ | $\langle p_1, p_2\rangle$ | conjunction |
| | | | $\texttt{or-T}[\texttt{l}](p)$ | $\texttt{l} \cdot p$ | disjunction left |
| | | | $\texttt{or-T}[\texttt{r}](p)$ | $\texttt{r} \cdot p$ | disjunction right |
| | | | $\texttt{not-T}(k)$ | $\texttt{not}(k)$ | negation |
| | | | $\texttt{imp-T}(x.p)$ | $\lambda\,(x)\,p$ | implication |
| | | | $\texttt{ccr}(u.(k \mathbin{\#} p))$ | $\texttt{ccr}(u.(k \mathbin{\#} p))$ | contradiction |
| Ref | $k$ | $::=$ | $\texttt{false-F}$ | $\texttt{abort}$ | falsehood |
| | | | $\texttt{and-F}[\texttt{l}](k)$ | $\texttt{fst}\,; k$ | conjunction left |
| | | | $\texttt{and-F}[\texttt{r}](k)$ | $\texttt{snd}\,; k$ | conjunction right |
| | | | $\texttt{or-F}(k_1; k_2)$ | $\texttt{case}(k_1; k_2)$ | disjunction |
| | | | $\texttt{not-F}(p)$ | $\texttt{not}(p)$ | negation |
| | | | $\texttt{imp-F}(p; k)$ | $\texttt{ap}(p)\,; k$ | implication |
| | | | $\texttt{ccp}(x.(k \mathbin{\#} p))$ | $\texttt{ccp}(x.(k \mathbin{\#} p))$ | contradiction |

Proofs serve as evidence for truth judgments, and refutations serve as evidence for false judgments. Contradictions are witnessed by the juxtaposition of a proof and a refutation.

A contradiction arises whenever a proposition is both true and false:

$$\frac{\Delta\ \Gamma \vdash k \div \phi \quad \Delta\ \Gamma \vdash p : \phi}{\Delta\ \Gamma \vdash k \mathbin{\#} p} \tag{32.2a}$$

Truth and falsity are defined symmetrically in terms of contradiction:

$$\frac{\Delta, u \div \phi\ \Gamma \vdash k \mathbin{\#} p}{\Delta\ \Gamma \vdash \texttt{ccr}(u.(k \mathbin{\#} p)) : \phi} \tag{32.2b}$$

Preview

$$\frac{\Delta\ \Gamma, x : \phi \vdash k \mathbin{\#} p}{\Delta\ \Gamma \vdash \mathtt{ccp}(x.(k \mathbin{\#} p)) \div \phi} \tag{32.2c}$$

Reflexivity corresponds to the use of a variable hypothesis:

$$\frac{}{\Delta, u \div \phi\ \Gamma \vdash u \div \phi} \tag{32.2d}$$

$$\frac{}{\Delta\ \Gamma, x : \phi \vdash x : \phi} \tag{32.2e}$$

The other structure properties are admissible.

Truth is trivially true, and cannot be refuted.

$$\frac{}{\Delta\ \Gamma \vdash \langle\rangle : \top} \tag{32.2f}$$

A conjunction is true if both conjuncts are true, and is false if either conjunct is false.

$$\frac{\Delta\ \Gamma \vdash p_1 : \phi_1 \quad \Delta\ \Gamma \vdash p_2 : \phi_2}{\Delta\ \Gamma \vdash \langle p_1, p_2\rangle : \phi_1 \wedge \phi_2} \tag{32.2g}$$

$$\frac{\Delta\ \Gamma \vdash k_1 \div \phi_1}{\Delta\ \Gamma \vdash \mathtt{fst}; k_1 \div \phi_1 \wedge \phi_2} \tag{32.2h}$$

$$\frac{\Delta\ \Gamma \vdash k_2 \div \phi_2}{\Delta\ \Gamma \vdash \mathtt{snd}; k_2 \div \phi_1 \wedge \phi_2} \tag{32.2i}$$

Falsity is trivially false, and cannot be proved.

$$\frac{}{\Delta\ \Gamma \vdash \mathtt{abort} \div \bot} \tag{32.2j}$$

A disjunction is true if either disjunct is true, and is false if both disjuncts are false.

$$\frac{\Delta\ \Gamma \vdash p_1 : \phi_1}{\Delta\ \Gamma \vdash \mathtt{l} \cdot p_1 : \phi_1 \vee \phi_2} \tag{32.2k}$$

$$\frac{\Delta\ \Gamma \vdash p_2 : \phi_2}{\Delta\ \Gamma \vdash \mathtt{r} \cdot p_2 : \phi_1 \vee \phi_2} \tag{32.2l}$$

$$\frac{\Delta\ \Gamma \vdash k_1 \div \phi_1 \quad \Delta\ \Gamma \vdash k_2 \div \phi_2}{\Delta\ \Gamma \vdash \mathtt{case}(k_1; k_2) \div \phi_1 \vee \phi_2} \tag{32.2m}$$

Negation inverts the sense of each judgment:

$$\frac{\Delta\ \Gamma \vdash k \div \phi}{\Delta\ \Gamma \vdash \mathtt{not}(k) : \neg\phi} \tag{32.2n}$$

$$\frac{\Delta\ \Gamma \vdash p : \phi}{\Delta\ \Gamma \vdash \mathtt{not}(p) \div \neg\phi} \tag{32.2o}$$

Preview

An implication is true if its conclusion is true whenever the assumption is true, and is false if its conclusion is false, yet its assumption is true.

$$\frac{\Delta\ \Gamma, x : \phi_1 \vdash p_2 : \phi_2}{\Delta\ \Gamma \vdash \lambda\,(x)\ p_2 : \phi_1 \supset \phi_2} \tag{32.2p}$$

$$\frac{\Delta\ \Gamma \vdash p_1 : \phi_1 \quad \Delta\ \Gamma \vdash k_2 \div \phi_2}{\Delta\ \Gamma \vdash \mathtt{ap}(p_1)\,;k_2 \div \phi_1 \supset \phi_2} \tag{32.2q}$$

## 32.2 Deriving Elimination Forms

The price of achieving a symmetry between truth and falsity in classical logic is that we must very often rely on the principle of indirect proof: to show that a proposition is true, we often must derive a contradiction from the assumption of its falsity. For example, a proof of

$$(\phi \wedge (\psi \wedge \theta)) \supset (\theta \wedge \phi)$$

in classical logic has the form

$$\lambda\,(w)\ \mathtt{ccr}(u.(k \mathbin{\#} w)),$$

where $k$ is the refutation

$$\mathtt{fst}\,;\mathtt{ccp}(x.(\mathtt{snd}\,;\mathtt{ccp}(y.(\mathtt{snd}\,;\mathtt{ccp}(z.(u \mathbin{\#} \langle z, x\rangle))) \mathbin{\#} y)) \mathbin{\#} w)).$$

And yet in constructive logic this proposition has a direct proof that avoids the circumlocutions of proof by contradiction:

$$\lambda\,(w)\ \langle w \cdot \mathtt{r} \cdot \mathtt{r}, w \cdot \mathtt{l}\rangle.$$

But this proof cannot be expressed (as is) in classical logic, because classical logic lacks the elimination forms of constructive logic.

However, we may package the use of indirect proof into a slightly more palatable form by deriving the elimination rules of constructive logic. For example, the rule

$$\frac{\Delta\ \Gamma \vdash \phi \wedge \psi\ \mathsf{true}}{\Delta\ \Gamma \vdash \phi\ \mathsf{true}}$$

is derivable in classical logic:

$$\dfrac{\dfrac{\dfrac{}{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \phi\ \mathsf{false}}}{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \phi \wedge \psi\ \mathsf{false}} \quad \dfrac{\Delta\ \Gamma \vdash \phi \wedge \psi\ \mathsf{true}}{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \phi \wedge \psi\ \mathsf{true}}}{\dfrac{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \#}{\Delta\ \Gamma \vdash \phi\ \mathsf{true}}}$$

The other elimination forms are derivable in a similar manner, in each case relying on indirect proof to construct a proof of the truth of a proposition from a derivation of a contradiction from the assumption of its falsity.



Preview

The derivations of the elimination forms of constructive logic are most easily exhibited using proof and refutation expressions, as follows:

$$\begin{aligned}
\mathtt{abort}(p) &= \mathtt{ccr}(u.(\mathtt{abort} \mathbin{\#} p)) \\
p \cdot \mathtt{l} &= \mathtt{ccr}(u.(\mathtt{fst}\,; u \mathbin{\#} p)) \\
p \cdot \mathtt{r} &= \mathtt{ccr}(u.(\mathtt{snd}\,; u \mathbin{\#} p)) \\
p_1(p_2) &= \mathtt{ccr}(u.(\mathtt{ap}(p_2)\,; u \mathbin{\#} p_1)) \\
\mathtt{case}\, p_1\, \{\mathtt{l} \cdot x \hookrightarrow p_2 \mid \mathtt{r} \cdot y \hookrightarrow p\} &= \mathtt{ccr}(u.(\mathtt{case}(\mathtt{ccp}(x.(u \mathbin{\#} p_2)); \mathtt{ccp}(y.(u \mathbin{\#} p))) \mathbin{\#} p_1))
\end{aligned}$$

The expected elimination rules are valid for these definitions. For example, the rule

$$\frac{\Delta\ \Gamma \vdash p_1 : \phi \supset \psi \quad \Delta\ \Gamma \vdash p_2 : \phi}{\Delta\ \Gamma \vdash p_1(p_2) : \psi} \tag{32.3}$$

is derivable using the definition of $p_1(p_2)$ given above. By suppressing proof terms, we may derive the corresponding provability rule

$$\frac{\Delta\ \Gamma \vdash \phi \supset \psi\ \mathsf{true} \quad \Delta\ \Gamma \vdash \phi\ \mathsf{true}}{\Delta\ \Gamma \vdash \psi\ \mathsf{true}}\ . \tag{32.4}$$

## 32.3 Proof Dynamics

The dynamics of classical logic arises from the simplification of the contradiction between a proof and a refutation of a proposition. To make this explicit we will define a transition system whose states are contradictions $k \mathbin{\#} p$ consisting of a proof, $p$, and a refutation, $k$, of the same proposition. The steps of the computation consist of simplifications of the contradictory state based on the form of $p$ and $k$.

The truth and falsity rules for the connectives play off one another in a pleasing manner:

$$\mathtt{fst}\,; k \mathbin{\#} \langle p_1, p_2 \rangle \longmapsto k \mathbin{\#} p_1 \tag{32.5a}$$
$$\mathtt{snd}\,; k \mathbin{\#} \langle p_1, p_2 \rangle \longmapsto k \mathbin{\#} p_2 \tag{32.5b}$$
$$\mathtt{case}(k_1; k_2) \mathbin{\#} \mathtt{l} \cdot p_1 \longmapsto k_1 \mathbin{\#} p_1 \tag{32.5c}$$
$$\mathtt{case}(k_1; k_2) \mathbin{\#} \mathtt{r} \cdot p_2 \longmapsto k_2 \mathbin{\#} p_2 \tag{32.5d}$$
$$\mathtt{not}(p) \mathbin{\#} \mathtt{not}(k) \longmapsto k \mathbin{\#} p \tag{32.5e}$$
$$\mathtt{ap}(p_1)\,; k \mathbin{\#} \lambda\,(x)\, p_2 \longmapsto k \mathbin{\#} [p_1/x]p_2 \tag{32.5f}$$

The rules of indirect proof give rise to the following transitions:

$$\mathtt{ccp}(x.(k_1 \mathbin{\#} p_1)) \mathbin{\#} p_2 \longmapsto [p_2/x]k_1 \mathbin{\#} [p_2/x]p_1 \tag{32.5g}$$
$$k_1 \mathbin{\#} \mathtt{ccr}(u.(k_2 \mathbin{\#} p_2)) \longmapsto [k_1/u]k_2 \mathbin{\#} [k_1/u]p_2 \tag{32.5h}$$

The first of these defines the behavior of the refutation of $\phi$ that proceeds by contradicting the assumption that $\phi$ is true. This refutation is activated by presenting it with a proof of $\phi$, which

Preview

is then substituted for the assumption in the new state. Thus, "ccp" stands for "call with current proof." The second transition defines the behavior of the proof of $\phi$ that proceeds by contradicting the assumption that $\phi$ is false. This proof is activated by presenting it with a refutation of $\phi$, which is then substituted for the assumption in the new state. Thus, "ccr" stands for "call with current refutation."

Rules (32.5g) to (32.5h) overlap in that there are two possible transitions for a state of the form

$$\mathtt{ccp}(x.(k_1 \mathbin{\#} p_1)) \mathbin{\#} \mathtt{ccr}(u.(k_2 \mathbin{\#} p_2)),$$

one to the state $[p/x]k_1 \mathbin{\#} [p/x]p_1$, where $p$ is $\mathtt{ccr}(u.(k_2 \mathbin{\#} p_2))$, and one to the state $[k/u]k_2 \mathbin{\#} [k/u]p_2$, where $k$ is $\mathtt{ccp}(x.(k_1 \mathbin{\#} p_1))$. The dynamics of classical logic is therefore non-deterministic. To avoid this one may impose a priority ordering among the two cases, preferring one transition over the other when there is a choice. Preferring the first corresponds to a "lazy" dynamics for proofs, because we pass the unevaluated proof, $p$, to the refutation on the left, which is thereby activated. Preferring the second corresponds to an "eager" dynamics for proofs, in which we pass the unevaluated refutation, $k$, to the proof, which is thereby activated.

All proofs in classical logic may be consider to proceed by contradicting the assumption that it is false, by proving the theorem. In terms of the classical logic machine the initial and final states of a computation are as follows:

$$\frac{}{\mathtt{halt}_\phi \mathbin{\#} p \ \mathsf{initial}} \tag{32.6a}$$

$$\frac{p \ \mathsf{canonical}}{\mathtt{halt}_\phi \mathbin{\#} p \ \mathsf{final}} \tag{32.6b}$$

where $p$ is a proof of $\phi$, and $\mathtt{halt}_\phi$ is the assumed refutation of $\phi$. The judgment $p \ \mathsf{canonical}$ states that $p$ is a canonical proof, which is defined to hold of any proof other than an indirect proof. Execution consists of driving a general proof to a canonical proof, under the assumption that the theorem is false.

**Theorem 32.1** (Preservation)**.** *If $k \div \phi$, $p : \phi$, and $k \mathbin{\#} p \longmapsto k' \mathbin{\#} p'$, then there exists $\phi'$ such that $k' \div \phi'$ and $p' : \phi'$.*

*Proof.* By rule induction on the dynamics of classical logic. □

**Theorem 32.2** (Progress)**.** *If $k \div \phi$ and $p : \phi$, then either $k \mathbin{\#} p$ final or $k \mathbin{\#} p \longmapsto k' \mathbin{\#} p'$.*

*Proof.* By rule induction on the statics of classical logic. □



Preview

## 32.4 Law of the Excluded Middle

The law of the excluded middle is derivable in classical logic:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\phi \vee \neg\phi \text{ false}, \phi \text{ true} \vdash \phi \text{ true}}}{\phi \vee \neg\phi \text{ false}, \phi \text{ true} \vdash \phi \vee \neg\phi \text{ true}} \quad \dfrac{}{\phi \vee \neg\phi \text{ false}, \phi \text{ true} \vdash \phi \vee \neg\phi \text{ false}}}{\phi \vee \neg\phi \text{ false}, \phi \text{ true} \vdash \#}}{\phi \vee \neg\phi \text{ false} \vdash \phi \text{ false}}}{\phi \vee \neg\phi \text{ false} \vdash \neg\phi \text{ true}}}{\phi \vee \neg\phi \text{ false} \vdash \phi \vee \neg\phi \text{ true}} \qquad \dfrac{}{\phi \vee \neg\phi \text{ false} \vdash \phi \vee \neg\phi \text{ false}}}{\dfrac{\phi \vee \neg\phi \text{ false} \vdash \#}{\phi \vee \neg\phi \text{ true}}}$$

When written out using explicit proofs and refutations, we obtain the proof term $p_0 : \phi \vee \neg\phi$:

$$\mathtt{ccr}(u.(u \mathbin{\#} \mathtt{r} \cdot \mathtt{not}(\mathtt{ccp}(x.(u \mathbin{\#} \mathtt{l} \cdot x))))).$$

To understand the computational meaning of this proof, let us juxtapose it with a refutation, $k \div \phi \vee \neg\phi$, and simplify it using the dynamics given in Section 32.3. The first step is the transition

$$\begin{gathered} k \mathbin{\#} \mathtt{ccr}(u.(u \mathbin{\#} \mathtt{r} \cdot \mathtt{not}(\mathtt{ccp}(x.(u \mathbin{\#} \mathtt{l} \cdot x))))) \\ \longmapsto \\ k \mathbin{\#} \mathtt{r} \cdot \mathtt{not}(\mathtt{ccp}(x.(k \mathbin{\#} \mathtt{l} \cdot x))), \end{gathered}$$

wherein we have replicated $k$ so that it occurs in two places in the result state. By virtue of its type the refutation $k$ must have the form $\mathtt{case}(k_1; k_2)$, where $k_1 \div \phi$ and $k_2 \div \neg\phi$. Continuing the reduction, we obtain:

$$\begin{gathered} \mathtt{case}(k_1; k_2) \mathbin{\#} \mathtt{r} \cdot \mathtt{not}(\mathtt{ccp}(x.(\mathtt{case}(k_1; k_2) \mathbin{\#} \mathtt{l} \cdot x))) \\ \longmapsto \\ k_2 \mathbin{\#} \mathtt{not}(\mathtt{ccp}(x.(\mathtt{case}(k_1; k_2) \mathbin{\#} \mathtt{l} \cdot x))). \end{gathered}$$

By virtue of its type $k_2$ must have the form $\mathtt{not}(p_2)$, where $p_2 : \phi$, and hence the transition proceeds as follows:

$$\begin{gathered} \mathtt{not}(p_2) \mathbin{\#} \mathtt{not}(\mathtt{ccp}(x.(\mathtt{case}(k_1; k_2) \mathbin{\#} \mathtt{l} \cdot x))) \\ \longmapsto \\ \mathtt{ccp}(x.(\mathtt{case}(k_1; k_2) \mathbin{\#} \mathtt{l} \cdot x)) \mathbin{\#} p_2. \end{gathered}$$

Preview

Observe that $p_2$ is a valid proof of $\phi$. Proceeding, we obtain

$$\begin{array}{c} \mathtt{ccp}(x.(\mathtt{case}(k_1;k_2) \mathbin{\#} \mathtt{l} \cdot x)) \mathbin{\#} p_2 \\ \longmapsto \\ \mathtt{case}(k_1;k_2) \mathbin{\#} \mathtt{l} \cdot p_2 \\ \longmapsto \\ k_1 \mathbin{\#} p_2 \end{array}$$

The first of these two steps is the crux of the matter: the refutation, $k = \mathtt{case}(k_1;k_2)$, which was replicated at the outset of the derivation, is re-used, but with a different argument. At the first use, the refutation, $k$, which is provided by the context of use of the law of the excluded middle, is presented with a proof $\mathtt{r} \cdot p_1$ of $\phi \vee \neg\phi$. That is, the proof behaves as though the right disjunct of the law is true, which is to say that $\phi$ is false. If the context is such that it inspects this proof, it can only be by providing the proof, $p_2$, of $\phi$ that refutes the claim that $\phi$ is false. Should this occur, the proof of the law of the excluded middle "backtracks" the context, providing instead the proof $\mathtt{l} \cdot p_2$ to $k$, which then passes $p_2$ to $k_1$ without further incident. The proof of the law of the excluded middle boldly asserts $\neg\phi$ true, regardless of the form of $\phi$. Then, if caught in its lie by the context providing a proof of $\phi$, it "changes its mind" and asserts $\phi$ to the original context, $k$, after all. No further reversion is possible, because the context has itself provided a proof, $p_2$, of $\phi$.

The law of the excluded middle illustrates that classical proofs are to be thought of as interactions between proofs and refutations, which is to say interactions between a proof and the context in which it is used. In programming terms this corresponds to an abstract machine with an explicit control stack, or continuation, representing the context of evaluation of an expression. That expression may access the context (stack, continuation) to effect backtracking as necessary to maintain the perfect symmetry between truth and falsity. The penalty is that a closed proof of a disjunction no longer need reveal which disjunct it proves, for as we have just seen, it may, on further inspection, "change its mind."

## 32.5 The Double-Negation Translation

One consequence of the greater expressiveness of constructive logic is that classical proofs may be translated systematically into constructive proofs of a classically equivalent proposition. This means that by systematically reorganizing the classical proof we may, without changing its meaning from a classical perspective, turn it into a constructive proof of a constructively weaker proposition. This shows that there is no loss in adhering to constructive proofs, because every classical proof can be seen as a constructive proof of a constructively weaker, but classically equivalent, proposition. Moreover, it proves that classical logic is weaker (less expressive) than constructive logic, contrary to a naïve interpretation which would say that the additional reasoning principles, such as the law of the excluded middle, afforded by classical logic makes it stronger. In programming language terms adding a "feature" does not necessarily strengthen (improve the expressive power) of your language; on the contrary, it may weaken it.

Preview

We will define a translation $\phi^*$ of propositions that provides an interpretation of classical into constructive logic according to the following correspondences:

| *Classical* | *Constructive* | |
|---|---|---|
| $\Delta\ \Gamma \vdash \phi\ \mathsf{true}$ | $\neg\Delta^*\ \Gamma^* \vdash \neg\neg\phi^*\ \mathsf{true}$ | truth |
| $\Delta\ \Gamma \vdash \phi\ \mathsf{false}$ | $\neg\Delta^*\ \Gamma^* \vdash \neg\phi^*\ \mathsf{true}$ | falsity |
| $\Delta\ \Gamma \vdash \#$ | $\neg\Delta^*\ \Gamma^* \vdash \bot\ \mathsf{true}$ | contradiction |

Classical truth is weakened to constructive irrefutability; classical falsehood is represented as constructive refutability; classical contradiction is represented by constructive falsehood. Falsity assumptions are negated after translation to express their falsehood; truth assumptions are merely translated as is. Because the double negations are classically cancellable, the translation will be easily seen to yield a classically equivalent proposition. But because $\neg\neg\phi$ is constructively weaker than $\phi$, we also see that a proof in classical logic is translated to a constructive proof of a weaker statement.

Many choices for the translation of propositions are available; we have chosen one that makes the proof of the correspondence between classical and constructive logic go smoothly:

$$\begin{aligned}
\top^* &= \top \\
(\phi_1 \wedge \phi_2)^* &= \phi_1^* \wedge \phi_2^* \\
\bot^* &= \bot \\
(\phi_1 \vee \phi_2)^* &= \phi_1^* \vee \phi_2^* \\
(\phi_1 \supset \phi_2)^* &= \phi_1^* \supset \neg\neg\phi_2^* \\
(\neg\phi)^* &= \neg\phi^*
\end{aligned}$$

It is straightforward to show by induction on the rules of classical logic that the correspondences summarized above hold. Some simple lemmas are required. For example, we must show that the entailment

$$\neg\neg\phi\ \mathsf{true}\ \neg\neg\psi\ \mathsf{true} \vdash \neg\neg(\phi \wedge \psi)\ \mathsf{true}$$

is derivable in constructive logic.

## 32.6 Notes

The computational interpretation of classical logic was first explored by Griffin (1990) and Murthy (1991). The account given here was influenced by Wadler (2003), transposed by Nanevski from sequent calculus to natural deduction using multiple forms of judgment. The terminology is inspired by Lakatos (1976), an insightful and inspiring analysis of the discovery of proofs and refutations of conjectures in mathematics. Versions of the double-negation translation were originally given by Gödel and Gentzen, and have been extended and modified in numerous other studies. The computational content of the double negation translation was first elucidated by Murthy (1991), who established the connection with the continuation-passing transformation used in compilers.

Preview

## Exercises

**32.1**. If the continuation type is read as negation, the types that you showed to be inhabited your solution to Exercise **30.2**, when interpreted under the proposition-as-types interpretation, look suspiciously like the following propositions:

(a) $\phi \vee \neg\phi$.

(b) $(\neg\neg\phi) \supset \phi$.

(c) $(\neg\phi_2 \supset \neg\phi_1) \supset (\phi_1 \supset \phi_2)$.

(d) $\neg(\phi_1 \vee \phi_2) \supset (\neg\phi_1 \wedge \neg\phi_2)$.

None of these propositions is true, in general, in constructive logic. Show that each of these propositions is true in classical logic by exhibiting a proof term for each. (The first one is done for you in Section 32.4; you need only do the other three.) Compare the proof term you obtain for each with the inhabitant of the corresponding type that you gave in your solution to Exercise **30.2**.

**32.2**. Complete the proof of the double-negation interpretation sketched in Section 32.5, providing explicit proof terms for clarity. Because $(\phi \vee \neg\phi)^* = \phi^* \vee \neg\phi^*$, the double-negation translation applied to the proof of LEM (for $\phi$) given in Section 32.4 yields a proof of the double negation of LEM (for $\phi^*$) in constructive logic. How does the translated proof compare to the one you derived by hand in Exercise **31.1**?

Preview

# Part XIII

# Symbolic Data

Preview

Preview

# Chapter 33

# Symbols

A *symbol* is an atomic datum with no internal structure. Whereas a variable is given meaning by substitution, a symbol is given meaning by a family of operations indexed by symbols. A symbol is therefore just a name, or index, for an instance of a family of operations. Many different interpretations may be given to symbols according to the operations we choose to consider, giving rise to concepts such as fluid binding, dynamic classification, mutable storage, and communication channels. To each symbol is associated a type whose interpretation depends on the particular application. The type of a symbol influences the type of its associated operations under each interpretation. For example, in the case of mutable storage, the type of a symbol constrains the contents of the cell named by that symbol to values of that type. It is important to bear in mind that a symbol is *not* a value of its associated type, but only a constraint on how that symbol may be interpreted by the operations associated with it.

In this chapter we consider two constructs for computing with symbols. The first is a means of *declaring* new symbols for use within a specified *scope*. The expresssion $\texttt{new}\,a \sim \rho\,\texttt{in}\,e$ introduces a "new" symbol $a$ with associated type $\rho$ for use within $e$. The declared symbol $a$ is "new" in the sense that it is bound by the declaration within $e$, and so may be renamed at will to ensure that it differs from any finite set of active symbols. Whereas the statics determines the scope of a declared symbol, its range of significance, or *extent*, is determined by the dynamics. There are two different dynamic interpretations of symbols, the *scoped* and the *free* (short for *scope-free*) dynamics. The scoped dynamics limits the extent of the symbol to its scope; the lifetime of the symbol is restricted to the evaluation of its scope. Alternatively, under the free dynamics the extent of a symbol exceeds its scope, extending to the entire computation of which it is a part. We may say that in the free dynamics a symbol "escapes its scope," but it is more accurate to say that its scope widens to encompass the rest of the computation.

The second construct associated with symbols is the concept of a *symbol reference*, a form of expression whose sole purpose is to refer to a particular symbol. Symbol references are values of a type $\rho\ \texttt{sym}$ and have the form $\&\,a$ for some symbol $a$ with associated type $\rho$.[1] The eliminatory form

[1] In Lisp and its descendants $\&\,a$ is written $\texttt{'}a$, and is called *quotation*. Quotation in this sense is not to be confused with other forms of quotation used to form a string of characters. It is, instead, simply a way to notate an abtract symbol thought

Preview

for the type $\rho$ `sym` is a conditional branch that determines whether or not a symbol reference refers to a statically specified symbol. Crucially, the statics of the eliminatory form is carefully designed so that, in the positive case, the type associated to the referenced symbol is made manifest, whereas in the negative case, no type information is gleaned because the referenced symbol could be of any type.

## 33.1 Symbol Declaration

We will consider here an extension, **SPCF**, of **PCF** with the means to allocate new symbols. This capability will be used in subsequent chapters that make use of symbols for other purposes. In **SPCF** we will be concerned solely with symbol allocation, and the introduction and elimination of symbols as values of a type.

The syntax for symbol declaration in **SPCF** is given by the following grammar:

$$\mathsf{Exp} \quad e \quad ::= \quad \mathtt{new}[\tau](a.e) \quad \mathtt{new}\, a \sim \tau\, \mathtt{in}\, e \quad \text{generation}$$

The statics of symbol declaration makes use of a *signature*, or *symbol context*, that associates a type to each of a finite set of symbols. We use the letter $\Sigma$ to range over signatures, which are finite sets of pairs $a \sim \tau$, where $a$ is a symbol and $\tau$ is a type. The typing judgment $\Gamma \vdash_\Sigma e : \tau$ is parameterized by a signature, $\Sigma$, associating types to symbols.

The statics of symbol declaration makes use of a judgment, $\tau$ mobile, whose definition depends on whether the dynamics is scoped or not. In a scoped dynamics mobility is defined so that the computed value of a mobile type cannot depend on any symbol. By constraining the scope of a declaration to have mobile type, we can, under this interpretation, ensure that the extent of a symbol is confined to its scope. In a free dynamics every type is deemed mobile, because the dynamics ensures that the scope of a symbol is widened to accommodate the possibility that the value returned from the scope of a declaration may depend on the declared symbol. The term "mobile" reflects the informal idea that symbols may or may not be "moved" from the scope of their declaration according to the dynamics given to them. A free dynamics allows symbols to be moved freely, whereas a scoped dynamics limits their range of motion.

The statics of symbol declaration itself is given by the following rule:

$$\frac{\Gamma \vdash_{\Sigma, a \sim \rho} e : \tau \quad \tau \text{ mobile}}{\Gamma \vdash_\Sigma \mathtt{new}[\rho](a.e) : \tau} \tag{33.1}$$

As mentioned, the condition on $\tau$ is to be chosen so as to ensure that the returned value is meaningful in which dynamics we are using.

### 33.1.1 Scoped Dynamics

The scoped dynamics of symbol declaration is given by a transition judgment of the form $e \underset{\Sigma}{\mapsto} e'$ indexed by a signature, $\Sigma$, specifying the active symbols of the transition. Either $e$ or $e'$ may involve

of as an expression.



Preview

the symbols declared in $\Sigma$, but no others.

$$\frac{e \underset{\Sigma,a\sim\rho}{\longmapsto} e'}{\mathtt{new}[\rho](a.e) \underset{\Sigma}{\mapsto} \mathtt{new}[\rho](a.e')} \tag{33.2a}$$

$$\frac{e \ \mathsf{val}_\Sigma}{\mathtt{new}[\rho](a.e) \underset{\Sigma}{\mapsto} e} \tag{33.2b}$$

Rule (33.2a) specifies that evaluation takes place within the scope of the declaration of a symbol. Rule (33.2b) specifies that the declared symbol is "forgotten" once its scope has been evaluated.

The definition of the judgment $\tau$ mobile must be chosen to ensure that the following *mobility condition* is satisfied:

*If $\tau$ mobile, $\vdash_{\Sigma,a\sim\rho} e : \tau$, and $e$ val$_{\Sigma,a\sim\rho}$, then $\vdash_\Sigma e : \tau$ and $e$ val$_\Sigma$.*

For example, in the presence of symbol references (see Section 33.2 below), a function type cannot be deemed mobile, because a function may contain a reference to a local symbol. The type `nat` may only be deemed mobile if the successor is evaluated eagerly, for otherwise a symbol reference may occur within a value of this type, invalidating the condition.

**Theorem 33.1** (Preservation)**.** *If $\vdash_\Sigma e : \tau$ and $e \underset{\Sigma}{\mapsto} e'$, then $\vdash_\Sigma e' : \tau$.*

*Proof.* By induction on the dynamics of symbol declaration. Rule (33.2a) follows directly by induction, applying rule (33.1). Rule (33.2b) follows directly from the condition on mobility. □

**Theorem 33.2** (Progress)**.** *If $\vdash_\Sigma e : \tau$, then either $e \underset{\Sigma}{\mapsto} e'$, or $e$ val$_\Sigma$.*

*Proof.* There is only one rule to consider, Rule (33.1). By induction we have either $e \underset{\Sigma,a\sim\rho}{\longmapsto} e'$, in which case rule (33.2a) applies, or $e$ val$_{\Sigma,a\sim\rho}$, in which case by the mobility condition we have $e$ val$_\Sigma$, and hence rule (33.2b) applies. □

### 33.1.2 Scope-Free Dynamics

The scope-free dynamics of symbols is defined by a transition system between states of the form $\nu\,\Sigma\,\{\,e\,\}$, where $\Sigma$ is a signature and $e$ is an expression over this signature. The judgment $\nu\,\Sigma\,\{\,e\,\} \longmapsto \nu\,\Sigma'\,\{\,e'\,\}$ states that evaluation of $e$ relative to symbols $\Sigma$ results in the expression $e'$ in the extension $\Sigma'$ of $\Sigma$.

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{new}[\rho](a.e)\,\} \longmapsto \nu\,\Sigma, a\sim\rho\,\{\,e\,\}} \tag{33.3}$$

Rule (33.3) specifies that symbol generation enriches the signature with the newly introduced symbol by extending the signature for all future transitions.



Preview

All other rules of the dynamics must be changed accordingly to account for the allocated symbols. For example, the dynamics of function application cannot simply be inherited from Chapter 15, but must be reformulated as follows:

$$\frac{\nu\,\Sigma\,\{\,e_1\,\} \longmapsto \nu\,\Sigma'\,\{\,e_1'\,\}}{\nu\,\Sigma\,\{\,e_1(e_2)\,\} \longmapsto \nu\,\Sigma'\,\{\,e_1'(e_2)\,\}} \tag{33.4a}$$

$$\frac{}{\nu\,\Sigma\,\{\,\lambda\,(x{:}\tau)\,e(e_2)\,\} \longmapsto \nu\,\Sigma\,\{\,[e_2/x]e\,\}} \tag{33.4b}$$

These rules shuffle around the signature so as to account for symbol declarations within the constituent expressions of the application. Similar rules must be given for all other constructs of **SPCF**.

**Theorem 33.3** (Preservation)**.** *If* $\nu\,\Sigma\,\{\,e\,\} \longmapsto \nu\,\Sigma'\,\{\,e'\,\}$ *and* $\vdash_\Sigma e : \tau$*, then* $\Sigma' \supseteq \Sigma$ *and* $\vdash_{\Sigma'} e' : \tau$*.*

*Proof.* There is only one rule to consider, rule (33.3), which is easily handled by inversion of rule (33.1). □

**Theorem 33.4** (Progress)**.** *If* $\vdash_\Sigma e : \tau$*, then either* $e$ *val*$_\Sigma$ *or* $\nu\,\Sigma\,\{\,e\,\} \longmapsto \nu\,\Sigma'\,\{\,e'\,\}$ *for some* $\Sigma'$ *and* $e'$*.*

*Proof.* Immediate, by rule (33.3). □

## 33.2 Symbol References

Symbols are not themselves values, but they may be used to form values. One useful example is provided by the type $\tau\,\mathtt{sym}$ of *symbol references*. A value of this type has the form $\&\,a$, where $a$ is a symbol in the signature. To compute with a reference we may branch according to whether it is a reference to a specified symbol or not. The syntax of symbol references is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | ::= | $\mathtt{sym}(\tau)$ | $\tau\,\mathtt{sym}$ | symbols |
| Exp | $e$ | | $\mathtt{sym}[a]$ | $\&\,a$ | reference |
| | | | $\mathtt{is}[a][t.\tau](e;e_1;e_2)$ | $\mathtt{if}\,e\,\mathtt{is}\,a\,\mathtt{then}\,e_1\,\mathtt{ow}\,e_2$ | comparison |

The expression $\mathtt{sym}[a]$ is a reference to the symbol $a$, a value of type $\mathtt{sym}(\tau)$. The expression $\mathtt{is}[a][t.\tau](e;e_1;e_2)$ compares the value of $e$, which must be a reference to some symbol $b$, with the given symbol, $a$. If $b$ is $a$, the expression evaluates to $e_1$, and otherwise to $e_2$.

### 33.2.1 Statics

The typing rules for symbol references are as follows:

$$\frac{}{\Gamma \vdash_{\Sigma,a\sim\rho} \mathtt{sym}[a] : \mathtt{sym}(\rho)} \tag{33.5a}$$



Preview

$$\frac{\Gamma \vdash_{\Sigma,a\sim\rho} e : \mathtt{sym}(\rho') \quad \Gamma \vdash_{\Sigma,a\sim\rho} e_1 : [\rho/t]\tau \quad \Gamma \vdash_{\Sigma,a\sim\rho} e_2 : [\rho'/t]\tau}{\Gamma \vdash_{\Sigma,a\sim\rho} \mathtt{is}[a][t.\tau](e;e_1;e_2) : [\rho'/t]\tau} \tag{33.5b}$$

Rule (33.5a) is the introduction rule for the type $\mathtt{sym}(\rho)$. It states that if $a$ is a symbol with associated type $\rho$, then $\mathtt{sym}[a]$ is an expression of type $\mathtt{sym}(\rho)$. Rule (33.5b) is the elimination rule for the type $\mathtt{sym}(\rho)$. The type associated to the given symbol, $a$, is not required to be the same as the type of the symbol referred to by the expression $e$. If $e$ evaluates to a reference to $a$, then these types will coincide, but if it refers to another symbol, $b \neq a$, then these types may well differ.

With this in mind, let us examine carefully rule (33.5b). *A priori* there is a discrepancy between the type, $\rho$, of $a$ and the type, $\rho'$, of the symbol referred to by $e$. This discrepancy is mediated by the type operator $t.\tau$.[2] Regardless of the outcome of the comparison, the overall type of the expression is $[\rho'/t]\tau$. To ensure safety, we must ensure that this is a valid type for the result, regardless of whether the comparison succeeds or fails. If $e$ evaluates to the symbol $a$, then we "learn" that the types $\rho'$ and $\rho$ coincide, because the specified and referenced symbol coincide. This is reflected by the type $[\rho/t]\tau$ for $e_1$. If $e$ evaluates to some other symbol, $a' \neq a$, then the comparison evaluates to $e_2$, which is required to have type $[\rho'/t]\tau$; no further information about the type of the symbol is acquired in this branch.

### 33.2.2 Dynamics

The (scoped) dynamics of symbol references is given by the following rules:

$$\frac{}{\mathtt{sym}[a]\ \mathsf{val}_{\Sigma,a\sim\rho}} \tag{33.6a}$$

$$\frac{}{\mathtt{is}[a][t.\tau](\mathtt{sym}[a];e_1;e_2) \xmapsto[\Sigma,a\sim\rho]{} e_1} \tag{33.6b}$$

$$\frac{(a \neq a')}{\mathtt{is}[a][t.\tau](\mathtt{sym}[a'];e_1;e_2) \xmapsto[\Sigma,a\sim\rho,a'\sim\rho']{} e_2} \tag{33.6c}$$

$$\frac{e \xmapsto[\Sigma,a\sim\rho]{} e'}{\mathtt{is}[a][t.\tau](e;e_1;e_2) \xmapsto[\Sigma,a\sim\rho]{} \mathtt{is}[a][t.\tau](e';e_1;e_2)} \tag{33.6d}$$

Rules (33.6b) and (33.6c) specify that $\mathtt{is}[a][t.\tau](e;e_1;e_2)$ branches according to whether the value of $e$ is a reference to the symbol, $a$, or not.

### 33.2.3 Safety

To ensure that the mobility condition is satisfied, it is important that symbol reference types *not* be deemed mobile.

[2] See Chapter 13 for a discussion of type operators.



Preview

**Theorem 33.5** (Preservation). *If* $\vdash_\Sigma e : \tau$ *and* $e \underset{\Sigma}{\mapsto} e'$, *then* $\vdash_\Sigma e' : \tau$.

*Proof.* By rule induction on rules (33.6). The most interesting case is rule (33.6b). When the comparison is positive, the types $\rho$ and $\rho'$ must be the same, because each symbol has at most one associated type. Therefore, $e_1$, which has type $[\rho'/t]\tau$, also has type $[\rho/t]\tau$, as required. □

**Lemma 33.6** (Canonical Forms). *If* $\vdash_\Sigma e : \mathtt{sym}(\rho)$ *and* $e$ $\mathsf{val}_\Sigma$, *then* $e = \mathtt{sym}[a]$ *for some* $a$ *such that* $\Sigma = \Sigma', a \sim \rho$.

*Proof.* By rule induction on rules (33.5), taking account of the definition of values. □

**Theorem 33.7** (Progress). *Suppose that* $\vdash_\Sigma e : \tau$. *Then either* $e$ $\mathsf{val}_\Sigma$, *or there exists* $e'$ *such that* $e \underset{\Sigma}{\mapsto} e'$.

*Proof.* By rule induction on rules (33.5). For example, consider rule (33.5b), in which we have that $\mathtt{is}[a][t.\tau](e; e_1; e_2)$ has some type $\tau$ and that $e : \mathtt{sym}(\rho)$ for some $\rho$. By induction either rule (33.6d) applies, or else we have that $e$ $\mathsf{val}_\Sigma$, in which case we are assured by Lemma 33.6 that $e$ is $\mathtt{sym}[a]$ for some symbol $b$ of type $\rho$ declared in $\Sigma$. But then progress is assured by rules (33.6b) and (33.6c), because equality of symbols is decidable (either $a$ is $b$ or it is not). □

## 33.3 Notes

The concept of a symbol in a programming language was considered by McCarthy in the original formulation of Lisp (McCarthy, 1965). Unfortunately, symbols were, and often continue to be, confused with variables, as they were in the original formulation of Lisp. Although symbols are frequently encountered in dynamically typed languages, the formulation given here makes clear that they are equally sensible in statically typed languages. The present account was influenced by Pitts and Stark (1993) and on the declaration of names in the $\pi$-calculus (Milner (1999).)

## Exercises

**33.1**. The eliminatory form for symbol references given in Section 33.2 is "one-sided" in the sense that one may compare a reference to an unknown symbol to a known symbol with a known type. An alternative eliminatory form provides an *equality test* on symbol references. Formulate such a variation.

**33.2**. A list of type $(\tau\ \mathtt{sym} \times \tau)\ \mathtt{list}$ is called an *association list*. Using your solution to Exercise **33.1** define a function `find` that sends an association list to a mapping of type $\tau\ \mathtt{sym} \rightharpoonup \tau\ \mathtt{opt}$.

**33.3**. It would be more efficient to represent an association list by a balanced tree associating values to symbols, but to do so would require a total ordering on symbols (at least among the symbols with the same associated type). What obstacles do you encounter in introducing a linear ordering on symbols?



Preview

**33.4**. In Lisp a *symbolic expression*, or *s-expression*, or *sexpr*, may be thought of as a value of the recursive type

$$\texttt{sexpr} \triangleq \texttt{rec}\, s\, \texttt{is}\, [\texttt{sym} \hookrightarrow \texttt{sym}(s)\,;\texttt{nil} \hookrightarrow \texttt{unit}\,;\texttt{cons} \hookrightarrow s \times s].$$

It is customary to write $\texttt{cons}(e_0;e_1)$ for $\texttt{fold}(\texttt{cons} \cdot \langle e_0, e_1\rangle)$, where $e_0 : \texttt{sexpr}$ and $e_1 : \texttt{sexpr}$, and to write $\texttt{nil}$ for $\texttt{fold}(\texttt{nil} \cdot \langle\rangle)$. The *list notation* $(e_0, \ldots, e_{n-1})$ is then used as shorthand for the s-expression

$$\texttt{cons}(e_0;\ldots\texttt{cons}(e_{n-1};\texttt{nil})\ldots).$$

Because lists involving symbols arise frequently, it is customary to extend the quotation notation from symbols to general s-expressions so that one need not quote each symbol contained within it. Give the definition of this extension, and work out its meaning for the special case of lists described above.

**33.5**. Considering symbol allocation to be an effect, give a modal formulation of **SPCF** along the lines of **MPCF** described in Chapter 29. Consider both a scoped and a scope-free extent for symbols.

Preview

Preview

# Chapter 34

# Fluid Binding

In this chapter we return to the concept of dynamic scoping of variables that was criticized in Chapter 8. There it was observed that dynamic scoping is problematic for at least two reasons:

- A bound variable may not always be renamed in an expression without changing its meaning.
- Because the scope of a variable is resolved dynamically, type safety is compromised.

These violations of the expected behavior of variables is intolerable, because it is at variance with mathematical practice and because it compromises modularity.

It is possible, however, to recover a type-safe analogue of dynamic scoping by divorcing it from the concept of a variable, and instead introducing a new mechanism, called *fluid*, or *dynamic*, *binding* of a symbol. Fluid binding associates to a symbol a value of its associated type within a specified scope. Upon exiting that scope, the binding is dropped (or, more accurately, reverted to its binding in the surrounding context).

## 34.1 Statics

To account for fluid binding we enrich **SPCF** defined in Chapter 33 with these additional constructs to obtain **FSPCF**:

| | | | | | |
|---|---|---|---|---|---|
| Exp | $e$ | $::=$ | $\mathtt{put}[a](e_1; e_2)$ | $\mathtt{put}\, e_1\, \mathtt{for}\, a\, \mathtt{in}\, e_2$ | binding |
| | | | $\mathtt{get}[a]$ | $\mathtt{get}\, a$ | retrieval |

The expression $\mathtt{get}[a]$ evaluates to the value of the current binding of $a$, if it has one, and is stuck otherwise. The expression $\mathtt{put}[a](e_1; e_2)$ binds the symbol $a$ to the value $e_1$ for the duration of the evaluation of $e_2$, at which point the binding of $a$ reverts to what it was prior to the execution. The symbol $a$ is not bound by the put expression, but is instead a parameter of it.

The statics of **FSPCF** is defined by judgments of the form

$$\Gamma \vdash_\Sigma e : \tau,$$

Preview

where $\Sigma$ is a finite set of symbol declarations of the form $a \sim \tau$ such that no symbol is declared more than once. The following rules govern the new expression forms:

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{get}[a] : \tau} \tag{34.1a}$$

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau_1} e_1 : \tau_1 \quad \Gamma \vdash_{\Sigma, a \sim \tau_1} e_2 : \tau_2}{\Gamma \vdash_{\Sigma, a \sim \tau_1} \mathtt{put}[a](e_1; e_2) : \tau_2} \tag{34.1b}$$

Rule (34.1b) specifies that the symbol $a$ is a parameter of the expression that must be declared in $\Sigma$.

## 34.2 Dynamics

The dynamics of **FSPCF** relies on a stack-like allocation of symbols in **SPCF**, and maintains an association of values to symbols that tracks this stack-like allocation discipline. To do so we define a family of transition judgments of the form $e \xmapsto[\Sigma]{\mu} e'$, where $\Sigma$ is as in the statics, and $\mu$ is a finite function mapping some subset of the symbols declared in $\Sigma$ to values of appropriate type. If $\mu$ is defined for some symbol $a$, then it has the form $\mu' \otimes a \hookrightarrow e$ for some $\mu'$ and value $e$. If, on the other hand, $\mu$ is undefined for some symbol $a$, we may regard it as having the form $\mu' \otimes a \hookrightarrow \bullet$. We will write $a \hookrightarrow _$ to stand ambiguously for either $a \hookrightarrow \bullet$ or $a \hookrightarrow e$ for some expression $e$.

The dynamics of **FSPCF** is defined by the following rules:

$$\mathtt{get}[a] \xmapsto[\Sigma, a \sim \tau]{\mu \otimes a \hookrightarrow e} e \tag{34.2a}$$

$$\frac{e_1 \xmapsto[\Sigma, a \sim \tau]{\mu} e_1'}{\mathtt{put}[a](e_1; e_2) \xmapsto[\Sigma, a \sim \tau]{\mu} \mathtt{put}[a](e_1'; e_2)} \tag{34.2b}$$

$$\frac{e_1 \ \mathsf{val}_{\Sigma, a \sim \tau} \quad e_2 \xmapsto[\Sigma, a \sim \tau]{\mu \otimes a \hookrightarrow e_1} e_2'}{\mathtt{put}[a](e_1; e_2) \xmapsto[\Sigma, a \sim \tau]{\mu \otimes a \hookrightarrow _} \mathtt{put}[a](e_1; e_2')} \tag{34.2c}$$

$$\frac{e_1 \ \mathsf{val}_{\Sigma, a \sim \tau} \quad e_2 \ \mathsf{val}_{\Sigma, a \sim \tau}}{\mathtt{put}[a](e_1; e_2) \xmapsto[\Sigma, a \sim \tau]{\mu} e_2} \tag{34.2d}$$

Rule (34.2a) specifies that $\mathtt{get}[a]$ evaluates to the current binding of $a$, if any. Rule (34.2b) specifies that the binding for the symbol $a$ is to be evaluated before the binding is created. Rule (34.2c) evaluates $e_2$ in an environment in which the symbol $a$ is bound to the value $e_1$, regardless of whether or not $a$ is already bound in the environment. Rule (34.2d) eliminates the fluid binding for $a$ once evaluation of the extent of the binding has completed.

Preview

According to the dynamics of **FSPCF** given by rules (34.2), there is no transition of the form $\mathtt{get}[a] \overset{\mu}{\underset{\Sigma}{\mapsto}} e$ if $\mu(a) = \bullet$. The judgment $e\ \mathsf{unbound}_\Sigma$ states that execution of $e$ will lead to such a "stuck" state, and is inductively defined by the following rules:

$$\frac{\mu(a) = \bullet}{\mathtt{get}[a]\ \mathsf{unbound}_\mu} \tag{34.3a}$$

$$\frac{e_1\ \mathsf{unbound}_\mu}{\mathtt{put}[a](e_1; e_2)\ \mathsf{unbound}_\mu} \tag{34.3b}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad e_2\ \mathsf{unbound}_\mu}{\mathtt{put}[a](e_1; e_2)\ \mathsf{unbound}_\mu} \tag{34.3c}$$

In a larger language it would also be necessary to include error propagation rules of the sort discussed in Chapter 6.

## 34.3 Type Safety

We first define the auxiliary judgment $\mu : \Sigma$ by the following rules:

$$\frac{}{\emptyset : \emptyset} \tag{34.4a}$$

$$\frac{\vdash_\Sigma e : \tau \quad \mu : \Sigma}{\mu \otimes a \hookrightarrow e : \Sigma, a \sim \tau} \tag{34.4b}$$

$$\frac{\mu : \Sigma}{\mu \otimes a \hookrightarrow \bullet : \Sigma, a \sim \tau} \tag{34.4c}$$

These rules specify that if a symbol is bound to a value, then that value must be of the type associated to the symbol by $\Sigma$. No demand is made in the case that the symbol is unbound (equivalently, bound to a "black hole").

**Theorem 34.1** (Preservation). *If $e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'$, where $\mu : \Sigma$ and $\vdash_\Sigma e : \tau$, then $\vdash_\Sigma e' : \tau$.*

*Proof.* By rule induction on rules (34.2). Rule (34.2a) is handled by the definition of $\mu : \Sigma$. Rule (34.2b) follows immediately by induction. Rule (34.2d) is handled by inversion of rules (34.1). Finally, rule (34.2c) is handled by inversion of rules (34.1) and induction. □

**Theorem 34.2** (Progress). *If $\vdash_\Sigma e : \tau$ and $\mu : \Sigma$, then either $e\ \mathsf{val}_\Sigma$, or $e\ \mathsf{unbound}_\mu$, or there exists $e'$ such that $e \overset{\mu}{\underset{\Sigma}{\mapsto}} e'$.*

*Proof.* By induction on rules (34.1). For rule (34.1a), we have $\Sigma \vdash a \sim \tau$ from the premise of the rule, and hence, because $\mu : \Sigma$, we have either $\mu(a) = \bullet$ or $\mu(a) = e$ for some $e$ such that $\vdash_\Sigma e : \tau$. In the former case we have $e\ \mathsf{unbound}_\mu$, and in the latter we have $\mathtt{get}[a] \overset{\mu}{\underset{\Sigma}{\mapsto}} e$. For rule (34.1b),



Preview

we have by induction that either $e_1$ $\mathsf{val}_\Sigma$ or $e_1$ $\mathsf{unbound}_\mu$, or $e_1 \xmapsto[\Sigma]{\mu} e_1'$. In the latter two cases we may apply rule (34.2b) or rule (34.3b), respectively. If $e_1$ $\mathsf{val}_\Sigma$, we apply induction to obtain that either $e_2$ $\mathsf{val}_\Sigma$, in which case rule (34.2d) applies; $e_2$ $\mathsf{unbound}_\mu$, in which case rule (34.3c) applies; or $e_2 \xmapsto[\Sigma]{\mu} e_2'$, in which case rule (34.2c) applies. □

## 34.4 Some Subtleties

The value of $\mathtt{put}\, e_1\, \mathtt{for}\, a\, \mathtt{in}\, e_2$ is the value of $e_2$, calculated in a context in which $a$ is bound to the value of $e_1$. If $e_2$ is of a basic type, such as $\mathtt{nat}$, then the reversion of the binding of $a$ cannot influence the meaning of the result.[1]

But what if the type of $\mathtt{put}\, e_1\, \mathtt{for}\, a\, \mathtt{in}\, e_2$ is a function type, so that the returned value is a $\lambda$-abstraction? The body of the returned $\lambda$ may refer to the binding of $a$, which is reverted upon return from the $\mathtt{put}$. For example, consider the expression

$$\mathtt{put}\, 17\, \mathtt{for}\, a\, \mathtt{in}\, \lambda\, (x{:}\mathtt{nat})\, x + \mathtt{get}\, a, \tag{34.5}$$

which has type $\mathtt{nat} \rightharpoonup \mathtt{nat}$, given that $a$ is a symbol of type $\mathtt{nat}$. Let us assume, for the sake of discussion, that $a$ is unbound at the point at which this expression is evaluated. Evaluating the $\mathtt{put}$ binds $a$ to the number 17, and returns the function $\lambda\, (x{:}\mathtt{nat})\, x + \mathtt{get}\, a$. But because $a$ is reverted to its unbound state upon exiting the $\mathtt{put}$, applying this function to an argument will result in an error, unless a binding for $a$ is provided. Thus, if $f$ is bound to the result of evaluating (34.5), then the expression

$$\mathtt{put}\, 21\, \mathtt{for}\, a\, \mathtt{in}\, f(7) \tag{34.6}$$

will evaluate successfully to 28, whereas evaluation of $f(7)$ in the absence of a surrounding binding for $a$ will incur an error.

Contrast this with the superficially similar expression

$$\mathtt{let}\, y\, \mathtt{be}\, 17\, \mathtt{in}\, \lambda\, (x{:}\mathtt{nat})\, x + y, \tag{34.7}$$

in which we have replaced the fluid-bound symbol, $a$, by a statically bound variable, $y$. This expression evaluates to $\lambda\, (x{:}\mathtt{nat})\, x + 17$, which adds 17 to its argument when applied. There is no possibility of an unbound symbol arising at execution time, precisely because variables are interpreted by substitution.

One way to think about this situation is to consider that fluid-bound symbols serve as an alternative to passing additional arguments to a function to specialize its value whenever it is called. To see this, let $e$ stand for the value of expression (34.5), a $\lambda$-abstraction whose body is dependent on the binding of the symbol $a$. To use this function safely, it is necessary that the programmer provide a binding for $a$ prior to calling it. For example, the expression

$$\mathtt{put}\, 7\, \mathtt{for}\, a\, \mathtt{in}\, (e(9))$$

[1] As long as the the successor is evaluated eagerly; if not, the following examples may be adapted to situations in which the value of $e_2$ is a lazily evaluated number.



Preview

evaluates to 16, and the expression

$$\texttt{put}\,8\,\texttt{for}\,a\,\texttt{in}\,(e(9))$$

evaluates to 17. Writing just $e(9)$, without a surrounding binding for $a$, results in a run-time error attempting to retrieve the binding of the unbound symbol $a$.

This behavior may be simulated by adding an additional argument to the function value that will be bound to the current binding of the symbol $a$ at the point where the function is called. Instead of using fluid binding, we would provide an additional argument at each call site, writing

$$e'(7)(9)$$

and

$$e'(8)(9),$$

respectively, where $e'$ is the $\lambda$-abstraction

$$\lambda\,(y{:}\texttt{nat})\,\lambda\,(x{:}\texttt{nat})\,x + y.$$

Additional arguments can be cumbersome, though, especially when several call sites provide the same binding for $a$. Using fluid binding we may write

$$\texttt{put}\,7\,\texttt{for}\,a\,\texttt{in}\,\langle e(8), e(9)\rangle,$$

whereas using an additional argument we must write

$$\langle e'(7)(8), e'(7)(9)\rangle.$$

However, such redundancy can be mitigated by simply factoring out the common part, writing

$$\texttt{let}\,f\,\texttt{be}\,e'(7)\,\texttt{in}\,\langle f(8), f(9)\rangle.$$

The awkwardness of this simulation is usually taken as an argument in favor of including fluid binding in a language. The drawback, which is often perceived as an advantage, is that nothing in the type of a function reveals its dependency on the binding of a symbol. It is therefore quite easy to forget that such a binding is required, leading to run-time failures that might better be caught at compile time.

## 34.5 Fluid References

The `get` and `put` operations for fluid binding are indexed by a symbol that must be given as part of the syntax of the operator. Rather than insist that the target symbol be given statically, it is useful to be able to defer until runtime the choice of fluid on which a `get` or `put` acts. This may be achieved by introducing *references* to fluids, which allow the name of a fluid to be represented as a value. References come equipped with analogues of the `get` and `put` primitives, but for a dynamically determined symbol.



Preview

We may extend **FSPCF** with fluid references by adding the following syntax:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{fluid}(\tau)$ | $\tau\,\mathtt{fluid}$ | fluid |
| Exp | $e$ | $::=$ | $\mathtt{fl}[a]$ | $\&\,a$ | reference |
| | | | $\mathtt{getfl}(e)$ | $\mathtt{getfl}\,e$ | retrieval |
| | | | $\mathtt{putfl}(e;e_1;e_2)$ | $\mathtt{putfl}\,e\,\mathtt{is}\,e_1\,\mathtt{in}\,e_2$ | binding |

The expression $\mathtt{fl}[a]$ is the symbol $a$ considered as a value of type $\mathtt{fluid}(\tau)$. The expressions $\mathtt{getfl}(e)$ and $\mathtt{putfl}(e;e_1;e_2)$ are analogues of the `get` and `put` operations for fluid-bound symbols.

The statics of these constructs is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma,a\sim\tau} \mathtt{fl}[a] : \mathtt{fluid}(\tau)} \tag{34.8a}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{fluid}(\tau)}{\Gamma \vdash_\Sigma \mathtt{getfl}(e) : \tau} \tag{34.8b}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{fluid}(\tau) \quad \Gamma \vdash_\Sigma e_1 : \tau \quad \Gamma \vdash_\Sigma e_2 : \tau_2}{\Gamma \vdash_\Sigma \mathtt{putfl}(e;e_1;e_2) : \tau_2} \tag{34.8c}$$

Because we are using a scoped dynamics, references to fluids cannot be deemed mobile.

The dynamics of references consists of resolving the referent and deferring to the underlying primitives acting on symbols.

$$\frac{}{\mathtt{fl}[a]\ \mathsf{val}_{\Sigma,a\sim\tau}} \tag{34.9a}$$

$$\frac{e \xmapsto[\Sigma]{\mu} e'}{\mathtt{getfl}(e) \xmapsto[\Sigma]{\mu} \mathtt{getfl}(e')} \tag{34.9b}$$

$$\frac{}{\mathtt{getfl}(\mathtt{fl}[a]) \xmapsto[\Sigma]{\mu} \mathtt{get}[a]} \tag{34.9c}$$

$$\frac{e \xmapsto[\Sigma]{\mu} e'}{\mathtt{putfl}(e;e_1;e_2) \xmapsto[\Sigma]{\mu} \mathtt{putfl}(e';e_1;e_2)} \tag{34.9d}$$

$$\frac{}{\mathtt{putfl}(\mathtt{fl}[a];e_1;e_2) \xmapsto[\Sigma]{\mu} \mathtt{put}[a](e_1;e_2)} \tag{34.9e}$$

Preview

## 34.6 Notes

The concept of dynamic binding arose from the confusion of variables and symbols in early dialects of Lisp. When properly separated, variables retain their substitutive meaning, and symbols give rise to a separate concept of fluid binding. Allen (1978) contains a thorough discussion of the implementation of fluid binding. The formulation given here also draws on Nanevski (2003).

## Exercises

**34.1**. *Deep binding* is an implementation of fluid binding in which the value associated to a symbol is stored on the control stack as part of a `put` frame, and is retrieved by finding the most recent such association. Define a stack machine for **FSPCF** that implements deep binding by extending the **FPCF** machine. Be sure to consider `new` as well as `put` and `get`. Attempting to get the binding for an unbound symbol should signal a failure; otherwise, its most recent binding should be returned. Where do the issues discussed in Section 34.4 arise? *Hint*: you will need to introduce an auxiliary judgment $k \geq k' \; ? \; a$, which searches for the binding of the symbol $a$ on the stack $k'$, returning its value (or failure) to the stack $k$.

**34.2**. *Shallow binding* is an implementation of fluid binding that maintains a mapping sending each active symbol to a *stack* of values, the topmost of which is the currently actively binding for that symbol. Define a stack machine for **FSPCF** that maintains such a mapping to facilitate access to the binding of a symbol. *Hint*: use evaluation states of the form $k \parallel \mu \triangleright e$, where $\mu$ is a mapping each symbol $a$ allocated on $k$ to a stack of values, the topmost element of which, if any, is the current binding of $a$. Use similar forms of return and fail states, and ensure that the mapping invariant is maintained.

**34.3**. It is often said that exception handlers (see Chapter 29) are an instance of fluid binding. The dynamics of exception handling given in Chapter 29 makes no use of fluid binding. It follows that there is no essential connection between the two concepts. However, it is possible to implement exception handling using fluid binding and continuations (see Chapter 30) by reserving a single fluid-bound symbol, say *hdlr*, that is bound to the currently active exception handler. The handler is represented by a continuation to which the exception value is thrown when an exception is raised; the appropriate continuation is `put` as the binding of *hdlr* when entering the scope of a handler. Formulate such an implementation of exceptions using fluid binding and continuations. What is the associated type of the fluid *hdlr*? *Hint*: it is important to ensure that the current handler is properly updated for both normal and exceptional returns.

Preview

Preview

# Chapter 35

# Dynamic Classification

In Chapters 11 and 26 we investigated the use of sums for the classification of values of disparate type. Every value of a classified type is labeled with a symbol that determines the type of the instance data. A classified value is decomposed by pattern matching against a known class, which reveals the type of the instance data. Under this representation the possible classes of an object are fully determined *statically* by its type. However, it is sometimes useful to allow the possible classes of data value to be determined *dynamically*. There are many uses for such a capability, some less apparent than others. The most obvious is simply extensibility, when we wish to introduce new classes of data during execution (and, presumably, define how methods act on values of those new classes).

A less obvious application exploits the fact that the new class is guaranteed to be distinct from any other class that has already been introduced. The class itself is a kind of "secret" that can be disclosed only if the computation that creates the class discloses its existence to another computation. In particular, the class is opaque to any computation to which this disclosure has not been explicitly made. This capability has a number of practical applications, including enforcement of private communication on a network, and ensuring that an exception reaches its intended handler, and no other.

Preview

## 35.1 Dynamic Classes

A dynamic class is a symbol that may be generated at run-time. A classified value consists of a symbol of type $\tau$ together with a value of that type. To compute with a classified value, it is compared with a known class. If the value is of this class, the underlying instance data is passed to the positive branch, otherwise the negative branch is taken, where it may be matched against other known classes.

### 35.1.1 Statics

The syntax of dynamic classification is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\texttt{clsfd}$ | $\texttt{clsfd}$ | classified |
| Exp | $e$ | $::=$ | $\texttt{in}[a](e)$ | $a \cdot e$ | instance |
| | | | $\texttt{isin}[a](e; x.e_1; e_2)$ | $\texttt{match}\, e\, \texttt{as}\, a \cdot x \hookrightarrow e_1\, \texttt{ow} \hookrightarrow e_2$ | comparison |

The expression $\texttt{in}[a](e)$ is a classified value with class $a$ and underlying value $e$. The expression $\texttt{isin}[a](e; x.e_1; e_2)$ checks whether the class of the value given by $e$ is $a$. If so, the classified value is passed to $e_1$; if not, the expression $e_2$ is evaluated instead.

The statics of dynamic classification is defined by these rules:

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau}{\Gamma \vdash_{\Sigma, a \sim \tau} \texttt{in}[a](e) : \texttt{clsfd}} \tag{35.1a}$$

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \texttt{clsfd} \quad \Gamma, x : \tau \vdash_{\Sigma, a \sim \tau} e_1 : \tau' \quad \Gamma \vdash_{\Sigma, a \sim \tau} e_2 : \tau'}{\Gamma \vdash_{\Sigma, a \sim \tau} \texttt{isin}[a](e; x.e_1; e_2) : \tau'} \tag{35.1b}$$

The type associated to the symbol in the signature determines the type of the instance data.

### 35.1.2 Dynamics

To maximize the flexibility in the use of dynamic classification, we will consider a free dynamics for symbol generation. Within this framework the dynamics of classification is given by the following rules:

$$\frac{e\ \mathsf{val}_\Sigma}{\texttt{in}[a](e)\ \mathsf{val}_\Sigma} \tag{35.2a}$$

$$\frac{\nu\, \Sigma \{\, e \,\} \longmapsto \nu\, \Sigma' \{\, e' \,\}}{\nu\, \Sigma \{\, \texttt{in}[a](e) \,\} \longmapsto \nu\, \Sigma' \{\, \texttt{in}[a](e') \,\}} \tag{35.2b}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\, \Sigma \{\, \texttt{isin}[a](\texttt{in}[a](e); x.e_1; e_2) \,\} \longmapsto \nu\, \Sigma \{\, [e/x]e_1 \,\}} \tag{35.2c}$$

$$\frac{e'\ \mathsf{val}_\Sigma \quad (a \neq a')}{\nu\, \Sigma \{\, \texttt{isin}[a](\texttt{in}[a'](e'); x.e_1; e_2) \,\} \longmapsto \nu\, \Sigma \{\, e_2 \,\}} \tag{35.2d}$$

$$\frac{\nu\, \Sigma \{\, e \,\} \longmapsto \nu\, \Sigma' \{\, e' \,\}}{\nu\, \Sigma \{\, \texttt{isin}[a](e; x.e_1; e_2) \,\} \longmapsto \nu\, \Sigma' \{\, \texttt{isin}[a](e'; x.e_1; e_2) \,\}} \tag{35.2e}$$

Throughout, if the states involved are well-formed, then there will be a declaration $a \sim \tau$ for some type $\tau$ in $\Sigma$.

The dynamics of the elimination form for the type `clsfd` relies on *disequality* of names (specifically, rule (35.2d)). Because disequality is not preserved under substitution, it is not sensible to

Preview

consider any language construct whose dynamics relies on such a substitution. To see what goes wrong, consider the expression

$$\texttt{match}\, b \cdot \langle\rangle\, \texttt{as}\, a \cdot _ \hookrightarrow \texttt{true}\, \texttt{ow} \hookrightarrow \texttt{match}\, b \cdot \langle\rangle\, \texttt{as}\, b \cdot _ \hookrightarrow \texttt{false}\, \texttt{ow} \hookrightarrow \texttt{true}.$$

This is easily seen to evaluate to `false`, because the outer conditional is on the class $a$, which is *a priori* different from $b$. However, if we substitute $b$ for $a$ in this expression we obtain

$$\texttt{match}\, b \cdot \langle\rangle\, \texttt{as}\, b \cdot _ \hookrightarrow \texttt{true}\, \texttt{ow} \hookrightarrow \texttt{match}\, b \cdot \langle\rangle\, \texttt{as}\, b \cdot _ \hookrightarrow \texttt{false}\, \texttt{ow} \hookrightarrow \texttt{true},$$

which evaluate to `true`, because now the outer conditional governs the evaluation.

### 35.1.3 Safety

**Theorem 35.1** (Safety)**.**

1. *If* $\vdash_\Sigma e : \tau$ *and* $\nu\, \Sigma\, \{\, e \,\} \longmapsto \nu\, \Sigma'\, \{\, e' \,\}$*, then* $\Sigma' \supseteq \Sigma$ *and* $\vdash_{\Sigma'} e' : \tau$*.*
2. *If* $\vdash_\Sigma e : \tau$*, then either* $e$ $\mathsf{val}_\Sigma$ *or* $\nu\, \Sigma\, \{\, e \,\} \longmapsto \nu\, \Sigma'\, \{\, e' \,\}$ *for some* $e'$ *and* $\Sigma'$*.*

*Proof.* Similar to the safety proofs given in Chapters 11, 12, and 33. □

## 35.2 Class References

The type $\texttt{class}(\tau)$ has as values references to classes.

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | ::= | $\texttt{class}(\tau)$ | $\tau\ \texttt{class}$ | class reference |
| Exp | $e$ | ::= | $\texttt{cls}[a]$ | $\&\, a$ | reference |
| | | | $\texttt{mk}(e_1; e_2)$ | $\texttt{mk}(e_1; e_2)$ | instance |
| | | | $\texttt{isofcls}(e_0; e_1; x.e_2; e_3)$ | $\texttt{isofcls}(e_0; e_1; x.e_2; e_3)$ | dispatch |

The statics of these constructs is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \texttt{cls}[a] : \texttt{class}(\tau)} \tag{35.3a}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \texttt{class}(\tau) \quad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma \texttt{mk}(e_1; e_2) : \texttt{clsfd}} \tag{35.3b}$$

$$\frac{\Gamma \vdash_\Sigma e_0 : \texttt{class}(\tau) \quad \Gamma \vdash_\Sigma e_1 : \texttt{clsfd} \quad \Gamma, x : \tau \vdash_\Sigma e_2 : \tau' \quad \Gamma \vdash_\Sigma e_3 : \tau'}{\Gamma \vdash_\Sigma \texttt{isofcls}(e_0; e_1; x.e_2; e_3) : \tau'} \tag{35.3c}$$

The corresponding dynamics is given by these rules:

$$\frac{\nu\, \Sigma\, \{\, e_1 \,\} \longmapsto \nu\, \Sigma'\, \{\, e_1' \,\}}{\nu\, \Sigma\, \{\, \texttt{mk}(e_1; e_2) \,\} \longmapsto \nu\, \Sigma'\, \{\, \texttt{mk}(e_1'; e_2) \,\}} \tag{35.4a}$$

Preview

$$\frac{e_1\ \mathsf{val}_\Sigma \quad \nu\,\Sigma\,\{\,e_2\,\} \longmapsto \nu\,\Sigma'\,\{\,e_2'\,\}}{\nu\,\Sigma\,\{\,\mathtt{mk}(e_1;e_2)\,\} \longmapsto \nu\,\Sigma'\,\{\,\mathtt{mk}(e_1;e_2')\,\}} \tag{35.4b}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\mathtt{mk}(\mathtt{cls}[a];e)\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{in}[a](e)\,\}} \tag{35.4c}$$

$$\frac{\nu\,\Sigma\,\{\,e_0\,\} \longmapsto \nu\,\Sigma'\,\{\,e_0'\,\}}{\nu\,\Sigma\,\{\,\mathtt{isofcls}(e_0;e_1;x.e_2;e_3)\,\} \longmapsto \nu\,\Sigma'\,\{\,\mathtt{isofcls}(e_0';e_1;x.e_2;e_3)\,\}} \tag{35.4d}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{isofcls}(\mathtt{cls}[a];e_1;x.e_2;e_3)\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{isin}[a](e_1;x.e_2;e_3)\,\}} \tag{35.4e}$$

Rules (35.4d) and (35.4e) specify that the first argument is evaluated to determine the target class, which is then used to check whether the second argument, a classified data value, is of the target class. This may be seen as a two-stage pattern matching process in which evaluation of $e_0$ determines the pattern against which to match the classified value of $e_1$.

## 35.3 Definability of Dynamic Classes

The type `clsfd` may be defined in terms of symbolic references, product types, and existential types by the type expression

$$\mathtt{clsfd} \triangleq \exists(t.t\ \mathtt{sym} \times t).$$

The introductory form $\mathtt{in}[a](e)$, in which $a$ is a symbol with associated type is $\tau$ and $e$ is an expression of type $\tau$, is defined to be the package

$$\mathtt{pack}\,\tau\,\mathtt{with}\,\langle \&\, a, e\rangle\,\mathtt{as}\,\exists(t.t\ \mathtt{sym} \times t).$$

The eliminatory form $\mathtt{isin}[a](e;x.e_1;e_2)$ is defined in terms of symbol comparison as defined in Chapter 33. Suppose that the overall type of the conditional is $\tau'$ and that the type associated to the symbol $a$ is $\tau$. The type of $e$ must be `clsfd`, defined as above, and the type of the branches $e_1$ and $e_2$ must be $\tau'$ with $x$ assumed to be of type $\tau$ in $e_1$. The conditional is defined to be the expression

$$\mathtt{open}\,e\,\mathtt{as}\,t\,\mathtt{with}\,\langle x, y\rangle{:}t\ \mathtt{sym} \times t\,\mathtt{in}\,(e_{body}(y)),$$

where $e_{body}$ is an expression to be defined shortly. The comparison opens the package $e$ representing the classified value, and decomposes it into a type $t$ a symbol reference $x$ of type $t\ \mathtt{sym}$, and an associated value $y$ of type $t$. The expression $e_{body}$ will have the type $t \rightharpoonup \tau'$ so that the application to $y$ is type correct.

The expression $e_{body}$ compares the symbolic reference $x$ to the symbol $a$ of type $\tau$, and yields a value of type $t \rightharpoonup \tau'$ regardless of the outcome. It is therefore defined to be the expression

$$\mathtt{is}[a][u.u \rightharpoonup \tau'](x;e_1';e_2')$$

where, in accordance with rule (33.5b), $e_1'$ has type $[\tau/u](u \rightharpoonup \tau') = \tau \rightharpoonup \tau'$, and $e_2'$ has type $[t/u](u \rightharpoonup \tau') = t \rightharpoonup \tau'$. The expression $e_1'$ "knows" that the abstract type $t$ is $\tau$, the type associated to the symbol $a$, because the comparison has come out positively. On the other hand $e_2'$ does not "learn" anything about the identity of $t$.

Preview

It remains to choose the expressions $e_1'$ and $e_2'$. In the case of a positive comparison, we wish to pass the classified value to the expression $e_1$ by substitution for the variable $x$. This is accomplished by defining $e_1'$ to be the expression

$$\lambda\,(x{:}\tau)\,e_1 : \tau \rightharpoonup \tau'.$$

In the case of a negative comparison no value is to be propagated to $e_2$. We therefore define $e_2'$ to be the expression

$$\lambda\,(_{:}t)\,e_2 : t \rightharpoonup \tau'.$$

We may then check that the statics and dynamics given in Section 35.1 are derivable, given the definitions of the type of classified values and its introductory and eliminatory forms.

## 35.4 Applications of Dynamic Classification

Dynamic classification has a number of interesting applications in programming. The most obvious is to generalize dynamic dispatch (Chapter 26) to support computation over a dynamically extensible type of heterogeneous values. Introducing a new class requires introducing a new row in the dispatch matrix defining the behavior of the methods on the newly defined class. To allow for this the rows of the matrix must be indexed by class references, rather than classes, so that it may be accessed without knowing statically the class in question.

Another application is to use dynamic classification as a form of "perfect encryption" that ensures that classified values can neither be constructed nor deconstructed without knowing the class in question. Abstract enryption of this form can be used to ensure privacy of communication among the parties in a computation. One example of such a scenario is in channel-based communication, as will be considered in Chapter 43. Another, less obvious, application is to ensure that an exception value may only be received by the intended handler, and no other.

### 35.4.1 Classifying Secrets

Dynamic classification may be used to enforce *confidentiality* and *integrity* of data values in a program. A value of type `clsfd` may only be constructed by *sealing* it with some class $a$ and may only be deconstructed by a case analysis that includes a branch for $a$. By controlling which parties in a multi-party interaction have access to the classifier $a$ we may control how classified values are created (ensuring their *integrity*) and how they are inspected (ensuring their *confidentiality*). Any party that lacks access to $a$ cannot decipher a value classified by $a$, nor may it create a classified value with this class. Because classes are dynamically generated symbols, they provide an absolute confidentiality guarantee among parties in a computation.[1]

Consider the following simple protocol for controlling the integrity and confidentiality of data in a program. A fresh symbol $a$ is introduced, and we return a pair of functions of type

$$(\tau \rightharpoonup \texttt{clsfd}) \times (\texttt{clsfd} \rightharpoonup \tau\ \texttt{opt}),$$

called the *constructor* and *destructor* functions for that class, which is accomplished by writing

[1] Of course, this guarantee is for programs written in conformance with the statics given here. If the abstraction imposed by the type system is violated, no guarantees of confidentiality can be made.



Preview

```
new a ∼ τ in
  ⟨ λ(x:τ) a · x,
    λ(x:clsfd) match x as a · y ↪ just(y) ow ↪ null ⟩.
```

The first function creates a value classified by $a$, and the second function recovers the instance data of a value classified by $a$. Outside of the scope of the declaration the symbol $a$ is an absolutely unguessable secret.

To enforce the *integrity* of a value of type $\tau$ it is sufficient to ensure that only trusted parties have access to the constructor. To enforce the *confidentiality* of a value of type $\tau$ it is sufficient to ensure that only trusted parties have access to the destructor. Ensuring the integrity of a value amounts to associating an invariant to it that is maintained by the trusted parties that may create an instance of that class. Ensuring the confidentiality of a value amounts to propagating the invariant to parties that may decipher it.

### 35.4.2 Exception Values

Exception handling may be seen as a communication between two agents, one that may raise an exception, and one that may handle it. We wish to ensure that an exception can be caught only by a designated handler, without fear that any intervening handler may intercept it. This can be achieved by dynamic class allocation. A new class is declared, with the capability to create an instance given only to the raising agent and the capability to match an instance given only to the handler. The exception value cannot be intercepted by any other handler, because no other handler is capable of matching it. This property is crucial to "black box" composition of programs from components. Without dynamic classification one can never be sure that alien code cannot intercept an exception intended for a handler within one's own code, or *vice versa*.

With this in mind, let us now reconsider the choice of the type $\tau_{\mathsf{exn}}$ of exception values specified in Chapter 29. There we distinguished the *closed-world* assumption, which amounts to taking $\tau_{\mathsf{exn}}$ to be a finite sum type shared by the whole program, from the *open-world* assumption, which may be realized by taking $\tau_{\mathsf{exn}}$ to be the type `clsfd` of dynamically classified values. This choice supports modularity and evolution by allowing fresh exceptions (classes) to be allocated at will, avoiding the need for an up-front agreement on the forms of exception. Another perspective is that dynamic classification treats an exception as a shared secret between the handler and the raiser of that exception. When an exception value is raised, it can only be intercepted and analyzed by a handler that can match the value against the specified class. It is only through the use of dynamic classification that one can gain control over the flow of information in a program that uses exceptions. Without it, an unintended handler can intercept an exception that was not intended for it, disrupting the logic of the program.

## 35.5 Notes

Dynamic classification appears in Standard ML (Milner et al., 1997) as the type `exn`. The importance of the type `exn` is obscured by its too-close association with the exception mechanism. The $\pi$-calculus (Milner, 1999) popularized the use of "name generation" and "channel passing" to



Preview

control the connectivity and information flow in a process network. In Chapter 43 we shall make explicit that this aspect of the $\pi$-calculus simply an application of dynamic classification, which is useful in its own right quite apart from concurrency.

## Exercises

**35.1**. Consider the following *open-world named exception* mechanism, which is typical of the exception mechanism found in many languages.

| | |
|---|---|
| $\texttt{exception}\,a\,\texttt{of}\,\tau\,\texttt{in}\,e$ | declare an exception $a$ of type $\tau$ in $e$ |
| $\texttt{raise}\,a\,\texttt{with}\,e$ | raise exception $a$ with value $e$ |
| $\texttt{try}\,e\,\texttt{ow}\,a_1(x_1) \hookrightarrow e_1 \mid \ldots \mid a_n(x_n) \hookrightarrow e_n \mid x \hookrightarrow e'$ | handle exceptions $a_1, \ldots, a_n$ |

Exceptions are declared by name, specifying the type of their associated values. Each execution of an exception declaration generates a *fresh* exception. An exception is raised by specifying the exception name and a value to associate with it. The handler intercepts any finite number of named exceptions, passing their associated values to handlers, and otherwise propagates the exception to the default handler.

The following rules define the statics of these constructs:

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau'}{\Gamma \vdash_{\Sigma} \texttt{exception}\,\tau\,\texttt{of}\,a\,\texttt{in}\,e : \tau'} \tag{35.5a}$$

$$\frac{\Sigma \vdash a \sim \tau \quad \Gamma \vdash_{\Sigma} e : \tau}{\Gamma \vdash_{\Sigma} \texttt{raise}\,a\,\texttt{with}\,e : \tau'} \tag{35.5b}$$

$$\frac{\begin{array}{c}\Sigma \vdash a_1 \sim \tau_1 \quad \ldots \quad \Sigma \vdash a_n \sim \tau_n \quad \Gamma \vdash_{\Sigma} e : \tau' \\ \Gamma, x_1 : \tau_1 \vdash_{\Sigma} e_1 : \tau' \quad \ldots \quad \Gamma, x_n : \tau_n \vdash e_n : \tau' \quad \Gamma, x : \tau_{\mathsf{exn}} \vdash e' : \tau'\end{array}}{\Gamma \vdash_{\Sigma} \texttt{try}\,e\,\texttt{ow}\,a_1(x_1) \hookrightarrow e_1 \mid \ldots \mid a_n(x_n) \hookrightarrow e_n \mid x \hookrightarrow e' : \tau'} \tag{35.5c}$$

Give an implementation of named exceptions in terms of dynamic classification and general value-passing exceptions (Chapter 29).

**35.2**. Open-world named exceptions obstruct exception tracking (as described in Exercise **29.3**).

(a) Show that it is not computable to track the exact set of exception names that may be raised by an expression.

(b) Show that it is impossible to finitely bound set of exceptions that may be thrown by an expression. *Hint*: Show that there are expressions for which any such upper bound is inaccurate.

Conclude that the evergreen proposal to refine types with "throws" clauses that state a finite bound on the exceptions that may be raised by an expression is doomed to failure.

Preview

Preview

# Part XIV

# Mutable State

Preview

Preview

# Chapter 36

# Reynolds's Algol

*Modernized Algol*, or **MA**, is an imperative, block-structured programming language based on the classic language Algol. **MA** may be seen as an extension to **PCF** with a new syntactic sort of *commands* that act on *assignables* by retrieving and altering their contents. Assignables are introduced by *declaring* them for use within a specified scope; this is the essence of block structure. Commands may be combined by sequencing, and may be iterated using recursion.

**MA** maintains a careful separation between *pure* expressions, whose meaning does not depend on any assignables, and *impure* commands, whose meaning is given in terms of assignables. This ensures that the evaluation order for expressions is not constrained by the presence of assignables in the language, and allows for expressions to be manipulated much as in **PCF**. Commands, on the other hand, have a tightly constrained execution order, because the execution of one may affect the meaning of another.

A distinctive feature of **MA** is that it adheres to the *stack discipline*, which means that assignables are allocated on entry to the scope of their declaration, and deallocated on exit, using a conventional stack discipline. This avoids the need for more complex forms of storage management, at the expense of reducing the expressiveness of the language.

## 36.1 Basic Commands

The syntax of the language **MA** of modernized Algol distinguishes pure *expressions* from impure *commands*. The expressions include those of **PCF** (as described in Chapter 15), augmented with one additional construct, and the commands are those of a simple imperative programming language based on assignment. The language maintains a sharp distinction between *variables* and *assignables*. Variables are introduced by $\lambda$-abstraction, and are given meaning by substitution. Assignables are introduced by a declaration, and are given meaning by assignment and retrieval of their *contents*, which is, for the time being, restricted to natural numbers. Expressions evaluate to values, and have no effect on assignables. Commands are executed for their effect on assignables, and also return a value. Composition of commands not only sequences their execution order, but also passes the value returned by the first to the second before it is executed. The returned value of

Preview

a command is, for the time being, restricted to the natural numbers. (But see Section 36.3 for the general case.)

The syntax of **MA** is given by the following grammar, from which we have omitted repetition of the expression syntax of **PCF** for the sake of brevity.

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{cmd}$ | $\mathtt{cmd}$ | command |
| Exp | $e$ | $::=$ | $\mathtt{cmd}(m)$ | $\mathtt{cmd}\,m$ | encapsulation |
| Cmd | $m$ | $::=$ | $\mathtt{ret}(e)$ | $\mathtt{ret}\,e$ | return |
| | | | $\mathtt{bnd}(e;x.m)$ | $\mathtt{bnd}\,x \hookleftarrow e\,;m$ | sequence |
| | | | $\mathtt{dcl}(e;a.m)$ | $\mathtt{dcl}\,a := e\,\mathtt{in}\,m$ | new assignable |
| | | | $\mathtt{get}[a]$ | $@\,a$ | fetch |
| | | | $\mathtt{set}[a](e)$ | $a := e$ | assign |

The expression $\mathtt{cmd}(m)$ consists of the unevaluated command, $m$, thought of as a value of type $\mathtt{cmd}$. The command, $\mathtt{ret}(e)$, returns the value of the expression $e$ without having any effect on the assignables. The command $\mathtt{bnd}(e;x.m)$ evaluates $e$ to an encapsulated command, then this command is executed for its effects on assignables, with its value substituted for $x$ in $m$. The command $\mathtt{dcl}(e;a.m)$ introduces a new assignable, $a$, for use within the command, $m$, whose initial contents is given by the expression, $e$. The command $\mathtt{get}[a]$ returns the current contents of the assignable, $a$, and the command $\mathtt{set}[a](e)$ changes the contents of the assignable $a$ to the value of $e$, and returns that value.

### 36.1.1 Statics

The statics of **MA** consists of two forms of judgment:

1. Expression typing: $\Gamma \vdash_\Sigma e : \tau$.
2. Command formation: $\Gamma \vdash_\Sigma m\ \mathsf{ok}$.

The context, $\Gamma$, specifies the types of variables, as usual, and the signature, $\Sigma$, consists of a finite set of assignables. These judgments are inductively defined by the following rules:

$$\frac{\Gamma \vdash_\Sigma m\ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{cmd}(m) : \mathtt{cmd}} \tag{36.1a}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{nat}}{\Gamma \vdash_\Sigma \mathtt{ret}(e)\ \mathsf{ok}} \tag{36.1b}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{cmd} \quad \Gamma\, x : \mathtt{nat} \vdash_\Sigma m\ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{bnd}(e;x.m)\ \mathsf{ok}} \tag{36.1c}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{nat} \quad \Gamma \vdash_{\Sigma,a} m\ \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{dcl}(e;a.m)\ \mathsf{ok}} \tag{36.1d}$$

$$\frac{}{\Gamma \vdash_{\Sigma,a} \mathtt{get}[a]\ \mathsf{ok}} \tag{36.1e}$$

Preview

$$\frac{\Gamma \vdash_{\Sigma,a} e : \mathtt{nat}}{\Gamma \vdash_{\Sigma,a} \mathtt{set}[a](e)\ \mathsf{ok}} \tag{36.1f}$$

Rule (36.1a) is the introductory rule for the type `cmd`, and rule (36.1c) is the corresponding eliminatory form. Rule (36.1d) introduces a new assignable for use within a specified command. The name, $a$, of the assignable is bound by the declaration, and hence may be renamed to satisfy the implicit constraint that it not already be present in $\Sigma$. Rule (36.1e) states that the command to retrieve the contents of an assignable, $a$, returns a natural number. Rule (36.1f) states that we may assign a natural number to an assignable.

### 36.1.2 Dynamics

The dynamics of **MA** is defined in terms of a *memory*, $\mu$, a finite function assigning a numeral to each of a finite set of assignables.

The dynamics of expressions consists of these two judgment forms:

1. $e\ \mathsf{val}_\Sigma$, stating that $e$ is a value relative to $\Sigma$.
2. $e \underset{\Sigma}{\mapsto} e'$, stating that the expression $e$ steps to the expression $e'$.

These judgments are inductively defined by the following rules, together with the rules defining the dynamics of **PCF** (see Chapter 15). It is important, however, that the successor operation be given an *eager*, rather than *lazy*, dynamics so that a closed value of type `nat` is a numeral (for reasons that will be explained in Section 36.3).

$$\frac{}{\mathtt{cmd}(m)\ \mathsf{val}_\Sigma} \tag{36.2a}$$

Rule (36.2a) states that an encapsulated command is a value.

The dynamics of commands is defined in terms of states $m \parallel \mu$, where $\mu$ is a memory mapping assignables to values, and $m$ is a command. There are two judgments governing such states:

1. $m \parallel \mu\ \mathsf{final}_\Sigma$. The state $m \parallel \mu$ is fully executed.
2. $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$. The state $m \parallel \mu$ steps to the state $m' \parallel \mu'$; the set of active assignables is given by the signature $\Sigma$.

These judgments are inductively defined by the following rules:

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{ret}(e) \parallel \mu\ \mathsf{final}_\Sigma} \tag{36.3a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{ret}(e) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{ret}(e') \parallel \mu} \tag{36.3b}$$

Preview

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{bnd}(e; x.m) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{bnd}(e'; x.m) \parallel \mu} \tag{36.3c}$$

$$\frac{e \ \mathsf{val}_\Sigma}{\mathtt{bnd}(\mathtt{cmd}(\mathtt{ret}(e)); x.m) \parallel \mu \underset{\Sigma}{\mapsto} [e/x]m \parallel \mu} \tag{36.3d}$$

$$\frac{m_1 \parallel \mu \underset{\Sigma}{\mapsto} m_1' \parallel \mu'}{\mathtt{bnd}(\mathtt{cmd}(m_1); x.m_2) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{bnd}(\mathtt{cmd}(m_1'); x.m_2) \parallel \mu'} \tag{36.3e}$$

$$\frac{}{\mathtt{get}[a] \parallel \mu \otimes a \hookrightarrow e \underset{\Sigma,a}{\longmapsto} \mathtt{ret}(e) \parallel \mu \otimes a \hookrightarrow e} \tag{36.3f}$$

$$\frac{e \underset{\Sigma,a}{\longmapsto} e'}{\mathtt{set}[a](e) \parallel \mu \underset{\Sigma,a}{\longmapsto} \mathtt{set}[a](e') \parallel \mu} \tag{36.3g}$$

$$\frac{e \ \mathsf{val}_{\Sigma,a}}{\mathtt{set}[a](e) \parallel \mu \otimes a \hookrightarrow _ \underset{\Sigma,a}{\longmapsto} \mathtt{ret}(e) \parallel \mu \otimes a \hookrightarrow e} \tag{36.3h}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{dcl}(e; a.m) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{dcl}(e'; a.m) \parallel \mu} \tag{36.3i}$$

$$\frac{e \ \mathsf{val}_\Sigma \quad m \parallel \mu \otimes a \hookrightarrow e \underset{\Sigma,a}{\longmapsto} m' \parallel \mu' \otimes a \hookrightarrow e'}{\mathtt{dcl}(e; a.m) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{dcl}(e'; a.m') \parallel \mu'} \tag{36.3j}$$

$$\frac{e \ \mathsf{val}_\Sigma \quad e' \ \mathsf{val}_{\Sigma,a}}{\mathtt{dcl}(e; a.\mathtt{ret}(e')) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{ret}(e') \parallel \mu} \tag{36.3k}$$

Rule (36.3a) specifies that a `ret` command is final if its argument is a value. Rules (36.3c) to (36.3e) specify the dynamics of sequential composition. The expression, $e$, must, by virtue of the type system, evaluate to an encapsulated command, which is to be executed to determine its return value, which is then substituted into $m$ before executing it.

Rules (36.3i) to (36.3k) define the concept of *block structure* in a programming language. Declarations adhere to the *stack discipline* in that an assignable is allocated for the duration of evaluation of the body of the declaration, and deallocated after evaluation of the body is complete. Therefore the lifetime of an assignable can be identified with its scope, and hence we may visualize the dynamic lifetimes of assignables as being nested inside one another, in the same manner as their static scopes are nested inside one another. This stack-like behavior of assignables is a characteristic feature of what are known as *Algol-like languages*.

Preview

### 36.1.3 Safety

The judgment $m \parallel \mu \ \mathsf{ok}_\Sigma$ is defined by the rule

$$\frac{\vdash_\Sigma m \ \mathsf{ok} \quad \mu : \Sigma}{m \parallel \mu \ \mathsf{ok}_\Sigma} \tag{36.4}$$

where the auxiliary judgment $\mu : \Sigma$ is defined by the rule

$$\frac{\forall a \in \Sigma \quad \exists e \quad \mu(a) = e \text{ and } e \ \mathsf{val}_\varnothing \text{ and } \vdash_\varnothing e : \mathtt{nat}}{\mu : \Sigma} \tag{36.5}$$

That is, the memory must bind a number to each location in $\Sigma$.

**Theorem 36.1** (Preservation)**.**

1. *If $e \underset{\Sigma}{\mapsto} e'$ and $\vdash_\Sigma e : \tau$, then $\vdash_\Sigma e' : \tau$.*

2. *If $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$, with $\vdash_\Sigma m \ \mathsf{ok}$ and $\mu : \Sigma$, then $\vdash_\Sigma m' \ \mathsf{ok}$ and $\mu' : \Sigma$.*

*Proof.* Simultaneously, by induction on rules (36.2) and (36.3).

Consider rule (36.3j). Assume that $\vdash_\Sigma \mathtt{dcl}(e; a.m) \ \mathsf{ok}$ and $\mu : \Sigma$. By inversion of typing we have $\vdash_\Sigma e : \mathtt{nat}$ and $\vdash_{\Sigma,a} m \ \mathsf{ok}$. Because $e \ \mathsf{val}_\Sigma$ and $\mu : \Sigma$, we have $\mu \otimes a \hookrightarrow e : \Sigma, a$. By induction we have $\vdash_{\Sigma,a} m' \ \mathsf{ok}$ and $\mu' \otimes a \hookrightarrow e : \Sigma, a$, from which the result follows immediately.

Consider rule (36.3k). Assume that $\vdash_\Sigma \mathtt{dcl}(e; a.\mathtt{ret}(e')) \ \mathsf{ok}$ and $\mu : \Sigma$. By inversion we have $\vdash_\Sigma e : \mathtt{nat}$, $\vdash_{\Sigma,a} \mathtt{ret}(e') \ \mathsf{ok}$, and hence that $\vdash_{\Sigma,a} e' : \mathtt{nat}$. But because $e' \ \mathsf{val}_{\Sigma,a}$, $e'$ is a numeral, and hence we also have $\vdash_\Sigma e' : \mathtt{nat}$, as required. □

**Theorem 36.2** (Progress)**.**

1. *If $\vdash_\Sigma e : \tau$, then either $e \ \mathsf{val}_\Sigma$, or there exists $e'$ such that $e \underset{\Sigma}{\mapsto} e'$.*

2. *If $\vdash_\Sigma m \ \mathsf{ok}$ and $\mu : \Sigma$, then either $m \parallel \mu \ \mathsf{final}_\Sigma$ or $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$ for some $\mu'$ and $m'$.*

*Proof.* Simultaneously, by induction on rules (36.1). Consider rule (36.1d). By the first inductive hypothesis we have either $e \underset{\Sigma}{\mapsto} e'$ or $e \ \mathsf{val}_\Sigma$. In the former case rule (36.3i) applies. In the latter, we have by the second inductive hypothesis,

$$m \parallel \mu \otimes a \hookrightarrow e \ \mathsf{final}_{\Sigma,a} \quad \text{or} \quad m \parallel \mu \otimes a \hookrightarrow e \underset{\Sigma,a}{\longmapsto} m' \parallel \mu' \otimes a \hookrightarrow e'.$$

In the former case we apply rule (36.3k), and in the latter, rule (36.3j). □

Preview

## 36.2 Some Programming Idioms

The language **MA** is designed to expose the elegant interplay between the execution of an expression for its value and the execution of a command for its effect on assignables. In this section we show how to derive several standard idioms of imperative programming in **MA**.

We define the *sequential composition* of commands, written $\{x \hookleftarrow m_1 \,;\, m_2\}$, to stand for the command $\mathtt{bnd}\, x \hookleftarrow \mathtt{cmd}\,(m_1)\,;\, m_2$. This generalizes to an $n$-ary form by defining

$$\{x_1 \hookleftarrow m_1 \,; \ldots x_{n-1} \hookleftarrow m_{n-1} \,;\, m_n\},$$

to stand for the iterated composition

$$\{x_1 \hookleftarrow m_1 \,; \ldots \{x_{n-1} \hookleftarrow m_{n-1} \,;\, m_n\}\}.$$

We sometimes write just $\{m_1 \,;\, m_2\}$ for the composition $\{_ \hookleftarrow m_1 \,;\, m_2\}$ in which the returned value from $m_1$ is ignored; this generalizes in the obvious way to an $n$-ary form.

A related idiom, the command $\mathtt{do}\, e$, executes an encapsulated command and returns its result. By definition $\mathtt{do}\, e$ stands for the command $\mathtt{bnd}\, x \hookleftarrow e\,;\mathtt{ret}\, x$.

The *conditional* command, $\mathtt{if}\,(m)\, m_1\, \mathtt{else}\, m_2$, executes either $m_1$ or $m_2$ according to whether the result of executing $m$ is zero or not:

$$\{x \hookleftarrow m\,;\mathtt{do}\,(\mathtt{ifz}\, x\, \{\mathtt{z} \hookleftarrow \mathtt{cmd}\, m_1 \mid \mathtt{s}(_) \hookleftarrow \mathtt{cmd}\, m_2\})\}.$$

The returned value of the conditional is the value returned by the selected command.

The *while loop* command, $\mathtt{while}\,(m_1)\, m_2$, repeatedly executes the command $m_2$ while the command $m_1$ yields a non-zero number. It is defined as follows:

$$\mathtt{do}\,(\mathtt{fix}\, \mathit{loop} : \mathtt{cmd}\, \mathtt{is}\, \mathtt{cmd}\,(\mathtt{if}\,(m_1)\,\{\mathtt{ret}\,\mathtt{z}\}\,\mathtt{else}\,\{m_2\,;\mathtt{do}\, \mathit{loop}\})).$$

This commands runs the self-referential encapsulated command that, when executed, first executes $m_1$, branching on the result. If the result is zero, the loop returns zero (arbitrarily). If the result is non-zero, the command $m_2$ is executed and the loop is repeated.

A *procedure* is a function of type $\tau \rightharpoonup \mathtt{cmd}$ that takes an argument of some type, $\tau$, and yields an unexecuted command as result. Many procedures have the form $\lambda\,(x{:}\tau)\,\mathtt{cmd}\, m$, which we abbreviate to $\mathtt{proc}\,(x:\tau)\, m$. A *procedure call* is the composition of a function application with the activation of the resulting command. If $e_1$ is a procedure and $e_2$ is its argument, then the procedure call $\mathtt{call}\, e_1(e_2)$ is defined to be the command $\mathtt{do}\,(e_1(e_2))$, which immediately runs the result of applying $e_1$ to $e_2$.

As an example, here is a procedure of type $\mathtt{nat} \rightharpoonup \mathtt{cmd}$ that returns the factorial of its argument:



Preview

```
proc (x:nat) {
  dcl r := 1 in
  dcl a := x in
  { while ( @ a ) {
      y ← @ r
    ; z ← @ a
    ; r := (x-z+1)× y
    ; a := z-1
    }
    ; x ← @ r
    ; ret x
  }
}
```

The loop maintains the invariant that the contents of `r` is the factorial of $x$ minus the contents of `a`. Initialization makes this invariant true, and it is preserved by each iteration of the loop, so that upon completion of the loop the assignable `a` contains 0 and `r` contains the factorial of $x$, as required.

## 36.3 Typed Commands and Typed Assignables

So far we have restricted the type of the returned value of a command, and the contents of an assignable, to be `nat`. Can this restriction be relaxed, while adhering to the stack discipline?

The key to admitting returned and assignable values of other types lies in the details of the proof of Theorem 36.1. The proof of this theorem relies on an eager interpretation of the successor to ensure that the value is well-typed even in the absence of the locally declared assignable, $a$. The proof breaks down, and indeed the preservation theorem is false, when the return type of a command or the contents type of an assignable is unrestricted.

For example, if we may return values of procedure type, then the following command violates safety:

$$\mathtt{dcl}\, a := \mathtt{z}\, \mathtt{in}\, \{\mathtt{ret}\, (\mathtt{proc}\, (x : \mathtt{nat})\, \{a := x\})\}.$$

This command, when executed, allocates a new assignable, $a$, and returns a procedure that, when called, assigns its argument to $a$. But this makes no sense, because the assignable, $a$, is deallocated when the body of the declaration returns, but the returned value still refers to it. If the returned procedure is called, execution will get stuck in the attempt to assign to $a$.

A similar example shows that admitting assignables of arbitrary type is also unsound. For example, suppose that $b$ is an assignable whose contents are of type $\mathtt{nat} \rightharpoonup \mathtt{cmd}$, and consider the command

$$\mathtt{dcl}\, a := \mathtt{z}\, \mathtt{in}\, \{b := \mathtt{proc}\, (x : \mathtt{nat})\, \mathtt{cmd}(a := x)\, ; \mathtt{ret}\, \mathtt{z}\}.$$

We assign to $b$ a procedure that uses a locally declared assignable, $a$, and then leaves the scope of the declaration. If we then call the procedure stored in $b$, execution will get stuck attempting to assign to the non-existent assignable, $a$.

Preview

To admit declarations to return values and to admit assignables of types other than `nat`, we must rework the statics of **MA** to record the returned type of a command and to record the type of the contents of each assignable. First, we generalize the finite set, $\Sigma$, of active assignables to assign a mobile type to each active assignable so that $\Sigma$ has the form of a finite set of assumptions of the form $a \sim \tau$, where $a$ is an assignable. Second, we replace the judgment $\Gamma \vdash_\Sigma m \ \mathtt{ok}$ by the more general form $\Gamma \vdash_\Sigma m \div \tau$, stating that $m$ is a well-formed command returning a value of type $\tau$. Third, the type `cmd` must be generalized to $\mathtt{cmd}(\tau)$, which is written in examples as $\tau$ `cmd`, to specify the return type of the encapsulated command.

The statics given in Section 36.1.1 may be generalized to admit typed commands and typed assignables, as follows:

$$\frac{\Gamma \vdash_\Sigma m \div \tau}{\Gamma \vdash_\Sigma \mathtt{cmd}(m) : \mathtt{cmd}(\tau)} \tag{36.6a}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau \quad \tau \ \mathsf{mobile}}{\Gamma \vdash_\Sigma \mathtt{ret}(e) \div \tau} \tag{36.6b}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{cmd}(\tau) \quad \Gamma\, x : \tau \vdash_\Sigma m \div \tau'}{\Gamma \vdash_\Sigma \mathtt{bnd}(e; x.m) \div \tau'} \tag{36.6c}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau \quad \tau \ \mathsf{mobile} \quad \Gamma \vdash_{\Sigma, a \sim \tau} m \div \tau'}{\Gamma \vdash_\Sigma \mathtt{dcl}(e; a.m) \div \tau'} \tag{36.6d}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{get}[a] \div \tau} \tag{36.6e}$$

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{set}[a](e) \div \tau} \tag{36.6f}$$

Apart from the generalization to track returned types and content types, the most important change is to require that the types of assignables and of returned values must be mobile.

As in Chapter 33, these rules make use of the judgment $\tau \ \mathsf{mobile}$, which states that the type $\tau$ is *mobile*. The definition of this judgment is guided by the following *mobility condition*:

$$\textit{if } \tau \ \mathsf{mobile}, \vdash_\Sigma e : \tau \textit{ and } e \ \mathsf{val}_\Sigma, \textit{ then } \vdash_\varnothing e : \tau \textit{ and } e \ \mathsf{val}_\varnothing. \tag{36.7}$$

That is, a value of mobile type does not depend on any active assignables.

Because the successor is evaluated eagerly, the type `nat` may be deemed mobile:

$$\frac{}{\mathtt{nat} \ \mathsf{mobile}} \tag{36.8}$$

Because the body of a procedure may involve an assignable, no procedure type may be considered mobile, nor may the type of commands returning a given type, for similar reasons. On the other hand, a product of mobile types may safely be deemed mobile, provided that pairing is evaluated eagerly:

$$\frac{\tau_1 \ \mathsf{mobile} \quad \tau_2 \ \mathsf{mobile}}{\tau_1 \times \tau_2 \ \mathsf{mobile}} \tag{36.9}$$

Preview

Similarly, sums may be deemed mobile so long as the injections are evaluated eagerly:

$$\frac{\tau_1 \text{ mobile} \quad \tau_2 \text{ mobile}}{\tau_1 + \tau_2 \text{ mobile}} \tag{36.10}$$

Laziness defeats mobility, because values may contain suspended computations that depend on an assignable. For example, if the successor operation for the natural numbers were evaluated lazily, then $\mathtt{s}(e)$ would be a value for any expression, $e$, including one that refers to an assignable, $a$. Similarly, if pairing were lazy, then products may not be deemed mobile, and if injections were evaluated lazily, then sums may not either.

What about function types other than procedure types? We may think they are mobile, because a pure expression cannot depend on an assignable. Although this is indeed the case, the mobility condition need not hold. For example, consider the following value of type $\mathtt{nat} \rightharpoonup \mathtt{nat}$:

$$\lambda\,(x{:}\mathtt{nat})\,(\lambda\,(_{:}\tau\ \mathtt{cmd})\,\mathtt{z})(\mathtt{cmd}\,\{@\,a\}).$$

Although the assignable $a$ is not actually needed to compute the result, it nevertheless occurs in the value, in violation of the safety condition.

The mobility restriction on the statics of assignable declaration ensures that the type associated to an assignable is always mobile. We may therefore assume, without loss of generality, that the types associated to the assignables in the signature $\Sigma$ are mobile.

**Theorem 36.3** (Preservation for Typed Commands)**.**

1. *If $e \underset{\Sigma}{\mapsto} e'$ and $\vdash_\Sigma e : \tau$, then $\vdash_\Sigma e' : \tau$.*
2. *If $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$, with $\vdash_\Sigma m \div \tau$ and $\mu : \Sigma$, then $\vdash_\Sigma m' \div \tau$ and $\mu' : \Sigma$.*

**Theorem 36.4** (Progress for Typed Commands)**.**

1. *If $\vdash_\Sigma e : \tau$, then either $e$ $\mathsf{val}_\Sigma$, or there exists $e'$ such that $e \underset{\Sigma}{\mapsto} e'$.*
2. *If $\vdash_\Sigma m \div \tau$ and $\mu : \Sigma$, then either $m \parallel \mu$ $\mathsf{final}_\Sigma$ or $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$ for some $\mu'$ and $m'$.*

The proofs of Theorems 36.3 and 36.4 follows very closely the proof of Theorems 36.1 and 36.2. The main difference is that we appeal to the mobility condition to ensure that returned values and stored values are independent of the active assignables.

## 36.4 Notes

Modernized Algol is a reformulation of Reynolds's Idealized Algol (Reynolds, 1981) in which we have maintained a clearer separation between computations that depend on the memory and those that do not, and have omitted reliance on call-by-name for function application. The modal distinction between expressions and commands was present in the original formulation of Algol

Preview

60. The same separation has been popularized by Haskell, under the rubric "the `IO` monad," which classifies expressions that may have an effect.

What are called here *assignables* are regrettably called *variables* in the programming language literature. It is preferable to retain the well-established meaning of a variable as standing for an unspecified object of a specified type, but to do so requires that we invent a new word for the name of a piece of mutable storage. The word *assignable* seems apt, and equally as convenient as the misappropriated word *variable*.

Perhaps the most significant departure from Reynolds's Idealized Algol is that we rescind the convention that assignables may serve as forms of expression, allowing assignables to behave more like variables than they do here. The penalty is that the meaning of an expression in Idealized Algol depends on the memory, whereas in Modernized Algol the only dependency is on the domain of the memory, not its contents.

The concept of mobility of a type was introduced in the ML5 language for distributed computing (Murphy et al., 2004), with the similar meaning that a value of a mobile type cannot depend on local resources. Here the mobility restriction is used to ensure that the language adheres to the stack discipline.

## Exercises

**36.1**. Originally Algol had both *scalar* assignables, whose contents are to be thought of as atomic values, and *array* assignables, which may be thought of as a finite sequence of scalar assignables. Like scalar assignables, array assignables are to be stack-allocated. Formulate the extension of **MA** with array assignables, ensuring that the language remains type safe, but allowing that computation may abort if a non-existent array element is accessed.

**36.2**. Consider carefully the behavior of assignable declarations within recursive procedures, as in the following expression

$$\mathtt{fix}\ p\ \mathtt{is}\ \lambda\ (p{:}\tau)\ \mathtt{dcl}\ a := e\ \mathtt{in}\ \mathtt{cmd}(m)$$

of type $\tau \rightharpoonup \rho\ \mathtt{cmd}$ for some $\rho$. Because $p$ is recursive, the body $m$ of the procedure may call itself during its execution, causing the *same* declaration to be executed more than once. Explain the dynamics of getting and setting $a$ in that scenario. This behavior exemplified the concept of *block structure* in Algol, which was thereafter deemed a *block-structured language*.

**36.3**. Originally Algol considered assignables to be forms of expression that stand for their contents in memory. Thus, if $a$ is an assignable containing a number, one could write expressions such as $a + a$ that would evaluate to twice the contents of $a$. Moreover, one could write commands such as $a := a + a$ to double the contents of $a$. These conventions encouraged programmers to think of assignables as variables, quite the opposite of their separation in **MA**. This convention, combined with an over-emphasis on concrete syntax, led to a conundrum about the different roles of $a$ in the above assignment command: its meaning on the left of the assignment is different from its meaning on the right. Inevitably these came to be called the *left-*, or *l-value*, and the *right-*, or *r-value* of the assignable $a$, corrresponding to its position



Preview

in the assignment statement. When viewed as abstract syntax, though, there is no ambiguity to be explained: the assignment operator is indexed by its target assignable, rather than taking as argument an expression that happens to be an assignable, so that the command is $\mathtt{set}[a](a+a)$, not $\mathtt{set}(a; a+a)$.

This still leaves the puzzle of how to regard assignables as forms of expression. As a first cut, reformulate the dynamics of **MA** to account for this. Reformulate the dynamics of expressions in terms of the judgments $e \parallel \mu \underset{\Sigma}{\mapsto} e' \parallel \mu'$ and $e \parallel \mu$ final that allow evaluation of $e$ to depend on the contents of the memory. Each use of an assignable as an expression should require one access to the memory. Then prove *memory invariance*:: if $e \parallel \mu \underset{\Sigma}{\mapsto} e' \parallel \mu'$, then $\mu' = \mu$.

A natural generalization is to permit any sequence of commands to be considered as an expression, provided that they are all *passive* in the sense that no assignments are permitted. Write $\mathtt{do}\,\{m\}$, where $m$ is a passive command, for a *passive block* whose evaluation consists of executing the command $m$ on the current memory, using its return value as the value of the expression. Observe that memory invariance holds for passive blocks.

The use of an assignable $a$ as an expression may now be rendered as the passive block $\mathtt{do}\,\{@a\}$. More complex uses of assignables as expressions admit several different interpretations using passive blocks. For example, an expression such as $a + a$ might be rendered in one of two ways:

(a) $\mathtt{do}\,\{@a\} + \mathtt{do}\,\{@a\}$, or

(b) $\mathtt{let}\ x\ \mathtt{be}\ \mathtt{do}\,\{@a\}\ \mathtt{in}\ x + x$.

The latter formulation accesses $a$ only once, but uses its value twice. Comment on the significance of there being two different interpretations of $a + a$.

**36.4**. Recursive procedures in Algol are *declared* using a command of the form $\mathtt{proc}\ p(x:\tau) : \rho\ \mathtt{is}\ m\ \mathtt{in}\ m'$, which is governed by the typing rule

$$\frac{\Gamma, p : \tau \rightharpoonup \rho\ \mathtt{cmd}, x : \tau \vdash_\Sigma m \div \rho \quad \Gamma, p : \tau \rightharpoonup \rho\ \mathtt{cmd} \vdash_\Sigma m' \div \tau'}{\Gamma \vdash_\Sigma \mathtt{proc}\ p(x:\tau) : \rho\ \mathtt{is}\ m\ \mathtt{in}\ m' \div \tau'}\ . \tag{36.11}$$

From the present viewpoint it is peculiar to insist on declaring procedures at all, since they are simply values of procedure type, and even more peculiar to insist that they be confined for use within a command. One justification for this limitation, though, is that Algol included a peculiar feature, called an *own variable*[1] that was declared for use within the procedure, but whose state persisted across calls to the procedure. One application would be to a procedure that generated pseudo-random numbers based on a stored seed that influenced the behavior of successive calls to it. Give a formulation in **MA** of the extended declaration

$$\mathtt{proc}\ p(x:\tau) : \rho\ \mathtt{is}\ \{\mathtt{own}\ a := e\ \mathtt{in}\ m\}\ \mathtt{in}\ m'$$

[1] That is to say, an *own assignable*.



Preview

in which $a$ is declared as an "own" of the procedure $p$. Contrast the meaning of the foregoing declaration with the following one:

$$\texttt{proc}\, p(x : \tau) : \rho\, \texttt{is}\, \{\texttt{dcl}\, a := e\, \texttt{in}\, m\}\, \texttt{in}\, m'.$$

**36.5**. A natural generalization of own assignables is to permit the creation of many such scenarios for a single procedure (or mutually recursive collection of procedures), with each instance creating its own persistent state. This motivated the concept of a *class* in Simula-67 as a collection of procedures, possibly mutually recursive, that shared common persistent state. Each instance of a class is called an *object* of that class; calls to its constituent procedures mutate the private persistent state. Formulate this 1967 precursor of imperative object-oriented programming in the context of **MA**.

**36.6**. There are several ways to formulate an abstract machine for **MA** that accounts for both the *control stack*, which sequentializes execution (as described in Chapter 28 for **PCF**), and the *data stack*, which records the contents of the assignables. A *consolidated stack* combines these two separate concepts into one, whereas *separated stacks* keeps the memory separate from the control stack, much as we have done in the structural dynamics given by Rules (36.3). In either case the storage required for an assignable is deallocated when the scope of that assignable is exited, a key benefit of the stack discipline for assignables in **MA**.

With a modal separation between expressions and commands it is natural to use a structural dynamics for expressions (given by the transition and value judgments, $e \longmapsto e'$ and $e\ \mathsf{val}$), and a stack machine dynamics for commands.

(a) Formulate a consolidated stack machine in which both assignables and stack frames are recorded on the same stack. Consider states $k \rhd_\Sigma m$, where $\vdash_\Sigma k \lhd: \tau$ and $\vdash_\Sigma m \div \tau$, and $k \lhd_\Sigma e$, where $\vdash_\Sigma k \lhd: \tau$ and $\vdash_\Sigma e : \tau$. Comment on the implementation methods required for a consolidated stack.

(b) Formulate a separated stack machine in which the memory is maintained separately from the control stack. Consider states of the form $\mu \parallel k \rhd_\Sigma m$, where $\mu : \Sigma$, $\vdash_\Sigma k \lhd: \tau$, and $\vdash_\Sigma m \div \tau$, and of the form $\mu \parallel k \lhd_\Sigma e$, where $\vdash_\Sigma k \lhd: \tau$, $\vdash_\Sigma e : \tau$, and $e\ \mathsf{val}$.

Preview

# Chapter 37

# Assignable References

A *reference* to an assignable, $a$, is a value, written $\&a$, of *reference type* that uniquely determines the assignable, $a$. A reference to an assignable provides the *capability* to get or set the contents of that assignable, even if the assignable itself is not in scope at the point at which it is used. Two references may also be compared for equality to test whether or not they govern the same underlying assignable. If two references are equal, then setting one will affect the result of getting the other; if they are not equal, then setting one cannot influence the result of getting from the other. Two references that govern the same underlying assignable are said to be *aliases*. The possibility of aliasing complicates reasoning about the correctness of code that uses references, for we must always consider for any two references whether or not they might be aliases.

Reference types are compatible with both a scoped and a scope-free allocation of assignables. When assignables are scoped, the range of significance of a reference type must be limited to the scope of the assignable to which it refers. This may be achieved by declaring that reference types are immobile, so that they cannot be returned from the body of a declaration, nor stored in an assignable. Although ensuring adherence to the stack discipline, this restriction precludes the use of references to create mutable data structures, those whose structure can be altered during execution. Mutable data structures have a number of applications in programming, including improving efficiency (often at the expense of expressiveness) and allowing the creation of cyclic (self-referential) structures. Supporting mutability requires that assignables be given a scope-free dynamics, so that their lifetime persists beyond the scope of their declaration. Consequently, all types may be regarded as mobile, and hence values of any type may be stored in assignables or returned from commands.

## 37.1 Capabilities

The commands $\texttt{get}[a]$ and $\texttt{set}[a](e)$ in **MA** operate on a statically specified assignable, $a$. To even write these commands requires that the assignable, $a$, be in scope at the point where the command occurs. But suppose that we wish to define a procedure that, say, updates an assignable to double its previous value, and returns the previous value. We can easily write such a procedure for any

Preview

given assignable, $a$, but what if we wish to write a generic procedure that works for any given assignable?

One way to do this is provide the procedure with the *capability* to get and set the contents of some caller-specified assignable. Such a capability is a pair consisting of a *getter* and a *setter* for that assignable. The getter for an assignable, $a$, is a command that, when executed, returns the contents of $a$. The setter for an assignable, $a$, is a procedure that, when applied to a value of suitable type, assigns that value to $a$. Thus, a capability for an assignable $a$ containing a value of type $\tau$ is a value of type

$$\tau\,\texttt{cap} \triangleq \tau\,\texttt{cmd} \times (\tau \rightharpoonup \tau\,\texttt{cmd}).$$

A capability for getting and setting an assignable $a$ containing a value of type $\tau$ is given by the pair

$$\langle \texttt{cmd}\,(@\,a), \texttt{proc}\,(x:\tau)\,a := x\rangle$$

of type $\tau\,\texttt{cap}$. Because a capability type is a product of a command type and a procedure type, no capability type is mobile. This means, in particular, that a capability cannot be returned from a command, nor stored into an assignable. This is as it should be, for otherwise we would violate the stack discipline for allocating assignables.

Using capabilities, the proposed generic doubling procedure may be programmed as follows:

$$\texttt{proc}\,(\langle \mathit{get}, \mathit{set}\rangle : \texttt{nat cmd} \times (\tau \rightharpoonup \tau\,\texttt{cmd}))\,\{x \hookleftarrow \texttt{do}\,\mathit{get}\,;\, y \hookleftarrow \texttt{do}\,(\mathit{set}(x+x))\,;\texttt{ret}\,x\}.$$

The procedure is to be called with the capability to access an assignable, $a$. When executed, it invokes the getter to obtain the contents of $a$, and then invokes the setter to assign to $a$, returning the previous value. Observe that the assignable, $a$, need not be directly accessible by this procedure; the capability provided by the caller comprises the commands required to get and set $a$.

## 37.2 Scoped Assignables

A weakness of using a capability to provide indirect access to an assignable is that there is no guarantee that a given getter/setter pair are in fact the capability for a particular assignable. For example, we might (deliberately or accidentally) pair the getter for $a$ with the setter for $b$, leading to unexpected behavior. There is nothing in the type system that prevents the creation of such mismatched pairs.

To avoid this we introduce the concept of a *reference* to an assignable. A reference is a value from which we may obtain the capability to get and set a particular assignable. Moreover, two references may be compared for equality to determine whether or not they act on the same assignable.[1] The *reference type* $\texttt{ref}(\tau)$ has as values references to assignables of type $\tau$. The introduction and

[1] The getter and setter are not quite sufficient to define equality, because not all types admit a run-time equality test. When they do, and when there are at least two distinct values of the contents type, we can determine whether they are aliases by assigning to one and checking whether the contents of the other is changed.



Preview

elimination forms for this type are given by the following syntax chart:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{ref}(\tau)$ | $\tau\,\mathtt{ref}$ | assignable |
| Exp | $e$ | $::=$ | $\mathtt{ref}[a]$ | $\&a$ | reference |
| Cmd | $m$ | $::=$ | $\mathtt{getref}(e)$ | $*e$ | contents |
| | | | $\mathtt{setref}(e_1;e_2)$ | $e_1 *= e_2$ | update |

The statics of reference types is defined by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma,a\sim\tau} \mathtt{ref}[a] : \mathtt{ref}(\tau)} \tag{37.1a}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{ref}(\tau)}{\Gamma \vdash_\Sigma \mathtt{getref}(e) \div \tau} \tag{37.1b}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathtt{ref}(\tau) \quad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma \mathtt{setref}(e_1;e_2) \div \tau} \tag{37.1c}$$

Rule (37.1a) specifies that the name of any active assignable is an expression of type $\mathtt{ref}(\tau)$.

The dynamics of reference types simply defers to the corresponding operations on assignables, and does not alter the underlying dynamics of assignables:

$$\frac{}{\mathtt{ref}[a]\ \mathsf{val}_{\Sigma,a\sim\tau}} \tag{37.2a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{getref}(e) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{getref}(e') \parallel \mu} \tag{37.2b}$$

$$\frac{}{\mathtt{getref}(\mathtt{ref}[a]) \parallel \mu \underset{\Sigma,a\sim\tau}{\longmapsto} \mathtt{get}[a] \parallel \mu} \tag{37.2c}$$

$$\frac{e_1 \underset{\Sigma}{\mapsto} e_1'}{\mathtt{setref}(e_1;e_2) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{setref}(e_1';e_2) \parallel \mu} \tag{37.2d}$$

$$\frac{}{\mathtt{setref}(\mathtt{ref}[a];e) \parallel \mu \underset{\Sigma,a\sim\tau}{\longmapsto} \mathtt{set}[a](e) \parallel \mu} \tag{37.2e}$$

A reference to an assignable is a value. The `getref` and `setref` operations on references defer to the corresponding operations on assignables once the reference has been determined.

Because references give rise to capabilities, the reference type is deemed to be immobile. Consequently, references cannot be stored in assignables or returned from commands. This ensures safety, as may be readily verified by extending the proof given in Chapter 36.

Preview

As an example of the use of references, the generic doubling procedure discussed in the preceding section may be programmed with references as follows:

$$\texttt{proc}\,(r:\texttt{nat ref})\,\{x \hookleftarrow *r\,;r *= x + x\,;\texttt{ret}\,x\}.$$

Because the argument is a reference, rather than a capability, there is no possibility that the getter and setter refer to different assignables.

The ability to pass references to procedures comes at a price. When handling two or more references at the same time, we must consider the possibility that they are aliases, which is to say that both refer to the same underlying assignable. Consider, for example, a procedure that, when given two references, $x$ and $y$, adds twice the contents of $y$ to the contents of $x$. One way to write this code creates no complications:

$$\lambda\,(x{:}\texttt{nat ref})\,\lambda\,(y{:}\texttt{nat ref})\,\texttt{cmd}\,\{x' \hookleftarrow *x\,;y' \hookleftarrow *y\,;x *= x' + y' + y'\}.$$

Even if $x$ and $y$ refer to the same assignable, the effect will be to set the contents of the assignable referenced by $x$ to the sum of its orginal contents and twice the contents of the assignable referenced by $y$.

But now consider the following apparently equivalent implementation of this procedure:

$$\lambda\,(x{:}\texttt{nat ref})\,\lambda\,(y{:}\texttt{nat ref})\,\texttt{cmd}\,\{x \mathrel{+}= y\,;x \mathrel{+}= y\},$$

where $x \mathrel{+}= y$ is the command

$$\{x' \hookleftarrow *x\,;y' \hookleftarrow *y\,;x *= x' + y'\}$$

that adds the contents of $y$ to the contents of $x$. The second implementation works properly provided that $x$ and $y$ do not refer to the same assignable. For if they both reference the same assignable, $a$, with contents $n$, the result is that $a$ is to set $4 \times n$, instead of the intended $3 \times n$, because the second get of $y$ is influenced by the first assignment to $x$.

In this case it is entirely obvious how to avoid the problem: use the first implementation, rather than the second. But the difficulty is not in fixing the problem once it has been uncovered, but rather noticing the problem in the first place. Wherever references (or capabilities) are used, the problems of interference lurk. Avoiding them requires very careful consideration of all possible aliasing relationships among all of the references in play at a given point of a computation. The problem is that the number of possible aliasing relationships among $n$ references grows quadratically in $n$, because we must consider all possible pairings.

## 37.3 Free Assignables

Although it is interesting to note that references and capabilities are compatible with the stack discipline, this is achieved at the expense of their utility. Because references are not mobile, it is not possible to build a data structure containing references internally. In particular, this restriction precludes programming mutable data structures (those whose structure changes during execution).



Preview

To allow for more flexible uses of references, we must relax the requirement that assignables are to be stack-allocated, and instead arrange that the lifetime of an assignable extends beyond the scope of its declaration. Such assignables are called *scope-free*, or just *free*, assignables. If all assignables are scope-free, then every type may safely be deemed mobile. Consequently, references may be used to implement mutable and cyclic data structures.

Supporting free assignables amounts to changing the dynamics so that allocation of assignables persists across transitions. This is achieved by employing transition judgments of the form

$$\nu\,\Sigma\,\{\,m \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,m' \parallel \mu'\,\}.$$

Execution of a command may allocate new assignables, may alter the contents of existing assignables, and may give rise to a new command to be executed at the next step. The rules defining the dynamics of free assignables are as follows:

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\mathtt{ret}(e) \parallel \mu\,\}\ \mathsf{final}} \tag{37.3a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{ret}(e) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{ret}(e') \parallel \mu\,\}} \tag{37.3b}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{bnd}(e; x.m) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{bnd}(e'; x.m) \parallel \mu\,\}} \tag{37.3c}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\mathtt{bnd}(\mathtt{cmd}(\mathtt{ret}(e)); x.m) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,[e/x]m \parallel \mu\,\}} \tag{37.3d}$$

$$\frac{\nu\,\Sigma\,\{\,m_1 \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,m_1' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{bnd}(\mathtt{cmd}(m_1); x.m_2) \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,\mathtt{bnd}(\mathtt{cmd}(m_1'); x.m_2) \parallel \mu'\,\}} \tag{37.3e}$$

$$\frac{}{\nu\,\Sigma, a \sim \tau\,\{\,\mathtt{get}[a] \parallel \mu \otimes a \hookrightarrow e\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,\mathtt{ret}(e) \parallel \mu \otimes a \hookrightarrow e\,\}} \tag{37.3f}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{set}[a](e) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{set}[a](e') \parallel \mu\,\}} \tag{37.3g}$$

$$\frac{e\ \mathsf{val}_{\Sigma, a \sim \tau}}{\nu\,\Sigma, a \sim \tau\,\{\,\mathtt{set}[a](e) \parallel \mu \otimes a \hookrightarrow _\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,\mathtt{ret}(e) \parallel \mu \otimes a \hookrightarrow e\,\}} \tag{37.3h}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{dcl}(e; a.m) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{dcl}(e'; a.m) \parallel \mu\,\}} \tag{37.3i}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\mathtt{dcl}(e; a.m) \parallel \mu\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,m \parallel \mu \otimes a \hookrightarrow e\,\}} \tag{37.3j}$$

Preview

The language **RMA** extends **MA** with references to free assignables. Its dynamics is similar to that of references to scoped assignables given earlier.

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{getref}(e) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{getref}(e') \parallel \mu\,\}} \tag{37.4a}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{getref}(\mathtt{ref}[a]) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{get}[a] \parallel \mu\,\}} \tag{37.4b}$$

$$\frac{e_1 \underset{\Sigma}{\mapsto} e_1'}{\nu\,\Sigma\,\{\,\mathtt{setref}(e_1; e_2) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{setref}(e_1'; e_2) \parallel \mu\,\}} \tag{37.4c}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{setref}(\mathtt{ref}[a]; e_2) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{set}[a](e_2) \parallel \mu\,\}} \tag{37.4d}$$

Observe that the evaluation of expressions cannot alter or extend the memory, only commands may do this.

As an example of the use of **RMA**, consider the command $\mathtt{newref}[\tau](e)$ defined by

$$\mathtt{dcl}\,a := e\,\mathtt{in}\,\mathtt{ret}\,(\&a). \tag{37.5}$$

This command allocates a fresh assignable, and returns a reference to it. Its static and dynamics may be derived from the foregoing rules, as follows:

$$\frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathtt{newref}[\tau](e) \mathrel{\dot{\sim}} \mathtt{ref}(\tau)} \tag{37.6}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{newref}[\tau](e) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{newref}[\tau](e') \parallel \mu\,\}} \tag{37.7a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\mathtt{newref}[\tau](e) \parallel \mu\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,\mathtt{ret}(\mathtt{ref}[a]) \parallel \mu \otimes a \hookrightarrow e\,\}} \tag{37.7b}$$

Oftentimes the command $\mathtt{newref}[\tau](e)$ is taken as primitive, and the declaration command is omitted. In that case all assignables are accessed by reference, and no direct access to assignables is provided.

## 37.4 Safety

Although the proof of safety for references to scoped assignables presents few difficulties, the safety for free assignables is suprisingly tricky. The main difficulty is to account for the possibility of cyclic dependencies of data structures in memory. The contents of one assignable may contain a



Preview

reference to itself, or a reference to another assignable that contains a reference to it, and so forth. For example, consider the following procedure, $e$, of type $\mathtt{nat} \rightharpoonup \mathtt{nat}\ \mathtt{cmd}$:

$$\mathtt{proc}\,(x:\mathtt{nat})\,\{\mathtt{if}\,(x)\,\mathtt{ret}\,(1)\,\mathtt{else}\,\{f \hookleftarrow @a\,;y \hookleftarrow f(x-1)\,;\mathtt{ret}\,(x*y)\}\}.$$

Let $\mu$ be a memory of the form $\mu' \otimes a \hookrightarrow e$ in which the contents of $a$ contains, via the body of the procedure, a reference to $a$ itself. Indeed, if the procedure $e$ is called with a non-zero argument, it will "call itself" by indirect reference through $a$.

The possibility of cyclic dependencies means that some care in the definition of the judgment $\mu : \Sigma$ is required. The following rule defines the well-formed states:

$$\frac{\vdash_\Sigma m \div \tau \quad \vdash_\Sigma \mu : \Sigma}{\nu\,\Sigma\,\{\,m \parallel \mu\,\}\ \mathsf{ok}} \tag{37.8}$$

The first premise of the rule states that the command $m$ is well-formed relative to $\Sigma$. The second premise states that the memory, $\mu$, conforms to $\Sigma$, *relative to the whole of* $\Sigma$ so that cyclic dependencies are permitted. The judgment $\vdash_{\Sigma'} \mu : \Sigma$ is defined as follows:

$$\frac{\forall a \sim \tau \in \Sigma \quad \exists e \quad \mu(a) = e \text{ and } \vdash_{\Sigma'} e : \tau}{\vdash_{\Sigma'} \mu : \Sigma} \tag{37.9}$$

**Theorem 37.1** (Preservation)**.**

1. *If* $\vdash_\Sigma e : \tau$ *and* $e \underset{\Sigma}{\mapsto} e'$, *then* $\vdash_\Sigma e' : \tau$.
2. *If* $\nu\,\Sigma\,\{\,m \parallel \mu\,\}$ *ok and* $\nu\,\Sigma\,\{\,m \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,m' \parallel \mu'\,\}$, *then* $\nu\,\Sigma'\,\{\,m' \parallel \mu'\,\}$ *ok.*

*Proof.* Simultaneously, by induction on transition. We prove the following stronger form of the second statement:

> If $\nu\,\Sigma\,\{\,m \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,m' \parallel \mu'\,\}$, where $\vdash_\Sigma m \div \tau$, $\vdash_\Sigma \mu : \Sigma$, then $\Sigma'$ extends $\Sigma$, and $\vdash_{\Sigma'} m' \div \tau$, and $\vdash_{\Sigma'} \mu' : \Sigma'$.

Consider, for example, the transition

$$\nu\,\Sigma\,\{\,\mathtt{dcl}(e; a.m) \parallel \mu\,\} \longmapsto \nu\,\Sigma, a \sim \rho\,\{\,m \parallel \mu \otimes a \hookrightarrow e\,\}$$

where $e\ \mathsf{val}_\Sigma$. By assumption and inversion of rule (36.6d) we have $\vdash_\Sigma e : \rho$, $\vdash_{\Sigma, a\sim\rho} m \div \tau$, and $\vdash_\Sigma \mu : \Sigma$. But because extension of $\Sigma$ with a fresh assignable does not affect typing, we also have $\vdash_{\Sigma,a\sim\rho} \mu : \Sigma$ and $\vdash_{\Sigma,a\sim\rho} e : \rho$, from which it follows by rule (37.9) that $\vdash_{\Sigma,a\sim\rho} \mu \otimes a \hookrightarrow e : \Sigma, a \sim \rho$.

The other cases follow a similar pattern, and are left as an exercise for the reader.

□

**Theorem 37.2** (Progress)**.**

1. *If* $\vdash_\Sigma e : \tau$, *then either* $e\ \mathsf{val}_\Sigma$ *or there exists* $e'$ *such that* $e \underset{\Sigma}{\mapsto} e'$.



Preview

2. *If* $\nu\,\Sigma\,\{\,m \parallel \mu\,\}$ *ok then either* $\nu\,\Sigma\,\{\,m \parallel \mu\,\}$ *final or* $\nu\,\Sigma\,\{\,m \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,m' \parallel \mu'\,\}$ *for some* $\Sigma'$, $\mu'$, *and* $m'$.

*Proof.* Simultaneously, by induction on typing. For the second statement we prove the stronger form

> If $\vdash_\Sigma m \div \tau$ and $\vdash_\Sigma \mu : \Sigma$, then either $\nu\,\Sigma\,\{\,m \parallel \mu\,\}$ final, or $\nu\,\Sigma\,\{\,m \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,m' \parallel \mu'\,\}$ for some $\Sigma'$, $\mu'$, and $m'$.

Consider, for example, the typing rule

$$\frac{\Gamma \vdash_\Sigma e : \rho \quad \Gamma \vdash_{\Sigma, a\sim\rho} m \div \tau}{\Gamma \vdash_\Sigma \mathtt{dcl}(e; a.m) \div \tau}$$

We have by the first inductive hypothesis that either $e\ \mathsf{val}_\Sigma$ or $e \underset{\Sigma}{\mapsto} e'$ for some $e'$. In the latter case we have by rule (37.3i)

$$\nu\,\Sigma\,\{\,\mathtt{dcl}(e; a.m) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{dcl}(e'; a.m) \parallel \mu\,\}.$$

In the former case we have by rule (37.3j) that

$$\nu\,\Sigma\,\{\,\mathtt{dcl}(e; a.m) \parallel \mu\,\} \longmapsto \nu\,\Sigma, a\sim\rho\,\{\,m \parallel \mu \otimes a \hookrightarrow e\,\}.$$

As another example, consider the typing rule

$$\frac{}{\Gamma \vdash_{\Sigma, a\sim\tau} \mathtt{get}[a] \div \tau}$$

By assumption $\vdash_{\Sigma, a\sim\tau} \mu : \Sigma, a \sim \tau$, and hence there exists $e\ \mathsf{val}_{\Sigma, a\sim\tau}$ such that $\mu = \mu' \otimes a \hookrightarrow e$ and $\vdash_{\Sigma, a\sim\tau} e : \tau$. By rule (37.3f)

$$\nu\,\Sigma, a \sim \tau\,\{\,\mathtt{get}[a] \parallel \mu' \otimes a \hookrightarrow e\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,\mathtt{ret}(e) \parallel \mu' \otimes a \hookrightarrow e\,\},$$

as required. The other cases are handled similarly. □

## 37.5 Benign Effects

The modal separation between commands and expressions ensures that the meaning of an expression does not depend on the (ever-changing) contents of assignables. Although this is advantageous in many, perhaps most, situations, it also precludes programming techniques that use storage effects to implement purely functional behavior. A prime example is memoization in a lazy language (which is described in detail in Chapter 38.) Externally, a suspended computation behaves exactly like the underlying computation; internally, an assignable is associated with the computation that stores the result of any evaluation of the computation for future use. Another

Preview

class of examples are self-adjusting data structures, which use state internally to improve their efficiency without changing their overall purely functional behavior. For example, a splay tree is a binary search tree that uses mutation internally to rebalance the tree as elements are inserted, deleted, and retrieved. This ensures that, for example, lookup operations take time proportional to the logarithm of the number of elements.

These are examples of *benign storage effects*, uses of mutation in a data structure to improve efficiency without disrupting its functional behavior. Because values are forms of expression, it is essential to relax the strict separation between expressions and commands that characterizes the modal type system for storage effects described in Chapter 36. Although several *ad hoc* methods have been considered in the literature, the most general approach is to simply do away with the distinction entirely, coalescing expressions and commands into a single syntactic category. The penalty is that the type system no longer ensures that an expression of type $\tau$ denotes a value of that type; it might, in addition, engender storage effects during evaluation. The benefit of this approach is that it is easy to implement benign effects that are impossible to implement when a strict modal separation between expressions and commands is maintained.

The language **RPCF** is an extension of **PCF** with references to free assignables. The following rules define the statics of the distinctive featurs of **RPCF**:

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau_1 \quad \Gamma \vdash_{\Sigma, a \sim \tau_1} e_2 : \tau_2}{\Gamma \vdash_\Sigma \mathtt{dcl}(e_1; a.e_2) : \tau_2} \tag{37.10a}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{get}[a] : \tau} \tag{37.10b}$$

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{set}[a](e) : \tau} \tag{37.10c}$$

Correspondingly, the dynamics of **RPCF** is given by transitions of the form

$$\nu\, \Sigma \, \{ \, e \parallel \mu \, \} \longmapsto \nu\, \Sigma' \, \{ \, e' \parallel \mu' \, \},$$

where $e$ is an expression, rather than a command. The rules defining the dynamics are very similar to those for **RMA**, but with commands and expressions integrated into a single category.

To illustrate the concept of a benign effect, consider the technique of *backpatching* to implement recursion. Here is a formulation of the factorial function in **RPCF** in which recursive calls are mediated by an assignable containing the function itself:

```
dcl a := λn:nat.0 in
  { f ← a := λn:nat.ifz(n, 1, n′.n×(@a)(n′))
  ; ret(f)
  }
```

This expression returns a function of type $\mathtt{nat} \rightharpoonup \mathtt{nat}$ that is obtained by (a) allocating a free assignable initialized arbitrarily (and immaterially) with a function of this type, (b) defining a $\lambda$-abstraction in which each "recursive call" consists of retrieving and applying the function stored in that assignable, (c) assigning this function to the assignable, and (d) returning that function. The

Preview

result is a function on the natural numbers, even though it uses state internally to its implementation.

Backpatching is not expressible in **RMA**, because it of the explicit reliance on state. Let us attempt to recode the previous example in **RMA**:

```
dcl a := proc(n:nat){ret 0} in
  { f ← a := ...
  ; ret(f)
  },
```

where the elided procedure assigned to $a$ is given by

```
proc(n:nat) {if (ret(n)) {ret(1)} else {f←@a; x←f(n-1); ret(n×x)}}.
```

The difficulty is that what we have is a command, rather than an expression. Moreover, the result of the command is of procedure type `nat` ⇀ (`nat cmd`), rather than function type `nat` ⇀ `nat`. This means that we cannot use the factorial "function" (so implemented) in an expression, but must instead execute it as a command, so that the factorial of $n$ is computed by writing

```
{ f ← fact; x ← f(n); ret(x) }.
```

## 37.6 Notes

Reynolds (1981) uses capabilities to provide indirect access to assignables; it is a short step from there to references in the form considered here. Often references are considered only for free assignables, but this is not essential. It is perfectly possible to have references to scoped assignables as well, provided that suitable mobility restrictions are imposed to ensure adherence to the stack discipline. The proof of safety of free references given here is inspired by Wright and Felleisen (1994) and Harper (1994).

Benign effects are central to the distinction between Haskell, which provides an Algol-like separation between commands and expressions, and ML, which integrates evaluation with execution. Each approach has its advantages and disadvantages; neither is uniformly superior to the other.

## Exercises

**37.1**. Consider scoped array assignables as described in Exercise **36.1**. Just as with scalar assignables, it is possible to have references to array assignables. Extend the treatment of array assignables in Exercise **36.1**, to account for array assignable references.

**37.2**. References to scope-free assignables are traditionally used to implement recursive data structures such as mutable lists and trees. Examine such data structures in the context of **RMA** enriched with sum, product, and recursive types.

Consider first the type of mutable lists, called *linked lists* in textbooks on data structures. Give *six* different types that could be considered a type of mutable lists, with the following characteristics:

Preview

(a) A mutable list may only be updated *in toto* by replacing it with another (immutable) list.

(b) A mutable list may be altered in one of two ways, to render it empty, or to change both its head and tail element simultaneously. The tail element is any other such mutable list, so, for example, circularities may arise.

(c) A mutable list is, permanently, either empty or not. If not, both its head and tail may be simultaneously updated.

(d) A mutable list is, permanently, either empty or not. If not, its tail, but not its head, can be set to another such list.

(e) A mutable list is, permanently, either empty or not. If not, either its head or its tail elements may be altered independently.

(f) A mutable list may be altered to become either empty or non-empty. If it is non-empty, either it head, or its tail, may be altered independently of one another.

Discuss the merits and deficiencies of each representation.

**37.3**. In light of Exercise 37.2, it is natural to ask for a clear definition of a *linked list*. There is no clear reason why any of those considered in that exercise should be considered canonical, yet textbooks often insist on a particular choice without any semantic justification. Worse, most textbooks are completely ignorant of sum types, and give a definition of linked list that is distinctly inferior to any of those considered in Exercise 37.2. The closest thing to a sum type in the standard accounts is the fallacious notion of the "null pointer" discussed in Chapter 11, on which data structure representations rely very heavily, and which are the source of innumerable errors in programs. Formulate the standard treatment of mutable linked lists in an extension of **RMA** with product, option, boolean, and recursive types. *Hint*: a list is represented by a reference to a labelled tuple containing three mutable components, one to indicate whether the end of the list has been reached, one for the head element of the list, and one for the tail (next component) of the list. All mutable fields are, implicitly, optional, because any of them could be "null", and are assignable independently of one another. Discuss the glaring deficiencies of the standard representation of mutable linked lists and ask yourself, why are we making such a colossal mess of such a simple idea?

Preview

Preview

# Part XV

# Laziness

Preview

Preview

# Chapter 38

# Lazy Evaluation

*Lazy evaluation* refers to a variety of concepts that seek to *defer* evaluation of an expression until it is definitely required, and to *share* the results of any such evaluation among all instances of a single deferred computation. The net result is that a computation is performed at most once among all of its instances. Laziness manifests itself in a number of ways.

One form of laziness is the *by-need* evaluation strategy for function application. Recall from Chapter 8 that the by-name evaluation order passes the argument to a function in unevaluated form so that it is only evaluated if it is actually used. But because the argument is replicated by substitution, it may be evaluated more than once. By-need evaluation ensures that the argument to a function is evaluated at most once, by ensuring that all copies of an argument share the result of evaluating any one copy.

Another form of laziness is the concept of a *lazy data structure*. As we have seen in Chapters 10, 11, and 16, we may choose to defer evaluation of the components of a data structure until they are actually required, rather than when the data structure is created. But if a component is required more than once, then the same computation will, without further provision, be repeated on each use. To avoid this, the deferred portions of a data structure are shared so an access to one will propagate its result to all occurrences of the same computation.

Yet another form of laziness arises from the concept of general recursion considered in Chapter 15. Recall that the dynamics of general recursion is given by unrolling, which replicates the recursive computation on each use. It would be preferable to share the results of such computation across unrollings. A lazy implementation of recursion avoids such replications by sharing those results.

Traditionally, languages are biased towards either eager or lazy evaluation. Eager languages use a by-value dynamics for function applications, and evaluate the components of data structures when they are created. Lazy languages adopt the opposite strategy, preferring a by-name dynamics for functions, and a lazy dynamics for data structures. The overhead of laziness is mitigated by managing sharing to avoid redundancy. Experience has shown, however, that the distinction is better drawn at the level of types, rather than at the level of a language design. What is important is to have available both lazy and eager types, so that the programmer may choose which to use in a given situation, rather than having the choice forced by the language designer.

Preview

In this chapter we make precise the means by which sharing of computations is achieved in the implementation of laziness. We then isolate these mechanisms into a type of suspended computations whose results are shared across all copies of a given suspension.

## 38.1 By-Need Dynamics for PCF

We begin by considering a lazy variant of **PCF**, called **LPCF**, in which functions are called by name, and the successor operator is evaluated lazily. The lazy dynamics, however, differs from that described in Chapter 15 in that it employs a memoized form of lazy evaluation, *call-by-need*. Under a lazy interpretation variables are bound to unevaluated expressions, and the argument to the successor left unevaluated: any successor is a value, regardless of whether the predecessor is or not. The idea of call-by-need is to *share* copies of the same computation, and to avoid *redundant* computation of results that have already been obtained. By-name function application replicates the unevaluated argument by substitution, which means that there can arise many copies of the same expression, each evaluated separately, if at all. By-need evaluation uses a technique called *memoization* to share all such copies of an argument and to ensure that if it is evaluated at all, its value is stored so that all other uses of it will avoid recomputation. The crucial point is that computations are *named* during evaluation, and are accessed by a level of indirection using this name to index the memo table, which records the expression and, if it is every evaluated, its value.

The dynamics of **LPCF** is based on a transition system with states of the form $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$, where $\Sigma$ is a finite set of hypotheses $a_1 \sim \tau_1, \ldots, a_n \sim \tau_n$ associating types to symbols, $e$ is an expression that may involve the symbols in $\Sigma$, and $\mu$ maps each symbol declared in $\Sigma$ to either an expression or a special symbol, $\bullet$, called the *black hole*. (The role of the black hole will be made clear below.) As a notational convenience, we employ a bit of legerdemain with the concrete syntax similar to that employed in Chapter 36. Specifically, the concrete syntax for the expression $\mathtt{via}(a)$, which fetches the contents of the assignable $a$, is $@\,a$.

The dynamics of **LPCF** is given by he following two forms of judgment:

1. $e\ \mathsf{val}_\Sigma$, stating that $e$ is a value that may involve the symbols in $\Sigma$.
2. $\nu\,\Sigma\,\{\,e \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}$, stating that one step of evaluation of the expression $e$ relative to memo table $\mu$ with the symbols declared in $\Sigma$ results in the expression $e'$ relative to the memo table $\mu'$ with symbols declared in $\Sigma'$.

The dynamics is defined so that the collection of active symbols grows monotonically, and so that the type of a symbol never changes. The memo table may be altered destructively during execution to reflect progress in the evaluation of the expression associated with a given symbol.

The judgment $e\ \mathsf{val}_\Sigma$ expressing that $e$ is a closed value is defined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}_\Sigma} \tag{38.1a}$$

$$\frac{}{\mathtt{s}(@\,a)\ \mathsf{val}_{\Sigma,a\sim\mathtt{nat}}} \tag{38.1b}$$



Preview

$$\frac{}{\lambda\,(x{:}\tau)\,e\ \mathsf{val}_\Sigma} \tag{38.1c}$$

Rules (38.1a) through (38.1c) specify that $\mathtt{z}$ is a value, any expression of the form $\mathtt{s}(@\,a)$, where $a$ is a symbol, is a value, and that any $\lambda$-abstraction, possibly containing symbols, is a value. It is important that symbols themselves are not values, rather they stand for (possibly unevaluated) expressions as specified by the memo table. The expression $@\,a$, which is short for $\mathtt{via}(a)$, is *not* a value. Rather, it must be accessed to determine, and possibly update, the binding of the symbol $a$ in memory.

The initial and final states of evaluation are defined as follows:

$$\frac{}{\nu\,\varnothing\,\{\,e \parallel \varnothing\,\}\ \mathsf{initial}} \tag{38.2a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{final}} \tag{38.2b}$$

Rule (38.2a) specifies that an initial state consists of an expression evaluated relative to an empty memo table. Rule (38.2b) specifies that a final state has the form $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$, where $e$ is a value relative to $\Sigma$.

The transition judgment for the dynamics of **LPCF** is defined by the following rules:

$$\frac{e\ \mathsf{val}_{\Sigma,a\sim\tau}}{\nu\,\Sigma, a\sim\tau\,\{\,@\,a \parallel \mu\otimes a\hookrightarrow e\,\} \longmapsto \nu\,\Sigma, a\sim\tau\,\{\,e \parallel \mu\otimes a\hookrightarrow e\,\}} \tag{38.3a}$$

$$\frac{\nu\,\Sigma, a\sim\tau\,\{\,e \parallel \mu\otimes a\hookrightarrow \bullet\,\} \longmapsto \nu\,\Sigma', a\sim\tau\,\{\,e' \parallel \mu'\otimes a\hookrightarrow \bullet\,\}}{\nu\,\Sigma, a\sim\tau\,\{\,@\,a \parallel \mu\otimes a\hookrightarrow e\,\} \longmapsto \nu\,\Sigma', a\sim\tau\,\{\,@\,a \parallel \mu'\otimes a\hookrightarrow e'\,\}} \tag{38.3b}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{s}(e) \parallel \mu\,\} \longmapsto \nu\,\Sigma, a\sim\mathtt{nat}\,\{\,\mathtt{s}(@\,a) \parallel \mu\otimes a\hookrightarrow e\,\}} \tag{38.3c}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{ifz}\,e\,\{\mathtt{z}\hookrightarrow e_0 \mid \mathtt{s}(x)\hookrightarrow e_1\}\parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,\mathtt{ifz}\,e'\,\{\mathtt{z}\hookrightarrow e_0 \mid \mathtt{s}(x)\hookrightarrow e_1\}\parallel \mu'\,\}} \tag{38.3d}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{ifz}\,\mathtt{z}\,\{\mathtt{z}\hookrightarrow e_0 \mid \mathtt{s}(x)\hookrightarrow e_1\}\parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,e_0 \parallel \mu\,\}} \tag{38.3e}$$

$$\frac{}{\left\{\begin{array}{c}\nu\,\Sigma, a\sim\mathtt{nat}\,\{\,\mathtt{ifz}\,\mathtt{s}(@\,a)\,\{\mathtt{z}\hookrightarrow e_0 \mid \mathtt{s}(x)\hookrightarrow e_1\}\parallel \mu\otimes a\hookrightarrow e\,\}\\ \longmapsto\\ \nu\,\Sigma, a\sim\mathtt{nat}\,\{\,[@\,a/x]e_1 \parallel \mu\otimes a\hookrightarrow e\,\}\end{array}\right\}} \tag{38.3f}$$

Preview

$$\frac{\nu\,\Sigma\,\{\,e_1 \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,e_1' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,e_1(e_2) \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,e_1'(e_2) \parallel \mu'\,\}} \tag{38.3g}$$

$$\left\{\begin{array}{c} \nu\,\Sigma\,\{\,\lambda\,(x{:}\tau)\,e(e_2) \parallel \mu\,\} \\ \longmapsto \\ \nu\,\Sigma, a \sim \tau\,\{\,[@\,a/x]e \parallel \mu \otimes a \hookrightarrow e_2\,\} \end{array}\right\} \tag{38.3h}$$

$$\nu\,\Sigma\,\{\,\mathtt{fix}\,x:\tau\,\mathtt{is}\,e \parallel \mu\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,@\,a \parallel \mu \otimes a \hookrightarrow [@\,a/x]e\,\} \tag{38.3i}$$

Rule (38.3a) governs a symbol whose associated expression is a value; the value of the symbol is the value associated to that symbol in the memo table. Rule (38.3b) specifies that if the expression associated to a symbol is not a value, then it is evaluated "in place" until such time as rule (38.3a) applies. This is achieved by switching the focus of evaluation to the associated expression, while at the same time associating the *black hole* to that symbol. The black hole represents the absence of a value for that symbol, so that any attempt to access it during evaluation of its associated expression cannot make progress. This signals a circular dependency that, if not caught using a black hole, would initiate an infinite regress. We may therefore think of the black hole as catching a particular form of non-termination that arises when the value of an expression associated to a symbol depends on the symbol itself.

Rule (38.3c) specifies that evaluation of $\mathtt{s}(e)$ allocates a fresh symbol, $a$, for the expression $e$, and yields the value $\mathtt{s}(@\,a)$. The value of $e$ is not determined until such time as the predecessor is required in a subsequent computation. This implements a lazy dynamics for the successor. Rule (38.3f), which governs a conditional branch on a successor, substitutes $@\,a$ for the variable $x$ when computing the predecessor of a non-zero number, ensuring that all occurrences of $x$ share the same predecessor computation.

Rule (38.3g) specifies that the value of the function position of an application must be determined before the application can be executed. Rule (38.3h) specifies that to evaluate an application of a $\lambda$-abstraction we allocate a fresh symbol, $a$, for the argument, and substitute $@\,a$ for the parameter of the function. The argument is evaluated only if it is needed in the subsequent computation, and then that value is shared among all occurrences of the parameter in the body of the function.

General recursion is implemented by rule (38.3i). Recall from Chapter 15 that the expression $\mathtt{fix}\,x:\tau\,\mathtt{is}\,e$ stands for the solution of the recursion equation $x = e$. Rule (38.3i) computes this solution by associating a fresh symbol, $a$, with the body, $e$, substituting $@\,a$ for $x$ within $e$ to effect the self-reference. It is this substitution that permits a named expression to depend on its own name. For example, the expression $\mathtt{fix}\,x:\tau\,\mathtt{is}\,x$ associates the expression $a$ to $a$ in the memo table, and returns $@\,a$. The next step of evaluation is stuck, because it seeks to evaluate $@\,a$ with $a$ bound to the black hole. In contrast an expression such as $\mathtt{fix}\,f:\tau' \rightharpoonup \tau\,\mathtt{is}\,\lambda\,(x{:}\tau')\,e$ does not get stuck, because the self-reference is "hidden" within the $\lambda$-abstraction, and hence need not be evaluated to determine the value of the binding.

Preview

## 38.2 Safety for By-Need PCF

We write $\Gamma \vdash_\Sigma e : \tau$ to mean that $e$ has type $\tau$ under the assumptions $\Gamma$, treating symbols declared in $\Sigma$ as expressions of their associated type. The rules are as in Chapter 15, with the addition of the following rule for accessing symbols:

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} @a : \tau} \tag{38.4}$$

This rule states that the demand for the binding of a symbol, $@a$, is a form of expression. It is a "delayed substitution" that lazily replaces a demand for $a$ by its binding.

The judgment $\nu\, \Sigma\, \{\, e \parallel \mu \,\}\ \mathsf{ok}$ is defined by the following rules:

$$\frac{\vdash_\Sigma e : \tau \quad \vdash_\Sigma \mu : \Sigma}{\nu\, \Sigma\, \{\, e \parallel \mu \,\}\ \mathsf{ok}} \tag{38.5a}$$

$$\frac{\forall a \sim \tau \in \Sigma \quad \mu(a) = e \neq \bullet \Longrightarrow \vdash_{\Sigma'} e : \tau}{\vdash_{\Sigma'} \mu : \Sigma} \tag{38.5b}$$

Rule (38.5b) permits self-reference through the memo table by allowing the expression associated to a symbol, $a$, to contain occurrences of $@a$. A symbol that is bound to the "black hole" is deemed to be of any type.

**Theorem 38.1** (Preservation)**.** *Suppose that* $\nu\, \Sigma\, \{\, e \parallel \mu \,\} \longmapsto \nu\, \Sigma'\, \{\, e' \parallel \mu' \,\}$. *If* $\nu\, \Sigma\, \{\, e \parallel \mu \,\}$ *ok, then* $\nu\, \Sigma'\, \{\, e' \parallel \mu' \,\}$ *ok.*

*Proof.* We prove by induction on rules (38.3) that if $\nu\, \Sigma\, \{\, e \parallel \mu \,\} \longmapsto \nu\, \Sigma'\, \{\, e' \parallel \mu' \,\}$ and $\vdash_\Sigma \mu : \Sigma$ and $\vdash_\Sigma e : \tau$, then $\Sigma' \supseteq \Sigma$ and $\vdash_{\Sigma'} \mu' : \Sigma'$ and $\vdash_{\Sigma'} e' : \tau$.

Consider rule (38.3b), for which we have $e = e' = @a$, $\mu = \mu_0 \otimes a \hookrightarrow e_0$, $\mu' = \mu_0' \otimes a \hookrightarrow e_0'$, and

$$\nu\, \Sigma, a \sim \tau\, \{\, e_0 \parallel \mu_0 \otimes a \hookrightarrow \bullet \,\} \longmapsto \nu\, \Sigma', a \sim \tau\, \{\, e_0' \parallel \mu_0' \otimes a \hookrightarrow \bullet \,\}.$$

Assume that $\vdash_{\Sigma, a \sim \tau} \mu : \Sigma, a \sim \tau$. It follows that $\vdash_{\Sigma, a \sim \tau} e_0 : \tau$ and $\vdash_{\Sigma, a \sim \tau} \mu_0 : \Sigma$, and hence that

$$\vdash_{\Sigma, a \sim \tau} \mu_0 \otimes a \hookrightarrow \bullet : \Sigma, a \sim \tau.$$

We have by induction that $\Sigma' \supseteq \Sigma$ and $\vdash_{\Sigma', a \sim \tau} e_0' : \tau'$ and

$$\vdash_{\Sigma', a \sim \tau} \mu_0 \otimes a \hookrightarrow \bullet : \Sigma, a \sim \tau.$$

But then

$$\vdash_{\Sigma', a \sim \tau} \mu' : \Sigma', a \sim \tau,$$

which suffices for the result.

Consider rule (38.3g), so that $e$ is the application $e_1(e_2)$ and

$$\nu\, \Sigma\, \{\, e_1 \parallel \mu \,\} \longmapsto \nu\, \Sigma'\, \{\, e_1' \parallel \mu' \,\}.$$

Suppose that $\vdash_\Sigma \mu : \Sigma$ and $\vdash_\Sigma e : \tau$. By inversion of typing $\vdash_\Sigma e_1 : \tau_2 \rightharpoonup \tau$ for some type $\tau_2$ such that $\vdash_\Sigma e_2 : \tau_2$. By induction $\Sigma' \supseteq \Sigma$ and $\vdash_{\Sigma'} \mu' : \Sigma'$ and $\vdash_{\Sigma'} e_1' : \tau_2 \rightharpoonup \tau$. By weakening we have $\vdash_{\Sigma'} e_2 : \tau_2$, so that $\vdash_{\Sigma'} e_1'(e_2) : \tau$, which is enough for the result.

□

Preview

The statement of the progress theorem allows for the possibility of encountering a black hole, representing a checkable form of non-termination. The judgment $\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{loops}$, stating that $e$ diverges by virtue of encountering the black hole, is defined by the following rules:

$$\frac{}{\nu\,\Sigma, a \sim \tau\,\{\,@a \parallel \mu \otimes a \hookrightarrow \bullet\,\}\ \mathsf{loops}} \tag{38.6a}$$

$$\frac{\nu\,\Sigma, a \sim \tau\,\{\,e \parallel \mu \otimes a \hookrightarrow \bullet\,\}\ \mathsf{loops}}{\nu\,\Sigma, a \sim \tau\,\{\,@a \parallel \mu \otimes a \hookrightarrow e\,\}\ \mathsf{loops}} \tag{38.6b}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{loops}}{\nu\,\Sigma\,\{\,\mathtt{ifz}\,e\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}\parallel \mu\,\}\ \mathsf{loops}} \tag{38.6c}$$

$$\frac{\nu\,\Sigma\,\{\,e_1 \parallel \mu\,\}\ \mathsf{loops}}{\nu\,\Sigma\,\{\,e_1(e_2) \parallel \mu\,\}\ \mathsf{loops}} \tag{38.6d}$$

To be sure, there are other ways of forming an infinite loop. The looping judgment simply codifies those cases in which the looping behavior is a self-dependency, which is mediated by a black hole.

**Theorem 38.2** (Progress)**.** *If $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$ ok, then either $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$ final, or $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$ loops, or there exists $\mu'$ and $e'$ such that $\nu\,\Sigma\,\{\,e \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}$.*

*Proof.* We proceed by induction on the derivations of $\vdash_\Sigma e : \tau$ and $\vdash_\Sigma \mu : \Sigma$ implicit in the derivation of $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$ ok.

Consider rule (15.1a), where the symbol, $a$, is declared in $\Sigma$. Thus $\Sigma = \Sigma_0, a \sim \tau$ and $\vdash_\Sigma \mu : \Sigma$. It follows that $\mu = \mu_0 \otimes a \hookrightarrow e_0$ with $\vdash_\Sigma \mu_0 : \Sigma_0$ and $\vdash_\Sigma e_0 : \tau$. Note that $\vdash_\Sigma \mu_0 \otimes a \hookrightarrow \bullet : \Sigma$. Applying induction to the derivation of $\vdash_\Sigma e_0 : \tau$, we consider three cases:

1. $\nu\,\Sigma\,\{\,e_0 \parallel \mu \otimes a \hookrightarrow \bullet\,\}$ final. By inversion of rule (38.2b) we have $e_0\ \mathsf{val}_\Sigma$, and hence by rule (38.3a) we obtain $\nu\,\Sigma\,\{\,@a \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,e_0 \parallel \mu\,\}$.
2. $\nu\,\Sigma\,\{\,e_0 \parallel \mu_0 \otimes a \hookrightarrow \bullet\,\}\ \mathsf{loops}$. By applying rule (38.6b) we obtain $\nu\,\Sigma\,\{\,@a \parallel \mu\,\}\ \mathsf{loops}$.
3. $\nu\,\Sigma\,\{\,e_0 \parallel \mu_0 \otimes a \hookrightarrow \bullet\,\} \longmapsto \nu\,\Sigma'\,\{\,e_0' \parallel \mu_0' \otimes a \hookrightarrow \bullet\,\}$. By applying rule (38.3b) we obtain
$$\nu\,\Sigma\,\{\,@a \parallel \mu \otimes a \hookrightarrow e_0\,\} \longmapsto \nu\,\Sigma'\,\{\,@a \parallel \mu' \otimes a \hookrightarrow e_0'\,\}.$$

□

## 38.3 By-Need Dynamics for FPC

By analogy with the by-need dynamics of **LPCF** we may also define **LFPC** to be **FPC** with a by-need dynamics. For example, the dynamics of product types in **LFPC** is given by the following rules:

$$\frac{}{\langle @a_1, @a_2\rangle\ \mathsf{val}_{\Sigma, a_1 \sim \tau_1, a_2 \sim \tau_2}} \tag{38.7a}$$



Preview

$$\left\{\begin{array}{c} \nu\,\Sigma\,\{\,\langle e_1, e_2\rangle \parallel \mu\,\} \\ \longmapsto \\ \nu\,\Sigma, a_1 \sim \tau_1, a_2 \sim \tau_2\,\{\,\langle @\,a_1, @\,a_2\rangle \parallel \mu \otimes a_1 \hookrightarrow e_1 \otimes a_2 \hookrightarrow e_2\,\} \end{array}\right\} \tag{38.7b}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,e \cdot \mathtt{l} \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,e' \cdot \mathtt{l} \parallel \mu'\,\}} \tag{38.7c}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{loops}}{\nu\,\Sigma\,\{\,e \cdot \mathtt{l} \parallel \mu\,\}\ \mathsf{loops}} \tag{38.7d}$$

$$\left\{\begin{array}{c} \nu\,\Sigma, a_1 \sim \tau_1, a_2 \sim \tau_2\,\{\,\langle @\,a_1, @\,a_2\rangle \cdot \mathtt{l} \parallel \mu\,\} \\ \longmapsto \\ \nu\,\Sigma, a_1 \sim \tau_1, a_2 \sim \tau_2\,\{\,@\,a_1 \parallel \mu\,\} \end{array}\right\} \tag{38.7e}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,e \cdot \mathtt{r} \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,e' \cdot \mathtt{r} \parallel \mu'\,\}} \tag{38.7f}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{loops}}{\nu\,\Sigma\,\{\,e \cdot \mathtt{r} \parallel \mu\,\}\ \mathsf{loops}} \tag{38.7g}$$

$$\left\{\begin{array}{c} \nu\,\Sigma, a_1 \sim \tau_1, a_2 \sim \tau_2\,\{\,\langle @\,a_1, @\,a_2\rangle \cdot \mathtt{r} \parallel \mu\,\} \\ \longmapsto \\ \nu\,\Sigma, a_1 \sim \tau_1, a_2 \sim \tau_2\,\{\,@\,a_2 \parallel \mu\,\} \end{array}\right\} \tag{38.7h}$$

A pair is considered a value only if its arguments are symbols (rule (38.7a)), which are introduced when the pair is created (rule (38.7b)). The first and second projections evaluate to one or the other symbol in the pair, inducing a demand for the value of that component (rules (38.7e) and (38.7h)).

Similar techniques may be used to give a by-need dynamics to sums and recursive types.

## 38.4 Suspensions

The dynamics of **LFPC** outlined in the previous section imposes a by-need interpretation on every type. A more flexible approach is to isolate the machinery of by-need evaluation by introducing a type $\tau\ \mathtt{susp}$ of memoized computations, called *suspensions*, of a value of type $\tau$ to an eager variant of **FPC**. Doing so allows the programmer to choose the extent to which a by-need dynamics is imposed.

Informally, the type $\tau\ \mathtt{susp}$ has as introductory form $\mathtt{susp}\,x : \tau\,\mathtt{is}\,e$ representing a suspended, self-referential, computation, $e$, of type $\tau$. It has as eliminatory form the operation $\mathtt{force}(e)$ that evaluates the suspended computation presented by $e$, records the value in a memo table, and returns that value as result. Using suspension types we may construct (partially or wholly) lazy

Preview

types at will. For example, the type of lazy pairs with components of type $\tau_1$ and $\tau_2$ is expressible as the type

$$\tau_1 \,\mathtt{susp} \times \tau_2 \,\mathtt{susp}$$

and the type of by-need functions with domain $\tau_1$ and range $\tau_2$ is expressible as the type

$$\tau_1 \,\mathtt{susp} \rightharpoonup \tau_2.$$

We may also express more complex combinations of eagerness and laziness, such as the type of "lazy lists" consisting of computations that, when forced, evaluate either to the empty list, or a non-empty list consisting of a natural number and another lazy list:

$$\mathtt{rec}\, t \,\mathtt{is}\, (\mathtt{unit} + (\mathtt{nat} \times t)) \,\mathtt{susp}.$$

This type should be contrasted with the type

$$\mathtt{rec}\, t \,\mathtt{is}\, (\mathtt{unit} + (\mathtt{nat} \times t \,\mathtt{susp}))$$

whose values are the empty list and a pair consisting of a natural number and a computation of another such value.

The language **SFPC** is an extension of **FPC** with a type of suspensions:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{susp}(\tau)$ | $\tau \,\mathtt{susp}$ | suspension |
| Exp | $e$ | $::=$ | $\mathtt{susp}[\tau](x.e)$ | $\mathtt{susp}\, x : \tau \,\mathtt{is}\, e$ | delay |
| | | | $\mathtt{force}(e)$ | $\downarrow e$ | force |
| | | | $\mathtt{susp}[a]$ | $\uparrow a$ | indirection |

Suspended computations may be self-referential; the bound variable, $x$, refers to the suspension itself. The expression $\mathtt{susp}[a]$ is a reference to the suspension named $a$.

The statics of **SFPC** is given using a judgment of the form $\Gamma \vdash_\Sigma e : \tau$, where $\Sigma$ assigns types to the names of suspensions. It is defined by the following rules:

$$\frac{\Gamma, x : \mathtt{susp}(\tau) \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathtt{susp}[\tau](x.e) : \mathtt{susp}(\tau)} \tag{38.8a}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{susp}(\tau)}{\Gamma \vdash_\Sigma \mathtt{force}(e) : \tau} \tag{38.8b}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{susp}[a] : \mathtt{susp}(\tau)} \tag{38.8c}$$

Rule (38.8a) checks that the expression, $e$, has type $\tau$ under the assumption that $x$, which stands for the suspension itself, has type $\mathtt{susp}(\tau)$.

The dynamics of **SFPC** is eager, with memoization confined to the suspension type as described by the following rules:

$$\frac{}{\mathtt{susp}[a] \;\mathsf{val}_{\Sigma, a \sim \tau}} \tag{38.9a}$$



Preview

$$\left\{\begin{array}{c} \nu\,\Sigma\,\{\,\mathtt{susp}[\tau](x.e) \parallel \mu\,\} \\ \longmapsto \\ \nu\,\Sigma, a \sim \tau\,\{\,\mathtt{susp}[a] \parallel \mu \otimes a \hookrightarrow [\mathtt{susp}[a]/x]e\,\} \end{array}\right\} \tag{38.9b}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{force}(e) \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,\mathtt{force}(e') \parallel \mu'\,\}} \tag{38.9c}$$

$$\frac{e\ \mathsf{val}_{\Sigma, a\sim\tau}}{\left\{\begin{array}{c} \nu\,\Sigma, a \sim \tau\,\{\,\mathtt{force}(\mathtt{susp}[a]) \parallel \mu \otimes a \hookrightarrow e\,\} \\ \longmapsto \\ \nu\,\Sigma, a \sim \tau\,\{\,e \parallel \mu \otimes a \hookrightarrow e\,\} \end{array}\right\}} \tag{38.9d}$$

$$\frac{\begin{array}{c} \nu\,\Sigma, a \sim \tau\,\{\,e \parallel \mu \otimes a \hookrightarrow \bullet\,\} \\ \longmapsto \\ \nu\,\Sigma', a \sim \tau\,\{\,e' \parallel \mu' \otimes a \hookrightarrow \bullet\,\} \end{array}}{\left\{\begin{array}{c} \nu\,\Sigma, a \sim \tau\,\{\,\mathtt{force}(\mathtt{susp}[a]) \parallel \mu \otimes a \hookrightarrow e\,\} \\ \longmapsto \\ \nu\,\Sigma', a \sim \tau\,\{\,\mathtt{force}(\mathtt{susp}[a]) \parallel \mu' \otimes a \hookrightarrow e'\,\} \end{array}\right\}} \tag{38.9e}$$

Rule (38.9a) specifies that a reference to a suspension is a value. Rule (38.9b) specifies that evaluation of a delayed computation consists of allocating a fresh symbol for it in the memo table, and returning a reference to that suspension. Rules (38.9c) to (38.9e) specify that demanding the value of a suspension forces evaluation of the suspended computation, which is then stored in the memo table and returned as the result.

## 38.5 Notes

The by-need dynamics given here is inspired by Ariola and Felleisen (1997), but with the crucial distinction that by-need cells are regarded as assignables, rather than variables. This is in keeping with the principle that a variable is given meaning by substitution. The goal of by-need evaluation is to limit substitution in the interest of avoiding redundant computations. As such it cannot properly be modeled using variables, but rather requires a form of assignable (introduced in Chapter 36) to which at most one assignment is ever performed.

## Exercises

**38.1**. Recall from Chapter 16 that, under a lazy interpretation, the recursive type

$$\mathtt{rec}\,t\,\mathtt{is}\,[\mathtt{z} \hookrightarrow \mathtt{unit}, \mathtt{s} \hookrightarrow t]$$



Preview

contains the "infinite number" $\omega \triangleq \mathtt{fix}\, x : \mathtt{nat}\, \mathtt{is}\, \mathtt{s}(x)$. Contrast the behavior of $\omega$ under the by-need interpretation given in this chapter with that by-name interpretation given in Chapters 15 and 16.

**38.2**. In **LFPC** the putative recursive type of "lists" of natural numbers,

$$\mathtt{rec}\, t\, \mathtt{is}\, [\mathtt{nil} \hookrightarrow \mathtt{unit}, \mathtt{cons} \hookrightarrow \mathtt{nat} \times t],$$

is, rather, the type of finite or infinite streams of natural numbers. To prove this, exhibit the stream of all natural numbers as an element of this type.

**38.3**. Complete the definition of **LFPC** by giving the by-need dynamics for unit, void, sum, and recursive types.

**38.4**. **LFPC** may be interpreted into **SFPC**, enriched with suspensions, by making explicit the use of laziness. Complete the following chart defining the interpretation, $\widehat{\tau}$, of the type $\tau$:

$$\begin{aligned} \mathtt{unit} &\triangleq \ldots \\ \tau_1 \times \tau_2 &\triangleq \ldots \\ \mathtt{void} &\triangleq \ldots \\ \tau_1 + \tau_2 &\triangleq \ldots \\ \mathtt{rec}\, t\, \mathtt{is}\, \tau &\triangleq \ldots. \end{aligned}$$

*Hint*: Characterize the *values* of the lazy types in the left column, and express those values as eager types in the right column, using suspensions where appropriate.

Preview

# Chapter 39

# Polarization

Up to this point we have frequently encountered arbitrary choices in the dynamics of various language constructs. For example, when specifying the dynamics of pairs, we must choose, rather arbitrarily, between the *lazy* dynamics, in which all pairs are values regardless of the value status of their components, and the *eager* dynamics, in which a pair is a value only if its components are both values. We could even consider a *half-eager* (or, equivalently, *half-lazy*) dynamics, in which a pair is a value only if, say, the first component is a value, but without regard to the second.

Similar questions arise with sums (all injections are values, or only injections of values are values), recursive types (all folds are values, or only folds of values are values), and function types (functions should be called by-name or by-value). Whole languages are built around adherence to one policy or another. For example, Haskell decrees that products, sums, and recursive types are to be lazy, and functions are to be called by name, whereas ML decrees the exact opposite policy. Not only are these choices arbitrary, but it is also unclear why they should be linked. For example, we could very sensibly decree that products, sums, and recursive types are lazy, yet impose a call-by-value discipline on functions. Or we could have eager products, sums, and recursive types, yet insist on call-by-name. It is not at all clear which of these points in the space of choices is right; each has its adherents, and each has its detractors.

Are we therefore stuck in a tarpit of subjectivity? No! The way out is to recognize that these distinctions should not be imposed by the language designer, but rather are choices that are to be made by the programmer. This may be achieved by recognizing that differences in dynamics reflect fundamental *type distinctions* that are being obscured by languages that impose one policy or another. We can have both eager and lazy pairs in the same language by simply distinguishing them as two distinct types, and similarly we can have both eager and lazy sums in the same language, and both by-name and by-value function spaces, by providing sufficient type distinctions as to make the choice available to the programmer.

Eager and lazy types are distinguished by their *polarity*, which is either *positive* or *negative* according to whether the type is defined by the values that inhabit the type or the behavior of expressions of that type. Positive types are *eager*, or *inductive*, in that they are defined by their values. Negative types are *lazy*, or *coinductive*, in that they are defined by the behavior of their elements. The polarity of types is made explicit using a technique called *focusing*. A focused

Preview

presentation of a programming language distinguishes three general forms of expression, *(positive and negative) values*, *(positive and negative) continuations*, and *(neutral) computations*.

## 39.1 Positive and Negative Types

Polarization consists of distinguishing positive from negative types according to the following two principles:

1. A positive type is defined by its introduction rules, which specify the *values* of that type in terms of other values. The elimination rules are *inversions* that specify a computation by pattern matching on values of that type.

2. A negative type is defined by its elimination rules, which specify the *observations* that may be performed on elements of that type. The introduction rules specify the *values* of that type by specifying how they respond to observations.

Based on this characterization we can anticipate that the type of natural numbers would be positive, because it is defined by zero and successor, whereas function types would be negative, because they are characterized by their behavior when applied, and not by their internal structure.

The language $\mathbf{PCF}^{\pm}$ is a polarized formulation of **PCF**. The syntax of types in this language is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| PTyp | $\tau^+$ | $::=$ | $\mathtt{dn}(\tau^-)$ | $\downarrow\tau^-$ | suspension |
| | | | $\mathtt{nat}$ | $\mathtt{nat}$ | naturals |
| NTyp | $\tau^-$ | $::=$ | $\mathtt{up}(\tau^+)$ | $\uparrow\tau^+$ | inclusion |
| | | | $\mathtt{parr}(\tau_1^+;\tau_2^-)$ | $\tau_1^+ \rightharpoonup \tau_2^-$ | partial function |

The types $\downarrow\tau^-$ and $\uparrow\tau^+$ effect a *polarity shift* from negative to positive and positive to negative, respectively. Intuitively, the shifted type $\uparrow\tau^+$ is just the inclusion of positive into negative values, whereas the shifted type $\downarrow\tau^-$ represents the type of suspended computations of negative type.

The domain of the negative function type is required to be positive, but its range is negative. This allows us to form right-iterated function types

$$\tau_1^+ \rightharpoonup (\tau_2^+ \rightharpoonup (\ldots (\tau_{n-1}^+ \rightharpoonup \tau_n^-)))$$

directly, but to form a left-iterated function type requires shifting,

$$\downarrow(\tau_1^+ \rightharpoonup \tau_2^-) \rightharpoonup \tau^-,$$

to turn the negative function type into a positive type. Conversely, shifting is needed to define a function whose range is positive, $\tau_1^+ \rightharpoonup \uparrow\tau_2^+$.



Preview

## 39.2 Focusing

The syntax of $\mathbf{PCF}^{\pm}$ is motivated by the polarization of its types. For each polarity we have a sort of values and a sort of continuations with which we may create (neutral) computations.

| | | | | | |
|---|---|---|---|---|---|
| PVal | $v^+$ | ::= | $\mathtt{z}$ | $\mathtt{z}$ | zero |
| | | | $\mathtt{s}(v^+)$ | $\mathtt{s}(v^+)$ | successor |
| | | | $\mathtt{del}^-(e)$ | $\mathtt{del}^-(e)$ | delay |
| PCont | $k^+$ | ::= | $\mathtt{ifz}(e_0; x.e_1)$ | $\mathtt{ifz}(e_0; x.e_1)$ | conditional |
| | | | $\mathtt{force}^-(k^-)$ | $\mathtt{force}^-(k^-)$ | evaluate |
| NVal | $v^-$ | ::= | $\mathtt{lam}[\tau^+](x.e)$ | $\lambda\,(x{:}\tau^+)\,e$ | abstraction |
| | | | $\mathtt{del}^+(v^+)$ | $\mathtt{del}^+(v^+)$ | inclusion |
| | | | $\mathtt{fix}(x.v^-)$ | $\mathtt{fix}\,x\,\mathtt{is}\,v^-$ | recursion |
| NCont | $k^-$ | ::= | $\mathtt{ap}(v^+; k^-)$ | $\mathtt{ap}(v^+; k^-)$ | application |
| | | | $\mathtt{force}^+(x.e)$ | $\mathtt{force}^+(x.e)$ | evaluate |
| Comp | $e$ | ::= | $\mathtt{ret}(v^-)$ | $\mathtt{ret}(v^-)$ | return |
| | | | $\mathtt{cut}^+(v^+; k^+)$ | $v^+ \triangleright k^+$ | positive cut |
| | | | $\mathtt{cut}^-(v^-; k^-)$ | $v^- \triangleright k^-$ | negative cut |

The positive values include the numerals, and the negative values include functions. In addition we may delay a computation of a negative value to form a positive value using $\mathtt{del}^-(e)$, and we may consider a positive value to be a negative value using $\mathtt{del}^+(v^+)$. The positive continuations include the conditional branch, sans argument, and the negative continuations include application sites for functions consisting of a positive argument value and a continuation for the negative result. In addition we include positive continuations to force the computation of a suspended negative value, and to extract an included positive value. Computations, which correspond to machine states, consist of returned negative values (these are final states), states passing a positive value to a positive continuation, and states passing a negative value to a negative continuation. General recursion appears as a form of negative value; the recursion is unrolled when it is made the subject of an observation.

## 39.3 Statics

The statics of $\mathbf{PCF}^{\pm}$ consists of a collection of rules for deriving judgments of the following forms:

- Positive values: $\Gamma \vdash v^+ : \tau^+$.
- Positive continuations: $\Gamma \vdash k^+ : \tau^+ > \gamma^-$.
- Negative values: $\Gamma \vdash v^- : \tau^-$.
- Negative continuations: $\Gamma \vdash k^- : \tau^- > \gamma^-$.
- Computations: $\Gamma \vdash e : \gamma^-$.

Preview

Throughout $\Gamma$ is a finite set of hypotheses of the form

$$x_1 : \tau_1^+, \ldots, x_n : \tau_n^+,$$

for some $n \geq 0$, and $\gamma^-$ is any negative type.

The typing rules for continuations specify both an argument type (on which values they act) and a result type (of the computation resulting from the action on a value). The typing rules for computations specify that the outcome of a computation is a negative type. All typing judgments specify that variables range over positive types. (These restrictions may always be met by appropriate use of shifting.)

The statics of positive values consists of the following rules:

$$\frac{}{\Gamma, x : \tau^+ \vdash x : \tau^+} \tag{39.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{39.1b}$$

$$\frac{\Gamma \vdash v^+ : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(v^+) : \mathtt{nat}} \tag{39.1c}$$

$$\frac{\Gamma \vdash e : \tau^-}{\Gamma \vdash \mathtt{del}^-(e) : \downarrow \tau^-} \tag{39.1d}$$

Rule (39.1a) specifies that variables range over positive values. Rules (39.1b) and (39.1c) specify that the values of type `nat` are just the numerals. Rule (39.1d) specifies that a suspended computation (necessarily of negative type) is a positive value.

The statics of positive continuations consists of the following rules:

$$\frac{\Gamma \vdash e_0 : \gamma^- \quad \Gamma, x : \mathtt{nat} \vdash e_1 : \gamma^-}{\Gamma \vdash \mathtt{ifz}(e_0; x.e_1) : \mathtt{nat} > \gamma^-} \tag{39.2a}$$

$$\frac{\Gamma \vdash k^- : \tau^- > \gamma^-}{\Gamma \vdash \mathtt{force}^-(k^-) : \downarrow \tau^- > \gamma^-} \tag{39.2b}$$

Rule (39.2a) governs the continuation that chooses between two computations according to whether a natural number is zero or non-zero. Rule (39.2b) specifies the continuation that forces a delayed computation with the specified negative continuation.

The statics of negative values is defined by these rules:

$$\frac{\Gamma, x : \tau_1^+ \vdash e : \tau_2^-}{\Gamma \vdash \lambda\,(x{:}\tau_1^+)\, e : \tau_1^+ \rightharpoonup \tau_2^-} \tag{39.3a}$$

$$\frac{\Gamma \vdash v^+ : \tau^+}{\Gamma \vdash \mathtt{del}^+(v^+) : \uparrow \tau^+} \tag{39.3b}$$

$$\frac{\Gamma, x : \downarrow \tau^- \vdash v^- : \tau^-}{\Gamma \vdash \mathtt{fix}\, x\, \mathtt{is}\, v^- : \tau^-} \tag{39.3c}$$



Preview

Rule (39.3a) specifies the statics of a $\lambda$-abstraction whose argument is a positive value, and whose result is a computation of negative type. Rule (39.3b) specifies the inclusion of positive values as negative values. Rule (39.3c) specifies that negative types admit general recursion.

The statics of negative continuations is defined by these rules:

$$\frac{\Gamma \vdash v_1^+ : \tau_1^+ \quad \Gamma \vdash k_2^- : \tau_2^- > \gamma^-}{\Gamma \vdash \mathtt{ap}(v_1^+; k_2^-) : \tau_1^+ \rightharpoonup \tau_2^- > \gamma^-} \tag{39.4a}$$

$$\frac{\Gamma, x : \tau^+ \vdash e : \gamma^-}{\Gamma \vdash \mathtt{force}^{+}(x.e) : \uparrow \tau^+ > \gamma^-} \tag{39.4b}$$

Rule (39.4a) is the continuation representing the application of a function to the positive argument, $v_1^+$, and executing the body with negative continuation, $k_2^-$. Rule (39.4b) specifies the continuation that passes a positive value, viewed as a negative value, to a computation.

The statics of computations is given by these rules:

$$\frac{\Gamma \vdash v^- : \tau^-}{\Gamma \vdash \mathtt{ret}(v^-) : \tau^-} \tag{39.5a}$$

$$\frac{\Gamma \vdash v^+ : \tau^+ \quad \Gamma \vdash k^+ : \tau^+ > \gamma^-}{\Gamma \vdash v^+ \triangleright k^+ : \gamma^-} \tag{39.5b}$$

$$\frac{\Gamma \vdash v^- : \tau^- \quad \Gamma \vdash k^- : \tau^- > \gamma^-}{\Gamma \vdash v^- \triangleright k^- : \gamma^-} \tag{39.5c}$$

Rule (39.5a) specifies the basic form of computation that simply returns the negative value $v^-$. Rules (39.5b) and (39.5c) specify computations that pass a value to a continuation of appropriate polarity.

## 39.4 Dynamics

The dynamics of **PCF**$^{\pm}$ is given by a transition system $e \longmapsto e'$ specifying the steps of computation. The rules are all axioms; no premises are required because the continuation is used to manage pending computations.

The dynamics consists of the following rules:

$$\overline{\mathtt{z} \triangleright \mathtt{ifz}(e_0; x.e_1) \longmapsto e_0} \tag{39.6a}$$

$$\overline{\mathtt{s}(v^+) \triangleright \mathtt{ifz}(e_0; x.e_1) \longmapsto [v^+/x]e_1} \tag{39.6b}$$

$$\overline{\mathtt{del}^{-}(e) \triangleright \mathtt{force}^{-}(k^-) \longmapsto e\,;k^-} \tag{39.6c}$$

$$\overline{\lambda\,(x{:}\tau^+)\,e \triangleright \mathtt{ap}(v^+; k^-) \longmapsto [v^+/x]e\,;k^-} \tag{39.6d}$$

$$\overline{\mathtt{del}^{+}(v^+) \triangleright \mathtt{force}^{+}(x.e) \longmapsto [v^+/x]e} \tag{39.6e}$$



Preview

$$\frac{}{\mathtt{fix}\,x\,\mathtt{is}\,v^- \triangleright k^- \longmapsto [\mathtt{del}^-(\mathtt{fix}\,x\,\mathtt{is}\,v^-)/x]v^- \triangleright k^-} \tag{39.6f}$$

These rules specify the interaction between values and continuations.

Rules (39.6) make use of two forms of substitution, $[v^+/x]e$ and $[v^+/x]v^-$, which are defined as in Chapter 1. They also employ a new form of *composition*, written $e\,;k_0^-$, which composes a computation with a continuation by attaching $k_0^-$ to the end of the computation specified by $e$. This composition is defined mutually recursive with the compositions $k^+\,;k_0^-$ and $k^-\,;k_0^-$, which essentially concatenate continuations (stacks).

$$\frac{}{\mathtt{ret}(v^-)\,;k_0^- = v^- \triangleright k_0^-} \tag{39.7a}$$

$$\frac{k^-\,;k_0^- = k_1^-}{(v^- \triangleright k^-)\,;k_0^- = v^- \triangleright k_1^-} \tag{39.7b}$$

$$\frac{k^+\,;k_0^- = k_1^+}{(v^+ \triangleright k^+)\,;k_0^- = v^+ \triangleright k_1^+} \tag{39.7c}$$

$$\frac{e_0\,;k^- = e_0' \quad x \mid e_1\,;k^- = e_1'}{\mathtt{ifz}(e_0; x.e_1)\,;k^- = \mathtt{ifz}(e_0'; x.e_1')} \tag{39.7d}$$

$$\frac{k^-\,;k_0^- = k_1^-}{\mathtt{force}^-(k^-)\,;k_0^- = \mathtt{force}^-(k_1^-)} \tag{39.7e}$$

$$\frac{k^-\,;k_0^- = k_1}{\mathtt{ap}(v^+;k^-)\,;k_0^- = \mathtt{ap}(v^+;k_1^-)} \tag{39.7f}$$

$$\frac{x \mid e\,;k_0^- = e'}{\mathtt{force}^+(x.e)\,;k_0^- = \mathtt{force}^+(x.e')} \tag{39.7g}$$

Rules (39.7d) and (39.7g) make use of the generic hypothetical judgment defined in Chapter 3 to express that the composition is defined uniformly in the bound variable.

## 39.5 Safety

The proof of preservation for $\mathbf{PCF}^{\pm}$ reduces to the proof of the typing properties of substitution and composition.

**Lemma 39.1** (Substitution)**.** *Suppose that* $\Gamma \vdash v^+ : \rho^+$.

1. *If* $\Gamma, x : \rho^+ \vdash e : \gamma^-$, *then* $\Gamma \vdash [v^+/x]e : \gamma^-$.
2. *If* $\Gamma, x : \rho^+ \vdash v^- : \tau^-$, *then* $\Gamma \vdash [v^+/x]v^- : \tau^-$.
3. *If* $\Gamma, x : \rho^+ \vdash k^+ : \tau^+ > \gamma^-$, *then* $\Gamma \vdash [v^+/x]k^+ : \tau^+ > \gamma^-$.
4. *If* $\Gamma, x : \rho^+ \vdash v_1^+ : \tau^+$, *then* $\Gamma \vdash [v^+/x]v_1^+ : \tau^+$.

Preview

5. *If* $\Gamma, x : \rho^+ \vdash k^- : \tau^- > \gamma^-$, *then* $\Gamma \vdash [v^+/x]k^- : \tau^- > \gamma^-$.

*Proof.* Simultaneously, by induction on the derivation of the typing of the target of the substitution. □

**Lemma 39.2** (Composition)**.**

1. *If* $\Gamma \vdash e : \tau^-$ *and* $\Gamma \vdash k^- : \tau^- > \gamma^-$, *then* $\Gamma \vdash e\,;k^- : \tau^- > \gamma^-$.
2. *If* $\Gamma \vdash k_0^+ : \tau^+ > \gamma_0^-$, *and* $\Gamma \vdash k_1^- : \gamma_0^- > \gamma_1^-$, *then* $\Gamma \vdash k_0^+\,;k_1^- : \tau^+ > \gamma_1^-$.
3. *If* $\Gamma \vdash k_0^- : \tau^- > \gamma_0^-$, *and* $\Gamma \vdash k_1^- : \gamma_0^- > \gamma_1^-$, *then* $\Gamma \vdash k_0^-\,;k_1^- : \tau^- > \gamma_1^-$.

*Proof.* Simultaneously, by induction on the derivations of the first premises of each clause of the lemma. □

**Theorem 39.3** (Preservation)**.** *If* $\Gamma \vdash e : \gamma^-$ *and* $e \longmapsto e'$, *then* $\Gamma \vdash e' : \gamma^-$.

*Proof.* By induction on transition, appealing to inversion for typing and Lemmas 39.1 and 39.2. □

The progress theorem reduces to the characterization of the values of each type. Focusing makes the required properties evident, because it defines directly the values of each type.

**Theorem 39.4** (Progress)**.** *If* $\Gamma \vdash e : \gamma^-$, *then either* $e = \mathtt{ret}(v^-)$ *for some* $v^-$, *or there exists* $e'$ *such that* $e \longmapsto e'$.

## 39.6 Notes

The concept of polarization originates with Andreoli (1992), which introduced focusing as a technique for proof search in linear logic. The formulation given here is inspired by Zeilberger (2008), wherein focusing is related to evaluation order in programming languages.

Preview

Preview

# Part XVI

# Parallelism

Preview

Preview

# Chapter 40

# Nested Parallelism

Parallel computation seeks to reduce the running times of programs by allowing many computations to be carried out simultaneously. For example, if we wish to add two numbers, each given by a complex computation, we may consider evaluating the addends simultaneously, then computing their sum. The ability to exploit parallelism is limited by the dependencies among parts of a program. Obviously, if one computation depends on the result of another, then we have no choice but to execute them sequentially so that we may propagate the result of the first to the second. Consequently, the fewer dependencies among sub-computations, the greater the opportunities for parallelism. This argues for functional models of computation, because the possibility of mutation of shared assignables imposes sequentialization constraints on imperative code.

In this chapter we discuss *nested parallelism* in which we nest parallel computations within one another in a hierarchical manner. Nested parallelism is sometimes called *fork-join* parallelism to emphasize the hierarchical structure arising from *forking* two (or more) parallel computations, then *joining* these computations to combine their results before proceeding. We will consider two forms of dynamics for nested parallelism. The first is a structural dynamics in which a single transition on a compound expression may involve multiple transitions on its constituent expressions. The second is a cost dynamics (introduced in Chapter 7) that focuses attention on the sequential and parallel complexity (also known as the *work* and *depth*) of a parallel program by associating a *series-parallel graph* with each computation.

## 40.1 Binary Fork-Join

We begin with a parallel language whose sole source of parallelism is the simultaneous evaluation of two variable bindings. This is modelled by a construct of the form `par` $x_1$ `=` $e_1$ `and` $x_2$ `=` $e_2$ `in` $e$, in which we bind two variables, $x_1$ and $x_2$, to two expressions, $e_1$ and $e_2$, respectively, for use within a single expression, $e$. This represents a simple fork-join primitive in which $e_1$ and $e_2$ may be evaluated independently of one another, with their results combined by the expression $e$. Some other forms of parallelism may be defined in terms of this primitive. As an example, *parallel pairing*

Preview

may be defined as the expression

$$\texttt{par}\ x_1 = e_1\ \texttt{and}\ x_2 = e_2\ \texttt{in}\ \langle x_1, x_2\rangle,$$

which evaluates the components of the pair in parallel, then constructs the pair itself from these values.

The abstract syntax of the parallel binding construct is given by the abstract binding tree

$$\texttt{par}(e_1; e_2; x_1.x_2.e),$$

which makes clear that the variables $x_1$ and $x_2$ are bound *only* within $e$, and not within their bindings. This ensures that evaluation of $e_1$ is independent of evaluation of $e_2$, and *vice versa*. The typing rule for an expression of this form is given as follows:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{par}(e_1; e_2; x_1.x_2.e) : \tau} \tag{40.1}$$

Although we emphasize the case of binary parallelism, it should be clear that this construct easily generalizes to $n$-way parallelism for any *static* value of $n$. We may also define an $n$-way parallel `let` construct from the binary parallel `let` by cascading binary splits. (For a treatment of $n$-way parallelism for a *dynamic* value of $n$, see Section 40.3.)

We will give both a *sequential* and a *parallel* dynamics of the parallel `let` construct. The definition of the sequential dynamics as a transition judgment of the form $e \mapsto_{\mathsf{seq}} e'$ is given as follows:

$$\frac{e_1 \longmapsto e_1'}{\texttt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{seq}} \texttt{par}(e_1'; e_2; x_1.x_2.e)} \tag{40.2a}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\texttt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{seq}} \texttt{par}(e_1; e_2'; x_1.x_2.e)} \tag{40.2b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\texttt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{seq}} [e_1, e_2/x_1, x_2]e} \tag{40.2c}$$

The parallel dynamics is given by a transition judgment of the form $e \mapsto_{\mathsf{par}} e'$, defined as follows:

$$\frac{e_1 \mapsto_{\mathsf{par}} e_1' \quad e_2 \mapsto_{\mathsf{par}} e_2'}{\texttt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{par}} \texttt{par}(e_1'; e_2'; x_1.x_2.e)} \tag{40.3a}$$

$$\frac{e_1 \mapsto_{\mathsf{par}} e_1' \quad e_2\ \mathsf{val}}{\texttt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{par}} \texttt{par}(e_1'; e_2; x_1.x_2.e)} \tag{40.3b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto_{\mathsf{par}} e_2'}{\texttt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{par}} \texttt{par}(e_1; e_2'; x_1.x_2.e)} \tag{40.3c}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\texttt{par}(e_1; e_2; x_1.x_2.e) \mapsto_{\mathsf{par}} [e_1, e_2/x_1, x_2]e} \tag{40.3d}$$

Preview

The parallel dynamics is idealized in that it abstracts away from any limitations on parallelism that would necessarily be imposed in practice by the availability of computing resources.

An important advantage of the present approach is captured by the *implicit parallelism theorem*, which states that the sequential and the parallel dynamics coincide. This means that we need never be concerned with the *semantics* of a parallel program (its meaning is determined by the sequential dynamics), but only with its *efficiency*. As a practical matter, this means that a program may be developed on a sequential platform, even if it is intended to run on a parallel platform, because the behavior is not affected by whether we execute it using a sequential or a parallel dynamics.

Because the sequential dynamics is deterministic (every expression has at most one value), the implicit parallelism theorem implies that the parallel dynamics is also deterministic. For this reason the implicit parallelism theorem is also known as the *deterministic parallelism theorem*. This clearly distinguishes *deterministic parallelism*, the subject of this chapter, from *non-deterministic concurrency*, the subject of Chapters 42 and 43.

A proof of the implicit parallelism theorem may be given by giving an evaluation dynamics, $e \Downarrow v$, in the style of Chapter 7, and showing that

$$e \mapsto^*_{\mathsf{par}} v \quad \text{iff} \quad e \Downarrow v \quad \text{iff} \quad e \mapsto^*_{\mathsf{seq}} v$$

(where $v$ is a closed expression such that $v$ val). The crucial rule of the evaluation dynamics is the one governing the parallel `let` construct:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad [v_1, v_2/x_1, x_2]e \Downarrow v}{\mathtt{par}(e_1; e_2; x_1.x_2.e) \Downarrow v} \tag{40.4}$$

It is easy to show that the sequential dynamics agrees with the evaluation dynamics by extending the proof of Theorem 7.2.

**Lemma 40.1.** *For all $v$ val, $e \mapsto^*_{seq} v$ if, and only if, $e \Downarrow v$.*

*Proof.* It suffices to show that if $e \mapsto_{\mathsf{seq}} e'$ and $e' \Downarrow v$, then $e \Downarrow v$, and that if $e_1 \mapsto^*_{\mathsf{seq}} v_1$ and $e_2 \mapsto^*_{\mathsf{seq}} v_2$ and $[v_1, v_2/x_1, x_2]e \mapsto^*_{\mathsf{seq}} v$, then

$$\mathtt{par}\, x_1 = e_1 \,\mathtt{and}\, x_2 = e_2 \,\mathtt{in}\, e \mapsto^*_{\mathsf{seq}} v.$$

□

By a similar argument we may show that the parallel dynamics also agrees with the evaluation dynamics, and hence with the sequential dynamics.

**Lemma 40.2.** *For all $v$ val, $e \mapsto^*_{par} v$ if, and only if, $e \Downarrow v$.*

*Proof.* It suffices to show that if $e \mapsto_{\mathsf{par}} e'$ and $e' \Downarrow v$, then $e \Downarrow v$, and that if $e_1 \mapsto^*_{\mathsf{par}} v_1$ and $e_2 \mapsto^*_{\mathsf{par}} v_2$ and $[v_1, v_2/x_1, x_2]e \mapsto^*_{\mathsf{par}} v$, then

$$\mathtt{par}\, x_1 = e_1 \,\mathtt{and}\, x_2 = e_2 \,\mathtt{in}\, e \mapsto^*_{\mathsf{par}} v.$$

The proof of the first is by induction on the parallel dynamics. The proof of the second proceeds by simultaneous induction on the derivations of $e_1 \mapsto^*_{\mathsf{par}} v_1$ and $e_2 \mapsto^*_{\mathsf{par}} v_2$. If $e_1 = v_1$ with $v_1$ val



Preview

and $e_2 = v_2$ with $v_2$ val, then the result follows immediately from the third premise. If $e_2 = v_2$ but $e_1 \mapsto_{\mathsf{par}} e_1' \mapsto^*_{\mathsf{par}} v_1$, then by induction we have that $\mathtt{par}\, x_1 = e_1' \,\mathtt{and}\, x_2 = v_2 \,\mathtt{in}\, e \mapsto^*_{\mathsf{par}} v$, and hence the result follows by an application of rule (40.3b). The symmetric case follows similarly by an application of rule (40.3c), and in the case that both $e_1$ and $e_2$ take a step, the result follows by induction and rule (40.3a). □

**Theorem 40.3** (Implicit Parallelism)**.** *The sequential and parallel dynamics coincide: for all $v$ val, $e \mapsto^*_{seq} v$ iff $e \mapsto^*_{par} v$.*

*Proof.* By Lemmas 40.1 and 40.2. □

The implicit parallelism theorem states that parallelism does not affect the semantics of a program, only the efficiency of its execution. Correctness concerns are factored out, focusing attention on complexity.

## 40.2 Cost Dynamics

In this section we define a *parallel cost dynamics* that assigns a *cost graph* to the evaluation of an expression. Cost graphs are defined by the following grammar:

| | | | | |
|---|---|---|---|---|
| Cost | $c$ | ::= | $\mathbf{0}$ | zero cost |
| | | | $\mathbf{1}$ | unit cost |
| | | | $c_1 \otimes c_2$ | parallel combination |
| | | | $c_1 \oplus c_2$ | sequential combination |

A cost graph is a form of *series-parallel* directed acyclic graph, with a designated *source* node and *sink* node. For $\mathbf{0}$ the graph consists of one node and no edges, with the source and sink both being the node itself. For $\mathbf{1}$ the graph consists of two nodes and one edge directed from the source to the sink. For $c_1 \otimes c_2$, if $g_1$ and $g_2$ are the graphs of $c_1$ and $c_2$, respectively, then the graph has two additional nodes, a source node with two edges to the source nodes of $g_1$ and $g_2$, and a sink node, with edges from the sink nodes of $g_1$ and $g_2$ to it. Finally, for $c_1 \oplus c_2$, where $g_1$ and $g_2$ are the graphs of $c_1$ and $c_2$, the graph has as source node the source of $g_1$, as sink node the sink of $g_2$, and an edge from the sink of $g_1$ to the source of $g_2$.

The intuition behind a cost graph is that nodes represent subcomputations of an overall computation, and edges represent *sequentiality constraints* stating that one computation depends on the result of another, and hence cannot be started before the one on which it depends completes. The product of two graphs represents *parallelism opportunities* in which there are no sequentiality constraints between the two computations. The assignment of source and sink nodes reflects the overhead of *forking* two parallel computations and *joining* them after they have both completed.

We associate with each cost graph two numeric measures, the *work*, $wk(c)$, and the *depth*, $dp(c)$.



Preview

The work is defined by the following equations:

$$wk(c) = \begin{cases} 0 & \text{if } c = \mathbf{0} \\ 1 & \text{if } c = \mathbf{1} \\ wk(c_1) + wk(c_2) & \text{if } c = c_1 \otimes c_2 \\ wk(c_1) + wk(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases} \tag{40.5}$$

The depth is defined by the following equations:

$$dp(c) = \begin{cases} 0 & \text{if } c = \mathbf{0} \\ 1 & \text{if } c = \mathbf{1} \\ \max(dp(c_1), dp(c_2)) & \text{if } c = c_1 \otimes c_2 \\ dp(c_1) + dp(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases} \tag{40.6}$$

Informally, the work of a cost graph determines the total number of computation steps represented by the cost graph, and thus corresponds to the *sequential complexity* of the computation. The depth of the cost graph determines the *critical path length*, the length of the longest dependency chain within the computation, which imposes a lower bound on the *parallel complexity* of a computation. The critical path length is the least number of sequential steps that can be taken, even if we have unlimited parallelism available to us, because of steps that can be taken only after the completion of another.

In Chapter 7 we introduced *cost dynamics* as a means of assigning time complexity to evaluation. The proof of Theorem 7.7 shows that $e \Downarrow^k v$ iff $e \longmapsto^k v$. That is, the step complexity of an evaluation of $e$ to a value $v$ is just the number of transitions required to derive $e \longmapsto^* v$. Here we use cost graphs as the measure of complexity, then relate these cost graphs to the structural dynamics given in Section 40.1.

The judgment $e \Downarrow^c v$, where $e$ is a closed expression, $v$ is a closed value, and $c$ is a cost graph specifies the cost dynamics. By definition we arrange that $e \Downarrow^{\mathbf{0}} e$ when $e$ val. The cost assignment for `let` is given by the following rule:

$$\frac{e_1 \Downarrow^{c_1} v_1 \quad e_2 \Downarrow^{c_2} v_2 \quad [v_1, v_2/x_1, x_2]e \Downarrow^{c} v}{\mathtt{par}(e_1; e_2; x_1.x_2.e) \Downarrow^{(c_1 \otimes c_2) \oplus \mathbf{1} \oplus c} v} \tag{40.7}$$

The cost assignment specifies that, under ideal conditions, $e_1$ and $e_2$ are to be evaluated in parallel, and that their results are to be propagated to $e$. The cost of fork and join is implicit in the parallel combination of costs, and assign unit cost to the substitution because we expect it to be implemented in practice by a constant-time mechanism for updating an environment. The cost dynamics of other language constructs is specified in a similar manner, using only sequential combination so as to isolate the source of parallelism to the `let` construct.

Two simple facts about the cost dynamics are important to keep in mind. First, the cost assignment does not influence the outcome.

**Lemma 40.4.** *$e \Downarrow v$ iff $e \Downarrow^c v$ for some $c$.*

Preview

*Proof.* From right to left, erase the cost assignments to obtain an evaluation derivation. From left to right, decorate the evaluation derivations with costs as determined by the rules defining the cost dynamics. □

Second, the cost of evaluating an expression is uniquely determined.

**Lemma 40.5.** *If $e \Downarrow^{c} v$ and $e \Downarrow^{c'} v$, then $c$ is $c'$.*

*Proof.* A routine induction on the derivation of $e \Downarrow^{c} v$. □

The link between the cost dynamics and the structural dynamics given in the preceding section is established by the following theorem, which states that the work cost is the sequential complexity, and the depth cost is the parallel complexity, of the computation.

**Theorem 40.6.** *If $e \Downarrow^{c} v$, then $e \mapsto^{w}_{\mathsf{seq}} v$ and $e \mapsto^{d}_{\mathsf{par}} v$, where $w = \mathit{wk}(c)$ and $d = \mathit{dp}(c)$. Conversely, if $e \mapsto^{w}_{\mathsf{seq}} v$, then there exists $c$ such that $e \Downarrow^{c} v$ with $\mathit{wk}(c) = w$, and if $e \mapsto^{d}_{\mathsf{par}} v'$, then there exists $c'$ such that $e \Downarrow^{c'} v'$ with $\mathit{dp}(c') = d$.*

*Proof.* The first part is proved by induction on the derivation of $e \Downarrow^{c} v$, the interesting case being rule (40.7). By induction we have $e_1 \mapsto^{w_1}_{\mathsf{seq}} v_1$, $e_2 \mapsto^{w_2}_{\mathsf{seq}} v_2$, and $[v_1, v_2/x_1, x_2]e \mapsto^{w}_{\mathsf{seq}} v$, where $w_1 = \mathit{wk}(c_1)$, $w_2 = \mathit{wk}(c_2)$, and $w = \mathit{wk}(c)$. By pasting together derivations we obtain a derivation

$$\begin{aligned}\mathtt{par}(e_1; e_2; x_1.x_2.e) &\mapsto^{w_1}_{\mathsf{seq}} \mathtt{par}(v_1; e_2; x_1.x_2.e)\\ &\mapsto^{w_2}_{\mathsf{seq}} \mathtt{par}(v_1; v_2; x_1.x_2.e)\\ &\mapsto_{\mathsf{seq}} [v_1, v_2/x_1, x_2]e\\ &\mapsto^{w}_{\mathsf{seq}} v.\end{aligned}$$

Noting that $\mathit{wk}((c_1 \otimes c_2) \oplus \mathbf{1} \oplus c) = w_1 + w_2 + 1 + w$ completes the proof. Similarly, we have by induction that $e_1 \mapsto^{d_1}_{\mathsf{par}} v_1$, $e_2 \mapsto^{d_2}_{\mathsf{par}} v_2$, and $[v_1, v_2/x_1, x_2]e \mapsto^{d}_{\mathsf{par}} v$, where $d_1 = \mathit{dp}(c_1)$, $d_2 = \mathit{dp}(c_2)$, and $d = \mathit{dp}(c)$. Assume, without loss of generality, that $d_1 \leq d_2$ (otherwise simply swap the roles of $d_1$ and $d_2$ in what follows). We may paste together derivations as follows:

$$\begin{aligned}\mathtt{par}(e_1; e_2; x_1.x_2.e) &\mapsto^{d_1}_{\mathsf{par}} \mathtt{par}(v_1; e_2'; x_1.x_2.e)\\ &\mapsto^{d_2 - d_1}_{\mathsf{par}} \mathtt{par}(v_1; v_2; x_1.x_2.e)\\ &\mapsto_{\mathsf{par}} [v_1, v_2/x_1, x_2]e\\ &\mapsto^{d}_{\mathsf{par}} v.\end{aligned}$$

Calculating $\mathit{dp}((c_1 \otimes c_2) \oplus \mathbf{1} \oplus c) = \max(d_1, d_2) + 1 + d$ completes the proof.

Turning to the second part, it suffices to show that if $e \mapsto_{\mathsf{seq}} e'$ with $e' \Downarrow^{c'} v$, then $e \Downarrow^{c} v$ with $\mathit{wk}(c) = \mathit{wk}(c') + 1$, and if $e \mapsto_{\mathsf{par}} e'$ with $e' \Downarrow^{c'} v$, then $e \Downarrow^{c} v$ with $\mathit{dp}(c) = \mathit{dp}(c') + 1$.

Suppose that $e = \mathtt{par}(e_1; e_2; x_1.x_2.e_0)$ with $e_1$ val and $e_2$ val. Then $e \mapsto_{\mathsf{seq}} e'$, where $e = [e_1, e_2/x_1, x_2]e_0$ and there exists $c'$ such that $e' \Downarrow^{c'} v$. But then $e \Downarrow^{c} v$, where $c = (\mathbf{0} \otimes \mathbf{0}) \oplus \mathbf{1} \oplus c'$, and a simple calculation shows that $\mathit{wk}(c) = \mathit{wk}(c') + 1$, as required. Similarly, $e \mapsto_{\mathsf{par}} e'$ for $e'$ as above, and hence $e \Downarrow^{c} v$ for some $c$ such that $\mathit{dp}(c) = \mathit{dp}(c') + 1$, as required.



Preview

Suppose that $e = \mathtt{par}(e_1; e_2; x_1.x_2.e_0)$ and $e \mapsto_{\mathsf{seq}} e'$, where $e' = \mathtt{par}(e_1'; e_2; x_1.x_2.e_0)$ and $e_1 \mapsto_{\mathsf{seq}} e_1'$. From the assumption that $e' \Downarrow^{c'} v$, we have by inversion that $e_1' \Downarrow^{c_1'} v_1$, $e_2 \Downarrow^{c_2'} v_2$, and $[v_1, v_2/x_1, x_2]e_0 \Downarrow^{c_0'} v$, with $c' = (c_1' \otimes c_2') \oplus \mathbf{1} \oplus c_0'$. By induction there exists $c_1$ such that $wk(c_1) = 1 + wk(c_1')$ and $e_1 \Downarrow^{c_1} v_1$. But then $e \Downarrow^{c} v$, with $c = (c_1 \otimes c_2') \oplus \mathbf{1} \oplus c_0'$.

By a similar argument, suppose that $e = \mathtt{par}(e_1; e_2; x_1.x_2.e_0)$ and $e \mapsto_{\mathsf{par}} e'$, where $e' = \mathtt{par}(e_1'; e_2'; x_1.x_2.e_0)$ and $e_1 \mapsto_{\mathsf{par}} e_1'$, $e_2 \mapsto_{\mathsf{par}} e_2'$, and $e' \Downarrow^{c'} v$. Then by inversion $e_1' \Downarrow^{c_1'} v_1$, $e_2' \Downarrow^{c_2'} v_2$, $[v_1, v_2/x_1, x_2]e_0 \Downarrow^{c_0} v$. But then $e \Downarrow^{c} v$, where $c = (c_1 \otimes c_2) \oplus \mathbf{1} \oplus c_0$, $e_1 \Downarrow^{c_1} v_1$ with $dp(c_1) = 1 + dp(c_1')$, $e_2 \Downarrow^{c_2} v_2$ with $dp(c_2) = 1 + dp(c_2')$, and $[v_1, v_2/x_1, x_2]e_0 \Downarrow^{c_0} v$. Calculating, we obtain

$$\begin{aligned} dp(c) &= \max(dp(c_1') + 1, dp(c_2') + 1) + 1 + dp(c_0) \\ &= \max(dp(c_1'), dp(c_2')) + 1 + 1 + dp(c_0) \\ &= dp((c_1' \otimes c_2') \oplus \mathbf{1} \oplus c_0) + 1 \\ &= dp(c') + 1, \end{aligned}$$

which completes the proof. □

**Corollary 40.7.** *If $e \mapsto_{seq}^{w} v$ and $e \mapsto_{par}^{d} v'$, then $v$ is $v'$ and $e \Downarrow^{c} v$ for some $c$ such that $wk(c) = w$ and $dp(c) = d$.*

## 40.3 Multiple Fork-Join

So far we have confined attention to binary fork/join parallelism induced by the parallel `let` construct. Although technically sufficient for many purposes, a more natural programming model admits an unbounded number of parallel tasks to be spawned simultaneously, rather than forcing them to be created by a cascade of binary forks and corresponding joins. Such a model, often called *data parallelism*, ties the source of parallelism to a data structure of unbounded size. The principal example of such a data structure is a *sequence* of values of a specified type. The primitive operations on sequences provide a natural source of unbounded parallelism. For example, we may consider a parallel map construct that applies a given function to every element of a sequence simultaneously, forming a sequence of the results.

We will consider here a simple language of sequence operations to illustrate the main ideas.

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | ::= | $\mathtt{seq}(\tau)$ | $\tau\,\mathtt{seq}$ | sequence |
| Exp | $e$ | ::= | $\mathtt{seq}(e_0, \ldots, e_{n-1})$ | $[e_0, \ldots, e_{n-1}]$ | sequence |
| | | | $\mathtt{len}(e)$ | $\lvert e \rvert$ | size |
| | | | $\mathtt{sub}(e_1; e_2)$ | $e_1[e_2]$ | element |
| | | | $\mathtt{tab}(x.e_1; e_2)$ | $\mathtt{tab}(x.e_1; e_2)$ | tabulate |
| | | | $\mathtt{map}(x.e_1; e_2)$ | $[e_1 \mid x \in e_2]$ | map |
| | | | $\mathtt{cat}(e_1; e_2)$ | $\mathtt{cat}(e_1; e_2)$ | concatenate |

The expression $\mathtt{seq}(e_0, \ldots, e_{n-1})$ evaluates to an $n$-sequence whose elements are given by the expressions $e_0, \ldots, e_{n-1}$. The operation $\mathtt{len}(e)$ returns the number of elements in the sequence given by $e$. The operation $\mathtt{sub}(e_1; e_2)$ retrieves the element of the sequence given by $e_1$ at the index given

Preview

by $e_2$. The tabulate operation, $\texttt{tab}(x.e_1; e_2)$, yields the sequence of length given by $e_2$ whose $i$th element is given by $[i/x]e_1$. The operation $\texttt{map}(x.e_1; e_2)$ computes the sequence whose $i$th element is given by $[e/x]e_1$, where $e$ is the $i$th element of the sequence given by $e_2$. The operation $\texttt{cat}(e_1; e_2)$ concatenates two sequences of the same type.

The statics of these operations is given by the following typing rules:

$$\frac{\Gamma \vdash e_0 : \tau \quad \ldots \quad \Gamma \vdash e_{n-1} : \tau}{\Gamma \vdash \texttt{seq}(e_0, \ldots, e_{n-1}) : \texttt{seq}(\tau)} \tag{40.8a}$$

$$\frac{\Gamma \vdash e : \texttt{seq}(\tau)}{\Gamma \vdash \texttt{len}(e) : \texttt{nat}} \tag{40.8b}$$

$$\frac{\Gamma \vdash e_1 : \texttt{seq}(\tau) \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{sub}(e_1; e_2) : \tau} \tag{40.8c}$$

$$\frac{\Gamma, x : \texttt{nat} \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{tab}(x.e_1; e_2) : \texttt{seq}(\tau)} \tag{40.8d}$$

$$\frac{\Gamma \vdash e_2 : \texttt{seq}(\tau) \quad \Gamma, x : \tau \vdash e_1 : \tau'}{\Gamma \vdash \texttt{map}(x.e_1; e_2) : \texttt{seq}(\tau')} \tag{40.8e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{seq}(\tau) \quad \Gamma \vdash e_2 : \texttt{seq}(\tau)}{\Gamma \vdash \texttt{cat}(e_1; e_2) : \texttt{seq}(\tau)} \tag{40.8f}$$

The cost dynamics of these constructs is defined by the following rules:

$$\frac{e_0 \Downarrow^{c_0} v_0 \quad \ldots \quad e_{n-1} \Downarrow^{c_{n-1}} v_{n-1}}{\texttt{seq}(e_0, \ldots, e_{n-1}) \Downarrow^{\bigotimes_{i=0}^{n-1} c_i} \texttt{seq}(v_0, \ldots, v_{n-1})} \tag{40.9a}$$

$$\frac{e \Downarrow^{c} \texttt{seq}(v_0, \ldots, v_{n-1})}{\texttt{len}(e) \Downarrow^{c \oplus \mathbf{1}} \texttt{num}[n]} \tag{40.9b}$$

$$\frac{e_1 \Downarrow^{c_1} \texttt{seq}(v_0, \ldots, v_{n-1}) \quad e_2 \Downarrow^{c_2} \texttt{num}[i] \quad (0 \leq i < n)}{\texttt{sub}(e_1; e_2) \Downarrow^{c_1 \oplus c_2 \oplus \mathbf{1}} v_i} \tag{40.9c}$$

$$\frac{e_2 \Downarrow^{c} \texttt{num}[n] \quad [\texttt{num}[0]/x]e_1 \Downarrow^{c_0} v_0 \quad \ldots \quad [\texttt{num}[n-1]/x]e_1 \Downarrow^{c_{n-1}} v_{n-1}}{\texttt{tab}(x.e_1; e_2) \Downarrow^{c \oplus \bigotimes_{i=0}^{n-1} c_i} \texttt{seq}(v_0, \ldots, v_{n-1})} \tag{40.9d}$$

$$\frac{e_2 \Downarrow^{c} \texttt{seq}(v_0, \ldots, v_{n-1}) \quad [v_0/x]e_1 \Downarrow^{c_0} v_0' \quad \ldots \quad [v_{n-1}/x]e_1 \Downarrow^{c_{n-1}} v_{n-1}'}{\texttt{map}(x.e_1; e_2) \Downarrow^{c \oplus \bigotimes_{i=0}^{n-1} c_i} \texttt{seq}(v_0', \ldots, v_{n-1}')} \tag{40.9e}$$

$$\frac{e_1 \Downarrow^{c_1} \texttt{seq}(v_0, \ldots, v_{m-1}) \quad e_2 \Downarrow^{c_2} \texttt{seq}(v_0', \ldots, v_{n-1}')}{\texttt{cat}(e_1; e_2) \Downarrow^{c_1 \oplus c_2 \oplus \bigotimes_{i=0}^{m+n} \mathbf{1}} \texttt{seq}(v_0, \ldots, v_{m-1}, v_0', \ldots, v_{n-1}')} \tag{40.9f}$$

The cost dynamics for sequence operations may be validated by introducing a sequential and parallel cost dynamics and extending the proof of Theorem 40.6 to cover this extension.

Preview

## 40.4 Provably Efficient Implementations

Theorem 40.6 states that the cost dynamics accurately models the dynamics of the parallel `let` construct, whether executed sequentially or in parallel. This validates the cost dynamics from the point of view of the dynamics of the language, and permits us to draw conclusions about the asymptotic complexity of a parallel program that abstracts away from the limitations imposed by a concrete implementation. Chief among these is the restriction to a fixed number, $p > 0$, of processors on which to schedule the workload. In addition to limiting the available parallelism this also imposes some synchronization overhead that must be accounted for in order to make accurate predictions of run-time behavior on a concrete parallel platform. A *provably efficient implementation* is one for which we may establish an asymptotic bound on the actual execution time once these overheads are taken into account.

A provably efficient implementation must take account of the limitations and capabilities of the actual hardware on which the program is to be run. Because we are only interested in asymptotic upper bounds, it is convenient to formulate an abstract machine model, and to show that the primitives of the language can be implemented on this model with guaranteed time (and space) bounds. One popular model is the *SMP*, or *shared-memory multiprocessor*, which consists of $p > 0$ sequential processors coordinated by an interconnect network that provides constant-time access to shared memory by each of the processors.[1] The multiprocessor is assumed to provide a constant-time synchronization primitive with which to control simultaneous access to a memory cell. There are a variety of such primitives, any of which is sufficient to provide a parallel fetch-and-add instruction that allows each processor to obtain the current contents of a memory cell and update it by adding a fixed constant in a single atomic operation—the interconnect serializes any simultaneous accesses by more than one processor.

Building a provably efficient implementation of parallelism involves two majors tasks. First, we must show that each of the primitives of the language may be implemented efficiently on the abstract machine model. Second, we must show how to schedule the workload across the processors so as to minimize execution time by maximizing parallelism. When working with a low-level machine model such as an SMP, both tasks involve a fair bit of technical detail to show how to use low-level machine instructions, including a synchronization primitive, to implement the language primitives and to schedule the workload. Collecting these together, we may then give an asymptotic bound on the time complexity of the implementation that relates the abstract cost of the computation to cost of implementing the workload on a $p$-way multiprocessor. The prototypical result of this kind is called *Brent's Theorem*.

**Theorem 40.8.** *If $e \Downarrow^c v$ with $wk(c) = w$ and $dp(c) = d$, then $e$ may be evaluated on a $p$-processor SMP in time $O(\max(w/p, d))$.*

The theorem tells us that we can never execute a program in fewer steps than its depth, $d$, and that, at best, we can divide the work up evenly into $w/p$ rounds of execution by the $p$ processors. Observe that if $p = 1$ then the theorem establishes an upper bound of $O(w)$ steps, the sequential complexity of the computation. Moreover, if the work is proportional to the depth, then we are unable to exploit parallelism, and the overall time is proportional to the work alone.

[1] A slightly weaker assumption is that each access may require up to $\lg p$ time to account for the overhead of synchronization, but we shall neglect this refinement in the present, simplified account.

PREVIEW

This motivates the definition of a useful figure of merit, called the *parallelizability ratio*, which is the ratio, $w/d$, of work to depth. If $w/d \gg p$, then the program is said to be *parallelizable*, because then $w/p \gg d$, and we may therefore reduce running time by using $p$ processors at each step. If, on the other hand, the parallelizability ratio is a constant, then $d$ will dominate $w/p$, and we will have little opportunity to exploit parallelism to reduce running time. It is not known, in general, whether a problem admits a parallelizable solution. The best we can say, on present knowledge, is that there are algorithms for some problems that have a high degree of parallelizability, and there are problems for which no such algorithm is known. It is a open problem in complexity theory to characterize which problems are parallelizable, and which are not.

To illustrate the essential ingredients of the proof of Brent's Theorem we will consider a dynamics that models the scheduling of work onto $p$ parallel processors, each of which implements the dynamics of **PCF** as described in Chapter 15. The parallel dynamics is defined on states $\nu\,\Sigma\,\{\,\mu\,\}$ of the form

$$\nu\, a_1 \sim \tau_1, \ldots a_n \sim \tau_n \,\{\, a_1 \hookrightarrow e_1 \otimes \ldots \otimes a_n \hookrightarrow e_n \,\},$$

where $n \geq 1$. Such a state represents a computation that has been decomposed into $n$ parallel tasks. Each task is given a *name*. The occurrence of a name within a task represents a dependency of that task on the named task. A task is said to be *blocked* on the tasks on which it depends; a task with no dependencies is said to be *ready*.

There are two forms of transition, the *local* and the *global*. The local transitions represent the steps of the individual processors. These consist of the steps of execution of expressions as defined in Chapter 15, augmented with transitions governing parallelism. The global transitions represent the simultaneous execution of local transitions on up to some fixed number, $p$, of processors.

Local transitions have the form

$$\nu\,\Sigma\, a \sim \tau \,\{\, \mu \otimes a \hookrightarrow e \,\} \longmapsto_{loc} \nu\,\Sigma'\, a \sim \tau \,\{\, \mu' \otimes a \hookrightarrow e' \,\},$$

where $e$ is ready, and either (a) $\Sigma$ and $\Sigma'$ are empty, and $\mu$ and $\mu'$ are empty; or (b) $\Sigma$ and $\mu$ are empty, and $\Sigma'$ and $\mu'$ declare the types and bindings of two distinct names; or (c) $\Sigma'$ and $\mu'$ are both empty, and $\Sigma$ and $\mu$ declare the types and bindings of two distinct names. These conditions correspond to the three possible outcomes of a local step by a task: (a) the task takes a step of computation in the sense of Chapter 15; (b) the task forks two new tasks, and waits for their completion; (c) the task joins two parallel tasks that have completed execution.

$$\frac{}{\nu\, a \sim \tau \,\{\, a \hookrightarrow (\lambda\,(x{:}\tau_2)\, e)(e_2) \,\} \longmapsto_{loc} \nu\, a \sim \tau \,\{\, a \hookrightarrow [e_2/x]e \,\}} \tag{40.10a}$$

$$\frac{}{\left\{\begin{array}{c} \nu\, a \sim \tau \,\{\, a \hookrightarrow \mathtt{par}(e_1; e_2; x_1.x_2.e) \,\} \\ \longmapsto_{loc} \\ \nu\, a_1 \sim \tau_1\, a_2 \sim \tau_2\, a \sim \tau \,\{\, a_1 \hookrightarrow e_1 \otimes a_2 \hookrightarrow e_2 \otimes a \hookrightarrow \mathtt{join}[a_1; a_2](x_1; x_2.e) \,\} \end{array}\right\}} \tag{40.10b}$$



Preview

$$\frac{e_1 \text{ val} \quad e_2 \text{ val}}{\left\{\begin{array}{c} \nu\, a_1 \sim \tau_1\, a_2 \sim \tau_2\, a \sim \tau\, \{\, a_1 \hookrightarrow e_1 \otimes a_2 \hookrightarrow e_2 \otimes a \hookrightarrow \mathtt{join}[a_1;a_2](x_1;x_2.e)\,\} \\ \longmapsto_{loc} \\ \nu\, a \sim \tau\, \{\, a \hookrightarrow [e_1, e_2/x_1, x_2]e\,\} \end{array}\right\}} \tag{40.10c}$$

Rule (40.10a) illustrates one rule for the dynamics of the non-parallel aspects of the language; additional rules are required to cover the other cases, allowing for nested uses of parallelism. Rule (40.10b) represents the creation of two parallel tasks on which the executing task depends. The expression $\mathtt{join}[a_1;a_2](x_2;x_2.e)$ is blocked on tasks $a_1$ and $a_2$, so that no local step applies to it. Rule (40.10c) synchronizes a task with the tasks on which it depends once their execution has completed; those tasks are no longer required, and are therefore eliminated from the state.

Each global transition represents the simultaneous execution of one step of computation on each of up to $p \geq 1$ processors.

$$\frac{\begin{array}{c} \nu\, \Sigma_1 a_1 \sim \tau_1\, \{\, \mu_1 \otimes a_1 \hookrightarrow e_1\,\} \longmapsto_{loc} \nu\, \Sigma_1' a_1 \sim \tau_1\, \{\, \mu_1' \otimes a_1 \hookrightarrow e_1'\,\} \\ \ldots \\ \nu\, \Sigma_n a_n \sim \tau_n\, \{\, \mu_n \otimes a_n \hookrightarrow e_n\,\} \longmapsto_{loc} \nu\, \Sigma_n' a_n \sim \tau_n\, \{\, \mu_n' \otimes a_n \hookrightarrow e_n'\,\} \end{array}}{\left\{\begin{array}{c} \nu\, \Sigma_0\, \Sigma_1\, a_1 \sim \tau_1\, \ldots\, \Sigma_n\, a_n \sim \tau_n\, \{\, \mu_0 \otimes \mu_1 \otimes a_1 \hookrightarrow e_1 \otimes \ldots \otimes \mu_n \otimes a_n \hookrightarrow e_n\,\} \\ \longmapsto_{glo} \\ \nu\, \Sigma_0\, \Sigma_1'\, a_1 \sim \tau_1\, \ldots\, \Sigma_n'\, a_n \sim \tau_n\, \{\, \mu_0 \otimes \mu_1' \otimes a_1 \hookrightarrow e_1' \otimes \ldots \otimes \mu_n' \otimes a_n \hookrightarrow e_n'\,\} \end{array}\right\}} \tag{40.11}$$

At each global step some number, $1 \leq n \leq p$, of ready tasks are scheduled for execution.[2] Because no two distinct tasks may depend on the same task, we may partition the $n$ tasks so that each scheduled task is grouped with the tasks on which it depends as necessary for any local join step. Any local fork step introduces two fresh tasks that are added to the state as a result of the global transition; any local join step eliminates two tasks whose execution has completed.

A subtle point is that it is implicit in our name binding conventions that the names of any created tasks are to be *globally unique*, even though they are *locally created*. In implementation terms this requires a synchronization step among the processors to ensure that task names are not acccidentally reused among the parallel processors.

The proof of Brent's Theorem for this high-level dynamics is now obvious, provided only that the global scheduling steps are performed greedily so as to maximize the use of processors at each round. If, at each stage of a computation, there are $p$ ready tasks, then the computation will complete in $w/p$ steps, where $w$ is the work complexity of the program. We may, however, be unable to make full use of all $p$ processors at any given stage. This would only be because the dependencies among computations, which are reflected in the variable occurrences and in the definition of the depth complexity of the computation, inhibits parallelism to the extent that evaluation cannot complete in fewer than $d$ rounds. This limitation is significant only to the extent that $d$ is larger than $w/p$; otherwise, the overall time is bounded by $w/p$, making maximal use of all $p$ processors.

[2] The rule does not require that $n$ be chosen as large as possible. A scheduler that always chooses the largest possible $1 \leq n \leq p$ is said to be *greedy*.

Preview

## 40.5 Notes

Parallelism should not be confused with concurrency. Parallelism is about efficiency, not semantics; the meaning of a program is independent of whether it is executed in parallel or not. Concurrency is about composition, not efficiency; the meaning of a concurrent program is very weakly specified so that we may compose it with other programs without altering its meaning. This distinction, and the formulation of it given here, was pioneered by Blelloch (1990). The concept of a cost semantics and the idea of a provably efficient implementation are derived from Blelloch and Greiner (1995, 1996).

Preview

# Chapter 41

# Futures and Speculations

A *future* is a computation whose evaluation is initiated in advance of any demand for its value. Like a suspension, a future represents a value that is to be determined later. Unlike a suspension, a future is always evaluated, regardless of whether its value is actually required. In a sequential setting futures are of little interest; a future of type $\tau$ is just an expression of type $\tau$. In a parallel setting, however, futures are of interest because they provide a means of initiating a parallel computation whose result is not needed until (presumably) much later, by which time it will have been completed.

The prototypical example of the use of futures is to implementing *pipelining*, a method for overlapping the stages of a multistage computation to the fullest extent possible. This minimizes the latency caused by one stage waiting for the completion of a previous stage by allowing the two stages to proceed in parallel until such time as an explicit dependency is encountered. Ideally, the computation of the result of an earlier stage is completed by the time a later stage requires it. At worst the later stage must be delayed until the earlier stage completes, incurring what is known as a *pipeline stall*.

A *speculation* is a delayed computation whose result may or may not be needed for the overall computation to finish. The dynamics for speculations executes suspended computations in parallel with the main thread of computation, without regard to whether the value of the speculation is actually required by the main thread. If the value of the speculation is required, then such a dynamics pays off, but if not, the effort to compute it is wasted.

Futures are *work efficient* in that the overall work done by a computation involving futures is no more than the work required by a sequential execution. Speculations, in contrast, are *work inefficient* in that speculative execution may be in vain—the overall computation may involve more steps than the work required to compute the result. For this reason speculation is a risky strategy for exploiting parallelism. It can make good use of available resources, but perhaps only at the expense of doing more work than necessary!

Preview

## 41.1 Futures

The syntax of futures is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{fut}(\tau)$ | $\tau\,\mathtt{fut}$ | future |
| Exp | $e$ | $::=$ | $\mathtt{fut}(e)$ | $\mathtt{fut}(e)$ | future |
| | | | $\mathtt{fsyn}(e)$ | $\mathtt{fsyn}(e)$ | synchronize |
| | | | $\mathtt{fut}[a]$ | $\mathtt{fut}[a]$ | indirection |

The type $\tau\,\mathtt{fut}$ is the type of futures of type $\tau$. Futures are introduced by the expression $\mathtt{fut}(e)$, which schedules $e$ for evaluation and returns a reference to it. Futures are eliminated by the expression $\mathtt{fsyn}(e)$, which synchronizes with the future referred to by $e$, returning its value. Indirect references to future values are represented by $\mathtt{fut}[a]$, indicating a future value to be stored at $a$.

### 41.1.1 Statics

The statics of futures is given by the following rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{fut}(e) : \mathtt{fut}(\tau)} \tag{41.1a}$$

$$\frac{\Gamma \vdash e : \mathtt{fut}(\tau)}{\Gamma \vdash \mathtt{fsyn}(e) : \tau} \tag{41.1b}$$

These rules are unsurprising, because futures add no new capabilities to the language beyond providing an opportunity for parallel evaluation.

### 41.1.2 Sequential Dynamics

The sequential dynamics of futures is easily defined. Futures are evaluated eagerly; synchronization returns the value of the future.

$$\frac{e\ \mathsf{val}}{\mathtt{fut}(e)\ \mathsf{val}} \tag{41.2a}$$

$$\frac{e \longmapsto e'}{\mathtt{fut}(e) \longmapsto \mathtt{fut}(e')} \tag{41.2b}$$

$$\frac{e \longmapsto e'}{\mathtt{fsyn}(e) \longmapsto \mathtt{fsyn}(e')} \tag{41.2c}$$

$$\frac{e\ \mathsf{val}}{\mathtt{fsyn}(\mathtt{fut}(e)) \longmapsto e} \tag{41.2d}$$

Under a sequential dynamics futures have little purpose: they simply introduce a pointless level of indirection.

Preview

## 41.2 Speculations

The syntax of (non-recursive) speculations is given by the following grammar:[1]

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{spec}(\tau)$ | $\tau\,\mathtt{spec}$ | speculation |
| Exp | $e$ | $::=$ | $\mathtt{spec}(e)$ | $\mathtt{spec}(e)$ | speculate |
| | | | $\mathtt{ssyn}(e)$ | $\mathtt{ssyn}(e)$ | synchronize |
| | | | $\mathtt{spec}[a]$ | $\mathtt{spec}[a]$ | indirection |

The type $\tau\,\mathtt{spec}$ is the type of speculations of type $\tau$. The introductory form, $\mathtt{spec}(e)$, creates a computation that may be speculatively evaluated, and the eliminatory form, $\mathtt{ssyn}(e)$, synchronizes with a speculation. A reference to the result of a speculative computation stored at location $a$ is written $\mathtt{spec}[a]$.

### 41.2.1 Statics

The statics of speculations is given by the following rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{spec}(e) : \mathtt{spec}(\tau)} \tag{41.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{spec}(\tau)}{\Gamma \vdash \mathtt{ssyn}(e) : \tau} \tag{41.3b}$$

Thus, the statics for speculations as given by rules (41.3) is essentially equivalent to the statics for futures given by rules (41.1).

### 41.2.2 Sequential Dynamics

The definition of the sequential dynamics of speculations is similar to that of futures, except that speculations are values.

$$\frac{}{\mathtt{spec}(e)\ \mathsf{val}} \tag{41.4a}$$

$$\frac{e \longmapsto e'}{\mathtt{ssyn}(e) \longmapsto \mathtt{ssyn}(e')} \tag{41.4b}$$

$$\frac{}{\mathtt{ssyn}(\mathtt{spec}(e)) \longmapsto e} \tag{41.4c}$$

Under a sequential dynamics speculations are simply a re-formlation of suspensions.

[1] We confine ourselves to the non-recursive case to facilitate the comparison with futures.



Preview

## 41.3 Parallel Dynamics

Futures are only interesting insofar as they admit a parallel dynamics that allows the computation of the future to proceed concurrently with some other computation. In this section we give a parallel dynamics of futures and speculation in which the creation, execution, and synchronization of tasks is made explicit. Interestingly, the parallel dynamics of futures and speculations is *identical*, except for the termination condition. Whereas futures require all tasks to be completed before termination, speculations may be abandoned before they are completed. For the sake of concision we will give the parallel dynamics of futures, remarking only where alterations must be made for the parallel dynamics of speculations.

The parallel dynamics of futures relies on a modest extension to the language given in Section 41.1 to introduce *names* for tasks. Let $\Sigma$ be a finite mapping assigning types to names. As mentioned earlier, the expression $\mathtt{fut}[a]$ is a value referring to the outcome of task $a$. The statics of this expression is given by the following rule:[2]

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{fut}[a] : \mathtt{fut}(\tau)} \tag{41.5}$$

Rules (41.1) carry over in the obvious way with $\Sigma$ recording the types of the task names.

States of the parallel dynamics have the form $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$, where $e$ is the *focus* of evaluation, and $\mu$ records the parallel futures (or speculations) that have been activated thus far in the computation. Formally, $\mu$ is a finite mapping assigning expressions to the task names declared in $\Sigma$. A state is well-formed according to the following rule:

$$\frac{\vdash_\Sigma e : \tau \quad (\forall a \in dom(\Sigma))\ \vdash_\Sigma \mu(a) : \Sigma(a)}{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{ok}} \tag{41.6}$$

As discussed in Chapter 37 this rule admits self-referential and mutually referential futures. A more refined condition could as well be given that avoids circularities; we leave this as an exercise for the reader.

The parallel dynamics is divided into two phases, the *local* phase, which defines the basic steps of evaluation of an expression, and the *global* phase, which executes all possible local steps in parallel. The local dynamics of futures is defined by the following rules:[3]

$$\frac{}{\mathtt{fut}[a]\ \mathsf{val}_{\Sigma, a \sim \tau}} \tag{41.7a}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{fut}(e) \parallel \mu\,\} \longmapsto_{loc} \nu\,\Sigma, a \sim \tau\,\{\,\mathtt{fut}[a] \parallel \mu \otimes a \hookrightarrow e\,\}} \tag{41.7b}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\} \longmapsto_{loc} \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{fsyn}(e) \parallel \mu\,\} \longmapsto_{loc} \nu\,\Sigma'\,\{\,\mathtt{fsyn}(e') \parallel \mu'\,\}} \tag{41.7c}$$

[2] A similar rule governs the analogous construct, $\mathtt{spec}[a]$, in the case of speculations.

[3] These rules must be augmented by a reformulation of the dynamics of the other constructs of the language phrased in terms of the present notion of state.



Preview

$$\frac{e' \ \mathsf{val}_{\Sigma, a \sim \tau}}{\left\{\begin{array}{c} \nu\, \Sigma, a \sim \tau \, \{\, \mathtt{fsyn}(\mathtt{fut}[a]) \parallel \mu \otimes a \hookrightarrow e' \,\} \\ \longmapsto_{loc} \\ \nu\, \Sigma, a \sim \tau \, \{\, e' \parallel \mu \otimes a \hookrightarrow e' \,\} \end{array}\right\}} \tag{41.7d}$$

Rule (41.7b) activates a future named $a$ executing the expression $e$ and returns a reference to it. Rule (41.7d) synchronizes with a future whose value has been determined. Note that a local transition always has the form

$$\nu\, \Sigma \, \{\, e \parallel \mu \,\} \longmapsto_{loc} \nu\, \Sigma\, \Sigma' \, \{\, e' \parallel \mu \otimes \mu' \,\}$$

where $\Sigma'$ is either empty or declares the type of a single symbol, and $\mu'$ is either empty or of the form $a \hookrightarrow e'$ for some expression $e'$.

A global step of the parallel dynamics consists of at most one local step for the focal expression and one local step for each of up to $p$ futures, where $p > 0$ is a fixed parameter representing the number of processors.

$$\frac{\begin{array}{c} \mu = \mu_0 \otimes a_1 \hookrightarrow e_1 \otimes \ldots \otimes a_n \hookrightarrow e_n \\ \mu'' = \mu_0 \otimes a_1 \hookrightarrow e_1' \otimes \ldots \otimes a_n \hookrightarrow e_n' \\ \nu\, \Sigma \, \{\, e \parallel \mu \,\} \longmapsto_{loc}^{\mathbf{0,1}} \nu\, \Sigma\, \Sigma' \, \{\, e' \parallel \mu \otimes \mu' \,\} \\ (\forall 1 \leq i \leq n \leq p) \quad \nu\, \Sigma \, \{\, e_i \parallel \mu \,\} \longmapsto_{loc} \nu\, \Sigma\, \Sigma_i' \, \{\, e_i' \parallel \mu \otimes \mu_i' \,\} \end{array}}{\left\{\begin{array}{c} \nu\, \Sigma \, \{\, e \parallel \mu \,\} \\ \longmapsto_{glo} \\ \nu\, \Sigma\, \Sigma'\, \Sigma_1' \, \ldots \, \Sigma_n' \, \{\, e' \parallel \mu'' \otimes \mu' \otimes \mu_1' \otimes \ldots \otimes \mu_n' \,\} \end{array}\right\}} \tag{41.8a}$$

Rule (41.8a) allows the focus expression to take either zero or one steps because it may be blocked awaiting the completion of evaluation of a parallel future (or synchronizing with a speculation). The futures allocated by the local steps of execution are consolidated in the result of the global step. We assume without loss of generality that the names of the new futures in each local step are pairwise disjoint so that the combination makes sense. In implementation terms satisfying this disjointness assumption means that the processors must synchronize their access to memory.

The initial state of a computation, whether for futures or speculations, is defined by the rule

$$\frac{}{\nu\, \varnothing \, \{\, e \parallel \varnothing \,\} \ \mathsf{initial}} \tag{41.9}$$

Final states differ according to whether we are considering futures or speculations. In the case of futures a state is final iff both the focus and all parallel futures have completed evaluation:

$$\frac{e \ \mathsf{val}_\Sigma \quad \mu \ \mathsf{val}_\Sigma}{\nu\, \Sigma \, \{\, e \parallel \mu \,\} \ \mathsf{final}} \tag{41.10a}$$

$$\frac{(\forall a \in dom(\Sigma)) \ \mu(a) \ \mathsf{val}_\Sigma}{\mu \ \mathsf{val}_\Sigma} \tag{41.10b}$$



Preview

In the case of speculations a state is final iff the focus is a value:

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{final}} \tag{41.11}$$

There may be unfinished speculations in such a final state.

## 41.4 Applications of Futures

Perhaps the simplest application of futures is a form of `let` binding in which the bound expression is evaluated simultaneously with its body:

$$\mathtt{letfut}\,x\,\mathtt{be}\,e_1\,\mathtt{in}\,e_2 \triangleq \mathtt{let}\,x'\,\mathtt{be}\,\mathtt{fut}(e_1)\,\mathtt{in}\,[\mathtt{fsyn}(x')/x]e_2.$$

Evaluation of the body may proceed until such time that the value of the bound variable is required, at which point computation may not proceed until that computation has finished.

As another simple example, we may use futures to encode binary nested parallelism by defining $\mathtt{par}(e_1; e_2; x_1.x_2.e)$ to stand for the expression

$$\mathtt{let}\ x_1'\ \mathtt{be}\ \mathtt{fut}(e_1)\ \mathtt{in}\ \mathtt{let}\ x_2\ \mathtt{be}\ e_2\ \mathtt{in}\ \mathtt{let}\ x_1\ \mathtt{be}\ \mathtt{fsyn}(x_1')\ \mathtt{in}\ e$$

The order of bindings is important to ensure that evaluation of $e_2$ proceeds in parallel with evaluation of $e_1$. Observe that evaluation of $e$ cannot, in any case, proceed until both are complete.

*Pipelining* is a more interesting example of the use of parallel futures. Consider a situation in which a *producer* builds a list whose elements represent units of work, and a *consumer* traverses the work list and acts on each element of that list. The elements of the work list can be thought of as "instructions" to the consumer, which maps a function over that list to carry out those instructions. An obvious sequential implementation first builds the work list, then traverses it to perform the work indicated by the list. This is fine as long as the elements of the list can be produced quickly, but if each element requires a substantial amount of computation, it would be preferable to overlap production of the next list element with execution of the previous unit of work. This can be easily programmed using futures.

Let `flist` be the recursive type $\mathtt{rec}\,t\,\mathtt{is}\,\mathtt{unit} + (\mathtt{nat} \times t\,\mathtt{fut})$, whose elements are `nil`, defined to be $\mathtt{fold}(\mathtt{l} \cdot \langle\rangle)$, and $\mathtt{cons}(e_1,e_2)$, defined to be $\mathtt{fold}(\mathtt{r} \cdot \langle e_1, \mathtt{fut}(e_2)\rangle)$. The producer is a recursive function that generates a value of type `flist`:

```
fix produce : (nat → nat opt) → nat → flist is
  λ f. λ i.
    case f(i) {
      null ↪ nil
    | just x ↪ cons(x, fut (produce f (i+1)))
    }
```

On each iteration the producer generates a parallel future to produce the tail. This computation proceeds after the producer returns so that it overlaps subsequent computation.

The consumer folds an operation over the work list as follows:



Preview

```
fix consume : ((nat×nat)→nat) → nat → flist → nat is
  λ g. λ a. λ xs.
    case xs {
      nil ↪ a
    | cons (x, xs) ↪ consume g (g (x, a)) (fsyn xs)
    }
```

The consumer synchronizes with the tail of the work list just at the point where it makes a recursive call and hence requires the head element of the tail to continue processing. At this point the consumer will block, if necessary, to await computation of the tail before continuing the recursion.

Speculations arise naturally in lazy languages. But although they provide opportunities for parallelism, they are not, in general, *work efficient*: a speculation may be evaluated even though its value is never required. An alternative is to combine suspensions (see Chapter 38) with futures so that the programmer may specify which suspensions ought profitably to be evaluated in parallel. This is achieved using *sparks*, which evaluate a computation in parallel solely for its effect on suspensions that are sure to be needed later. Specifically, we may define

$$\texttt{spark}(e_1; e_2) \triangleq \texttt{letfut}\ _\ \texttt{be}\ \texttt{force}(e_1)\ \texttt{in}\ e_2,$$

where $e_1 : \tau_1\ \texttt{susp}$ and $e_2 : \tau_2$. The expression $\texttt{force}(e_1)$ is evaluated in parallel, forcing the evaluation of $e_1$, in hopes that it will have completed evaluation before its value is required by $e_2$.

As an example, consider the type `strm` of streams of numbers defined by the recursive type $\texttt{rec}\, t\ \texttt{is}\ (\texttt{unit} + (\texttt{nat} \times t))\ \texttt{susp}$. Elements of this type are suspended computations that, when forced, either signals the end of stream, or produces a number and another such stream. Suppose that $s$ is such a stream, and assume that we know, for reasons of its construction, that it is finite. We wish to compute $\texttt{map}(f)(s)$ for some function $f$, and to overlap this computation with the production of the stream elements. We will make use of a function `mapforce` that forces successive elements of the input stream, but yields no useful output. The computation

$$\texttt{letfut}\ _\ \texttt{be}\ \texttt{map}(\texttt{force})(s)\ \texttt{in}\ \texttt{map}(f)(s)$$

forces the elements of the stream in parallel with the computation of $\texttt{map}(f)(s)$, with the intention that all suspensions in $s$ are forced before their values are required by the main computation.

## 41.5 Notes

Futures were introduced by Friedman and Wise (1976), and featured in the MultiLisp language (Halstead, 1985) for parallel programming. A similar concept is proposed by Arvind et al. (1986) under the name "I-structures." The formulation given here is derived from Greiner and Blelloch (1999). Sparks were introduced by Trinder, *et al.* Trinder et al. (1998) in Haskell, a lazy language.

Preview

Preview

# Part XVII

# Concurrency and Distribution

Preview

Preview

# Chapter 42

# Milner's Process Calculus

So far we have mainly studied the statics and dynamics of programs in isolation, without regard to their interaction with the world. But to extend this analysis to even the most rudimentary forms of input and output requires that we consider external agents that interact with the program. After all, the whole purpose of a computer is, ultimately, to interact with a person!

To extend our investigations to interactive systems, we begin with the study of *process calculi*, which are abstract formalisms that capture the essence of interaction among independent agents. The development will proceed in stages, starting with simple action models, then extending to interacting concurrent processes, and finally to synchronous and asynchronous communication. The calculus consists of two main syntactic categories, *processes* and *events*. The basic form of process is one that awaits the arrival of an event. Processes are closed under parallel composition (the product of processes), replication, and declaration of a channel. The basic forms of event are signaling on a channel and querying a channel; these are later generalized to sending and receiving data on a channel. Events are closed under a finite choice (sum) of events. When enriched with types of messages and channel references, the process calculus may be seen to be universal in that it is at least as powerful as the untyped $\lambda$-caclulus.

## 42.1 Actions and Events

Our treatment of concurrent interaction is based on the notion of an *event*, which specifies the *actions* that a process is prepared to undertake in concert with another process. Two processes interact by undertaking two complementary actions, which may be thought of as a *signal* and a *query* on a *channel*. The processes synchronize when one signals on a channel that the other is querying, after which they both proceed independently to interact with other processes.

To begin with we will focus on sequential processes, which simply await the arrival of one of

Preview

several possible actions, known as an event.

| | | | | | |
|---|---|---|---|---|---|
| Proc | $P$ | ::= | $\texttt{await}(E)$ | $\$\,E$ | synchronize |
| Evt | $E$ | ::= | $\texttt{null}$ | $\mathbf{0}$ | null |
| | | | $\texttt{or}(E_1;E_2)$ | $E_1 + E_2$ | choice |
| | | | $\texttt{que}[a](P)$ | $?a;P$ | query |
| | | | $\texttt{sig}[a](P)$ | $!a;P$ | signal |

The variable $a$ ranges over symbols serving as *channel names* that mediate communication among the processes.

We will not distinguish between events that differ only up to *structural congruence*, which is defined to be the strongest equivalence relation closed under these rules:

$$\frac{E \equiv E'}{\$\,E \equiv \$\,E'} \tag{42.1a}$$

$$\frac{E_1 \equiv E_1' \quad E_2 \equiv E_2'}{E_1 + E_2 \equiv E_1' + E_2'} \tag{42.1b}$$

$$\frac{P \equiv P'}{?a;P \equiv ?a;P'} \tag{42.1c}$$

$$\frac{P \equiv P'}{!a;P \equiv !a;P'} \tag{42.1d}$$

$$\frac{}{E + \mathbf{0} \equiv E} \tag{42.1e}$$

$$\frac{}{E_1 + E_2 \equiv E_2 + E_1} \tag{42.1f}$$

$$\frac{}{E_1 + (E_2 + E_3) \equiv (E_1 + E_2) + E_3} \tag{42.1g}$$

Imposing structural congruence on sequential processes enables us to think of an event as having the form

$$!a;P_1 + \ldots + ?a;Q_1 + \ldots$$

consisting of a sum of signal and query events, with the sum of no events being the null event, $\mathbf{0}$.

An illustrative example of Milner's is a simple vending machine that may take in a 2p coin, then optionally either permit selection of a cup of tea, or take another 2p coin, then permit selection of a cup of coffee.

$$V = \$\,(?\texttt{2p};\$\,(!\texttt{tea};V + ?\texttt{2p};\$\,(!\texttt{cof};V)))$$

As the example indicates, we permit recursive definitions of processes, with the understanding that a defined identifier may always be replaced with its definition wherever it occurs. (Later we will show how to avoid reliance on recursive definitions.)



Preview

Because the computation occurring within a process is suppressed, sequential processes have no dynamics on their own, but only through their interaction with other processes. For the vending machine to operate there must be another process (you) who initiates the events expected by the machine, causing both your state (the coins in your pocket) and its state (as just described) to change as a result.

## 42.2 Interaction

Processes become interesting when they are allowed to interact with one another to achieve a common goal. To account for interaction we enrich the language of processes with *concurrent composition*:

| | | | | | |
|---|---|---|---|---|---|
| Proc | $P$ | ::= | $\mathtt{await}(E)$ | $\$\,E$ | synchronize |
| | | | $\mathtt{stop}$ | $\mathbf{1}$ | inert |
| | | | $\mathtt{par}(P_1; P_2)$ | $P_1 \parallel P_2$ | composition |

The process $\mathbf{1}$ represents the inert process, and the process $P_1 \parallel P_2$ represents the concurrent composition of $P_1$ and $P_2$. We may identify $\mathbf{1}$ with $\$\,\mathbf{0}$, the process that awaits the event that will never occur, but we prefer to treat the inert process as a primitive concept.

We will identify processes up to structural congruence, which is defined to be the strongest equivalence relation closed under these rules:

$$\frac{}{P \parallel \mathbf{1} \equiv P} \tag{42.2a}$$

$$\frac{}{P_1 \parallel P_2 \equiv P_2 \parallel P_1} \tag{42.2b}$$

$$\frac{}{P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3} \tag{42.2c}$$

$$\frac{P_1 \equiv P_1' \quad P_2 \equiv P_2'}{P_1 \parallel P_2 \equiv P_1' \parallel P_2'} \tag{42.2d}$$

Up to structural congruence every process has the form

$$\$\,E_1 \parallel \ldots \parallel \$\,E_n$$

for some $n \geq 0$, it being understood that when $n = 0$ this stands for the null process, $\mathbf{1}$.

Interaction between processes consists of synchronization of two complementary actions. The dynamics of interaction is defined by two forms of judgment. The transition judgment $P \longmapsto P'$ states that the process $P$ evolves to the process $P'$ as a result of a single step of computation. The family of transition judgments, $P \overset{\alpha}{\longmapsto} P'$, where $\alpha$ is an *action*, states that the process $P$ may evolve to the process $P'$ provided that the action $\alpha$ is permissible in the context in which the transition occurs (in a sense to be made precise momentarily). As a notational convenience, we often regard the unlabeled transition to be the labeled transition corresponding to the special silent action.

Preview

The possible actions are given by the following grammar:

$$\begin{array}{llcll} \text{Act} & \alpha & ::= & \mathtt{que}[a] & a\,? & \text{query} \\ & & & \mathtt{sig}[a] & a\,! & \text{signal} \\ & & & \mathtt{sil} & \varepsilon & \text{silent} \end{array}$$

The *query action*, $a\,?$, and the *signal action*, $a\,!$, are complementary, and the *silent action*, $\varepsilon$, is self-complementary. We define the *complementary* action to $\alpha$ to be the action $\overline{\alpha}$ given by the equations $\overline{a\,?} = a\,!$, $\overline{a\,!} = a\,?$, and $\overline{\varepsilon} = \varepsilon$.

$$\frac{}{\$\,(!a;P + E) \stackrel{a\,!}{\mapsto} P} \tag{42.3a}$$

$$\frac{}{\$\,(?a;P + E) \stackrel{a\,?}{\mapsto} P} \tag{42.3b}$$

$$\frac{P_1 \stackrel{\alpha}{\mapsto} P_1'}{P_1 \parallel P_2 \stackrel{\alpha}{\mapsto} P_1' \parallel P_2} \tag{42.3c}$$

$$\frac{P_1 \stackrel{\alpha}{\mapsto} P_1' \quad P_2 \stackrel{\overline{\alpha}}{\mapsto} P_2'}{P_1 \parallel P_2 \longmapsto P_1' \parallel P_2'} \tag{42.3d}$$

Rules (42.3a) and (42.3b) specify that any of the events on which a process is synchronizing may occur. Rule (42.3d) synchronizes two processes that take complementary actions.

As an example, let us consider the interaction of the vending machine, $V$, with the user process, $U$, defined as follows:

$$U = \$\,!\mathtt{2p};\$\,!\mathtt{2p};\$\,?\mathtt{cof};\mathbf{1}.$$

Here is a trace of the interaction between $V$ and $U$:

$$\begin{aligned} V \parallel U &\longmapsto \$\,(!\mathtt{tea};V + ?\mathtt{2p};\$\,!\mathtt{cof};V) \parallel \$\,!\mathtt{2p};\$\,?\mathtt{cof};\mathbf{1} \\ &\longmapsto \$\,!\mathtt{cof};V \parallel \$\,?\mathtt{cof};\mathbf{1} \\ &\longmapsto V \end{aligned}$$

Preview

These steps are justified, respectively, by the following pairs of labeled transitions:

$$U \overset{\texttt{2p!}}{\longmapsto} U' = \$\,!\texttt{2p};\$\,?\texttt{cof};\mathbf{1}$$
$$V \overset{\texttt{2p?}}{\longmapsto} V' = \$\,(!\texttt{tea};V + ?\texttt{2p};\$\,!\texttt{cof};V)$$

$$U' \overset{\texttt{2p!}}{\longmapsto} U'' = \$\,?\texttt{cof};\mathbf{1}$$
$$V' \overset{\texttt{2p?}}{\longmapsto} V'' = \$\,!\texttt{cof};V$$

$$U'' \overset{\texttt{cof?}}{\longmapsto} \mathbf{1}$$
$$V'' \overset{\texttt{cof!}}{\longmapsto} V$$

We have suppressed uses of structural congruence in the above derivations to avoid clutter, but it is important to see its role in managing the non-deterministic choice of events by a process.

## 42.3 Replication

Some presentations of process calculi forego reliance on defining equations for processes in favor of a *replication* construct, which we write $*P$. This process stands for as many concurrently executing copies of $P$ as we may require, which may be modeled by the structural congruence

$$*P \equiv P \parallel *P. \tag{42.4}$$

Understood as a principle of structural congruence, this rule hides the steps of process creation, and gives no hint as to how often it can or should be applied. We could alternatively build replication into the dynamics to model the details of replication more closely:

$$*P \longmapsto P \parallel *P. \tag{42.5}$$

Because the application of this rule is unconstrained, it may be applied at any time to effect a new copy of the replicated process $P$.

So far we have been using recursive process definitions to define processes that interact repeatedly according to some protocol. Rather than take recursive definition as a primitive notion, we may instead use replication to model repetition. This may be achieved by introducing an "activator" process that is contacted to effect the replication. Consider the recursive definition $X = P(X)$, where $P$ is a process expression that may refer to itself as $X$. Such a self-referential process may be simulated by defining the activator process

$$A = *\,\$\,(?a;P(\$\,(!a;\mathbf{1}))),$$

in which we have replaced occurrences of $X$ within $P$ by an initiator process that signals the event $a$ to the activator. Observe that the activator, $A$, is structurally congruent to the process $A' \parallel A$, where $A'$ is the process

$$\$\,(?a;P(\$\,(!a;\mathbf{1}))).$$



Preview

To start process $P$ we concurrently compose the activator, $A$, with an initiator process, $\$(!a;\mathbf{1})$. Observe that

$$A \parallel \$(!a;\mathbf{1}) \longmapsto A \parallel P(\$\,!a;\mathbf{1}),$$

which starts the process $P$ while maintaining a running copy of the activator, $A$.

As an example, let us consider Milner's vending machine written using replication, rather than using recursive process definition:

$$V_0 = \$(!v;\mathbf{1}) \tag{42.6}$$
$$V_1 = *\,\$(?v;V_2) \tag{42.7}$$
$$V_2 = \$(?\mathtt{2p};\$(!\mathtt{tea};V_0 + ?\mathtt{2p};\$(!\mathtt{cof};V_0))) \tag{42.8}$$

The process $V_1$ is a replicated server that awaits a signal on channel $v$ to create another instance of the vending machine. The recursive calls are replaced by signals along $v$ to re-start the machine. The original machine, $V$, is simulated by the concurrent composition $V_0 \parallel V_1$.

This example motivates a commonly-considered restriction on replication that avoids the indeterminacy inherent in choosing when and whether to expand a replication into a parallel composition. To avoid this, we may replace general replication by *replicated synchronization*, which is governed by the following rules:

$$\frac{}{*\,\$(!a;P+E) \stackrel{a!}{\longmapsto} P \parallel *\,\$(!a;P+E)} \tag{42.9a}$$

$$\frac{}{*\,\$(?a;P+E) \stackrel{a?}{\longmapsto} P \parallel *\,\$(?a;P+E)} \tag{42.9b}$$

The process $*\,\$(E)$ is to be regarded not as a composition of replication and synchronization, but as the inseparable combination of these two constructs. The advantage is that the replication occurs only as needed, precisely when a synchronization with another process is possible. This avoids the need to "guess", either by structural congruence or an explicit step, when to replicate a process.

## 42.4 Allocating Channels

It is often useful (particularly once we have introduced inter-process communication) to introduce new channels within a process, rather than assume that all channels of interaction are given *a priori*. To allow for this, the syntax of processes is enriched with a channel declaration primitive:

$$\mathsf{Proc} \quad P \quad ::= \quad \mathtt{new}(a.P) \quad \nu\, a.P \quad \text{new channel}$$

The channel, $a$, is bound within the process $P$, and hence may be renamed at will (avoiding conflicts) within $P$. To simplify notation we sometimes write $\nu\, a_1, \ldots, a_k.P$ for the iterated declaration $\nu\, a_1 \ldots \nu\, a_k.P$.



Preview

Structural congruence is extended with the following rules:

$$\frac{P =_\alpha P'}{P \equiv P'} \tag{42.10a}$$

$$\frac{P \equiv P'}{\nu\, a.P \equiv \nu\, a.P'} \tag{42.10b}$$

$$\frac{a \notin P_2}{(\nu\, a.P_1) \parallel P_2 \equiv \nu\, a.(P_1 \parallel P_2)} \tag{42.10c}$$

$$\overline{\nu\, a.\nu\, b.P \equiv \nu\, b.\nu\, a.P} \tag{42.10d}$$

$$\frac{(a \notin P)}{\nu\, a.P \equiv P} \tag{42.10e}$$

Rule (42.10c), called *scope extrusion*, will be especially important in Section 42.6. Rule (42.10e) states that channels may be de-allocated once they are no longer in use.

To account for the scopes of names (and to prepare for later generalizations) it is useful to introduce a static semantics for processes that ensures that names are properly scoped. A *signature*, $\Sigma$, is, for the time being, a finite set of channels. The judgment $\vdash_\Sigma P$ proc states that a process, $P$, is well-formed relative to the channels declared in the signature, $\Sigma$.

$$\overline{\vdash_\Sigma \mathbf{1}\ \mathsf{proc}} \tag{42.11a}$$

$$\frac{\vdash_\Sigma P_1\ \mathsf{proc} \quad \vdash_\Sigma P_2\ \mathsf{proc}}{\vdash_\Sigma P_1 \parallel P_2\ \mathsf{proc}} \tag{42.11b}$$

$$\frac{\vdash_\Sigma E\ \mathsf{event}}{\vdash_\Sigma \$\,E\ \mathsf{proc}} \tag{42.11c}$$

$$\frac{\vdash_{\Sigma,a} P\ \mathsf{proc}}{\vdash_\Sigma \nu\, a.P\ \mathsf{proc}} \tag{42.11d}$$

The foregoing rules make use of an auxiliary judgment, $\vdash_\Sigma E$ event, stating that $E$ is a well-formed event relative to $\Sigma$.

$$\overline{\vdash_\Sigma \mathbf{0}\ \mathsf{event}} \tag{42.12a}$$

$$\frac{\vdash_{\Sigma,a} P\ \mathsf{proc}}{\vdash_{\Sigma,a} ?a;P\ \mathsf{event}} \tag{42.12b}$$

$$\frac{\vdash_{\Sigma,a} P\ \mathsf{proc}}{\vdash_{\Sigma,a} !a;P\ \mathsf{event}} \tag{42.12c}$$

$$\frac{\vdash_\Sigma E_1\ \mathsf{event} \quad \vdash_\Sigma E_2\ \mathsf{event}}{\vdash_\Sigma E_1 + E_2\ \mathsf{event}} \tag{42.12d}$$

Preview

We shall also have need of the judgment $\vdash_\Sigma \alpha$ action stating that $\alpha$ is a well-formed action relative to $\Sigma$:

$$\frac{}{\vdash_{\Sigma,a} a\,?\ \mathsf{action}} \tag{42.13a}$$

$$\frac{}{\vdash_{\Sigma,a} a\,!\ \mathsf{action}} \tag{42.13b}$$

$$\frac{}{\vdash_{\Sigma} \varepsilon\ \mathsf{action}} \tag{42.13c}$$

The dynamics is correspondingly generalized to keep track of the set of active channels. The judgment $P \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P'$ states that $P$ transitions to $P'$ with action $\alpha$ relative to channels $\Sigma$. The rules defining the dynamics are indexed forms of those given above, augmented by an additional rule governing the declaration of a channel. We give the complete set of rules here for the sake of clarity.

$$\frac{}{\$\,(!a;P + E) \overset{a!}{\underset{\Sigma,a}{\longmapsto}} P} \tag{42.14a}$$

$$\frac{}{\$\,(?a;P + E) \overset{a?}{\underset{\Sigma,a}{\longmapsto}} P} \tag{42.14b}$$

$$\frac{P_1 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P_1'}{P_1 \parallel P_2 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P_1' \parallel P_2} \tag{42.14c}$$

$$\frac{P_1 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P_1' \quad P_2 \overset{\overline{\alpha}}{\underset{\Sigma}{\longmapsto}} P_2'}{P_1 \parallel P_2 \underset{\Sigma}{\longmapsto} P_1' \parallel P_2'} \tag{42.14d}$$

$$\frac{P \overset{\alpha}{\underset{\Sigma,a}{\longmapsto}} P' \quad \vdash_\Sigma \alpha\ \mathsf{action}}{\nu\, a.P \overset{\alpha}{\underset{\Sigma}{\longmapsto}} \nu\, a.P'} \tag{42.14e}$$

Rule (42.14e) states that no process may interact with $\nu\, a.P$ along the locally-allocated channel, $a$, because to do so would require that $a$ already be declared in $\Sigma$, which is precluded by the freshness convention on binders.

As an example, let us consider again the definition of the vending machine using replication, rather than recursion. The channel, $v$, used to initialize the machine should be considered private to the machine itself, and not be made available to a user process. This is naturally expressed by the process expression $\nu\, v.(V_0 \parallel V_1)$, where $V_0$ and $V_1$ are as defined above using the designated

Preview

channel, $v$. This process correctly simulates the original machine, $V$, because it precludes interaction with a user process on channel $v$. If $U$ is a user process, the interaction begins as follows:

$$(\nu\, v.(V_0 \parallel V_1)) \parallel U \underset{\Sigma}{\mapsto} (\nu\, v.V_2) \parallel U \equiv \nu\, v.(V_2 \parallel U).$$

(The processes $V_0$, $V_1$, and $V_2$ are those defined earlier.) The interaction continues as before, albeit within the scope of the binder, provided that $v$ has been chosen (by structural congruence) to be apart from $U$, ensuring that it is private to the internal workings of the machine.

## 42.5 Communication

*Synchronization* is the coordination of the execution of two processes that are willing to undertake the complementary actions of signalling and querying a common channel. *Synchronous communication* is a natural generalization of synchronization to allow more than one bit of data to be communicated between two coordinating processes, a *sender* and a *receiver*. In principle any type of data may be communicated from one process to another, and we can give a uniform account of communication that is independent of the type of data communicated between processes. Communication becomes more interesting in the presence of a type of *channel references*, which allow access to a communication channel to be propagated from one process to another, allowing alteration of the interconnection topology among processes during execution. (Channel references will be discussed in Section 42.6.)

To account for interprocess communication we must enrich the language of processes to include *variables*, as well as *channels*, in the formalism. Variables range, as always, over types, and are given meaning by substitution. Channels, on the other hand, are assigned types that classify the data carried on that channel, and are given meaning by send and receive events that generalize the signal and query events considered earlier. The abstract syntax of communication events is given by the following grammar:

$$\begin{array}{llcll} \mathsf{Evt} & E & ::= & \mathtt{snd}[a](e;P) & !\,a(e;P) \quad \text{send} \\ & & & \mathtt{rcv}[a](x.P) & ?\,a(x.P) \quad \text{receive} \end{array}$$

The event $\mathtt{rcv}[a](x.P)$ represents the receipt of a value, $x$, on the channel $a$, passing $x$ to the process $P$. The variable, $x$, is bound within $P$, and hence may be chosen freely, subject to the usual restrictions on the choice of names of bound variables. The event $\mathtt{snd}[a](e;P)$ represents the transmission of (the value of) the expression $e$ on channel $a$, continuing with the process $P$ only once this value has been received.

To account for the type of data that may be sent on a channel, the syntax of channel declaration is generalized to associate a type with each channel name.

$$\mathsf{Proc} \quad P \quad ::= \quad \mathtt{new}[\tau](a.P) \quad \nu\, a \sim \tau.P \quad \text{typed channel}$$

The process $\mathtt{new}[\tau](a.P)$ introduces a new channel name, $a$, with associated type $\tau$ for use within the process $P$. The name, $a$, is bound within $P$, and hence may be chosen at will, subject only to avoidance of confusion of distinct names.



Preview

The statics of communication extends that of synchronization by associating types to channels and by considering variables that range over a type. The judgment $\Gamma \vdash_\Sigma P$ proc states that $P$ is a well-formed process involving the channels declared in $\Sigma$ and the variables declared in $\Gamma$. It is inductively defined by the following rules, wherein we assume that the typing judgment $\Gamma \vdash_\Sigma e : \tau$ is given separately.

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{1} \text{ proc}} \tag{42.15a}$$

$$\frac{\Gamma \vdash_\Sigma P_1 \text{ proc} \quad \Gamma \vdash_\Sigma P_2 \text{ proc}}{\Gamma \vdash_\Sigma P_1 \parallel P_2 \text{ proc}} \tag{42.15b}$$

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau} P \text{ proc}}{\Gamma \vdash_\Sigma \nu\, a \sim \tau . P \text{ proc}} \tag{42.15c}$$

$$\frac{\Gamma \vdash_\Sigma E \text{ event}}{\Gamma \vdash_\Sigma \$\, E \text{ proc}} \tag{42.15d}$$

Rules (42.15) make use of the auxiliary judgment $\Gamma \vdash_\Sigma E$ event, stating that $E$ is a well-formed event relative to $\Gamma$ and $\Sigma$, which is defined as follows:

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{0} \text{ event}} \tag{42.16a}$$

$$\frac{\Gamma \vdash_\Sigma E_1 \text{ event} \quad \Gamma \vdash_\Sigma E_2 \text{ event}}{\Gamma \vdash_\Sigma E_1 + E_2 \text{ event}} \tag{42.16b}$$

$$\frac{\Gamma, x : \tau \vdash_{\Sigma, a \sim \tau} P \text{ proc}}{\Gamma \vdash_{\Sigma, a \sim \tau} ?\, a(x.P) \text{ event}} \tag{42.16c}$$

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau} e : \tau \quad \Gamma \vdash_{\Sigma, a \sim \tau} P \text{ proc}}{\Gamma \vdash_{\Sigma, a \sim \tau} !\, a(e;P) \text{ event}} \tag{42.16d}$$

Rule (42.16d) makes use of a typing judgment for expressions that ensures that the type of a channel is respected by communication.

The dynamics of synchronous communication is similarly an extension of the dynamics of synchronization. Actions are generalized to include the transmitted value, as well as the channel and its orientation:

| | | | | | |
|---|---|---|---|---|---|
| Act | $\alpha$ | ::= | $\mathtt{rcv}[a](e)$ | $a\,?\,e$ | receive |
| | | | $\mathtt{snd}[a](e)$ | $a\,!\,e$ | send |
| | | | $\mathtt{sil}$ | $\varepsilon$ | silent |

Complementarity is defined, essentially as before, to switch the orientation of an action: $\overline{a\,?\,e} = a\,!\,e$, $\overline{a\,!\,e} = a\,?\,e$, and $\overline{\varepsilon} = \varepsilon$.

The statics ensures that the expression associated with these actions is a value of a type suitable for the channel:

$$\frac{\vdash_{\Sigma, a \sim \tau} e : \tau \quad e \text{ val}_{\Sigma, a \sim \tau}}{\vdash_{\Sigma, a \sim \tau} a\,!\,e \text{ action}} \tag{42.17a}$$

Preview

$$\frac{\vdash_{\Sigma,a\sim\tau} e : \tau \quad e\ \mathsf{val}_{\Sigma,a\sim\tau}}{\vdash_{\Sigma,a\sim\tau} a\,?\,e\ \mathsf{action}} \tag{42.17b}$$

$$\frac{}{\vdash_{\Sigma} \varepsilon\ \mathsf{action}} \tag{42.17c}$$

The dynamics of synchronous communication is defined by replacing rules (42.14a) and (42.14b) with the following rules:

$$\frac{e \xmapsto[\Sigma,a\sim\tau]{} e'}{\$\,(!\,a(e;P) + E) \xmapsto[\Sigma,a\sim\tau]{} \$\,(!\,a(e';P) + E)} \tag{42.18a}$$

$$\frac{e\ \mathsf{val}_{\Sigma,a\sim\tau}}{\$\,(!\,a(e;P) + E) \xmapsto[\Sigma,a\sim\tau]{a!e} P} \tag{42.18b}$$

$$\frac{e\ \mathsf{val}_{\Sigma,a\sim\tau}}{\$\,(?\,a(x.P) + E) \xmapsto[\Sigma,a\sim\tau]{a?e} [e/x]P} \tag{42.18c}$$

Rule (42.18c) is non-deterministic in that it "guesses" the value, $e$, to be received along channel $a$. Rules (42.18) make reference to the dynamics of expressions, which is left unspecified here so as to avoid an *a priori* commitment as to the nature of values communicated on a channel.

Using synchronous communication, both the sender and the receiver of a message are blocked until the interaction is completed. This means that the sender must be notified whenever a message is received, and hence there must be an implicit reply channel from the receiver to the sender for the notification. This suggests that synchronous communication may be decomposed into a simpler *asynchronous send* operation, which transmits a message on a channel without waiting for its receipt, together with *channel passing* to transmit an acknowledgement channel along with the message data.

*Asynchronous communication* is defined by removing the synchronous send event from the process calculus, and adding a new form of process that simply sends a message on a channel. The syntax of asynchronous send is as follows:

$$\mathsf{Proc} \quad P \quad ::= \quad \mathtt{asnd}[a](e) \quad !\,a(e) \quad \text{send}$$

The process $\mathtt{asnd}[a](e)$ sends the message $e$ on channel $a$, and then terminates immediately. Without the synchronous send event, every event is, up to structural congruence, a choice of zero or more read events. The statics of asynchronous send is given by the following rule:

$$\frac{\Gamma \vdash_{\Sigma,a\sim\tau} e : \tau}{\Gamma \vdash_{\Sigma,a\sim\tau} !\,a(e)\ \mathsf{proc}} \tag{42.19}$$

The dynamics is given similarly:

$$\frac{e\ \mathsf{val}_{\Sigma}}{!\,a(e) \xmapsto[\Sigma]{a!e} \mathbf{1}} \tag{42.20}$$

Preview

The rule for interprocess communication given earlier remains unchanged, because the action associated with the asychronous send is the same as in the synchronous case. We may regard a pending asynchronous send as a "buffer" in which the message is held until a receiver is selected.

## 42.6 Channel Passing

An interesting case of interprocess communication arises when one process passes one channel to another along a common channel. The channel passed by the sending process need not have been known *a priori* to the receiving process. This allows for new patterns of communication to be established among processes. For example, two processes, $P$ and $Q$, may share a channel, $a$, along which they may send and receive messages. If the scope of $a$ is limited to these processes, then no other process, $R$, may communicate on that channel; it is, in effect, a *private* channel between $P$ and $Q$.

It frequently arises, however, that $P$ and $Q$ wish to include the process $R$ in their conversation in a controlled manner. This may be accomplished by first expanding the scope of the channel $a$ to encompass $R$, then sending (a reference to) the channel $a$ to $R$ along a pre-arranged channel. Upon receipt of the channel reference, $R$ may communicate with $P$ and $Q$ using send and receive operations that act on channel references. Bearing in mind that channels are not themselves forms of expression, such a scenario can be enacted by introducing a type, $\tau\,\mathtt{chan}$, whose values are references to channels carrying values of type $\tau$. The elimination forms for the channel type are send and receive operations that act on references, rather than explicitly given channels.[1]

Such a situation may be described schematically by the process expression

$$(\nu\, a \sim \tau.(P \parallel Q)) \parallel R,$$

in which the process $R$ is initially excluded from the scope of the channel $a$, whose scope encompasses both the processes $P$ and $Q$. The type $\tau$ represents the type of data communicated along channel $a$; it may be chosen arbitrarily for the sake of this example. The processes $P$ and $Q$ may communicate with each other by sending and receiving along channel $a$. If these two processes wish to include $R$ in the conversation, then they must communicate the identity of channel $a$ to the process $R$ along some pre-arranged channel, $b$. If $a$ is a channel carrying values of type $\tau$, then $b$ will be a channel carrying values of type $\tau\,\mathtt{chan}$, which are references to $\tau$-carrying channels. The channel $b$ must be known to at least one of $P$ and $Q$, and also to $R$. This can be described by the following process expression:

$$\nu\, b \sim \tau\,\mathtt{chan}.((\nu\, a \sim \tau.(P \parallel Q)) \parallel R).$$

Suppose that $P$ wishes to include $R$ in the conversation by sending a reference to the channel $a$ along $b$. The process $R$ correspondingly receives a *reference* to a channel on the channel $b$, and commences communication with $P$ and $Q$ along that channel. Thus $P$ has the form $\$\,(!\,b(\&\,a;P'))$ and $R$ has the form $\$\,(?\,b(x.R'))$. The overall process has the form

$$\nu\, b \sim \tau\,\mathtt{chan}.(\nu\, a \sim \tau.(\$\,(!\,b(\&\,a;P')) \parallel Q) \parallel \$\,(?\,b(x.R'))).$$

[1] It may be helpful to compare channel types with reference types as described in Chapters 36 and 37. Channels correspond to assignables, and channel types correspond to reference types.



Preview

Note carefully that the declaration of the channel $b$ specifies that it *carries* a channel reference, not that it *is* a channel reference.

The process $P$ is prepared to send a reference to the channel $a$ along the channel $b$, where it may be received by the process $R$. But the scope of $a$ is limited to processes $P$ and $Q$, so in order for the communication to succeed, we must first expand its scope to encompass $R$ using the concept of scope extrusion introduced in Section 42.4 to obtain the structurally equivalent process

$$\nu\, b \sim \tau\ \mathtt{chan}.\nu\, a \sim \tau.(\$\,(!\, b(\&\, a; P')) \parallel Q \parallel \$\,(?\, b(x.R'))).$$

The scope of $a$ has been expanded to encompass $R$, preparing the ground for communication between $P$ and $R$, which results in the process

$$\nu\, b \sim \tau\ \mathtt{chan}.\nu\, a \sim \tau.(P' \parallel Q \parallel [\&\, a/x]R').$$

The reference to the channel $a$ has been substituted for the variable $x$ within $R'$.

The process $R$ may now communicate with $P$ and $Q$ by sending and receiving messages along the channel referenced by $x$. This is accomplished using dynamic forms of send and receive in which the channel on which to communicate is determined by evaluation of an expression, rather than specified statically by an explicit channel name. For example, to send a message $e$ of type $\tau$ along the channel referred to by $x$, the process $R'$ would have the form

$$\$\,(!!\,(x; e; R'')).$$

Similarly, to receive along the referenced channel, the process $R'$ would have the form

$$\$\,(??\,(x; y.R'')).$$

In both cases the dynamic communication forms evolve to the static communication forms once the referenced channel has been determined.

The syntax of channel types is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{chan}(\tau)$ | $\tau\ \mathtt{chan}$ | channel type |
| Exp | $e$ | $::=$ | $\mathtt{ch}[a]$ | $\&\, a$ | reference |
| Evt | $E$ | $::=$ | $\mathtt{sndref}(e_1; e_2; P)$ | $!!\,(e_1; e_2; P)$ | send |
| | | | $\mathtt{rcvref}(e; x.P)$ | $??\,(e; x.P)$ | receive |

The events $\mathtt{sndref}(e_1; e_2; P)$ and $\mathtt{rcvref}(e; x.P)$ are dynamic versions of the events $\mathtt{snd}[a](e; P)$ and $\mathtt{rcv}[a](x.P)$ in which the channel is determined dynamically by evaluation of an expression, rather than statically as a fixed parameter of the event.

The statics of channel references is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \&\, a : \tau\ \mathtt{chan}} \tag{42.21a}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau\ \mathtt{chan} \quad \Gamma \vdash_\Sigma e_2 : \tau \quad \Gamma \vdash_\Sigma P\ \mathsf{proc}}{\Gamma \vdash_\Sigma !!\,(e_1; e_2; P)\ \mathsf{event}} \tag{42.21b}$$



Preview

$$\frac{\Gamma \vdash_\Sigma e : \tau\ \mathtt{chan} \quad \Gamma, x : \tau \vdash_\Sigma P\ \mathsf{proc}}{\Gamma \vdash_\Sigma ??\,(e;x.P)\ \mathsf{event}} \tag{42.21c}$$

The introduction of channel references requires that events be evaluated to determine the referent of a dynamically determined channel. This is accomplished by adding the following rules for evaluation of a synchronizing process:

$$\overline{\$\,(!!\,(\&\,a;e;P) + E) \xmapsto[\Sigma, a\sim\tau]{} \$\,(!\,a(e;P) + E)} \tag{42.22a}$$

$$\overline{\$\,(??\,(\&\,a;x.P) + E) \xmapsto[\Sigma, a\sim\tau]{} \$\,(?\,a(x.P) + E)} \tag{42.22b}$$

In addition we require rules for evaluating each of the constituent expressions of a dynamically determined event; these rules are omitted here for the sake of concision.

## 42.7 Universality

In the presence of both channel references and recursive types the process calculus with communication is a universal programming language. One way to prove this is to show that it is capable of encoding the untyped $\lambda$-calculus with a call-by-name dynamics (see Chapter 17). The main idea of the encoding is to associate with each untyped $\lambda$-term, $u$, a process that represents it. This encoding is defined by induction on the structure of the untyped term, $u$. For the sake of the induction, the representation is defined relative to a channel reference that represents the context in which the term occurs. Because every term in the untyped $\lambda$-calculus is a function, the context consists of an *argument* and the *continuation* for the result of the application. Because of the by-name interpretation of application, variables are represented by references to "servers" that listen on a channel for a channel reference representing a call site, and activate their bindings with that channel reference.

We will write $u\,@\,z$, where $u$ is an untyped $\lambda$-term and $z$ is a channel reference representing the continuation of $u$. The free variables of $u$ will be represented by channels on which we may pass an argument and a continuation. Thus, the channel reference $z$ will be a value of type $\pi$, and a free variable, $x$, will be a value of type $\pi\,\mathtt{chan}$. The type $\pi$ is chosen to satisfy the isomorphism

$$\pi \cong (\pi\,\mathtt{chan} \times \pi)\,\mathtt{chan}.$$

That is, a continuation is a channel on which is passed an argument and another continuation. An argument, in turn, is a channel on which is passed a continuation.

The encoding of untyped $\lambda$-terms as processes is given by the following equations:

$$\begin{aligned}
x\,@\,z &= !!\,(x;z)\\
\lambda\,(x)\,u\,@\,z &= \$\,??\,(\mathtt{unfold}(z);\langle x, z'\rangle.u\,@\,z')\\
u_1(u_2)\,@\,z &= \\
&\qquad \nu\,a_1 \sim \pi\,\mathtt{chan} \times \pi.(u_1\,@\,\mathtt{fold}(\&\,a_1)) \parallel \nu\,a \sim \pi.*\,\$\,?\,a(z_2.u_2\,@\,z_2) \parallel !\,a_1(\langle \&\,a, z\rangle)
\end{aligned}$$



Preview

Here we have taken a few liberties with the syntax for the sake of readability. We use the asynchronous form of dynamic send operation, because there is no need to be aware of the receipt of the message. Moreover, we use a product pattern, rather than explicit projections, in the dynamic receive to obtain the components of a pair.

The use of static and dynamic communication operations in the translation merits careful examination. The call site of a $\lambda$-term is determined dynamically; we cannot predict at translation time the continuation of the term. In particular, the binding of a variable may be used at many different call sites, corresponding to the multiple possible uses of that variable. On the other hand the channel associated to an argument is determined statically. The server associated to the variable listens on a statically determined channel for a continuation in which to evaluate its binding, which, as just remarked, is determined dynamically.

As a quick check on the correctness of the representation, consider the following derivation:

$$\begin{aligned}(\lambda\,(x)\,x)(y)\,@\,z &\longmapsto^* \\ &\quad \nu\,a_1 \sim \tau.(\$\,?\,a_1(\langle x, z'\rangle.!!\,(x;z'))) \parallel \nu\,a \sim \pi.*\,\$\,?\,a(z_2.!!\,(y;z_2)) \parallel\, !\,a_1(\langle \&\,a, z\rangle) \\ &\longmapsto^* \nu\,a \sim \pi.*\,\$\,?\,a(z_2.!!\,(y;z_2)) \parallel\, !\,a(z) \\ &\longmapsto^* \nu\,a \sim \pi.*\,\$\,?\,a(z_2.!!\,(y;z_2)) \parallel\, !!\,(y;z)\end{aligned}$$

Apart from the idle server process listening on channel $a$, this is just the translation $y\,@\,z$. (Using the methods to be developed in detail in Chapter 51, we may show that the result of the computation step is "bisimilar" to the translation of $y\,@\,z$, and hence equivalent to it for all purposes.)

## 42.8 Notes

Process calculi as models of concurrency and interaction were introduced and extensively developed by Hoare (1978) and Milner (1999). Milner's original formulation, CCS, was introduced to model pure synchronization, whereas Hoare's, CSP, included value-passing. CCS was subsequently extended to become the $\pi$-calculus (Milner, 1999), which includes channel-passing. Dozens upon dozens of variations and extensions of CSP, CCS, and the $\pi$-calculus have been considered in the literature, and continue to be a subject of intensive study. (See Engberg and Nielsen (2000) for an account of some of the critical developments in the area.)

The process calculus given here is derived from the $\pi$-calculus as presented in Milner (1999). (In particular, the vending machine example is adapted from Milner's monograph.) Unlike Milner's account (but like related formalisms such as Abadi and Fournet (2001)) we enforce a distinction between variables and names. Variables are given meaning by substitution; a type is the range of significance of a variable, the collection of values that may be substituted for it. Names, on the other hand, are given meaning by the operations associated with them; the type associated with a name is the type of data associated with the operations defined on it. The distinction between variables and names is important, because disequality is well-defined for names, but not for variables. We use the general concept of a reference to pass channel names as data; this is sufficient to ensure the universality of the process calculus.

The distinction drawn here between static and dynamic events (that is, those that are given syntactically versus those that arise by evaluation) flows naturally from the prior distinction be-

Preview

tween variables and names. It is possible to formulate the process calculus so that all uses of names are suppressed, but then the dynamics cannot be expressed using only the machinery of the calculus itself, but instead must be augmented by an internal concept of names. It seems preferable, in the interest of maintaining a structural operational semantics, to work with a formalism that is closed under its own execution rules. The concept of dynamic events is taken one step further in Concurrent ML (Reppy, 1999), wherein events are values of an event type (see also Chapter 43).

Preview

# Chapter 43

# Concurrent Algol

In this chapter we integrate concurrency into the framework of Modernized Algol described in Chapter 36. The resulting language, called Concurrent Algol, **CA**, illustrates the integration of the mechanisms of the process calculus described in Chapter 42 into a practical programming language. To avoid distracting complications, we drop assignables from Modernized Algol entirely. (There is no loss of generality, however, because free assignables are definable in Concurrent Algol using processes as cells.)

The process calculus described in Chapter 42 is intended as a self-standing model of concurrent computation. When viewed in the context of a programming language, however, it is possible to streamline the machinery to take full advantage of types that are in any case required for other purposes. In particular the concept of a *channel*, which features prominently in Chapter 42, may be identified with the concept of a *dynamic class* as described in Chapter 35. More precisely, we take *broadcast communication* of dynamically classified values as the basic synchronization mechanism of the language. Being dynamically classified, messages consist of a *payload* tagged with a *class*, or *channel*. The type of the channel determines the type of the payload. Importantly, only those processes that have access to the channel may decode the message; all others must treat it as inscrutable data that may be passed around but not examined. In this way we can model not only the mechanisms described in Chapter 42, but also formulate an abstract account of encryption and decryption in a network using the methods described in Chapter 42.

The formulation of Concurrent Algol is based on a modal separation between commands and expressions, much as in Modernized Algol. It is also possible to consolidate these two levels (so as to allow benign concurrency effects), but we do not develop this approach in detail here.

Preview

## 43.1 Concurrent Algol

The syntax of **CA** is obtained by stripping out assignables from **MA**, and adding a syntactic level of *processes*:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{cmd}(\tau)$ | $\tau\,\mathtt{cmd}$ | commands |
| Exp | $e$ | $::=$ | $\mathtt{cmd}(m)$ | $\mathtt{cmd}\,m$ | command |
| Cmd | $m$ | $::=$ | $\mathtt{ret}\,e$ | $\mathtt{ret}\,e$ | return |
| | | | $\mathtt{bnd}(e; x.m)$ | $\mathtt{bnd}\,x \hookleftarrow e\,;m$ | sequence |
| Proc | $p$ | $::=$ | $\mathtt{stop}$ | $\mathbf{1}$ | idle |
| | | | $\mathtt{proc}(m)$ | $\mathtt{proc}(m)$ | atomic |
| | | | $\mathtt{par}(p_1; p_2)$ | $p_1 \parallel p_2$ | parallel |
| | | | $\mathtt{new}[\tau](a.p)$ | $\nu\, a \sim \tau.p$ | new channel |

The process $\mathtt{proc}(m)$ is an atomic process executing the command, $m$. The other forms of process are adapted from Chapter 42. If $\Sigma$ has the form $a_1 \sim \tau_1, \ldots, a_n \sim \tau_n$, then we sometimes write $\nu\,\Sigma\{p\}$ for the iterated form $\nu\, a_1 \sim \tau_1 \ldots \nu\, a_n \sim \tau_n.p$.

The statics is given by the judgments $\Gamma \vdash_\Sigma e : \tau$ and $\Gamma \vdash_\Sigma m \div \tau$ introduced in Chapter 36, augmented by the judgment $\vdash_\Sigma p\ \mathsf{proc}$ stating that $p$ is a well-formed process over the signature $\Sigma$. The latter judgment is defined by the following rules:

$$\frac{}{\vdash_\Sigma \mathbf{1}\ \mathsf{proc}} \tag{43.1a}$$

$$\frac{\vdash_\Sigma m \div \tau}{\vdash_\Sigma \mathtt{proc}(m)\ \mathsf{proc}} \tag{43.1b}$$

$$\frac{\vdash_\Sigma p_1\ \mathsf{proc} \quad \vdash_\Sigma p_2\ \mathsf{proc}}{\vdash_\Sigma p_1 \parallel p_2\ \mathsf{proc}} \tag{43.1c}$$

$$\frac{\vdash_{\Sigma, a\sim\tau} p\ \mathsf{proc}}{\vdash_\Sigma \nu\, a \sim \tau.p\ \mathsf{proc}} \tag{43.1d}$$

Processes are tacitly identified up to structural equivalence, as described in Chapter 42.

The transition judgment $p \overset{\alpha}{\underset{\Sigma}{\mapsto}} p'$ states that the process $p$ evolves in one step to the process $p'$ with associated action $\alpha$. The particular actions are specified when specific commands are introduced in Section 43.2. As in Chapter 42 we assume that to each action is associated a complementary action, and that the silent action indexes the unlabeled transition judgment.

$$\frac{m \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} \nu\,\Sigma'\,\{\, m' \parallel p \,\}}{\mathtt{proc}(m) \overset{\alpha}{\underset{\Sigma}{\mapsto}} \nu\,\Sigma'\{\mathtt{proc}(m') \parallel p\}} \tag{43.2a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{proc}(\mathtt{ret}\,e) \underset{\Sigma}{\mapsto} \mathbf{1}} \tag{43.2b}$$

Preview

$$\frac{p_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}} p_1'}{p_1 \parallel p_2 \overset{\alpha}{\underset{\Sigma}{\mapsto}} p_1' \parallel p_2} \tag{43.2c}$$

$$\frac{p_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}} p_1' \quad p_2 \overset{\overline{\alpha}}{\underset{\Sigma}{\mapsto}} p_2'}{p_1 \parallel p_2 \underset{\Sigma}{\mapsto} p_1' \parallel p_2'} \tag{43.2d}$$

$$\frac{p \overset{\alpha}{\underset{\Sigma, a \sim \tau}{\longmapsto}} p' \quad \vdash_\Sigma \alpha \text{ action}}{\nu\, a \sim \tau.p \overset{\alpha}{\underset{\Sigma}{\mapsto}} \nu\, a \sim \tau.p'} \tag{43.2e}$$

Rule (43.2a) states that a step of execution of the atomic process $\mathtt{proc}(m)$ consists of a step of execution of the command $m$, which may result in the allocation of some set, $\Sigma'$, of symbols and the creation of a concurrent process, $p$. This rule implements scope extrusion for classes (channels) by expanding the scope of the declaration of a channel to the context in which the command, $m$, occurs. Rule (43.2b) states that a completed command evolves to the inert (stopped) process; processes are executed solely for their effect, and not for their value. The remaining rules are those of the process calculus that define the interaction between processes and the allocation of symbols within a process.

The auxiliary judgment $m \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} \nu\, \Sigma' \{ m' \parallel p' \}$ defines the execution behavior of commands. It states that the command, $m$, transitions to the command, $m'$, while creating new channels, $\Sigma'$, and new processes, $p'$. The action, $\alpha$, specifies the interactions of which $m$ is capable when executed. As a notational convenience we drop mention of the new channels or processes when either are trivial. It is important that the right-hand side of this judgment be construed as a triple consisting of $\Sigma'$, $m'$, and $p'$, rather than as a process expression comprising these parts.

The general rules defining this auxiliary judgment are as follows:

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{ret}\, e \overset{\varepsilon}{\underset{\Sigma}{\Rightarrow}} \mathtt{ret}\, e'} \tag{43.3a}$$

$$\frac{m_1 \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} \nu\, \Sigma' \{ m_1' \parallel p' \}}{\mathtt{bnd}\, x \hookrightarrow \mathtt{cmd}\, m_1 \,;\, m_2 \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} \nu\, \Sigma' \{ \mathtt{bnd}\, x \hookrightarrow \mathtt{cmd}\, m_1' \,;\, m_2 \parallel p' \}} \tag{43.3b}$$

$$\frac{e \text{ val}_\Sigma}{\mathtt{bnd}\, x \hookrightarrow \mathtt{cmd}\, \mathtt{ret}\, e \,;\, m_2 \overset{\varepsilon}{\underset{\Sigma}{\Rightarrow}} [e/x] m_2} \tag{43.3c}$$

$$\frac{e_1 \underset{\Sigma}{\mapsto} e_1'}{\mathtt{bnd}\, x \hookrightarrow e_1 \,;\, m_2 \overset{\varepsilon}{\underset{\Sigma}{\Rightarrow}} \mathtt{bnd}\, x \hookrightarrow e_1' \,;\, m_2} \tag{43.3d}$$

These generic rules are supplemented by rules governing commands for communication and synchronization among processes.

Preview

## 43.2 Broadcast Communication

In this section we consider a very general form of process synchronization called *broadcast*. Processes emit and accept messages of type `clsfd`, the type of dynamically classified values considered in Chapter 35. A message consists of a *channel*, which is its class, and a *payload*, which is a value of the type associated with the channel (class). Recipients may pattern match against a message to determine whether it is of a given class, and, if so, recover the associated payload. No process that lacks access to the class of a message may recover the payload of that message. (See Section 35.4.1 for a discussion of how to enforce confidentiality and integrity restrictions using dynamic classification).

The syntax of the commands pertinent to broadcast communication is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Cmd | $m$ | $::=$ | $\mathtt{spawn}(e)$ | $\mathtt{spawn}(e)$ | spawn |
| | | | $\mathtt{emit}(e)$ | $\mathtt{emit}(e)$ | emit message |
| | | | $\mathtt{acc}$ | $\mathtt{acc}$ | accept message |
| | | | $\mathtt{newch}[\tau]$ | $\mathtt{newch}$ | new channel |

The command $\mathtt{spawn}(e)$ spawns a process that executes the encapsulated command given by $e$. The commands $\mathtt{emit}(e)$ and $\mathtt{acc}$ emit and accept messages, which are classified values whose class is the channel on which the message is sent. The command $\mathtt{newch}[\tau]$ returns a reference to a fresh class carrying values of type $\tau$.

The statics of broadcast communication is given by the following rules:

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{cmd}(\mathtt{unit})}{\Gamma \vdash_\Sigma \mathtt{spawn}(e) \div \mathtt{unit}} \tag{43.4a}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{clsfd}}{\Gamma \vdash_\Sigma \mathtt{emit}(e) \div \mathtt{unit}} \tag{43.4b}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathtt{acc} \div \mathtt{clsfd}} \tag{43.4c}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathtt{newch}[\tau] \div \mathtt{class}(\tau)} \tag{43.4d}$$

The execution of commands for broadcast communication is defined by these rules:

$$\frac{}{\mathtt{spawn}(\mathtt{cmd}(m)) \overset{\varepsilon}{\underset{\Sigma}{\Rightarrow}} \mathtt{ret}\,\langle\rangle \parallel \mathtt{proc}(m)} \tag{43.5a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{spawn}(e) \overset{\varepsilon}{\underset{\Sigma}{\Rightarrow}} \mathtt{spawn}(e')} \tag{43.5b}$$

Preview

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{emit}(e) \overset{e!}{\underset{\Sigma}{\Longrightarrow}} \mathtt{ret}\ \langle\rangle} \tag{43.5c}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\mathtt{emit}(e) \overset{\varepsilon}{\underset{\Sigma}{\Longrightarrow}} \mathtt{emit}(e')} \tag{43.5d}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{acc} \overset{e?}{\underset{\Sigma}{\Longrightarrow}} \mathtt{ret}\, e} \tag{43.5e}$$

$$\frac{}{\mathtt{newch}[\tau] \overset{\varepsilon}{\underset{\Sigma}{\Longrightarrow}} \nu\, a \sim \tau.\mathtt{ret}\ (\&\, a)} \tag{43.5f}$$

Rule (43.5c) specifies that $\mathtt{emit}(e)$ has the effect of emitting the message $e$. Correspondingly, rule (43.5e) specifies that `acc` may accept (any) message that is being sent.

As usual, the preservation theorem for **CA** ensures that well-typed programs remain well-typed during execution. The proof of preservation requires a lemma governing the execution of commands. First, let us define the judgment $\vdash_\Sigma \alpha\ \mathsf{action}$ by the following rules:

$$\frac{}{\vdash_\Sigma \varepsilon\ \mathsf{action}} \tag{43.6a}$$

$$\frac{\vdash_\Sigma e : \mathtt{clsfd}}{\vdash_\Sigma e!\ \mathsf{action}} \tag{43.6b}$$

$$\frac{\vdash_\Sigma e : \mathtt{clsfd}}{\vdash_\Sigma e?\ \mathsf{action}} \tag{43.6c}$$

**Lemma 43.1.** *If $m \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} \nu\, \Sigma'\, \{\, m' \parallel p' \,\}$, $\vdash_\Sigma m \div \tau$, and $\vdash_\Sigma \alpha$ action, then $\vdash_{\Sigma\Sigma'} m' \div \tau$, and $\vdash_{\Sigma\Sigma'} p'$ proc.*

*Proof.* By induction on rules (43.3). □

With this in hand the proof of preservation proceeds along familiar lines.

**Theorem 43.2** (Preservation)**.** *If $\vdash_\Sigma p$ proc and $p \underset{\Sigma}{\longmapsto} p'$, then $\vdash_\Sigma p'$ proc.*

*Proof.* By induction on transition, appealing to Lemma 43.1 for the crucial steps. □

Typing does not, however, guarantee progress with respect to unlabeled transition, for the simple reason that there may be no other process with which to communicate. By extending progress to labeled transitions we may state that this is the *only* way for the execution of a process to get stuck. But some care must be taken to account for the allocation of new channels.

**Theorem 43.3** (Progress)**.** *If $\vdash_\Sigma p$ proc, then either $p \equiv \mathbf{1}$, or $p \equiv \nu\, \Sigma'\{p'\}$ such that $p' \overset{\alpha}{\underset{\Sigma\Sigma'}{\longmapsto}} p''$ for some $\vdash_{\Sigma\Sigma'} p''$ and some $\vdash_{\Sigma\Sigma'} \alpha$ action.*



Preview

*Proof.* By induction on rules (43.1) and (43.4). □

The statement of progress amounts to saying that no process may become stuck for reasons other than its inability to communicate with another process. For example, a process that receives on a channel for which there is no sender is "stuck", but this is not regarded as a violation of progress because the type system does not attempt to rule out such situations.

## 43.3 Selective Communication

Broadcast communication provides no means of restricting acceptance to messages of a particular class (that is, of messages on a particular channel). Using broadcast communication we may restrict attention to a particular channel, $a$, of type, $\tau$, by running the following command:

$$\mathtt{fix}\, loop : \tau\, \mathtt{cmd}\, \mathtt{is}\, \{x \hookrightarrow \mathtt{acc}\,;\mathtt{match}\, x\, \mathtt{as}\, a \cdot y \hookrightarrow \mathtt{ret}\, y\, \mathtt{ow} \hookrightarrow \mathtt{emit}(x)\,;\mathtt{do}\, loop\}$$

This command is always capable of receiving a broadcast message. When one arrives, it is examined to determine whether it is classified by the class, $a$. If so, the underlying value is returned; otherwise the message is re-broadcast to make it available to another process that may be executing a similar command. *Polling* consists of repeatedly executing the above command until such time as a message of channel $a$ is successfully accepted, if ever.

Polling is evidently impractical in most situations. An alternative is to change the language to allow for *selective communication*. Rather than accept any broadcast message, we may confine attention to messages that are sent on any of several possible channels. This may be accomplished by introducing a type, $\mathtt{event}(\tau)$, of *events* consisting of a finite choice of accepts, all of whose associated payload has the type $\tau$.

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{event}(\tau)$ | $\tau\,\mathtt{event}$ | events |
| Exp | $e$ | $::=$ | $\mathtt{rcv}[a]$ | $?\,a$ | select |
| | | | $\mathtt{never}[\tau]$ | $\mathtt{never}$ | null |
| | | | $\mathtt{or}(e_1; e_2)$ | $e_1\,\mathtt{or}\,e_2$ | choice |
| Cmd | $m$ | $::=$ | $\mathtt{sync}(e)$ | $\mathtt{sync}(e)$ | synchronize |

Events in **CA** correspond directly to those of the *asynchronous* process calculus described in Chapter 42. One difference is that the select event need not carry with it a continuation, as it does in the process calculus; this is handled by the ambient modal structure of commands. (However, note that all events in a choice share the same continuation, whereas in process calculus a separate continuation is associated to each event in a choice.) Another difference between the two formalisms is that in **CA** events are values of the type $\tau\,\mathtt{event}$, whereas in the process calculus events are not regarded as a form of expression.

The statics of event expressions is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{rcv}[a] : \mathtt{event}(\tau)} \tag{43.7a}$$



Preview

$$\frac{}{\Gamma \vdash_\Sigma \mathtt{never}[\tau] : \mathtt{event}(\tau)} \tag{43.7b}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathtt{event}(\tau) \quad \Gamma \vdash_\Sigma e_2 : \mathtt{event}(\tau)}{\Gamma \vdash_\Sigma \mathtt{or}(e_1; e_2) : \mathtt{event}(\tau)} \tag{43.7c}$$

The corresponding dynamics is defined by these rules:

$$\frac{}{\mathtt{rcv}[a]\ \mathsf{val}_{\Sigma, a \sim \tau}} \tag{43.8a}$$

$$\frac{}{\mathtt{never}[\tau]\ \mathsf{val}_\Sigma} \tag{43.8b}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad e_2\ \mathsf{val}_\Sigma}{\mathtt{or}(e_1; e_2)\ \mathsf{val}_\Sigma} \tag{43.8c}$$

$$\frac{e_1 \underset{\Sigma}{\mapsto} e_1'}{\mathtt{or}(e_1; e_2) \underset{\Sigma}{\mapsto} \mathtt{or}(e_1'; e_2)} \tag{43.8d}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad e_2 \underset{\Sigma}{\mapsto} e_2'}{\mathtt{or}(e_1; e_2) \underset{\Sigma}{\mapsto} \mathtt{or}(e_1; e_2')} \tag{43.8e}$$

Event values are identified up to structural congruence as described in Chapter 42. This ensures that the ordering of events in a choice is immaterial.

Channel references (see Section 35.2) give rise to an additional form of event, $\mathtt{rcvref}(e)$, in which the argument, $e$, is a reference to the channel on which to accept a message. Its statics is given by the rule

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{class}(\tau)}{\Gamma \vdash_\Sigma \mathtt{rcvref}(e) : \mathtt{event}(\tau)} \tag{43.9}$$

Its dynamics is defined to dereference its argument and evaluate to an accept event for the referenced channel:

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{rcvref}(e) \underset{\Sigma}{\mapsto} \mathtt{rcvref}(e')} \tag{43.10a}$$

$$\frac{}{\mathtt{rcvref}(\&a) \underset{\Sigma, a \sim \tau}{\longmapsto} \mathtt{rcv}[a]} \tag{43.10b}$$

Turning now to the synchronization command, the statics is given by the following rule:

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{event}(\tau)}{\Gamma \vdash_\Sigma \mathtt{sync}(e) \div \tau} \tag{43.11a}$$

Preview

Its execution is defined by these rules:

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\mathtt{sync}(e) \underset{\Sigma}{\overset{\varepsilon}{\Longrightarrow}} \mathtt{sync}(e')} \tag{43.12a}$$

$$\frac{e \underset{\Sigma}{\overset{\alpha}{\Longrightarrow}} m}{\mathtt{sync}(e) \underset{\Sigma}{\overset{\alpha}{\Longrightarrow}} m} \tag{43.12b}$$

Rule (43.12b) specifies that synchronization may take any action engendered by the event given as argument.

The possible actions engendered by an event value are defined by the judgment $e \underset{\Sigma}{\overset{\alpha}{\Longrightarrow}} m$, which states that the event value $e$ engenders action $\alpha$ and activates command $m$. It is defined by the following rules:

$$\frac{e\ \mathsf{val}_{\Sigma,a\sim\tau} \quad \vdash_{\Sigma,a\sim\tau} e : \tau}{\mathtt{rcv}[a] \underset{\Sigma,a\sim\tau}{\overset{a\cdot e\,?}{\Longrightarrow}} \mathtt{ret}(e)} \tag{43.13a}$$

$$\frac{e_1 \underset{\Sigma}{\overset{\alpha}{\Longrightarrow}} m_1}{\mathtt{or}(e_1; e_2) \underset{\Sigma}{\overset{\alpha}{\Longrightarrow}} m_1} \tag{43.13b}$$

Rule (43.13a) states that an acceptance on a channel $a$ may synchronize only with messages classified by $a$. In conjunction with the identification of event values up to structural congruence. Rule (43.13b) states that any event among a set of choices may engender an action.

Selective communication and dynamic events may be used together to implement a communication protocol in which a channel reference is passed on a channel in order to establish a communication path with the recipient. Let $a$ be a channel carrying values of type $\mathtt{class}(\tau)$, and let $b$ be a channel carrying values of type $\tau$, so that $\&\, b$ may be passed as a message along channel $a$. A process that wishes to accept a channel reference on $a$ and then accept on that channel has the form

$$\{x \hookrightarrow \mathtt{sync}(?\, a)\,;\, y \hookrightarrow \mathtt{sync}(??\, x)\,;\, \ldots\}.$$

The event $?\, a$ specifies a selective receipt on channel $a$. Once the value, $x$, has been accepted, the event $??\, x$ specifies a selective receipt on the channel referenced by $x$. So, if $\&\, b$ is sent along $a$, then the event $??\, \&\, b$ evaluates to $?\, b$, which accepts selectively on channel $b$, even though the receiving process may have no direct access to the channel $b$ itself.

Selective communication may be seen as a simple form of pattern matching in which patterns are restricted to $a \cdot x$, where $a$ is a channel carrying values of some type $\tau$, and $x$ is a variable of type $\tau$. The idea is that selective communication filters for messages that match a pattern of this form, and proceeds by returning the associated value, $x$. From this point of view it is natural to generalize selective communication to allow arbitrary patterns of type $\mathtt{clsfd}$. Because different patterns may bind different variables, it is then natural to associate a separate continuation with

Preview

each pattern, as in Chapter 12. Basic events are of the form $p \hookrightarrow m$, where $p$ is a pattern of type $\texttt{clsfd}$, $x_1, \ldots, x_k$ are its variables, and $m$ is a command involving these variables. Compound events are compositions of such rules, written $r_1 \mid \ldots \mid r_n$, quotiented by structural congruence to ensure that the order of rules is insignificant.

The statics of pattern-driven events may be readily derived from the statics of pattern matching given in Chapter 12. The dynamics is defined by the following rule defining the action engendered by an event:

$$\frac{e\ \mathsf{val}_\Sigma \quad \vdash_\Sigma e : \texttt{clsfd} \quad \theta \Vdash p \triangleleft e}{p \hookrightarrow m \mid rs \xRightarrow[\Sigma]{e?} \hat{\theta}(m)} \tag{43.14}$$

This rule states that we may choose any accept action by a value matching the pattern $p$, continuing with the corresponding instance of the continuation of the rule.

## 43.4 Free Assignables as Processes

Scope-free assignables are definable in **CA** by associating to each assignable a server process that sets and gets the contents of the assignable. To each assignable, $a$, of type $\tau$ is associated a server that selectively accepts a message on channel $a$ with one of two forms:

1. $\texttt{get} \cdot (\& b)$, where $b$ is a channel of type $\tau$. This message requests that the contents of $a$ be sent on channel $b$.
2. $\texttt{set} \cdot (\langle e, \& b \rangle)$, where $e$ is a value of type $\tau$, and $b$ is a channel of type $\tau$. This message requests that the contents of $a$ be set to $e$, and that the new contents be transmitted on channel $b$.

In other words, $a$ is a channel of type $\tau_{srvr}$ given by

$$[\texttt{get} \hookrightarrow \tau\ \texttt{class}, \texttt{set} \hookrightarrow \tau \times \tau\ \texttt{class}].$$

The server selectively accepts on channel $a$, then dispatches on the class of the message to satisfy the request.

The server associated with the assignable, $a$, of type $\tau$ maintains the contents of $a$ using recursion. When called with the current contents of the assignable, the server selectively accepts on channel $a$, dispatching on the associated request, and calling itself recursively with the (updated, if necessary) contents:

$$\lambda\,(u{:}\tau_{srvr}\ \texttt{class})\ \texttt{fix}\, srvr : \tau \rightharpoonup \texttt{void}\ \texttt{cmd}\ \texttt{is}\ \lambda\,(x{:}\tau)\ \texttt{cmd}\,\{y \hookrightarrow \texttt{sync}(??\,u)\,;\, e_{(43.16)}\}. \tag{43.15}$$

The server is a procedure that takes an argument of type $\tau$, the current contents of the assignable, and yields a command that never terminates, because it restarts the server loop after each request. The server selectively accepts a message on channel $a$, and dispatches on it as follows:

$$\texttt{case}\, y\,\{\texttt{get} \cdot z \hookrightarrow e_{(43.17)} \mid \texttt{set} \cdot \langle x', z \rangle \hookrightarrow e_{(43.18)}\}. \tag{43.16}$$

A request to get the contents of the assignable $a$ is served as follows:

$$\{_ \hookrightarrow \texttt{emit}(\texttt{mk}(z; x))\,;\texttt{do}\, srvr(x)\} \tag{43.17}$$



Preview

A request to set the contents of the assignable $a$ is served as follows:

$$\{_ \hookrightarrow \mathtt{emit}(\mathtt{mk}(z; x'))\,;\mathtt{do}\,srvr(x')\} \tag{43.18}$$

The type $\tau\,\mathtt{ref}$ is defined to be $\tau_{srvr}\,\mathtt{class}$, the type of channels (classes) to servers providing a cell containing a value of type $\tau$. A new free assignable is created by the command $\mathtt{ref}\,e_0$, which is defined to be

$$\{x \hookrightarrow \mathtt{newch}\,;_ \hookrightarrow \mathtt{spawn}(e_{(43.15)}(x)(e_0))\,;\mathtt{ret}\,x\}. \tag{43.19}$$

A channel carrying a value of type $\tau_{srvr}$ is allocated to serve as the name of the assignable, and a new server is spawned that accepts requests on that channel, with initial value $e_0$ of type $\tau_0$.

The commands $*\,e_0$ and $e_0 *= e_1$ send a message to the server to get and set the contents of an assignable. The code for $*\,e_0$ is as follows:

$$\{x \hookrightarrow \mathtt{newch}\,;_ \hookrightarrow \mathtt{emit}(\mathtt{mk}(e_0; \mathtt{get} \cdot x))\,;\mathtt{sync}(??\,(x))\} \tag{43.20}$$

A channel is allocated for the return value, the server is contacted with a `get` message specifying this channel, and the result of receiving on this channel is returned. Similarly, the code for $e_0 *= e_1$ is as follows:

$$\{x \hookrightarrow \mathtt{newch}\,;_ \hookrightarrow \mathtt{emit}(\mathtt{mk}(e_0; \mathtt{set} \cdot \langle e_1, x\rangle))\,;\mathtt{sync}(??\,(x))\} \tag{43.21}$$

## 43.5 Notes

Concurrent Algol is a synthesis of process calculus and Modernized Algol, and may be seen as an "Algol-like" formulation of Concurrent ML (Reppy, 1999) in which interaction is confined to the command modality. The design is influenced by Parallel Algol (Brookes, 2002). The reduction of channels to dynamic classification appears to be new. Most work on concurrent interaction seems to take the notion of communication channel as a central concept (but see Linda (Gelernter, 1985) for an alternative viewpoint, albeit in a unityped setting).

Preview

# Chapter 44

# Distributed Algol

A *distributed* computation is one that takes place at many different *sites*, each of which controls some *resources* located at that site. For example, the sites might be nodes on a network, and a resource might be a device or sensor located at that site, or a database controlled by that site. Only programs that execute at a particular site may access the resources situated at that site. Consequently, command execution always takes place at a particular site, called the *locus of execution*. Access to resources at a remote site from a local site is achieved by moving the locus of execution to the remote site, running code to access the local resource, and returning a value to the local site.

In this chapter we consider the language **DA**, an extension of Concurrent Algol with a *spatial* type system that mediates access to located resources on a network. The type safety theorem ensures that all accesses to a resource controlled by a site are through a program executing at that site, even though references to local resources may be freely passed around to other sites on the network. The key idea is that channels and events are *located* at a particular site, and that synchronization on an event may only occur at the site appropriate to that event. Issues of concurrency, which are to do with non-deterministic composition, are thereby cleanly separated from those of distribution, which are to do with the locality of resources on a network.

The concept of location in **DA** is sufficiently abstract that it admits another useful interpretation that can be useful in computer security settings. The "location" of a computation may also be thought of as the *principal* on whose behalf the computation is executing. From this point of view, a local resource is one that is accessible to a particular principal, and a mobile computation is one that may be executed by any principal. Movement from one location to another may then be interpreted as executing a piece of code on behalf of another principal, returning its result to the principal that initiated the transfer.

## 44.1 Statics

The statics of **DA** is inspired by the *possible worlds* interpretation of modal logic. Under that interpretation the truth of a proposition is relative to a *world*, which determines the state of affairs described by that proposition. A proposition may be true in one world, and false in another. For

Preview

example, one may use possible worlds to model counterfactual reasoning, in which one postulates that certain facts that happen to be true in this, the *actual*, world, might be otherwise in some other, *possible*, world. For instance, in the actual world you, the reader, are reading this book, but in a possible world you may never have taken up the study of programming languages at all. Of course not everything is possible: there is no possible world in which $2 + 2$ is other than 4, for example. Moreover, once a commitment has been made to one counterfactual, others are ruled out. We say that one world is *accessible* from another when the first is a sensible counterfactual relative to the first. So, for example, one may consider that relative to a possible world in which you are the king, there is no further possible world in which someone else is also the king (there being only one sovereign).

In **DA** we shall interpret possible worlds as sites on a network, with accessibility between worlds expressing network connectivity. We postulate that every site is connected to itself (reflexivity); that if one site is reachable from another, then the second is also reachable from the first (symmetry); and that if a site is reachable from a reachable site, then this site is itself reachable from the first (transitivity). From the point of view of modal logics, the type system of **DA** is derived from the logic **S5**, for which accessibility is an equivalence relation.

The syntax of **DA** is a modification of that of **CA**. The following grammar summarizes the key changes:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | $::=$ | $\mathtt{cmd}[w](\tau)$ | $\tau\,\mathtt{cmd}[w]$ | commands |
| | | | $\mathtt{chan}[w](\tau)$ | $\tau\,\mathtt{chan}[w]$ | channels |
| | | | $\mathtt{event}[w](\tau)$ | $\tau\,\mathtt{event}[w]$ | events |
| Cmd | $m$ | $::=$ | $\mathtt{at}[w](m)$ | $\mathtt{at}\,w\,\{m\}$ | change site |

The command, channel, and event types are indexed by the site, $w$, to which they pertain. There is a new form of command, $\mathtt{at}[w](m)$, that changes the locus of execution from one site to another.

A signature, $\Sigma$, in **DA**, consists of a finite set of declarations of the form $a \sim \tau \mathbin{@} w$, where $\tau$ is a type and $w$ is a site. Such a declaration specifies that $a$ is a channel carrying a payload of type $\tau$ located at the site $w$. We may think of a signature, $\Sigma$, as a family of signatures, $\Sigma_w$, one for each world $w$, containing the decalarations of the channels located at that world. This partitioning corresponds to the idea that channels are *located* resources in that they are uniquely associated with a site. They may be handled passively at other sites, but their only active role is at the site at which they are declared.

The statics of **DA** is given by the following two judgment forms:

| | |
|---|---|
| $\Gamma \vdash_\Sigma e : \tau$ | expression typing |
| $\Gamma \vdash_\Sigma m \div \tau \mathbin{@} w$ | command typing |

The expression typing judgment is independent of the site. This corresponds to the idea that the values of a type have a site-independent meaning: the number 3 is the number 3, regardless of where it is used. On the other hand commands can only be executed at a particular site, because they depend on the state located at that site.

A representative selection of the rules defining the statics of **DA** is given below:

$$\frac{\Gamma \vdash_\Sigma m \div \tau \mathbin{@} w}{\Gamma \vdash_\Sigma \mathtt{cmd}(m) : \mathtt{cmd}[w](\tau)} \tag{44.1a}$$



Preview

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau @ w} \mathtt{ch}[a] : \mathtt{chan}[w](\tau)} \tag{44.1b}$$

$$\frac{}{\Gamma \vdash_{\Sigma} \mathtt{never}[\tau] : \mathtt{event}[w](\tau)} \tag{44.1c}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau @ w} \mathtt{rcv}[a] : \mathtt{event}[w](\tau)} \tag{44.1d}$$

$$\frac{\Gamma \vdash_{\Sigma} e : \mathtt{chan}[w](\tau)}{\Gamma \vdash_{\Sigma} \mathtt{rcvref}(e) : \mathtt{event}[w](\tau)} \tag{44.1e}$$

$$\frac{\Gamma \vdash_{\Sigma} e_1 : \mathtt{event}[w](\tau) \quad \Gamma \vdash_{\Sigma} e_2 : \mathtt{event}[w](\tau)}{\Gamma \vdash_{\Sigma} \mathtt{or}(e_1; e_2) : \mathtt{event}[w](\tau)} \tag{44.1f}$$

$$\frac{\Gamma \vdash_{\Sigma} e : \mathtt{event}[w](\tau)}{\Gamma \vdash_{\Sigma} \mathtt{sync}(e) \div \tau \mathbin{@} w} \tag{44.1g}$$

$$\frac{\Gamma \vdash_{\Sigma} m' \div \tau' \mathbin{@} w'}{\Gamma \vdash_{\Sigma} \mathtt{at}[w'](m') \div \tau' \mathbin{@} w} \tag{44.1h}$$

Rule (44.1a) states that the type of an encapsulated command records the site at which the command is to be executed. Rules (44.1d) and (44.1e) specify that the type of a (static or dynamic) receive event records the site at which the channel resides. Rules (44.1c) and (44.1f) state that a choice can only be made between events at the same site; there are no cross-site choices. Rule (44.1g) states that the `sync` command returns a value of the same type as that of the event, and may be executed only at the site to which the given event pertains. Finally, rule (44.1h) states that to execute a command at a site, $w'$, requires that the command pertain to that site. The returned value is then passed to the original site.

## 44.2 Dynamics

The dynamics is given by a labeled transition judgment between processes, much as in Chapter 43. The principal difference is that the atomic process consisting of a single command has the form $\mathtt{proc}[w](m)$, which specifies the site, $w$, at which the command, $m$, is to be executed. The dynamics of processes remains much as in Chapter 43, except for the following rules governing the atomic process:

$$\frac{m \xrightarrow[\Sigma, w]{\alpha} \nu\, \Sigma' \{ m' \parallel p \}}{\mathtt{proc}[w](m) \xmapsto[\Sigma]{\alpha} \nu\, \Sigma' \{ \mathtt{proc}[w](m') \parallel p \}} \tag{44.2a}$$

$$\frac{}{\mathtt{proc}[w](\mathtt{ret}(\langle\rangle)) \xmapsto[\Sigma]{\varepsilon} \mathtt{stop}} \tag{44.2b}$$



Preview

The command execution judgment

$$m \xRightarrow[\Sigma, w]{\alpha} \nu\, \Sigma' \{\, m' \parallel p \,\}$$

states that the command, $m$, when executed at site, $w$, may undertake the action, $\alpha$, and in the process create new channels, $\Sigma'$, and a new process, $p$. (The result of the transition is not a process expression, but rather should be construed as a structure having three parts, the newly allocated channels, the newly created processes, and a new command; we omit any part when it is trivial.) This may be understood as a family of judgments indexed by signatures, $\Sigma$, and sites, $w$. At each site there is an associated labeled transition system defining concurrent interaction of processes *at that site*.

The command execution judgment is defined by the following rules:

$$\frac{}{\mathtt{spawn}(\mathtt{cmd}(m)) \xRightarrow[\Sigma, w]{\varepsilon} \mathtt{ret}(\langle\rangle) \parallel \mathtt{proc}[w](m)} \tag{44.3a}$$

$$\frac{}{\mathtt{newch}[\tau] \xRightarrow[\Sigma, w]{\varepsilon} \nu\, a \sim \tau \mathbin{@} w.\mathtt{ret}\,(\&\, a)} \tag{44.3b}$$

$$\frac{m \xRightarrow[\Sigma, w']{\alpha} \nu\, \Sigma' \{\, m' \parallel p' \,\}}{\mathtt{at}[w'](m) \xRightarrow[\Sigma, w]{\alpha} \nu\, \Sigma' \{\, \mathtt{at}[w'](m') \parallel p' \,\}} \tag{44.3c}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{at}[w'](\mathtt{ret}(e)) \xRightarrow[\Sigma, w]{\varepsilon} \mathtt{ret}(e)} \tag{44.3d}$$

$$\frac{e \overset{\alpha}{\Rightarrow}_\Sigma m}{\mathtt{sync}(e) \xRightarrow[\Sigma, w]{\alpha} m} \tag{44.3e}$$

Rule (44.3a) states that new processes created at a site remain at that site—the new process executes the given command at the current site. Rules (44.3c) and (44.3d) state that the command $\mathtt{at}[w'](m)$ is executed at site $w$ by executing $m$ at site $w'$, and returning the result to the site $w$. Rule (44.3e) states that an action may be undertaken at site $w$ if the given event engenders that action. Notice that no cross-site synchronization is possible. Movement between sites is handled separately from synchronization among the processes at a site.

## 44.3 Safety

The safety theorem for **DA** ensures that synchronization on a channel may only occur at the site on which the channel resides, even though channel references may be propagated from one site



Preview

to another during a computation. By the time the reference is resolved and synchronization is attempted the computation will, as a consequence of typing, be located at the appropriate site.

The key to the safety proof is the definition of a well-formed process. The judgment $\vdash_\Sigma p$ proc states that the process $p$ is well-formed. Most importantly, the following rule governs the formation of atomic processes:

$$\frac{\vdash_\Sigma m \div \mathtt{unit} \mathbin{@} w}{\vdash_\Sigma \mathtt{proc}[w](m)\ \mathsf{proc}} \tag{44.4}$$

That is, an atomic process is well-formed if and only if the command it is executing is well-formed at the site at which the process is located.

The proof of preservation relies on a lemma stating the typing properties of the execution judgment.

**Lemma 44.1** (Execution). *Suppose that* $m \xrightarrow[\Sigma,w]{\alpha} \nu\,\Sigma'\,\{\, m' \parallel p \,\}$. *If* $\vdash_\Sigma m \div \tau \mathbin{@} w$, *then* $\vdash_\Sigma \alpha$ *action and* $\vdash_\Sigma \nu\,\Sigma\{\mathtt{proc}[w](m') \parallel p\}$ *proc.*

*Proof.* By induction on rules (44.3). □

**Theorem 44.2** (Preservation). *If* $p \xmapsto[\Sigma]{\alpha} p'$ *and* $\vdash_\Sigma p$ *proc, then* $\vdash_\Sigma p'$ *proc.*

*Proof.* By induction on rules (44.1), appealing to Lemma 44.1 for atomic processes. □

The progress theorem states that the only impediment to execution of a well-typed program is the possiblity of synchronizing on an event that will never arise.

**Theorem 44.3** (Progress). *If* $\vdash_\Sigma p$ *proc, then either* $p \equiv \mathbf{1}$ *or there exists* $\alpha$ *and* $p'$ *such that* $p \xmapsto[\Sigma]{\alpha} p'$.

## 44.4 Situated Types

The foregoing formulation of **DA** relies on indexing command, channel, and event types by the site to which they pertain so that values of these types may be passed around at will without fear of misinterpretation. The price to pay, however, is that the command, channel, and event types are indexed by the site to which they pertain, leading to repetition and redundancy. One way to mitigate this cost is to separate out the *skeleton*, $\phi$, of a type from its *specialization* to a particular site, $w$, which is written $\phi\langle w\rangle$.

We will now reformulate the statics of **DA** using judgments of the form $\Phi \vdash_\Sigma e : \phi \mathbin{@} w$ and $\Phi \vdash_\Sigma m \div \phi \mathbin{@} w$, where $\Phi$ consists of hypotheses of the form $x_i : \phi_i \mathbin{@} w_i$. The type of an expression or command is factored into two parts, the skeleton, $\phi$, and a site, $w$, at which to specialize it. The meaning of the factored judgments is captured by the following conditions:

1. If $\Phi \vdash_\Sigma e : \phi \mathbin{@} w$, then $\widehat{\Phi} \vdash_\Sigma e : \phi\langle w\rangle$.
2. If $\Phi \vdash_\Sigma m \div \phi \mathbin{@} w$, then $\widehat{\Phi} \vdash_\Sigma m \div \phi\langle w\rangle \mathbin{@} w$.



Preview

If $\Phi$ is a context of the form $x_1 : \phi_1 \mathbin{@} w_1, \ldots, x_n : \phi_n \mathbin{@} w_n$, then $\widehat{\Phi}$ is the context $x_1 : \phi_1\langle w_1\rangle, \ldots, x_n : \phi_n\langle w_n\rangle$.

The syntax of skeletons is similar to that for types, with the addition of a means of specializing a skeleton to a particular site.

| | | | | | |
|---|---|---|---|---|---|
| Fam | $\phi$ | ::= | $\mathtt{nat}$ | $\mathtt{nat}$ | numbers |
| | | | $\mathtt{parr}(\phi_1;\phi_2)$ | $\phi_1 \rightharpoonup \phi_2$ | functions |
| | | | $\mathtt{cmd}(\phi)$ | $\phi\,\mathtt{cmd}$ | computations |
| | | | $\mathtt{chan}(\phi)$ | $\phi\,\mathtt{chan}$ | channels |
| | | | $\mathtt{event}(\phi)$ | $\phi\,\mathtt{event}$ | events |
| | | | $\mathtt{at}[w](\phi)$ | $\phi\,\mathtt{at}\,w$ | situated |

The situated type $\phi\,\mathtt{at}\,w$ fixes the interpretation of $\phi$ at the site $w$.

The instantiation of a family, $\phi$, at a site, $w$, is written $\phi\langle w\rangle$, and is inductively defined by the following rules:

$$\frac{}{\mathtt{nat}\langle w\rangle = \mathtt{nat}} \tag{44.5a}$$

$$\frac{\phi_1\langle w\rangle = \tau_1 \quad \phi_2\langle w\rangle = \tau_2}{(\phi_1 \rightharpoonup \phi_2)\langle w\rangle = \tau_1 \rightharpoonup \tau_2} \tag{44.5b}$$

$$\frac{\phi\langle w\rangle = \tau}{\phi\,\mathtt{cmd}\langle w\rangle = \tau\,\mathtt{cmd}[w]} \tag{44.5c}$$

$$\frac{\phi\langle w\rangle = \tau}{\phi\,\mathtt{chan}\langle w\rangle = \tau\,\mathtt{chan}[w]} \tag{44.5d}$$

$$\frac{\phi\langle w\rangle = \tau}{\phi\,\mathtt{event}\langle w\rangle = \tau\,\mathtt{event}[w]} \tag{44.5e}$$

$$\frac{\phi\langle w'\rangle = \tau'}{(\phi\,\mathtt{at}\,w')\langle w\rangle = \tau'} \tag{44.5f}$$

Crucially, rule (44.5f) states that the situated family $\phi\,\mathtt{at}\,w'$ is to be interpreted at $w$ by the interpretation of $\phi$ at $w'$. Otherwise instantiation serves merely to decorate the constituent command, channel, and event skeletons with the site at which they are being interpreted.

Any type, $\tau$, of **DA** may be embedded as a constant family, $\mathtt{exactly}(\tau)$, such that $\mathtt{exactly}(\tau)\langle w\rangle = \tau$ for any site $w$. The constant family is inductively defined by the following rules:

$$\frac{}{\mathtt{exactly}(\mathtt{nat}) = \mathtt{nat}} \tag{44.6a}$$

$$\frac{\mathtt{exactly}(\tau_1) = \phi_1 \quad \mathtt{exactly}(\tau_2) = \phi_2}{\mathtt{exactly}(\tau_1 \rightharpoonup \tau_2) = \phi_1 \rightharpoonup \phi_2} \tag{44.6b}$$

$$\frac{\mathtt{exactly}(\tau) = \phi}{\mathtt{exactly}(\tau\,\mathtt{cmd}[w]) = \phi\,\mathtt{cmd}\,\mathtt{at}\,w} \tag{44.6c}$$



Preview

$$\frac{\mathtt{exactly}(\tau) = \phi}{\mathtt{exactly}(\tau\ \mathtt{chan}[w]) = \phi\ \mathtt{chan}\ \mathtt{at}\ w} \tag{44.6d}$$

$$\frac{\mathtt{exactly}(\tau) = \phi}{\mathtt{exactly}(\tau\ \mathtt{event}[w]) = \phi\ \mathtt{event}\ \mathtt{at}\ w} \tag{44.6e}$$

It is easy to check that $\mathtt{exactly}(\tau)$ is a constant family:

**Lemma 44.4.** *For any site $w$, $\mathtt{exactly}(\tau)\langle w\rangle = \tau$.*

The statics of **DA** may be given in factored form, as is illustrated by the following selection of typing rules:

$$\frac{\Phi \vdash_\Sigma e : \phi\ @\ w}{\Phi \vdash_\Sigma \mathtt{ret}\ e \mathrel{\dot{\sim}} \phi\ @\ w} \tag{44.7a}$$

$$\frac{\Phi \vdash_\Sigma e_1 : \phi_1\ @\ w \quad \Phi, x : \phi_1\ @\ w \vdash_\Sigma m_2 \mathrel{\dot{\sim}} \phi_2\ @\ w}{\Phi \vdash_\Sigma \mathtt{bnd}\ x \hookleftarrow e_1\ ;\ m_2 \mathrel{\dot{\sim}} \phi_2\ @\ w} \tag{44.7b}$$

$$\frac{\Phi \vdash_\Sigma m \mathrel{\dot{\sim}} \phi\ @\ w}{\Phi \vdash_\Sigma \mathtt{cmd}\ m : \phi\ \mathtt{cmd}\ @\ w} \tag{44.7c}$$

$$\frac{\mathtt{exactly}(\tau) = \phi}{\Phi \vdash_{\Sigma, a \sim \tau @ w} \mathbin{\&} a : \phi\ \mathtt{chan}\ @\ w} \tag{44.7d}$$

$$\frac{}{\Phi \vdash_\Sigma \mathtt{never} : \phi\ \mathtt{event}\ @\ w} \tag{44.7e}$$

$$\frac{\mathtt{exactly}(\tau) = \phi}{\Phi \vdash_{\Sigma, a \sim \tau @ w} ?\ a : \phi\ \mathtt{event}\ @\ w} \tag{44.7f}$$

$$\frac{\Phi \vdash_\Sigma e : \phi\ \mathtt{chan}\ @\ w}{\Phi \vdash_\Sigma ??\ e : \phi\ \mathtt{event}\ @\ w} \tag{44.7g}$$

$$\frac{\Phi \vdash_\Sigma e_1 : \phi\ \mathtt{event}\ @\ w \quad \Phi \vdash_\Sigma e_2 : \phi\ \mathtt{event}\ @\ w}{\Phi \vdash_\Sigma e_1\ \mathtt{or}\ e_2 : \phi\ \mathtt{event}\ @\ w} \tag{44.7h}$$

$$\frac{\Phi \vdash_\Sigma e : \phi\ \mathtt{event}\ @\ w}{\Phi \vdash_\Sigma \mathtt{sync}(e) \mathrel{\dot{\sim}} \phi\ @\ w} \tag{44.7i}$$

$$\frac{\Phi \vdash_\Sigma m' \mathrel{\dot{\sim}} \phi'\ @\ w' \quad \phi'\ \mathsf{mobile}}{\Phi \vdash_\Sigma \mathtt{at}\ w'\ \{m'\} \mathrel{\dot{\sim}} \phi'\ @\ w} \tag{44.7j}$$

Rule (44.7d) specifies that a reference to a channel carrying a value of type $\tau$ is classified by the *constant* family yielding the type $\tau$ at each site. Rule (44.7j) is the most interesting rule, because it include a restriction on the family $\phi'$. To see how this arises, inductively we have that $\widehat{\Phi} \vdash_\Sigma m' \mathrel{\dot{\sim}} \phi'\langle w'\rangle\ @\ w'$, which is enough to ensure that $\widehat{\Phi} \vdash_\Sigma \mathtt{at}\ w'\ \{m'\} \mathrel{\dot{\sim}} \phi'\langle w'\rangle\ @\ w$. But we are required to show that $\widehat{\Phi} \vdash_\Sigma \mathtt{at}\ w'\ \{m'\} \mathrel{\dot{\sim}} \phi'\langle w\rangle\ @\ w$! This will only be the case if $\phi'\langle w\rangle = \phi'\langle w'\rangle$, which is to say that $\phi'$ is a constant family, whose meaning does not depend on the site at which it is instantiated.

Preview

The judgment $\phi$ mobile states that $\phi$ is a mobile family. It is inductively defined by the following rules:

$$\frac{}{\texttt{nat mobile}} \tag{44.8a}$$

$$\frac{\phi_1 \text{ mobile} \quad \phi_2 \text{ mobile}}{\phi_1 \rightharpoonup \phi_2 \text{ mobile}} \tag{44.8b}$$

$$\frac{}{\phi \,\texttt{at}\, w \text{ mobile}} \tag{44.8c}$$

The remaining families are not mobile, precisely because their instantiation specifies the site of their instances; these do not determine constant families.

**Lemma 44.5.**

1. *If $\phi$ mobile, then for every $w$ and $w'$, $\phi\langle w\rangle = \phi\langle w'\rangle$.*
2. *For any type $\tau$, $\texttt{exactly}(\tau)$ mobile.*

We may then verify that the intended interpretation is valid:

**Theorem 44.6.**

1. *If $\Phi \vdash_\Sigma e : \phi \mathbin{@} w$, then $\widehat{\Phi} \vdash_\Sigma e : \phi\langle w\rangle$.*
2. *If $\Phi \vdash_\Sigma m \mathrel{\dot{\sim}} \phi \mathbin{@} w$, then $\widehat{\Phi} \vdash_\Sigma m \mathrel{\dot{\sim}} \phi\langle w\rangle \mathbin{@} w$.*

*Proof.* By induction on rules (44.7). □

## 44.5 Notes

The use of a spatial modality to express locality and mobility constraints in a distributed program was inspired by ML5 (Murphy et al., 2004). The separation of locality concerns from concurrency concerns is expressed here by supporting communication and synchronization within a site, and treating movement between sites separately. The formulation of situated types is based on Licata and Harper (2010).

Preview

# Part XVIII

# Separate Compilation and Modularity

Preview

Preview

# Chapter 45

# Components and Linking

*Modularity* is the most important technique for controlling the complexity of programs. Programs are decomposed into separate *components* with precisely specified, and tightly controlled, interactions. The pathways for interaction among components determine dependencies that constrain the process by which the components are integrated, or *linked*, to form a complete system. Different systems may use the same components, and a single system may use multiple instances of a single component. Sharing of components amortizes the cost of their development across systems, and helps limit errors by limiting coding effort.

Modularity is not limited to programming languages. In mathematics the proof of a theorem is decomposed into a collection of definitions and lemmas. Cross-references among lemmas determine a dependency structure that constrains their integration to form a complete proof of the main theorem. Of course, one person's theorem is another person's lemma; there is no intrinsic limit on the depth and complexity of the hierarchies of results in mathematics. Mathematical structures are themselves composed of separable parts, as, for example a Lie group is a group structure on a manifold.

Modularity arises from the structural properties of the hypothetical and general judgments. Dependencies among components are expressed by free variables whose typing assumptions state the presumed properties of the component. Linking amounts to substitution to discharge the hypothesis.

## 45.1 Simple Units and Linking

Decomposing a program into units amounts to exploiting the transitivity of the hypothetical judgment (see Chapter 3). The decomposition may be expressed as an interaction between two parties, the *client* and the *implementor*, that is mediated by an agreed-upon contract, called an *interface*. The client *assumes* that the implementor upholds the contract, and the implementor *guarantees* that the contract will be upheld. The assumption made by the client amounts to a declaration of its dependence on the implementor that is discharged by *linking* the two parties in accordance with their agreed-upon contract.

Preview

The interface that mediates the interaction between a client and an implementor is a *type*. Linking is may be thought of as the implementation of the composite structural rules of substitution and transitivity:

$$\frac{\Gamma \vdash e_{impl} : \tau_{intf} \quad \Gamma, x : \tau_{intf} \vdash e_{client} : \tau_{client}}{\Gamma \vdash [e_{impl}/x]e_{client} : \tau_{client}} \tag{45.1}$$

The type $\tau_{intf}$ is the interface type. It defines the capabilities to be provided by the implementor, $e_{impl}$, that are relied upon by the client, $e_{client}$. The free variable, $x$, expresses the dependency of $e_{client}$ on $e_{impl}$. That is, the client accesses the implementation by using the variable, $x$.

The interface type, $\tau_{intf}$, is the contract between the client and the implementor. It determines the properties of the implementation on which the client may depend and, at the same time, determines the obligations that the implementor must fulfill. The simplest form of interface type is a finite product type of the form $\langle f_1 \hookrightarrow \tau_1, \ldots, f_n \hookrightarrow \tau_n \rangle$, specifying a component with components $f_i$ of type $\tau_i$. Such a type is commonly called an *application program interface*, or *API*, because it determines the operations that the client (application) may expect from the implementor. A more sophisticated form of interface is one that defines an abstract type of the form $\exists(t.\langle f_1 \hookrightarrow \tau_1, \ldots, f_n \hookrightarrow \tau_n \rangle)$, which defines an abstract type, $t$, representing the internal state of an "abstract machine" whose "instruction set" consists of the operations $f_1, \ldots, f_n$ whose types may involve $t$. Being abstract, the type $t$ is not revealed to the client, but is known only to the implementor.[1]

Conceptually, linking is just substitution, but practically this can be implemented in a variety of ways. One method is called *separate compilation*. The expressions $e_{client}$ and $e_{impl}$, called in this context *source modules*, are translated (compiled) into another, lower-level, language, resulting in *object modules*. Linking consists of performing the required substitution at the level of the object language in such a way that the result corresponds to the translation of $[e_{impl}/x]e_{client}$.[2] Another method, called *separate checking*, shifts the requirement for translation to the linker. The client and implementor units are ensured to be type-correct with respect to the interface requirements, but are not translated into lower-level form. Linking then consists of translating the composite program as a whole, often resulting in a more efficient outcome than would be possible when compiling separately.

A more sophisticated, and widely used, implementation of substitution is called *dynamic linking*. Informally, this means that execution of the client commences before the implementation of the components on which it depends are provided. Rather than link prior to execution, we instead execute and link "on the fly." At first blush this might seem to be a radical departure from the methodology developed in this book, because we have consistently required that execution be defined only on expressions with no free variables. But looks can be deceiving. What is really going on with dynamic linking is that the client is implemented by a *stub* that forwards accesses to a stored implementation (typically, in a "file system" or similar data structure). The actual implementation code is not accessed until the client requests it, which may not happen at all. This tends to reduce latency and makes it possible to replace the implementation without recompiling the client.

[1] See Chapters 21 and 50 for a discussion of type abstraction.

[2] The correspondence need not be exact, but must be equivalent for all practical purposes, in the sense discussed in Chapter 49.

Preview

What is important is not how linking is implemented, but rather that the linking principle enables *separate development*. Once the common interface has been agreed upon, the client and implementor are free to proceed with their work independently of one another. All that is required is that both parties complete their work before the system as a whole can be built.

## 45.2 Initialization and Effects

Linking resolves the dependencies among the components of a program by substitution. This view is valid so long as the components are given by pure expressions, those that evaluate to a value without inducing any effects. For in such cases there is no problem with the replication, or complete omission, of a component arising from repeated, or absent, uses of a variable representing it. But what if the expression defining the implementation of a component has an effect when evaluated? At a minimum replication of the component implies replication of its effects. Worse, effects introduce *implicit dependencies* among components that are not apparent from their types. For example, if each of two components mutates a shared assignable, the order in which they are linked with a client program affects the behavior of the whole.

This may raise doubts about the treatment of linking as substitution, but on closer inspection it becomes clear that implicit dependencies are naturally expressed by the modal distinction between expressions and commands introduced in Chapter 36. Specifically, a component that may have an effect when executed does not have type $\tau_{intf}$ of implementations of the interface type, but rather the type $\tau_{intf}\,\texttt{cmd}$ of encapsulated commands that, when executed, have effects and yield implementations. Being encapsulated, a value of this type is itself free of effects, but it may have effects when evaluated.

The distinction between the types $\tau_{intf}$ and $\tau_{intf}\,\texttt{cmd}$ is mediated by the sequentialization command introduced in Chapter 36. For the sake of generality, let us assume that the client is itself an encapsulated command of type $\tau_{client}\,\texttt{cmd}$, so that it may itself have effects when executed, and may serve as a component of a yet larger system. Assuming that the client refers to the encapsulated implementation by the variable $x$, the command

$$\texttt{bnd}\,x \hookleftarrow x\,\texttt{;}\,\texttt{do}\,e_{client}$$

first determines the implementation of the interface by running the encapsulated command, $x$, then running the client code with the result bound to $x$. The implicit dependencies of the client on the implementor are made explicit by the sequentialization command, which ensures that the implementor's effects occur prior to those of the client, precisely because the client depends on the implementor for its execution.

More generally, to manage such interactions in a large program it is common to isolate an *initialization procedure* whose role is to stage the effects engendered by the various components according to some policy or convention. Rather than attempt to survey all possible policies, which are numerous and complex, let us simply observe that the upshot of such conventions is that the initialization procedure is a command of the form

$$\{x_1 \hookleftarrow x_1\,\texttt{;}\ldots x_n \hookleftarrow x_n\,\texttt{;}\,m_{main}\},$$



Preview

where $x_1, \ldots, x_n$ represent the components of the system and $m_{main}$ is the main (startup) routine. After linking the initialization procedure has the form

$$\{x_1 \hookrightarrow e_1 \,;\, \ldots x_n \hookrightarrow e_n \,;\, m_{main}\},$$

where $e_1, \ldots, e_n$ are the encapsulated implementations of the linked components. When the initialization procedure is executed, it results in the substitution

$$[v_1, \ldots, v_n / x_1, \ldots, x_n] m_{main},$$

where the expressions $v_1, \ldots, v_n$ represent the values resulting from executing $e_1, \ldots, e_n$, respectively, and the implicit effects have occurred in the order specified by the initializer.

## 45.3 Notes

The relationship between the structural properties of entailment and the practical problem of separate development was implicit in much early work on programming languages, but became explicit once the correspondence between propositions and types was developed. There are many indications of this correspondence, for example in *Proofs and Types* (Girard, 1989) and *Intuitionistic Type Theory* (Martin-Löf, 1984), but it was first made explicit by Cardelli (1997).

Preview

# Chapter 46

# Type Abstractions and Type Classes

An interface is a contract that specifies the rights of a client and the responsibilities of an implementor. Being a specification of behavior, an interface is a type. In principle any type may serve as an interface, but in practice it is usual to structure code into *modules* consisting of separable and reusable components. An interface specifies the behavior of a module expected by a client and imposed on the implementor. It is the fulcrum on which is balanced the tension between separability and integration. As a rule, a module should have a well-defined behavior that can be understood separately, but it is equally important that it be easy to combine modules to form an integrated whole.

A fundamental question is, what is the type of a module? That is, what form should an interface take? One long-standing idea is for an interface to be a labeled tuple of functions and procedures with specified types. The types of the fields of the tuple are traditionally called *function headers*, because they summarize the call and return types of each function. Using interfaces of this form is called *procedural abstraction*, because it limits the dependencies between modules to a specified set of procedures. We may think of the fields of the tuple as being the instruction set of a virtual machine. The client makes use of these instructions in its code, and the implementor agrees to provide their implementations.

The problem with procedural abstraction is that it does not provide as much insulation as one might like. For example, a module that implements a dictionary must expose in the types of its operations the exact representation of the tree as, say, a recursive type (or, in more rudimentary languages, a pointer to a structure that itself may contain such pointers). Yet the client really should not depend on this representation: the whole point of abstraction is to eliminate such dependencies. The solution, as discussed in Chapter 21, is to extend the abstract machine metaphor to allow the internal state of the machine to be hidden from the client. In the case of a dictionary the representation of the dictionary as a binary search tree is hidden by existential quantification. This is called *type abstraction*, because the type of the underlying data (state of the abstract machine) is hidden.

Type abstraction is a powerful method for limiting the dependencies among the modules that constitute a program. It is very useful in many circumstances, but is not universally applicable. It is often useful to expose, rather than obscure, type information across a module boundary. A

Preview

typical example is the implementation of a dictionary, which is a mapping from keys to values. To use, say, a binary search tree to implement a dictionary, we require that the key type admit a total ordering with which keys can be compared. The dictionary abstraction does not depend on the exact type of the keys, but only requires that the key type be constrained to provide a comparison operation. A *type class* is a specification of such a requirement. The class of comparable types, for example, specifies a type, $t$, together with an operation, `leq`, of type $(t \times t) \to \mathtt{bool}$ with which to compare them. Superficially, such a specification looks like a type abstraction, because it specifies a type and one or more operations on it, but with the important difference that the type, $t$, is not hidden from the client. For if it were, the client would only be able to compare keys using `leq`, but would have no means of obtaining keys to compare. A type class, in contrast to a type abstraction, is not intended to be an exhaustive specification of the operations on a type, but rather as a constraint on its behavior expressed by demanding that certain operations, such as comparison, be available, without limiting the other operations that might be defined on it.

Type abstractions and type classes are extremal cases of a general concept of module type that we shall discuss in detail in this chapter. The crucial idea is the *controlled revelation* of type information across module boundaries. Type abstractions are opaque; type classes are transparent. These are both instances of *translucency*, which arises from the combination of existential types (Chapter 21), subtyping (Chapter 23), and singleton kinds and subkinding (Chapter 25). Unlike in Chapter 21, however, we will distinguish the types of modules, which we will called *signatures*, from the types of ordinary values. The distinction is not essential, but it will be helpful to keep the two concepts separate at the outset, deferring discussion of how to relax the segregation once the basic concepts are in place.

## 46.1 Type Abstraction

Type abstraction is captured by a form of existential type quantification similar to that described in Chapter 21. For example, a dictionary with keys of type $\tau_{\mathsf{key}}$ and values of type $\tau_{\mathsf{val}}$ implements the signature, $\sigma_{\mathsf{dict}}$, defined as follows:

$$[t :: \mathtt{T}; \langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow \tau_{\mathsf{key}} \times \tau_{\mathsf{val}} \times t \to t, \mathtt{fnd} \hookrightarrow \tau_{\mathsf{key}} \times t \to \tau_{\mathsf{val}}\ \mathtt{opt} \rangle].$$

The type variable, $t$, of kind `T` is the abstract type of dictionaries on which are defined three operations, `emp`, `ins`, and `fnd`, with the specified types. It is not essential to fix the type $\tau_{\mathsf{val}}$, because the dictionary operations impose no restrictions on it; we will do so only for the sake of simplicity. However, it is essential, at this stage, that the key type, $\tau_{\mathsf{key}}$, be fixed, for reasons that will become clearer as we proceed.

An implementation of the signature $\sigma_{\mathsf{dict}}$ is a *structure*, $M_{\mathsf{dict}}$, of the form

$$[\tau_{\mathsf{dict}}; \langle \mathtt{emp} \hookrightarrow \ldots, \mathtt{ins} \hookrightarrow \ldots, \mathtt{fnd} \hookrightarrow \ldots \rangle],$$

where the elided parts implement the dictionary operations in terms of the chosen representation type, $\tau_{\mathsf{dict}}$. For example, $\tau_{\mathsf{dict}}$ might be a recursive type defining a balanced binary search tree, such as a red-black tree. The dictionary operations work on the underlying representation of the dictionary as such a tree, just as would a package of existential type discussed in Chapter 21.

Preview

To ensure that the representation of the dictionary is hidden from a client, the structure $M_{\mathsf{dict}}$ is *sealed* with the signature $\sigma_{\mathsf{dict}}$ to obtain the module

$$M_{\mathsf{dict}} \rceil \sigma_{\mathsf{dict}}.$$

The effect of sealing is to ensure that the *only* information about $M_{\mathsf{dict}}$ that is propagated to the client is given by $\sigma_{\mathsf{dict}}$. In particular, because $\sigma_{\mathsf{dict}}$ only specifies that the type, $t$, have kind $\mathtt{T}$, no information about the choice of $t$ as $\tau_{\mathsf{dict}}$ in $M_{\mathsf{dict}}$ is made available to the client.

A module is a *two-phase* object consisting of a *static part* and a *dynamic part*. The static part is a constructor of a specified kind; the dynamic part is a value of a specified type. There are two elimination forms that extract the static and dynamic parts of a module. These are, respectively, a form of constructor and a form of expression. More precisely, the constructor $M \cdot \mathsf{s}$ stands for the static part of $M$, and the expression $M \cdot \mathsf{d}$ stands for its dynamic part. According to the inversion principle, if a module, $M$, has introductory form, then $M \cdot \mathsf{s}$ should be equivalent to the static part of $M$. So, for example, $M_{\mathsf{dict}} \cdot \mathsf{s}$ should be equivalent to $\tau_{\mathsf{dict}}$.

But consider the static part of a sealed module, which has the form $(M_{\mathsf{dict}} \rceil \sigma_{\mathsf{dict}}) \cdot \mathsf{s}$. Because sealing hides the representation of an abstract type, this constructor should not be equivalent to $\tau_{\mathsf{dict}}$. If $M'_{\mathsf{dict}}$ is another implementation of $\sigma_{\mathsf{dict}}$, should $(M_{\mathsf{dict}} \rceil \sigma_{\mathsf{dict}}) \cdot \mathsf{s}$ be equivalent to $(M'_{\mathsf{dict}} \rceil \sigma_{\mathsf{dict}}) \cdot \mathsf{s}$? To ensure reflexivity of type equivalence this equation should hold whenever $M$ and $M'$ are equivalent modules. But this violates representation independence for abstract types by making equivalence of abstract types sensitive to their implementation.

It would seem, then, that there is a fundamental contradiction between two very fundamental concepts, type equivalence and representation independence. The way out of this conundrum is to *disallow* reference to the static part of a sealed module: the type expression $M \rceil \sigma \cdot \mathsf{s}$ is deemed ill-formed. More generally, we disallow formation of $M \cdot \mathsf{s}$ unless $M$ is a *module value*, whose static part is always manifest. An explicit structure is a module value, as is any module variable (provided that module variables are bound by-value).

One effect of this restriction is that sealed modules must be bound to a variable before they are used. Because module variables are bound by-value, doing so has the effect of imposing abstraction at the binding site. In fact, we may think of sealing as a kind of computational effect that "occurs" at the binding site, much as the bind operation in Algol discussed in Chapter 36 engenders the effects induced by an encapsulated command. A consequence of this is that two distinct bindings of the same sealed module result in two distinct abstract types. The type system willfully ignores the identity of the two occurrences of the same module in order to ensure that their representations can be changed independently of one another without disrupting the behavior of any client code (because the client cannot rely on their identity, it must be prepared for them to be different).

## 46.2 Type Classes

Type abstraction is an essential tool for limiting dependencies among modules in a program. The signature of a type abstraction determines all that is known about a module by a client; no other uses of the values of an abstract type are permissible. A complementary tool is to use a signature to partially specify the capabilities of a module. Such a mechanism is called a *type class*, or a *view*, in



Preview

which case an implementation is called an *instance* of the type class or view. Because the signature of a type class serves only as a constraint specifying the minimum capabilities of an unknown module, some other means of working with values of that type must be available. The key to achieving this is to expose, rather than hide, the identity of the static part of a module. In this sense type classes are the "opposite" of type abstractions, but we shall see below that there is a smooth progression between them, mediated by a subsignature judgment.

Let us consider the implementation of dictionaries as a client of the implementation of its keys. To implement a dictionary using a binary search tree, for example, the only requirement is that keys come equipped with a total ordering given by a comparison operation. This can be expressed by a signature, $\sigma_{\mathsf{ord}}$, given by

$$[t :: \mathtt{T}; \langle \mathtt{leq} \hookrightarrow (t \times t) \to \mathtt{bool} \rangle].$$

Because a given type may be ordered in many ways, it is essential that the ordering be packaged with the type to determine a type of keys.

The implementation of dictionaries as binary search trees takes the form

$$X : \sigma_{\mathsf{ord}} \vdash M^X_{\mathsf{bstdict}} : \sigma^X_{\mathsf{dict}},$$

where $\sigma^X_{\mathsf{dict}}$ is the signature

$$[t :: \mathtt{T}; \langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow X \cdot \mathtt{s} \times \tau_{\mathsf{val}} \times t \to t, \mathtt{fnd} \hookrightarrow X \cdot \mathtt{s} \times t \to \tau_{\mathsf{val}}\, \mathtt{opt} \rangle],$$

and $M^X_{\mathsf{bstdict}}$ is a structure (not given explicitly here) that implements the dictionary operations using binary search trees.[1] Within $M^X_{\mathsf{bstdict}}$, the static and dynamic parts of the module $X$ are accessed by writing $X \cdot \mathtt{s}$ and $X \cdot \mathtt{d}$, respectively. In particular, the comparison operation on keys is accessed by the projection $X \cdot \mathtt{d} \cdot \mathtt{leq}$.

The declared signature of the module variable, $X$, expresses a constraint on the capabilities of a key type by specifying an upper bound on its signature in the subsignature ordering. This implies that any module bound to $X$ must provide a type of keys and a comparison operation on that type, but nothing else is assumed of it. Because this is all we know about the unknown module, $X$, the dictionary implementation is constrained to rely only on these specified capabilities, and no others. When linking with a module defining $X$, the implementation need not be sealed with this signature, but must rather have a signature that is no larger than it in the subsignature relation. Indeed, the signature $\sigma_{\mathsf{ord}}$ is useless for sealing, as is easily seen by example. Suppose that $M_{\mathsf{natord}} : \sigma_{\mathsf{ord}}$ is an instance of the class of ordered types under the usual ordering. If we seal $M_{\mathsf{natord}}$ with $\sigma_{\mathsf{ord}}$ by writing

$$M_{\mathsf{natord}} \upharpoonright \sigma_{\mathsf{ord}},$$

the resulting module is *useless*, because we would then have no way to create values of the key type.

We see, then, that a type class amounts to a categorization of a pre-existing type, not a means of introducing a new type. Rather than obscure the identity of the static part of $M_{\mathsf{natord}}$, we wish

[1] Here and elsewhere in this chapter and the next, the superscript $X$ serves as a reminder that the module variable, $X$, may occur free in the annotated module or signature.



Preview

to propagate its identity as nat while specifying a comparison with which to order them. This may be achieved using singleton kinds (Chapter 25). Specifically, the most precise, or *principal*, signature of a structure is the one that exposes its static part using a singleton kind. In the case of the module $M_{\mathsf{natord}}$, the principal signature is the signature, $\sigma_{\mathsf{natord}}$, given by

$$[t :: \mathtt{S}(\mathtt{nat});\mathtt{leq} \hookrightarrow (t \times t) \to \mathtt{bool}],$$

which, by the rules of equivalence (defined formally in Section 46.3), is equivalent to the signature

$$[_ :: \mathtt{S}(\mathtt{nat});\mathtt{leq} \hookrightarrow (\mathtt{nat} \times \mathtt{nat}) \to \mathtt{bool}].$$

The dictionary implementation, $M^X_{\mathsf{bstdict}}$ expects a module, $X$, with signature $\sigma_{\mathsf{ord}}$, but the module $M_{\mathsf{natord}}$ provides the signature $\sigma_{\mathsf{natord}}$. Applying the rules of subkinding given in Chapter 25, together with the evident covariance principle for signatures, we obtain the subsignature relationship

$$\sigma_{\mathsf{natord}} <: \sigma_{\mathsf{ord}}.$$

By the subsumption principle, a module of signature $\sigma_{\mathsf{natord}}$ may be provided whenever a module of signature $\sigma_{\mathsf{ord}}$ is required. Therefore $M_{\mathsf{natord}}$ may be linked to $X$ in $M^X_{\mathsf{bstdict}}$.

The combination of subtyping and sealing provides a smooth gradation between type classes and type abstractions. The principal signature for $M^X_{\mathsf{bstdict}}$ is the signature $\rho^X_{\mathsf{dict}}$ given by

$$[t :: \mathtt{S}(\tau^X_{\mathsf{bst}});\langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow X \cdot \mathtt{s} \times \tau_{\mathsf{val}} \times t \to t, \mathtt{fnd} \hookrightarrow X \cdot \mathtt{s} \times t \to \tau_{\mathsf{val}}\,\mathtt{opt}\rangle],$$

where $\tau^X_{\mathsf{bst}}$ is the type of binary search trees with keys given by the module $X$ of signature $\sigma_{ord}$. This is a subsignature of $\sigma^X_{\mathsf{dict}}$ given earlier, so that the sealed module

$$M^X_{\mathsf{bstdict}} \upharpoonright \sigma^X_{\mathsf{dict}}$$

is well-formed, and has type $\sigma^X_{\mathsf{dict}}$, which hides the representation type of the dictionary abstraction.

After linking $X$ to $M_{\mathsf{natord}}$, the signature of the dictionary is specialized by propagating the identity of the static part of $M_{\mathsf{natord}}$. This, too, is achieved by using the subsignature judgment. As remarked earlier, the dictionary implementation satisfies the typing

$$X : \sigma_{\mathsf{ord}} \vdash M^X_{\mathsf{bstdict}} : \sigma^X_{\mathsf{dict}}.$$

But because $\sigma_{\mathsf{natord}} <: \sigma_{\mathsf{ord}}$, we have, by contravariance, that

$$X : \sigma_{\mathsf{natord}} \vdash M^X_{\mathsf{bstdict}} : \sigma^X_{\mathsf{dict}}.$$

is also a valid typing judgment. If $X : \sigma_{\mathsf{natord}}$, then $X \cdot \mathtt{s}$ is equivalent to nat, because it has kind $\mathtt{S}(\mathtt{nat})$, and hence the following typing is also valid:

$$X : \sigma_{\mathsf{natord}} \vdash M^X_{\mathsf{bstdict}} : \sigma_{\mathsf{natdict}}.$$



Preview

Here $\sigma_{\mathsf{natdict}}$ is the closed signature

$$[t :: \mathtt{T};\langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow \mathtt{nat} \times \tau_{\mathsf{val}} \times t \to t, \mathtt{fnd} \hookrightarrow \mathtt{nat} \times t \to \tau_{\mathsf{val}}\, \mathtt{opt}\rangle]$$

in which the representation of dictionaries is held abstract, but the representation of keys as natural numbers is publicized. The dependency on $X$ has been eliminated by replacing all occurrences of $X \cdot \mathtt{s}$ within $\sigma^X_{\mathsf{dict}}$ by the type $\mathtt{nat}$. Having derived this typing we may link $X$ with $M_{natord}$ as described in Chapter 45 to obtain a composite module, $M_{\mathsf{natdict}}$, of signature $\sigma_{\mathsf{natdict}}$, in which keys are natural numbers ordered as specified by $M_{\mathsf{natord}}$.

It is convenient to exploit subtyping for labeled tuple types to avoid creating an *ad hoc* module specifying the standard ordering on the natural numbers. Instead we can extract the required module directly from the implementation of the abstract type of numbers using subsumption. As an illustration, let $X_{\mathsf{nat}}$ be a module variable of signature $\sigma_{\mathsf{nat}}$, which has the form

$$[t :: \mathtt{T};\langle \mathtt{zero} \hookrightarrow t, \mathtt{succ} \hookrightarrow t \to t, \mathtt{leq} \hookrightarrow (t \times t) \to \mathtt{bool}, \ldots\rangle]$$

The fields of the tuple provide all and only the operations that are available on the abstract type of natural numbers. Among them is the comparison operation, `leq`, which is required by the dictionary module. Applying the subtyping rules for labeled tuples given in Chapter 23, together with the covariance of signatures, we obtain the subsignature relationship

$$\sigma_{\mathsf{nat}} <: \sigma_{\mathsf{ord}},$$

so that by subsumption the variable, $X_{\mathsf{nat}}$, may be linked to the variable, $X$, postulated by the dictionary implementation. Subtyping takes care of extracting the required `leq` field from the abstract type of natural numbers, demonstrating that the natural numbers are an instance of the class of ordered types. Of course, this approach only works if we wish to order the natural numbers in the natural way provided by the abstract type. If, instead, we wish to use another ordering, then we must construct instances of $\sigma_{\mathsf{ord}}$ "by hand" to define the appropriate ordering.

## 46.3 A Module Language

The module language, **Mod**, is a codification of the ideas outlined in the preceding section. The syntax is divided into five levels: expressions classified by types, constructors classified by kinds, and modules classified by signatures. The expression and type level consists of various language mechanisms described earlier in this book, including at least product, sum, and partial function types. The constructor and kind level is as described in Chapters 22 and 25, with singleton and dependent kinds. The syntax of modules is summarized by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Sig | $\sigma$ | ::= | $\mathtt{sig}[\kappa](t.\tau)$ | $[t :: \kappa;\tau]$ | signature |
| Mod | $M$ | ::= | $X$ | $X$ | variable |
| | | | $\mathtt{str}(c;e)$ | $[c;e]$ | structure |
| | | | $\mathtt{seal}[\sigma](M)$ | $M \upharpoonright \sigma$ | seal |
| | | | $\mathtt{let}[\sigma](M_1; X.M_2)$ | $(\mathtt{let}\ X\ \mathtt{be}\ M_1\ \mathtt{in}\ M_2) : \sigma$ | definition |
| Con | $c$ | ::= | $\mathtt{stat}(M)$ | $M \cdot \mathtt{s}$ | static part |
| Exp | $e$ | ::= | $\mathtt{dyn}(M)$ | $M \cdot \mathtt{d}$ | dynamic part |



Preview

The statics of **Mod** consists of the following forms of judgment, in addition to those governing the kind and type levels:

| | |
|---|---|
| $\Gamma \vdash \sigma\ \mathsf{sig}$ | well-formed signature |
| $\Gamma \vdash \sigma_1 \equiv \sigma_2$ | equivalent signatures |
| $\Gamma \vdash \sigma_1 <: \sigma_2$ | subsignature |
| $\Gamma \vdash M : \sigma$ | well-formed module |
| $\Gamma \vdash M\ \mathsf{val}$ | module value |
| $\Gamma \vdash e\ \mathsf{val}$ | expression value |

Rather than segregate hypotheses into zones, we instead admit the following three forms of hypothesis groups:

| | |
|---|---|
| $X : \sigma, X\ \mathsf{val}$ | module value variable |
| $u :: \kappa$ | constructor variable |
| $x : \tau, x\ \mathsf{val}$ | expression value variable |

It is important that module and expression variables are always regarded as values to ensure that type abstraction is properly enforced. Correspondingly, each module and expression variable appears in $\Gamma$ paired with the hypothesis that it is a value. As a notational convenience we will not explicitly state the value hypotheses associated with module and expression variables, under the convention that all such variables implicitly come paired with such an assumption.

The formation, equivalence, and subsignature judgments are defined by the following rules:

$$\frac{\Gamma \vdash \kappa\ \mathsf{kind} \quad \Gamma, u :: \kappa \vdash \tau\ \mathsf{type}}{\Gamma \vdash [u :: \kappa;\tau]\ \mathsf{sig}} \tag{46.1a}$$

$$\frac{\Gamma \vdash \kappa_1 \equiv \kappa_2 \quad \Gamma, u :: \kappa_1 \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [u :: \kappa_1;\tau_1] \equiv [u :: \kappa_2;\tau_2]} \tag{46.1b}$$

$$\frac{\Gamma \vdash \kappa_1 :<: \kappa_2 \quad \Gamma, u :: \kappa_1 \vdash \tau_1 <: \tau_2}{\Gamma \vdash [u :: \kappa_1;\tau_1] <: [u :: \kappa_2;\tau_2]} \tag{46.1c}$$

Most importantly, signatures are covariant in both the kind and type positions: subkinding and subtyping are preserved by the formation of a signature. It is a consequence of rule (46.1b) that

$$[u :: \mathtt{S}(c);\tau] \equiv [_ :: \mathtt{S}(c);[c/u]\tau]$$

and, further, it is a consequence of rule (46.1c) that

$$[_ :: \mathtt{S}(c);[c/u]\tau] <: [_ :: \mathtt{T};[c/u]\tau]$$

and therefore

$$[u :: \mathtt{S}(c);\tau] <: [_ :: \mathtt{T};[c/u]\tau].$$

It is also the case that

$$[u :: \mathtt{S}(c);\tau] <: [u :: \mathtt{T};\tau].$$

Preview

But the two supersignatures of $[u :: \mathtt{S}(c);\tau]$ are *incomparable* with respect to the subsignature judgment. This fact is important in the statics of module definitions, as will be detailed shortly.

The statics of expressions of **Mod** is given by the following rules:

$$\frac{}{\Gamma, X : \sigma \vdash X : \sigma} \tag{46.2a}$$

$$\frac{\Gamma \vdash c :: \kappa \quad \Gamma \vdash e :: [c/u]\tau}{\Gamma \vdash [c;e] : [u :: \kappa;\tau]} \tag{46.2b}$$

$$\frac{\Gamma \vdash \sigma \ \mathsf{sig} \quad \Gamma \vdash M : \sigma}{\Gamma \vdash M \upharpoonright \sigma : \sigma} \tag{46.2c}$$

$$\frac{\Gamma \vdash \sigma \ \mathsf{sig} \quad \Gamma \vdash M_1 : \sigma_1 \quad \Gamma, X : \sigma_1 \vdash M_2 : \sigma}{\Gamma \vdash (\mathtt{let}\ X\ \mathtt{be}\ M_1\ \mathtt{in}\ M_2) : \sigma : \sigma} \tag{46.2d}$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash \sigma <: \sigma'}{\Gamma \vdash M : \sigma'} \tag{46.2e}$$

In rule (46.2b) it is always possible to choose $\kappa$ to be the most specific kind of $c$ in the subkind ordering, which uniquely determines $c$ up to constructor equivalence. For such a choice, the signature $[u :: \kappa;\tau]$ is equivalent to $[_ :: \kappa;[c/u]\tau]$, which propagates the identity of the static part of the module expression into the type of its dynamic part. Rule (46.2c) is to be used in conjunction with subsumption (rule (46.2e)) to ensure that $M$ has the specified signature.

The need for a signature annotation on a module definition is a manifestation of the *avoidance problem*. Rule (46.2d) would be perfectly sensible were the signature, $\sigma$, omitted from the syntax of the definition. However, omitting this information greatly complicates type checking. If $\sigma$ were omitted from the syntax of the definition, the type checker would be required to find a signature, $\sigma$, for the body of the definition that *avoids* the module variable, $X$. Inductively, we may suppose that we have found a signature, $\sigma_1$, for the module $M_1$, and a signature, $\sigma_2$, for the module $M_2$, under the assumption that $X$ has signature $\sigma_1$. To find a signature for an unadorned definition, we must find a supersignature, $\sigma$, of $\sigma_2$ that avoids $X$. To ensure that all possible choices of $\sigma$ are accounted for, we seek to find the least (most precise) such signature with respect to the subsignature relation; this is called the *principal signature* of a module. The problem is that there may be no least supersignature of a given signature that avoids a specified variable. (Consider the example above of a signature with two incomparable supersignatures. The example may be chosen so that the supersignatures avoid a variable, $X$, that occurs in the subsignature.) Consequently, modules do not have principal signatures, a significant complication for type checking. To avoid this problem, we insist that the avoiding supersignature, $\sigma$, be given by the programmer so that the type checker is not required to find one.

In the presence of modules we have a new form of constructor expression, $M \cdot \mathsf{s}$, and a new form of value expression, $M \cdot \mathsf{d}$. These operations, respectively, extract the static and dynamic parts of the module $M$. Their formation rules are as follows:

$$\frac{\Gamma \vdash M \ \mathsf{val} \quad \Gamma \vdash M : [u :: \kappa;\tau]}{\Gamma \vdash M \cdot \mathsf{s} :: \kappa} \tag{46.3a}$$

Preview

$$\frac{\Gamma \vdash M : [_ :: \kappa;\tau]}{\Gamma \vdash M \cdot \mathsf{d} : \tau} \tag{46.3b}$$

Rule (46.3a) requires that the module expression, $M$, be a value in accordance with the following rules:

$$\frac{}{\Gamma, X : \sigma, X\ \mathsf{val} \vdash X\ \mathsf{val}} \tag{46.4a}$$

$$\frac{\Gamma \vdash e\ \mathsf{val}}{\Gamma \vdash [c;e]\ \mathsf{val}} \tag{46.4b}$$

(It is not strictly necessary to insist that the dynamic part of a structure be a value in order for the structure itself to be a value, but we impose this requirement to be consistent with the general policy to employ eager evaluation, and to obtain laziness through types, as described in Chapter 38.)

Rule (46.3a) specifies that only structure values have well-defined static parts, and hence precludes reference to the static part of a sealed structure, which is not a value. This ensures representation independence for abstract types, as discussed in Section 46.1. For if $M \cdot \mathsf{s}$ were admissible when $M$ is a sealed module, it would be a type whose identity depends on the underlying implementation, in violation of the abstraction principle. Module variables are, on the other hand, values, so that if $X : [t :: \mathtt{T};\tau]$ is a module variable, then $X \cdot \mathsf{s}$ is a well-formed type. What this means in practice is that sealed modules must be bound to variables before they can be used. It is for this reason that we include definitions among module expressions.

Rule (46.3b) requires that the signature of the module, $M$, be non-dependent, so that the result type, $\tau$, does not depend on the static part of the module. This may not always be the case. For example, if $M$ is a sealed module, say $N \upharpoonright [t :: \mathtt{T};t]$ for some module $N$, then projection $M \cdot \mathsf{d}$ is ill-formed. For if it were to be well-formed, its type would be $M \cdot \mathsf{s}$, which would violate representation independence for abstract types. But if $M$ is a module value, then it is always possible to derive a non-dependent signature for it, provided that we include the following rule of *self-recognition*:

$$\frac{\Gamma \vdash M : [u :: \kappa;\tau] \quad \Gamma \vdash M\ \mathsf{val}}{\Gamma \vdash M : [u :: \mathtt{S}(M \cdot \mathsf{s} :: \kappa);\tau]} \tag{46.5}$$

This rule propagates the identity of the static part of a module value into its signature. The dependency of the type of the dynamic part on the static part is then eliminable by sharing propagation.

The following rule of constructor equivalence states that a type projection from a module value is eliminable:

$$\frac{\Gamma \vdash [c;e] : [t :: \kappa;\tau] \quad \Gamma \vdash [c;e]\ \mathsf{val}}{\Gamma \vdash [c;e] \cdot \mathsf{s} \equiv c :: \kappa} \tag{46.6}$$

The requirement that the expression, $e$, be a value, which is implicit in the second premise of the rule, is not strictly necessary, but does no harm. A consequence of this rule is that apparent dependencies of closed constructors (or kinds) on modules may always be eliminated. In particular the identity of the constructor $[c;e] \cdot \mathsf{s}$ is independent of $e$, as would be expected if representation independence is to be assured.

The dynamics of modules is given as follows:

$$\frac{e \longmapsto e'}{[c;e] \longmapsto [c;e']} \tag{46.7a}$$

Preview

$$\frac{e \text{ val}}{[c;e] \cdot \mathsf{d} \longmapsto e} \tag{46.7b}$$

There is no need to evaluate constructors at run-time, because the dynamics of expressions does not depend on their types. It is not difficult to prove type safety for this dynamics relative to the foregoing statics.

## 46.4 First- and Second-Class

It is common to draw a distinction between *first-class* and *second-class* modules based on whether or not signatures are types, and hence whether modules are just a form of expression like any other. When modules are first-class they may be computed based on run-time conditions that may even depend on the state of the world, and may be stored in data structures, or stored in assignables. When modules are second-class signatures are a separate form of classifier from types, and module expressions may not be used in the same way as ordinary expressions. For example, it may not be possible to compute a module based on the phase of the moon.

Superficially, it seems as though first-class modules are uniformly superior to second-class modules, because you can do more with them. But on closer examination we see that the "less is more" principle applies here as well, much as in the distinction between dynamic and static languages discussed in Chapters 18 and 19. In particular if modules are first-class, then one must adopt a "pessimistic" attitude towards expressions that compute them, precisely because they represent fully general, even state-dependent, computations. One consequence is that it is difficult, or even impossible, to track the identity of the static part of a module during type checking. A general module expression need not have a well-defined static component, precluding its use in type expressions. Second-class modules, on the other hand, may be very permissive in the use of the static components of modules in types, precisely because the range of possible computations is reduced. In this respect second-class modules are more powerful than first-class, despite initial impressions. More importantly, a second-class module system can always be enriched to *permit* first-class modules, without requiring that they *all* be first-class. This gives us the best of both worlds: all of the flexibility of first-class modules and all of the precision of second-class modules. In short you pay for only what you use: if you use first-class capabilities, you should expect to pay a cost, but if you do not, you should not have to be taxed on the unrealized gain.

First-class modules may be added to **Mod** in the following manner. First, enrich the type system with existential types, as described in Chapter 21, so that "first-class modules" are just packages of existential type. A second-class module $M$ of signature $[t :: \kappa;\tau]$ may be made first-class by forming the package $\mathtt{pack}\, M \cdot \mathtt{s}\, \mathtt{with}\, M \cdot \mathsf{d}\, \mathtt{as}\, \exists(t.\tau)$ of type $\exists\, t :: \kappa.\tau$ consisting of the static and dynamic parts of $M$. Second, to allow packages to behave as modules, we introduce the module expression $\mathtt{open}\, e$ that opens the contents of a package as a module:

$$\frac{\Gamma \vdash e : \exists\, t :: \kappa.\tau}{\Gamma \vdash \mathtt{open}\, e : [t :: \kappa;\tau]} \tag{46.8}$$

Because the package $e$ is an arbitrary expression of existential type, the module expression $\mathtt{open}\, e$ may not be regarded as a value, and hence does not have a well-defined static part. Instead we

PREVIEW

must generally bind it to a variable before it is used, mimicking the composite behavior of the existential elimination form given in Chapter 21.

## 46.5 Notes

The use of dependent types to express modularity was first proposed by MacQueen (1986). Subsequent studies extended this proposal to model the *phase distinction* between compile- and run-time (Harper et al., 1990), and to account for type abstraction as well as type classes (Harper and Lillibridge, 1994; Leroy, 1994). The avoidance problem was first isolated by Castagna and Pierce (1994) and by Harper and Lillibridge (1994). It has come to play a central role in subsequent work on modules, such as Lillibridge (1997) and Dreyer (2005). The self-recognition rule was introduced by Harper and Lillibridge (1994) and by Leroy (1994). It was subsequently identified as a manifestation of higher-order singletons (Stone and Harper, 2006). A consolidation of these ideas was used as the foundation for a mechanization of the metatheory of modules (Lee et al., 2007). A thorough summary of the main issues in module system design is given in Dreyer (2005).

The presentation given here focuses attention on the type structure required to support modularity. An alternative formulation is based on *elaboration*, a translation of modularity constructs into more primitive notions such as polymorphism and higher-order functions. *The Definition of Standard ML* (Milner et al., 1997) pioneered the elaboration approach. Building on earlier work of Russo, a more rigorous type-theoretic formulation was given by Rossberg et al. (2010). The advantage of the elaboration-based approach is that it can make do with a simpler type theory as the target language, but at the expense of making the explanation of modularity more complex. It seems clear that some form of elaboration is required (to handle identifier scope resolution and type inference, for example), but it is as yet unclear where best to draw the line.

Preview

Preview

# Chapter 47

# Hierarchy and Parameterization

To be adequately expressive it is essential that a module system support the construction of module *hierarchies*. Hierarchical structure arises naturally in programming, both as an organizational device for partitioning of a large program into manageable pieces, and as a localization device that allows one type abstraction or type class to be layered on top of another. In such a scenario the lower layer plays an auxiliary role relative to the upper layer, and we may think of the upper layer as being parameterized by the lower in the sense that any implementation of the lower layer induces an instance of the upper layer corresponding to that instance. The pattern of dependency of one abstraction on another may be captured by an *abstraction*, or *parameterization*, mechanism that allows the implementation of one abstraction to be considered a function of the implementation of another. Hierarchies and parameterization work in tandem to provide an expressive language for organizing programs.

## 47.1 Hierarchy

It is common in modular programming to layer a type class or a type abstraction on top of a type class. For example, the class of *equality types*, which are those that admit a boolean equivalence test, is described by the signature $\sigma_{\mathtt{eq}}$ defined as follows:

$$[t :: \mathtt{T}; \langle \mathtt{eq} \hookrightarrow (t \times t) \to \mathtt{bool} \rangle].$$

Instances of this class consist of a type together with a binary equality operation defined on it. Such instances are modules with a subsignature of $\sigma_{\mathtt{eq}}$, for example $\sigma_{\mathtt{nateq}}$ given by

$$[t :: \mathtt{S}(\mathtt{nat}); \langle \mathtt{eq} \hookrightarrow (t \times t) \to \mathtt{bool} \rangle].$$

A module value of this signature has the form

$$[\mathtt{nat}; \langle \mathtt{eq} \hookrightarrow \ldots \rangle],$$

Preview

where the elided expression implements an equivalence relation on the natural numbers. All other instance values of the class $\sigma_{\mathsf{eq}}$ have a similar form, differing in the choice of type, and/or the choice of comparison operation.

The class of *ordered types* may be considered to be an extension of the class of equality types with an additional binary operation for the (strict) comparison of two elements of that type. One way to formulate this is as the signature

$$[t :: \mathtt{T}; \langle \mathtt{eq} \hookrightarrow (t \times t) \to \mathtt{bool}, \mathtt{lt} \hookrightarrow (t \times t) \to \mathtt{bool} \rangle],$$

which is a subsignature of $\sigma_{\mathsf{eq}}$ according to the rules of subtyping given in Chapter 23. This relationship amounts to the requirement that every ordered type is *a fortiori* an equality type.

This is well and good, but it would be even better if there were a way to incrementally extend the equality type class to the ordered type class without having to rewrite the signature as we have done in the foregoing example. Instead, we would like to *layer* the comparison aspect on top of an equality type class to obtain the ordered type class. This is achieved using a *hierarchical signature*, $\sigma_{\mathsf{eqord}}$, of the form

$$\sum X{:}\sigma_{\mathsf{eq}} \cdot \sigma^X_{\mathsf{ord}}.$$

In this signature we write $\sigma^X_{\mathsf{ord}}$ for the signature

$$[t :: \mathtt{S}(X \cdot \mathtt{s}); \langle \mathtt{lt} \hookrightarrow (t \times t) \to \mathtt{bool} \rangle],$$

which refers to the static part of $X$, namely the type on which the equality relation is defined. The notation $\sigma^X_{\mathsf{ord}}$ emphasizes that this signature has a free module variable, $X$, occurring within it, and hence is only meaningful in a context in which $X$ has been declared.

A value of the signature $\sigma_{\mathsf{eqord}}$ is a pair of modules, $\langle M_{\mathsf{eq}} ; M_{\mathsf{ord}} \rangle$, in which $M_{\mathsf{eq}}$ comprises a type equipped with an equality relation on it, and the second comprises a type equipped with an ordering relation on it. Crucially, the second type is constrained by the singleton kind in $\sigma^X_{\mathsf{ord}}$ to be the *same* as the first type. Such a constraint is called a *sharing specification*. The process of drawing out of the consequences of a sharing specification is called *sharing propagation*.

Sharing propagation is achieved by a combination of subkinding (as described in Chapter 25) and subtyping for signatures. For example, a particular ordering, $M_{\mathsf{natord}}$, of the natural numbers is a module with signature

$$\sum X{:}\sigma_{\mathsf{nateq}} \cdot \sigma^X_{\mathsf{ord}}.$$

By covariance of the hierarchical signature, this signature is a subsignature of $\sigma_{\mathsf{eqord}}$, so that by subsumption we may regard $M_{\mathsf{natord}}$ as a module of the latter signature. The static part of the subsignature is a singleton, so we may apply the rules of sharing propagation given in Chapter 25 to show that the subsignature is equivalent to the signature

$$\sum X{:}\sigma_{\mathsf{nateq}} \cdot \sigma_{\mathsf{natord}},$$

where $\sigma_{\mathsf{natord}}$ is the closed signature

$$[t :: \mathtt{S}(\mathtt{nat}); \langle \mathtt{lt} \hookrightarrow (t \times t) \to \mathtt{bool} \rangle].$$

Preview

Notice that sharing propagation has replaced the type $X \cdot \mathtt{s}$ in the signature with `nat`, eliminating the dependency on the module variable $X$. After another round of sharing propagation, this signature may be shown to be equivalent to the signature $\rho_{\mathsf{natord}}$ given by

$$[_ :: \mathtt{S}(\mathtt{nat}); \langle \mathtt{lt} \hookrightarrow (\mathtt{nat} \times \mathtt{nat}) \to \mathtt{bool} \rangle].$$

Here we have replaced both occurrences of $t$ in the type of the comparison operation with `nat` as a consequence of the kind of $t$. The net effect is to propagate the identity of the static part of $M_{\mathsf{natord}}$ to the signature of the second component of $M_{\mathsf{natord}}$.

Although its value is a pair, which seems symmetric, a module of signature $\sigma_{\mathsf{eqord}}$ is asymmetric in that the signature of the second component is dependent on the first component itself. This dependence is made manifest by the occurrence of the module variable, $X$, in the signature $\sigma_{\mathsf{ord}}$. Thus, for $\langle M_{\mathsf{eq}} \,;\, M_{\mathsf{ord}} \rangle$ to be a well-formed module of signature $\sigma_{\mathsf{eqord}}$, the first component, $M_{\mathsf{eq}}$, must have signature $\sigma_{\mathsf{eq}}$, which is meaningful independently of the other component of the pair. On the other hand, the second component, $M_{\mathsf{ord}}$, must have signature $\sigma_{\mathsf{eq}}^{X}$, with the understanding that $X$ stands for the module $M_{\mathsf{eq}}$. In general this signature is not meaningful independently of $M_{\mathsf{eq}}$ itself, and hence it may not be possible to handle $M_{\mathsf{ord}}$ independently of $M_{\mathsf{eq}}$.

Turning this the other way around, if $M$ is *any* module of signature $\sigma_{\mathsf{eqord}}$, then it is always sensible to project it onto its first coordinate to obtain a module, $M \cdot 1$, of signature $\sigma_{\mathsf{eq}}$. But it is not always sensible to project it onto its second coordinate, because it may not be possible to give a signature to $M \cdot 2$ in the case that the dependency on the first component cannot be resolved statically. This can happen if, for example, the $M \cdot 1$ is a sealed module, whose static part cannot be formed in order to ensure representation independence. In such a situation the dependence of the signature $\sigma_{\mathsf{ord}}^{X}$ on the module variable, $X$, cannot be eliminated, and so no signature may be given to the second projection. For this reason the first component of a module hierarchy is called a *submodule* of the hierarchy, whereas the second component may or may not be a submodule of it. Put in other terms, the second component of a hierarchy is "projectible" exactly when the dependence of its signature on the first component is eliminable by sharing propagation. That is, we may know enough about the first component statically to ensure that an independent type for the second component may be given. In that case the second component may be considered to be a submodule of the pair; otherwise, the second is inseparable from the first, and therefore cannot be projected from the pair.

Consider, for example, a module $M_{\mathsf{natord}}$ of signature $\sigma_{\mathsf{natord}}$, which, we noted earlier, is a subsignature of $\sigma_{\mathsf{eqord}}$. The first projection $M_{\mathsf{natord}} \cdot 1$ is a well-formed module of closed signature $\sigma_{\mathsf{eq}}$ and hence is a submodule of $M_{\mathsf{natord}}$. The situation is less clear for the second projection, $M_{\mathsf{natord}} \cdot 2$, because its signature, $\sigma_{\mathsf{ord}}^{X}$, depends on the first component via the variable $X$. However, we noted above that the signature $\sigma_{\mathsf{natord}}$ is equivalent to the signature

$$\sum _ : \sigma_{\mathsf{nateq}} \cdot \rho_{\mathsf{natord}}$$

in which the dependency on $X$ has been eliminated by sharing propagation. This, too, is a valid signature for $M_{\mathsf{natord}}$, and hence the second projection, $M_{\mathsf{natord}} \cdot 2$ is a well-formed module of closed signature $\rho_{\mathsf{natord}}$. If, on the other hand, the *only* signature available for $M_{\mathsf{natord}}$ were $\sigma_{\mathsf{eqord}}$, then the second projection would be ill-formed—the second component would not be separable from the first, and hence could not be considered a submodule of the pair.

Preview

The hierarchical dependency of the signature of the second component of a pair on the first component gives rise to a useful alternative interpretation of a hierarchical module signature as describing a *family of modules* given by the second component thought of as being indexed by the first component. In the case at hand, the totality of modules of the signature $\sigma_{\mathsf{eqord}}$ gives rise to a family of modules of signature $\sigma^X_{\mathsf{ord}}$, where $X$ ranges over $\sigma_{\mathsf{eq}}$. That is, to each choice, $M_{\mathsf{eq}}$, of signature $\sigma_{\mathsf{eq}}$, we associate the collection of choices, $M_{\mathsf{ord}}$, coherent with the first choice in accordance with the sharing constraint in $\sigma^X_{\mathsf{ord}}$, taking $X$ to be $M_{\mathsf{ord}}$. This collection is called the *fibre over* $M_{eq}$, and the totality of modules of signature $\sigma_{\mathsf{eqord}}$ is said to be *fibred over* $\sigma_{eq}$ (by the first projection).

The preceding example illustrates the layering of one type class on top of another. It is also useful to layer a type abstraction over a type class. A good example is provided by a dictionary abstraction in which the type of keys is required to be of the class of ordered types, but is otherwise unspecified. The signature, $\sigma_{\mathsf{keydict}}$, of such a dictionary is given as follows:

$$\sum X{:}\sigma_{\mathsf{eqord}} \cdot \sigma^X_{\mathsf{dict}},$$

where $\sigma_{\mathsf{eqord}}$ is the signature of ordered equality types (in either of the two forms discussed above), and $\sigma^X_{\mathsf{dict}}$ is the signature of dictionaries of some type $\tau$ given as follows:

$$[t :: \mathtt{T}; \langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow X \cdot \mathtt{s} \times \tau \times t \to t, \mathtt{fnd} \hookrightarrow X \cdot \mathtt{s} \times t \to \tau\ \mathtt{opt} \rangle].$$

The `ins` and `fnd` operations make use of the type $X \cdot \mathtt{s}$ of keys given by the submodule of the dictionary module. We may think of $\sigma_{\mathsf{keydict}}$ as specifying a family of dictionary modules, one for each choice of the ordered type of keys. Regardless of the interpretation, an implementation of the signature $\sigma_{\mathsf{keydict}}$ consists of a two-level hierarchy of the form $\langle M_1 \,;\, M_2 \rangle$, where $M_1$ specifies the key type and its ordering, and $M_2$ implements the dictionary for keys of this type in terms of this ordering.

## 47.2 Parameterizaton

The signature $\sigma_{\mathsf{keydict}}$ may be understood as describing a family of dictionary modules indexed by a module of ordered keys. The totality of such modules evaluate to pairs consisting of the ordered type of keys together with the dictionary *per se*, specialized to that choice of keys. Although it is possible that the code of the dictionary operations differs for each choice of keys, it is more often the case that the *same* implementation may be used for *all* choices of keys, the only difference being that references to, say, $X \cdot \mathtt{lt}$ refers to a different function for each choice of key module, $X$.

Such a uniform implementation of dictionaries is provided by the concept of a *parameterized module*, or *functor*. A functor is a module expressed as a function of an unknown module of specified signature. The uniform dictionary module would be expressed as a functor parameterized over the module implemeting keys, which is to say as a $\lambda$-abstraction of the form

$$\lambda\, Z{:}\sigma_{\mathsf{eqord}} \cdot M_{\mathsf{keydict}}.$$



Preview

Here $M_{\mathsf{keydict}}$ is the generic implementation of dictionaries in terms of an unspecified module, $Z$, of signature $\sigma_{\mathsf{eqord}}$. The signature of $Z$ expresses the requirement that the dictionary implementation relies on the keys being an ordered type, but makes no other requirement on it.

A functor is a form of module, and hence has a signature as well, called (oddly enough) a *functor signature*. The signature, $\sigma_{\mathsf{dictfun}}$, of the functor $M_{\mathsf{keydict}}$ has the form

$$\prod Z{:}\sigma_{\mathsf{eqord}} \cdot \rho^{Z}_{\mathsf{keydict}},$$

which specifies that its *domain* is the signature, $\sigma_{\mathsf{eqord}}$, of ordered types, and whose *range* is a signature, $\rho^{Z}_{\mathsf{keydict}}$, depends on the module $Z$.

The range, $\rho^{Z}_{\mathsf{keydict}}$, of the dictionary functor is defined to be a subsignature of $\sigma_{\mathsf{keydict}}$ in which the key type of the result is constrained to be the same as the key type given as argument. This constraint may be expressed by defining $\rho^{Z}_{\mathsf{keydict}}$ to be the hierarchical signature

$$\sum X{:}\rho^{Z}_{\mathsf{eqord}} \cdot \sigma^{X}_{\mathsf{dict}},$$

where $\rho^{Z}_{\mathsf{eqord}}$ is a subsignature of $\sigma_{\mathsf{eqord}}$ imposing the desired sharing constraint. This is itself a hierarchical signature of the form

$$\sum X{:}\rho^{Z}_{\mathsf{eq}} \cdot \sigma^{X}_{\mathsf{ord}},$$

where $\rho^{Z}_{\mathsf{eq}}$ is the subsignature of $\sigma_{\mathsf{eq}}$ given by

$$[t :: \mathtt{S}(Z \cdot \mathtt{1} \cdot \mathtt{s}); \langle \mathsf{eq} \hookrightarrow (t \times t) \to \mathtt{bool} \rangle].$$

The singleton kind in $\rho^{Z}_{\mathsf{eq}}$ expresses the required sharing constraint between the key type in the result of the functor and the key type given as its argument.

It is evidently rather tedious to write out $\rho^{Z}_{\mathsf{eqord}}$ separately from $\sigma_{\mathsf{eqord}}$, because it requires repetition of so much that is already present in the latter signature. To lighten the notation it is preferable to express more directly the close relationship between the two signatures using *signature modification* to impose a type sharing constraint on a given signature. In the present case we may define $\rho^{Z}_{\mathsf{eqord}}$ to be the modification of $\sigma_{\mathsf{eqord}}$ given by

$$Y : \sigma_{\mathsf{eqord}} \,/\, Y \cdot \mathtt{1} \cdot \mathtt{s} = Z \cdot \mathtt{1} \cdot \mathtt{s}.$$

The modification states that the signature $\sigma_{\mathsf{eqord}}$ is to be altered by imposing the constraint that the static part of its equality part is to share with the static part of the equality part of $Z$, which, recall, also has signature $\sigma^{\mathsf{eqord}}$. We may similarly define the range signature, $\rho^{Z}_{\mathsf{keydict}}$, of the dictionary functor using signature modification as follows:

$$Y : \sigma_{\mathsf{keydict}} \,/\, Y \cdot \mathtt{1} \cdot \mathtt{1} \cdot \mathtt{s} = Z \cdot \mathtt{1} \cdot \mathtt{s}.$$

The left-hand side of the sharing constraint refers to the type of keys in the submodule of the dictionary, and the right-hand side refers to the type of keys given as argument to the functor.

The dictionary functor, $M_{\mathsf{dictfun}}$, defines a generic implementation of dictionaries in terms of an ordered type of keys. An instance of the dictionary for a specific choice of keys is obtained by

Preview

*applying*, or *instantiating*, it with a module of its domain signature, $\sigma_{\mathsf{eqord}}$. For example, because $M_{\mathsf{natord}}$, the type of natural numbers ordered in the usual way, is such a module, we may form the instance $M_{\mathsf{keydict}}\,(M_{\mathsf{natord}})$ to obtain a dictionary with numeric keys. By choosing other modules of signature $\sigma_{\mathsf{eqord}}$ we may obtain corresponding instances of the dictionary functor. More generally, if $M$ is any module of signature $\sigma_{\mathsf{dictfun}}$, then it is a functor that we may apply it to any module, $M_{\mathsf{key}}$, of signature $\sigma_{\mathsf{eqord}}$ to obtain the instance $M\,(M_{\mathsf{key}})$.

But what is the signature of such an instance, and how may it be deduced? Recall that the result signature of $\sigma_{\mathsf{dictfun}}$ is dependent on the argument itself, and not just its signature. It is therefore not immediately clear what signature to assign to the instance; the dependency on the argument must be resolved in order to obtain a signature that makes sense independently of the argument. The situation is broadly similar to the problem of computing the signature of the second component of a hiearchical module, and similar methods are used to resolve the dependencies, namely to exploit subtyping for signatures to obtain a specialization of the result signature appropriate to the argument.

This is best illustrated by example. First, we note that by contravariance of subtyping for functor signatures, we may *weaken* a functor signature by *strengthening* its domain signature. In the case of the signature $\sigma_{\mathsf{dictfun}}$ of the dictionary functor, we may obtain a supersignature $\sigma_{\mathsf{natdictfun}}$ by strengthening its domain to require that the key type be the type of natural numbers:

$$\prod Z{:}\sigma_{\mathsf{natord}} \cdot \rho^{Z}_{\mathsf{keydict}}.$$

Fixing $Z$ to be a module variable of the specialized signature $\sigma_{\mathsf{natord}}$, the range signature, $\rho^{Z}_{\mathsf{keydict}}$, is given by the modification

$$Y : \sigma_{\mathsf{keydict}} \,/\, Y \cdot \mathtt{1} \cdot \mathtt{1} \cdot \mathtt{s} = Z \cdot \mathtt{1} \cdot \mathtt{s}.$$

By sharing propagation this is equivalent to the closed signature, $\rho_{\mathsf{natdict}}$, given by

$$Y : \sigma_{\mathsf{keydict}} \,/\, Y \cdot \mathtt{1} \cdot \mathtt{1} \cdot \mathtt{s} = \mathtt{nat},$$

because we may derive the equivalence of $Z \cdot \mathtt{1} \cdot \mathtt{s}$ and `nat` once the signature of $Z$ is specialized to $\sigma_{\mathsf{natord}}$.

Now by subsumption if $M$ is a module of signature $\sigma_{\mathsf{dictfun}}$, then $M$ is also a module of the supersignature

$$\prod Z{:}\sigma_{\mathsf{natord}} \cdot \rho^{Z}_{\mathsf{keydict}}.$$

We have just shown that the latter signature is equivalent to the non-dependent functor signature

$$\prod _{:}\sigma_{\mathsf{natord}} \cdot \rho_{\mathsf{natdict}}.$$

The range is now given independently of the argument, so we may deduce that if $M_{\mathsf{natkey}}$ has signature $\sigma_{\mathsf{natord}}$, then the application $M\,(M_{\mathsf{natkey}})$ has the signature $\rho_{\mathsf{natdict}}$.

The crucial point is that the dependence of the range signature on the domain signature is eliminated by propagating knowledge about the type components of the argument itself. Absent this knowledge, the functor application cannot be regarded as well-formed, much as the second projection from a hierarchy cannot be admitted if the dependency of its signature on the first component cannot be eliminated. If the argument to the functor is a value, then it is always possible to find a signature for it that maximizes the propagation of type sharing information so that the dependency of the range on the argument can always be eliminated.

Preview

## 47.3 Extending Modules with Hierarchies and Parameterization

In this section we sketch the extension of the module language introduced in Chapter 46 to account for module hierarchies and module parameterization.

The syntax of **Mod** is enriched with the following clauses:

| | | | | | |
|---|---|---|---|---|---|
| Sig | $\sigma$ | ::= | $\texttt{hier}(\sigma_1;X.\sigma_2)$ | $\sum X{:}\sigma_1 \cdot \sigma_2$ | hierarchy |
| | | | $\texttt{fun}(\sigma_1;X.\sigma_2)$ | $\prod X{:}\sigma_1 \cdot \sigma_2$ | functor |
| Mod | $M$ | ::= | $\texttt{hier}(M_1;M_2)$ | $\langle M_1 ; M_2 \rangle$ | hierarchy |
| | | | $\texttt{fst}(M)$ | $M \cdot 1$ | first component |
| | | | $\texttt{snd}(M)$ | $M \cdot 2$ | second component |
| | | | $\texttt{fun}[\sigma](X.M)$ | $\lambda\, X{:}\sigma \cdot M$ | functor |
| | | | $\texttt{app}(M_1;M_2)$ | $M_1\,(M_2)$ | instance |

The syntax of signatures is extended to include hierarchies and functors, and the syntax of modules is correspondingly extended with introduction and elimination forms for these signatures.

The judgment $M$ projectible states that the module, $M$, is *projectible* in the sense that its constituent types may be referenced by compositions of projections, including the static part of a structure. This judgment is inductively defined by the following rules:

$$\frac{}{\Gamma, X : \sigma \vdash x \text{ projectible}} \tag{47.1a}$$

$$\frac{\Gamma \vdash M_1 \text{ projectible} \quad \Gamma \vdash M_2 \text{ projectible}}{\Gamma \vdash \langle M_1 ; M_2 \rangle \text{ projectible}} \tag{47.1b}$$

$$\frac{\Gamma \vdash M \text{ projectible}}{\Gamma \vdash M \cdot 1 \text{ projectible}} \tag{47.1c}$$

$$\frac{\Gamma \vdash M \text{ projectible}}{\Gamma \vdash M \cdot 2 \text{ projectible}} \tag{47.1d}$$

All module variables are deemed projectible, even though this condition is only relevant for hierarchies of basic structures. Because the purpose of sealing is to hide the representation of an abstract type, no sealed module is deemed projectible. Furthermore, no functor is projectible, because there is no concept of projection for a functor. More importantly, no functor instance is projectible either. This ensures that any two instances of the same functor define distinct abstract types; functors are therefore said to be *generative*. (See Section 47.4 for a discussion of an alternative treatment of functors.)

The signature formation judgment is extended to include these rules:

$$\frac{\Gamma \vdash \sigma_1 \text{ sig} \quad \Gamma, X : \sigma_1 \vdash \sigma_2 \text{ sig}}{\Gamma \vdash \sum X{:}\sigma_1 \cdot \sigma_2 \text{ sig}} \tag{47.2a}$$

$$\frac{\Gamma \vdash \sigma_1 \text{ sig} \quad \Gamma, X : \sigma_1 \vdash \sigma_2 \text{ sig}}{\Gamma \vdash \prod X{:}\sigma_1 \cdot \sigma_2 \text{ sig}} \tag{47.2b}$$

Preview

Signature equivalence is defined to be compatible with the two new forms of signature:

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_1' \quad \Gamma, X : \sigma_1 \vdash \sigma_2 \equiv \sigma_2'}{\Gamma \vdash \sum X{:}\sigma_1 \cdot \sigma_2 \equiv \sum X{:}\sigma_1' \cdot \sigma_2'} \tag{47.3a}$$

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_1' \quad \Gamma, X : \sigma_1 \vdash \sigma_2 \equiv \sigma_2'}{\Gamma \vdash \prod X{:}\sigma_1 \cdot \sigma_2 \equiv \prod X{:}\sigma_1' \cdot \sigma_2'} \tag{47.3b}$$

The subsignature judgment is augmented with the following rules:

$$\frac{\Gamma \vdash \sigma_1 <: \sigma_1' \quad \Gamma, X : \sigma_1 \vdash \sigma_2 <: \sigma_2'}{\Gamma \vdash \sum X{:}\sigma_1 \cdot \sigma_2 <: \sum X{:}\sigma_1' \cdot \sigma_2'} \tag{47.4a}$$

$$\frac{\Gamma \vdash \sigma_1' <: \sigma_1 \quad \Gamma, X : \sigma_1' \vdash \sigma_2 <: \sigma_2'}{\Gamma \vdash \prod X{:}\sigma_1 \cdot \sigma_2 <: \prod X{:}\sigma_1' \cdot \sigma_2'} \tag{47.4b}$$

Rule (47.4a) specifies that the hierarchical signature is covariant in both positions, whereas rule (47.4b) specifies that the functor signature is contravariant in its domain and covariant in its range.

The statics of module expressions is extended by the following rules:

$$\frac{\Gamma \vdash M_1 : \sigma_1 \quad \Gamma \vdash M_2 : \sigma_2}{\Gamma \vdash \langle M_1 \,;\, M_2 \rangle : \sum _{:}\sigma_1 \cdot \sigma_2} \tag{47.5a}$$

$$\frac{\Gamma \vdash M : \sum X{:}\sigma_1 \cdot \sigma_2}{\Gamma \vdash M \cdot 1 : \sigma_1} \tag{47.5b}$$

$$\frac{\Gamma \vdash M : \sum _{:}\sigma_1 \cdot \sigma_2}{\Gamma \vdash M \cdot 2 : \sigma_2} \tag{47.5c}$$

$$\frac{\Gamma, X : \sigma_1 \vdash M_2 : \sigma_2}{\Gamma \vdash \lambda\, X{:}\sigma_1 \cdot M_2 : \prod X{:}\sigma_1 \cdot \sigma_2} \tag{47.5d}$$

$$\frac{\Gamma \vdash M_1 : \prod _{:}\sigma_2 \cdot \sigma \quad \Gamma \vdash M_2 : \sigma_2}{\Gamma \vdash M_1\,(M_2) : \sigma} \tag{47.5e}$$

Rule (47.5a) states that an explicit module hierarchy is given a signature in which there is no dependency of the signature of the second component on the first component (indicated here by the underscore in place of the module variable). A dependent signature may be given to a hierarchy by sealing, which makes it into a non-value, even if the components are values. Rule (47.5b) states that the first projection is defined for general hierarchical signatures. On the other hand, rule (47.5c) restricts the second projection to non-dependent hierarchies, as discussed in the preceding section. Similarly, rule (47.5e) restricts instantiation to functors whose types are non-dependent, forcing any dependencies to be resolved using the subsignature relation and sharing propagation prior to application.

The self-recognition rules given in Chapter 46 are extended to account for the formation of hierarchical module value by the following rules:

$$\frac{\Gamma \vdash M \text{ projectible} \quad \Gamma \vdash M : \sum X{:}\sigma_1 \cdot \sigma_2 \quad \Gamma \vdash M \cdot 1 : \sigma_1'}{\Gamma \vdash M : \sum X{:}\sigma_1' \cdot \sigma_2} \tag{47.6a}$$

Preview

$$\frac{\Gamma \vdash M \text{ projectible} \quad \Gamma \vdash M : \sum _ : \sigma_1 \cdot \sigma_2 \quad \Gamma \vdash M \cdot 2 : \sigma_2'}{\Gamma \vdash M : \sum _ : \sigma_1 \cdot \sigma_2'} \tag{47.6b}$$

Rules (47.6a) and (47.6b) permit the specialization of the signature of a hierarchical module value to express that its constructor components are equivalent to their projections from the module itself.

## 47.4 Applicative Functors

In the module language just described functors are regarded as generative in the sense that any two instances, even with arguments, are considered to "generate" distinct abstract types. This is ensured by treating a functor application, $M(M_1)$, to be non-projectible, so that if it defines an abstract type in the result, that type cannot be referenced without first binding the application to a variable. Any two such bindings are necessarily to distinct variables, $X$ and $Y$, and so the abstract types $X \cdot \mathsf{s}$ and $Y \cdot \mathsf{s}$ are distinct, regardless of their bindings.

The justification for this design decision merits careful consideration. By treating functors as generative, we are ensuring that a client of the functor cannot in any way rely on the implementation of that functor. That is, we are extending the principle of representation independence for abstract types to functors in a natural way. One consequence of this policy is that the module language is compatible with extensions such as a *conditional module* that branches on an arbitrary dynamic condition that might even depend on external conditions such as the phase of the moon! A functor with such an implementation *must* be considered generative, because the abstract types arising from any instance cannot be regarded as well-defined until the moment when the application is evaluated, which amounts to the point at which it is bound to a variable. By regarding all functors as generative we are, in effect, maximizing opportunities to exploit changes of representation without disrupting the behavior of clients of the functor, a bedrock principle of modular decomposition.

But because the module language considered in the preceding section does not include anything so powerful as a conditional module, we might consider that the restriction to generative functors is too severe, and may be usefully relaxed. One such alternative is the concept of an *applicative* functor. An applicative functor is one for which instances by values are regarded as projectible:[1]

$$\frac{M \text{ projectible} \quad M_1 \text{ val}}{M(M_1) \text{ projectible}} \tag{47.7}$$

It is important to bear in mind that because of this rule applicative functors are *not compatible* with conditional modules. Thus, a module language based on applicative functors is inherently restricted as compared to one based on generative functors.

The benefit of regarding a functor instance as projectible is that we may form types such as $(M(M_1)) \cdot \mathsf{s}$, which projects the static part of the instance. But this raises the question of when two such type expressions are to be deemed equivalent? The difficulty is that the answer to this question depends on the functor argument. For suppose that $F$ is an applicative functor variable,

[1] We may, in addition, regard functor abstractions as projectible, but because all variables are projectible, there is no harm in omitting this and instead insisting that functors be bound to variables before being used.



Preview

under what conditions should $(F(M_1)) \cdot \mathsf{s}$ and $(F(M_2)) \cdot \mathsf{s}$ be regarded as the same type? In the case of generative functors we did not have face this question, because the instances are not projectible, but for applicative functors the question cannot be dodged, but must be addressed. We will return to this point in a moment, after considering one further complication that raises a similar issue.

The difficulty is that the body of an applicative functor cannot be sealed to impose abstraction, and, according to the rules given in the preceding section, no sealed module is projectible. Because sealing is the only means of imposing abstraction, we must relax this condition and allow sealed projectible modules to be projectible:

$$\frac{M \text{ projectible}}{M \upharpoonright \sigma \text{ projectible}} \tag{47.8}$$

Thus, we may form type expressions of the form $(M \upharpoonright \sigma) \cdot \mathsf{s}$, which project the static part of a sealed module. And once again we are faced with the issue that the equivalence of two such types must involve the equivalence of the sealed modules themselves, in apparent violation of representation independence.

Summarizing, if we are to deem functors to be applicative, then some compromise of the principle of representation independence for abstract types is required. We must define equivalence for the static parts of sealed modules, and doing so requires at least checking whether the underlying modules are identical. This has two consequences. Because the underlying modules have both static and dynamic parts, this means comparing their executable code for equivalence during type checking. More significantly, because the formation of a client may depend on the equivalence of two modules, we cannot change the representation of a sealed module without fear of disrupting the typing or behavior of the client. This undermines the very purpose of having a module system in the first place!

## 47.5 Notes

Module hierarchies and functors in the form discussed here were introduced by Milner et al. (1997), which also employed the reading of a module hierarchy as an indexed family of modules. The theory of hierarchies and functors was first studied by Harper and Lillibridge (1994) and Leroy (1994), building on earlier work by Mitchell and Plotkin (1988) on existential types. The concept of an applicative functor was introduced by Leroy (1995) and is central to the module system of O'Caml (OCaml).

Preview

# Part XIX

# Equational Reasoning

Preview

Preview

# Chapter 48

# Equational Reasoning for T

The beauty of functional programming is that equality of expressions in a functional language corresponds very closely to familiar patterns of mathematical reasoning. For example, in the language **T** of Chapter 9 in which we can express addition as the function `plus`, the expressions

$$\lambda\,(x{:}\mathtt{nat})\,\lambda\,(y{:}\mathtt{nat})\,\mathtt{plus}(x)(y)$$

and

$$\lambda\,(x{:}\mathtt{nat})\,\lambda\,(y{:}\mathtt{nat})\,\mathtt{plus}(y)(x)$$

are equal. In other words, the addition function *as programmed in* **T** is commutative.

This may seem to be obviously true, but *why*, precisely, is it so? More importantly, what do we even *mean* for two expressions to be equal in this sense? It is intuitively obvious that these two expressions are not *definitionally* equivalent, because they cannot be shown equivalent by symbolic execution. We may say that these two expressions are definitionally inequivalent because they describe different algorithms: one proceeds by recursion on $x$, the other by recursion on $y$. On the other hand, the two expressions are interchangeable in any complete computation of a natural number, because the only use we can make of them is to apply them to arguments and compute the result. Two functions are *logically equivalent* if they give equal results for equal arguments—in particular, they agree on all possible arguments. Because their behavior on arguments is all that matters for calculating observable results, we may expect that logically equivalent functions are equal in the sense of being interchangeable in all complete programs. Thinking of the programs in which these functions occur as *observations* of their behavior, these functions are said to be *observationally equivalent*. The main result of this chapter is that observational and logical equivalence coincide for a variant of **T** in which the successor is evaluated eagerly, so that a value of type `nat` is a numeral.

## 48.1 Observational Equivalence

When are two expressions equal? Whenever we cannot tell them apart! This may seem tautological, but it is not, because it depends on what we consider to be a means of telling expressions

Preview

apart. What "experiment" are we permitted to perform on expressions in order to distinguish them? What counts as an observation that, if different for two expressions, is a sure sign that they are different?

If we permit ourselves to consider the syntactic details of the expressions, then very few expressions could be considered equal. For example, if it is deemed significant that an expression contains, say, more than one function application, or that it has an occurrence of $\lambda$-abstraction, then very few expressions would come out as equivalent. But such considerations seem silly, because they conflict with the intuition that the significance of an expression lies in its contribution to the *outcome* of a computation, and not to the process of obtaining that outcome. In short, if two expressions make the same contribution to the outcome of a complete program, then they ought to be regarded as equal.

We must fix what we mean by a complete program. Two considerations inform the definition. First, the dynamics of **T** is given only for expressions without free variables, so a complete program should clearly be a *closed* expression. Second, the outcome of a computation should be *observable*, so that it is evident whether the outcome of two computations differs or not. We define a *complete program* to be a closed expression of type `nat`, and define the *observable behavior* of the program to be the numeral to which it evaluates.

An *experiment* on, or *observation* about, an expression is any means of using that expression within a complete program. We define an *expression context* to be an expression with a "hole" in it serving as a placeholder for another expression. The hole is permitted to occur anywhere, including within the scope of a binder. The bound variables within whose scope the hole lies are said to be *exposed to capture* by the expression context. These variables may be assumed, without loss of generality, to be distinct from one another. A *program context* is a closed expression context of type `nat`—that is, it is a complete program with a hole in it. The meta-variable $\mathcal{C}$ stands for any expression context.

*Replacement* is the process of filling a hole in an expression context, $\mathcal{C}$, with an expression, $e$, which is written $\mathcal{C}\{e\}$. Importantly, the free variables of $e$ that are exposed by $\mathcal{C}$ are *captured* by replacement (which is why replacement is not a form of substitution, which is defined so as to avoid capture). If $\mathcal{C}$ is a program context, then $\mathcal{C}\{e\}$ is a complete program iff all free variables of $e$ are captured by the replacement. For example, if $\mathcal{C} = \lambda\,(x{:}\mathtt{nat})\,\circ$, and $e = x + x$, then

$$\mathcal{C}\{e\} = \lambda\,(x{:}\mathtt{nat})\,x + x.$$

The free occurrences of $x$ in $e$ are captured by the $\lambda$-abstraction as a result of the replacement of the hole in $\mathcal{C}$ by $e$.

We sometimes write $\mathcal{C}\{\circ\}$ to emphasize the occurrence of the hole in $\mathcal{C}$. Expression contexts are closed under *composition* in that if $\mathcal{C}_1$ and $\mathcal{C}_2$ are expression contexts, then so is

$$\mathcal{C}\{\circ\} \triangleq \mathcal{C}_1\{\mathcal{C}_2\{\circ\}\},$$

and we have $\mathcal{C}\{e\} = \mathcal{C}_1\{\mathcal{C}_2\{e\}\}$. The *trivial*, or *identity*, expression context is the "bare hole", written $\circ$, for which $\circ\{e\} = e$.

The statics of expressions of **T** is extended to expression contexts by defining the typing judgment

$$\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$$



Preview

so that if $\Gamma \vdash e : \tau$, then $\Gamma' \vdash \mathcal{C}\{e\} : \tau'$. This judgment may be inductively defined by a collection of rules derived from the statics of **T** (see rules (9.1)). Some representative rules are as follows:

$$\frac{}{\circ : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma \rhd \tau)} \tag{48.1a}$$

$$\frac{\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat})}{\mathtt{s}(\mathcal{C}) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat})} \tag{48.1b}$$

$$\frac{\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \mathtt{nat}) \quad \Gamma' \vdash e_0 : \tau' \quad \Gamma', x : \mathtt{nat}, y : \tau' \vdash e_1 : \tau'}{\mathtt{rec}\,\mathcal{C}\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x)\,\mathtt{with}\,y \hookrightarrow e_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{48.1c}$$

$$\frac{\Gamma' \vdash e : \mathtt{nat} \quad \mathcal{C}_0 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau') \quad \Gamma', x : \mathtt{nat}, y : \tau' \vdash e_1 : \tau'}{\mathtt{rec}\,e\,\{\mathtt{z} \hookrightarrow \mathcal{C}_0 \mid \mathtt{s}(x)\,\mathtt{with}\,y \hookrightarrow e_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{48.1d}$$

$$\frac{\Gamma' \vdash e : \mathtt{nat} \quad \Gamma' \vdash e_0 : \tau' \quad \mathcal{C}_1 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma', x : \mathtt{nat}, y : \tau' \rhd \tau')}{\mathtt{rec}\,e\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x)\,\mathtt{with}\,y \hookrightarrow \mathcal{C}_1\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{48.1e}$$

$$\frac{\mathcal{C}_2 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma', x : \tau_1 \rhd \tau_2)}{\lambda\,(x{:}\tau_1)\,\mathcal{C}_2 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_1 \to \tau_2)} \tag{48.1f}$$

$$\frac{\mathcal{C}_1 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_2 \to \tau') \quad \Gamma' \vdash e_2 : \tau_2}{\mathcal{C}_1(e_2) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{48.1g}$$

$$\frac{\Gamma' \vdash e_1 : \tau_2 \to \tau' \quad \mathcal{C}_2 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_2)}{e_1(\mathcal{C}_2) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')} \tag{48.1h}$$

**Lemma 48.1.** *If $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$, then $\Gamma' \subseteq \Gamma$, and if $\Gamma \vdash e : \tau$, then $\Gamma' \vdash \mathcal{C}\{e\} : \tau'$.*

Contexts are closed under composition, with the trivial context acting as an identity for it.

**Lemma 48.2.** *If $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$, and $\mathcal{C}' : (\Gamma' \rhd \tau') \rightsquigarrow (\Gamma'' \rhd \tau'')$, then $\mathcal{C}'\{\mathcal{C}\{\circ\}\} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma'' \rhd \tau'')$.*

**Lemma 48.3.** *If $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$ and $x \notin dom(\Gamma)$, then $\mathcal{C} : (\Gamma, x : \tau'' \rhd \tau) \rightsquigarrow (\Gamma', x : \tau'' \rhd \tau')$.*

*Proof.* By induction on rules (48.1). □

A *complete program* is a closed expression of type `nat`.

**Definition 48.4.** *Two complete programs, $e$ and $e'$, are* Kleene equal, *written $e \simeq e'$, iff there exists $n \geq 0$ such that $e \longmapsto^* \overline{n}$ and $e' \longmapsto^* \overline{n}$.*

Kleene equality is evidently reflexive and symmetric; transitivity follows from determinacy of evaluation. Closure under converse evaluation also follows directly from determinacy. It is immediate from the definition that $\overline{0} \not\simeq \overline{1}$.

**Definition 48.5.** *Suppose that $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$ are two expressions of the same type. Two such expressions are* observationally equivalent, *written $\Gamma \vdash e \cong e' : \tau$, iff $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$ for every program context $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\emptyset \rhd \mathtt{nat})$.*



Preview

In other words, for all possible experiments, the outcome of an experiment on $e$ is the same as the outcome on $e'$. This is obviously an equivalence relation. For the sake of brevity, we often write $e \cong_\tau e'$ for $\emptyset \vdash e \cong e' : \tau$.

A family of equivalence relations $\Gamma \vdash e_1 \mathcal{E} e_2 : \tau$ is a *congruence* iff it is preserved by all contexts. That is,

$$\text{if } \Gamma \vdash e \mathcal{E} e' : \tau, \text{ then } \Gamma' \vdash \mathcal{C}\{e\} \mathcal{E} \mathcal{C}\{e'\} : \tau'$$

for every expression context $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$. Such a family of relations is *consistent* iff $\emptyset \vdash e \mathcal{E} e' : \mathtt{nat}$ implies $e \simeq e'$.

**Theorem 48.6.** *Observational equivalence is the coarsest consistent congruence on expressions.*

*Proof.* Consistency follows directly from the definition by noting that the trivial context is a program context. Observational equivalence is obviously an equivalence relation. To show that it is a congruence, we need only observe that type-correct composition of a program context with an arbitrary expression context is again a program context. Finally, it is the coarsest such equivalence relation, for if $\Gamma \vdash e \mathcal{E} e' : \tau$ for some consistent congruence $\mathcal{E}$, and if $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$, then by congruence $\emptyset \vdash \mathcal{C}\{e\} \mathcal{E} \mathcal{C}\{e'\} : \mathtt{nat}$, and hence by consistency $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$. □

A *closing substitution*, $\gamma$, for the typing context $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ is a finite function assigning closed expressions $e_1 : \tau_1, \ldots, e_n : \tau_n$ to $x_1, \ldots, x_n$, respectively. We write $\hat{\gamma}(e)$ for the substitution $[e_1, \ldots, e_n / x_1, \ldots, x_n]e$, and write $\gamma : \Gamma$ to mean that if $x : \tau$ occurs in $\Gamma$, then there exists a closed expression, $e$, such that $\gamma(x) = e$ and $e : \tau$. We write $\gamma \cong_\Gamma \gamma'$, where $\gamma : \Gamma$ and $\gamma' : \Gamma$, to express that $\gamma(x) \cong_{\Gamma(x)} \gamma'(x)$ for each $x$ declared in $\Gamma$.

**Lemma 48.7.** *If $\Gamma \vdash e \cong e' : \tau$ and $\gamma : \Gamma$, then $\hat{\gamma}(e) \cong_\tau \hat{\gamma}(e')$. Moreover, if $\gamma \cong_\Gamma \gamma'$, then $\hat{\gamma}(e) \cong_\tau \widehat{\gamma'}(e)$ and $\hat{\gamma}(e') \cong_\tau \widehat{\gamma'}(e')$.*

*Proof.* Let $\mathcal{C} : (\emptyset \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$ be a program context; we are to show that $\mathcal{C}\{\hat{\gamma}(e)\} \simeq \mathcal{C}\{\hat{\gamma}(e')\}$. Because $\mathcal{C}$ has no free variables, this is equivalent to showing that $\hat{\gamma}(\mathcal{C}\{e\}) \simeq \hat{\gamma}(\mathcal{C}\{e'\})$. Let $\mathcal{D}$ be the context

$$\lambda\,(x_1{:}\tau_1) \ldots \lambda\,(x_n{:}\tau_n)\, \mathcal{C}\{\circ\}(e_1) \ldots (e_n),$$

where $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ and $\gamma(x_1) = e_1, \ldots, \gamma(x_n) = e_n$. By Lemma 48.3 we have $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma \triangleright \mathtt{nat})$, from which it follows directly that $\mathcal{D} : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$. Because $\Gamma \vdash e \cong e' : \tau$, we have $\mathcal{D}\{e\} \simeq \mathcal{D}\{e'\}$. But by construction $\mathcal{D}\{e\} \simeq \hat{\gamma}(\mathcal{C}\{e\})$, and $\mathcal{D}\{e'\} \simeq \hat{\gamma}(\mathcal{C}\{e'\})$, so $\hat{\gamma}(\mathcal{C}\{e\}) \simeq \hat{\gamma}(\mathcal{C}\{e'\})$. Because $\mathcal{C}$ is arbitrary, it follows that $\hat{\gamma}(e) \cong_\tau \hat{\gamma}(e')$.

Defining $\mathcal{D}'$ similarly to $\mathcal{D}$, but based on $\gamma'$, rather than $\gamma$, we may also show that $\mathcal{D}'\{e\} \simeq \mathcal{D}'\{e'\}$, and hence $\widehat{\gamma'}(e) \cong_\tau \widehat{\gamma'}(e')$. Now if $\gamma \cong_\Gamma \gamma'$, then by congruence we have $\mathcal{D}\{e\} \cong_{\mathtt{nat}} \mathcal{D}'\{e\}$, and $\mathcal{D}\{e'\} \cong_{\mathtt{nat}} \mathcal{D}'\{e'\}$. It follows that $\mathcal{D}\{e\} \cong_{\mathtt{nat}} \mathcal{D}'\{e'\}$, and so, by consistency of observational equivalence, we have $\mathcal{D}\{e\} \simeq \mathcal{D}'\{e'\}$, which is to say that $\hat{\gamma}(e) \cong_\tau \widehat{\gamma'}(e')$. □

Theorem 48.6 licenses the principle of *proof by coinduction*: to show that $\Gamma \vdash e \cong e' : \tau$, it is enough to exhibit a consistent congruence, $\mathcal{E}$, such that $\Gamma \vdash e \mathcal{E} e' : \tau$. It can be difficult to construct such a relation. In the next section we will provide a general method for doing so that exploits types.

Preview

## 48.2 Logical Equivalence

The key to simplifying reasoning about observational equivalence is to exploit types. Informally, we may classify the uses of expressions of a type into two broad categories, the *passive* and the *active* uses. The passive uses are those that merely manipulate expressions without actually inspecting them. For example, we may pass an expression of type $\tau$ to a function that merely returns it. The active uses are those that operate on the expression itself; these are the elimination forms associated with the type of that expression. For the purposes of distinguishing two expressions, it is only the active uses that matter; the passive uses merely manipulate expressions at arm's length, affording no opportunities to distinguish one from another.

This leads to the definition of logical equivalence alluded to in the introduction.

**Definition 48.8.** Logical equivalence *is a family of relations $e \sim_\tau e'$ between closed expressions of type $\tau$. It is defined by induction on $\tau$ as follows:*

$$e \sim_{\mathtt{nat}} e' \quad \textit{iff} \quad e \simeq e'$$

$$e \sim_{\tau_1 \to \tau_2} e' \quad \textit{iff} \quad \textit{if } e_1 \sim_{\tau_1} e_1', \textit{ then } e(e_1) \sim_{\tau_2} e'(e_1')$$

The definition of logical equivalence at type `nat` licenses the following principle of *proof by `nat`-induction*. To show that $\mathcal{E}\ (e, e')$ whenever $e \sim_{\mathtt{nat}} e'$, it is enough to show that

1. $\mathcal{E}\ (\overline{0}, \overline{0})$, and
2. if $\mathcal{E}\ (\overline{n}, \overline{n})$, then $\mathcal{E}\ (\overline{n+1}, \overline{n+1})$.

This is, of course, justified by mathematical induction on $n \geq 0$, where $e \longmapsto^* \overline{n}$ and $e' \longmapsto^* \overline{n}$ by the definition of Kleene equivalence.

**Lemma 48.9.** *Logical equivalence is symmetric and transitive: if $e \sim_\tau e'$, then $e' \sim_\tau e$, and if $e \sim_\tau e'$ and $e' \sim_\tau e''$, then $e \sim_\tau e''$.*

*Proof.* Simultaneously, by induction on the structure of $\tau$. If $\tau = \mathtt{nat}$, the result is immediate. If $\tau = \tau_1 \to \tau_2$, then we may assume that logical equivalence is symmetric and transitive at types $\tau_1$ and $\tau_2$. For symmetry, assume that $e \sim_\tau e'$; we wish to show $e' \sim_\tau e$. Assume that $e_1' \sim_{\tau_1} e_1$; it suffices to show that $e'(e_1') \sim_{\tau_2} e(e_1)$. By induction we have that $e_1 \sim_{\tau_1} e_1'$. Therefore by assumption $e(e_1) \sim_{\tau_2} e'(e_1')$, and hence by induction $e'(e_1') \sim_{\tau_2} e(e_1)$. For transitivity, assume that $e \sim_\tau e'$ and $e' \sim_\tau e''$; we are to show $e \sim_\tau e''$. Suppose that $e_1 \sim_{\tau_1} e_1''$; it is enough to show that $e(e_1) \sim_\tau e''(e_1'')$. By symmetry and transitivity we have $e_1 \sim_{\tau_1} e_1$, so by assumption $e(e_1) \sim_{\tau_2} e'(e_1)$. We also have by assumption $e'(e_1) \sim_{\tau_2} e''(e_1'')$. By transitivity we have $e'(e_1) \sim_{\tau_2} e''(e_1'')$, which suffices for the result. □

Logical equivalence is extended to open terms by substitution of related closed terms to obtain related results. If $\gamma$ and $\gamma'$ are two substitutions for $\Gamma$, we define $\gamma \sim_\Gamma \gamma'$ to hold iff $\gamma(x) \sim_{\Gamma(x)} \gamma'(x)$ for every variable, $x$, such that $\Gamma \vdash x : \tau$. *Open logical equivalence*, written $\Gamma \vdash e \sim e' : \tau$, is defined to mean that $\hat{\gamma}(e) \sim_\tau \hat{\gamma'}(e')$ whenever $\gamma \sim_\Gamma \gamma'$.



Preview

**Lemma 48.10.** *Open logical equivalence is symmetric and transitive.*

*Proof.* Follows immediately from Lemma 48.9 and the definition of open logical equivalence. □

At this point we are "two thirds of the way" to justifying the use of the name "open logical equivalence." The remaining third, reflexivity, is established in the next section.

## 48.3 Logical and Observational Equivalence Coincide

In this section we prove the coincidence of observational and logical equivalence.

**Lemma 48.11** (Converse Evaluation)**.** *Suppose that $e \sim_\tau e'$. If $d \longmapsto e$, then $d \sim_\tau e'$, and if $d' \longmapsto e'$, then $e \sim_\tau d'$.*

*Proof.* By induction on the structure of $\tau$. If $\tau = \mathtt{nat}$, then the result follows from the closure of Kleene equivalence under converse evaluation. If $\tau = \tau_1 \to \tau_2$, then suppose that $e \sim_\tau e'$, and $d \longmapsto e$. To show that $d \sim_\tau e'$, we assume $e_1 \sim_{\tau_1} e_1'$ and show $d(e_1) \sim_{\tau_2} e'(e_1')$. It follows from the assumption that $e(e_1) \sim_{\tau_2} e'(e_1')$. Noting that $d(e_1) \longmapsto e(e_1)$, the result follows by induction. □

**Lemma 48.12** (Consistency)**.** *If $e \sim_{\mathtt{nat}} e'$, then $e \cong e'$.*

*Proof.* Immediate, from Definition 48.8. □

**Theorem 48.13** (Reflexivity)**.** *If $\Gamma \vdash e : \tau$, then $\Gamma \vdash e \sim e : \tau$.*

*Proof.* We are to show that if $\Gamma \vdash e : \tau$ and $\gamma \sim_\Gamma \gamma'$, then $\hat{\gamma}(e) \sim_\tau \widehat{\gamma'}(e)$. The proof proceeds by induction on typing derivations; we consider two representative cases.

Consider the case of rule (8.4a), in which $\tau = \tau_1 \to \tau_2$ and $e = \lambda\,(x{:}\tau_1)\,e_2$. We are to show that

$$\lambda\,(x{:}\tau_1)\,\hat{\gamma}(e_2) \sim_{\tau_1 \to \tau_2} \lambda\,(x{:}\tau_1)\,\widehat{\gamma'}(e_2).$$

Assume that $e_1 \sim_{\tau_1} e_1'$; by Lemma 48.11, it is enough to show that $[e_1/x]\hat{\gamma}(e_2) \sim_{\tau_2} [e_1'/x]\widehat{\gamma'}(e_2)$. Let $\gamma_2 = \gamma \otimes x \hookrightarrow e_1$ and $\gamma_2' = \gamma' \otimes x \hookrightarrow e_1'$, and observe that $\gamma_2 \sim_{\Gamma, x:\tau_1} \gamma_2'$. Therefore, by induction we have $\hat{\gamma}_2(e_2) \sim_{\tau_2} \hat{\gamma}_2'(e_2)$, from which the result follows directly.

Now consider the case of rule (9.1d), for which we are to show that

$$\mathtt{rec}(\hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1); \hat{\gamma}(e)) \sim_\tau \mathtt{rec}(\widehat{\gamma'}(e_0); x.y.\widehat{\gamma'}(e_1); \widehat{\gamma'}(e)).$$

By the induction hypothesis applied to the first premise of rule (9.1d), we have

$$\hat{\gamma}(e) \sim_{\mathtt{nat}} \widehat{\gamma'}(e).$$

We proceed by $\mathtt{nat}$-induction. It suffices to show that

$$\mathtt{rec}(\hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1); \mathtt{z}) \sim_\tau \mathtt{rec}(\widehat{\gamma'}(e_0); x.y.\widehat{\gamma'}(e_1); \mathtt{z}), \tag{48.2}$$



Preview

and that

$$\mathtt{rec}(\hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1); \mathtt{s}(\overline{n})) \sim_\tau \mathtt{rec}(\widehat{\gamma'}(e_0); x.y.\widehat{\gamma'}(e_1); \mathtt{s}(\overline{n})), \tag{48.3}$$

assuming

$$\mathtt{rec}(\hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1); \overline{n}) \sim_\tau \mathtt{rec}(\widehat{\gamma'}(e_0); x.y.\widehat{\gamma'}(e_1); \overline{n}). \tag{48.4}$$

To show (48.2), by Lemma 48.11 it is enough to show that $\hat{\gamma}(e_0) \sim_\tau \widehat{\gamma'}(e_0)$. This is assured by the outer inductive hypothesis applied to the second premise of rule (9.1d).

To show (48.3), define

$$\delta = \gamma \otimes x \hookrightarrow \overline{n} \otimes y \hookrightarrow \mathtt{rec}(\hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1); \overline{n})$$

and

$$\delta' = \gamma' \otimes x \hookrightarrow \overline{n} \otimes y \hookrightarrow \mathtt{rec}(\widehat{\gamma'}(e_0); x.y.\widehat{\gamma'}(e_1); \overline{n}).$$

By (48.4) we have $\delta \sim_{\Gamma, x:\mathtt{nat}, y:\tau} \delta'$. Consequently, by the outer inductive hypothesis applied to the third premise of rule (9.1d), and Lemma 48.11, the required follows. □

**Corollary 48.14** (Equivalence)**.** *Open logical equivalence is an equivalence relation.*

**Corollary 48.15** (Termination)**.** *If $e : \mathtt{nat}$, then $e \longmapsto^* e'$ for some $e'$ val.*

**Lemma 48.16** (Congruence)**.** *If $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_0)$, and $\Gamma \vdash e \sim e' : \tau$, then $\Gamma_0 \vdash \mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_0$.*

*Proof.* By induction on the derivation of the typing of $\mathcal{C}_0$. We consider a representative case in which $\mathcal{C}_0 = \lambda\,(x{:}\tau_1)\,\mathcal{C}_2$ so that $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_1 \to \tau_2)$ and $\mathcal{C}_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0, x : \tau_1 \triangleright \tau_2)$. Assuming $\Gamma \vdash e \sim e' : \tau$, we are to show that

$$\Gamma_0 \vdash \mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_1 \to \tau_2,$$

which is to say

$$\Gamma_0 \vdash \lambda\,(x{:}\tau_1)\,\mathcal{C}_2\{e\} \sim \lambda\,(x{:}\tau_1)\,\mathcal{C}_2\{e'\} : \tau_1 \to \tau_2.$$

We know, by induction, that

$$\Gamma_0, x : \tau_1 \vdash \mathcal{C}_2\{e\} \sim \mathcal{C}_2\{e'\} : \tau_2.$$

Suppose that $\gamma_0 \sim_{\Gamma_0} \gamma_0'$, and that $e_1 \sim_{\tau_1} e_1'$. Let $\gamma_1 = \gamma_0 \otimes x \hookrightarrow e_1$, $\gamma_1' = \gamma_0' \otimes x \hookrightarrow e_1'$, and observe that $\gamma_1 \sim_{\Gamma_0, x:\tau_1} \gamma_1'$. By Definition 48.8 it is enough to show that

$$\hat{\gamma_1}(\mathcal{C}_2\{e\}) \sim_{\tau_2} \hat{\gamma_1'}(\mathcal{C}_2\{e'\}),$$

which follows immediately from the inductive hypothesis.

□

**Theorem 48.17.** *If $\Gamma \vdash e \sim e' : \tau$, then $\Gamma \vdash e \cong e' : \tau$.*

*Proof.* By Lemmas 48.12 and 48.16, and Theorem 48.6. □

**Corollary 48.18.** *If $e : \mathtt{nat}$, then $e \cong_{\mathtt{nat}} \overline{n}$, for some $n \geq 0$.*



Preview

*Proof.* By Theorem 48.13 we have $e \sim_{\mathtt{nat}} e$. Hence for some $n \geq 0$, we have $e \sim_{\mathtt{nat}} \overline{n}$, and so by Theorem 48.17, $e \cong_{\mathtt{nat}} \overline{n}$. □

**Lemma 48.19.** *For closed expressions $e : \tau$ and $e' : \tau$, if $e \cong_\tau e'$, then $e \sim_\tau e'$.*

*Proof.* We proceed by induction on the structure of $\tau$. If $\tau = \mathtt{nat}$, consider the empty context to obtain $e \simeq e'$, and hence $e \sim_{\mathtt{nat}} e'$. If $\tau = \tau_1 \to \tau_2$, then we are to show that whenever $e_1 \sim_{\tau_1} e_1'$, we have $e(e_1) \sim_{\tau_2} e'(e_1')$. By Theorem 48.17 we have $e_1 \cong_{\tau_1} e_1'$, and hence by congruence of observational equivalence it follows that $e(e_1) \cong_{\tau_2} e'(e_1')$, from which the result follows by induction. □

**Theorem 48.20.** *If $\Gamma \vdash e \cong e' : \tau$, then $\Gamma \vdash e \sim e' : \tau$.*

*Proof.* Assume that $\Gamma \vdash e \cong e' : \tau$, and that $\gamma \sim_\Gamma \gamma'$. By Theorem 48.17 we have $\gamma \cong_\Gamma \gamma'$, so by Lemma 48.7 $\hat{\gamma}(e) \cong_\tau \widehat{\gamma'}(e')$. Therefore, by Lemma 48.19, $\hat{\gamma}(e) \sim_\tau \hat{\gamma}(e')$. □

**Corollary 48.21.** *$\Gamma \vdash e \cong e' : \tau$ iff $\Gamma \vdash e \sim e' : \tau$.*

The principle of *symbolic evaluation* states that definitional equivalence is sufficient for observational equivalence:

**Theorem 48.22.** *If $\Gamma \vdash e \equiv e' : \tau$, then $\Gamma \vdash e \sim e' : \tau$, and hence $\Gamma \vdash e \cong e' : \tau$.*

*Proof.* By an argument similar to that used in the proof of Theorem 48.13 and Lemma 48.16, then appealing to Theorem 48.17. □

**Corollary 48.23.** *If $e \equiv e' : \mathtt{nat}$, then there exists $n \geq 0$ such that $e \longmapsto^* \overline{n}$ and $e' \longmapsto^* \overline{n}$.*

*Proof.* By Theorem 48.22 we have $e \sim_{\mathtt{nat}} e'$ and hence $e \simeq e'$. □

## 48.4 Some Laws of Equality

In this section we summarize some useful principles of observational equivalence for **T**. For the most part these may be proved as laws of logical equivalence, and then transferred to observational equivalence by appeal to Corollary 48.21. The laws are presented as inference rules with the meaning that if all of the premises are true judgments about observational equivalence, then so are the conclusions. In other words each rule is admissible as a principle of observational equivalence.



Preview

### 48.4.1 General Laws

Logical equivalence is indeed an equivalence relation: it is reflexive, symmetric, and transitive.

$$\frac{}{\Gamma \vdash e \cong e : \tau} \tag{48.5a}$$

$$\frac{\Gamma \vdash e' \cong e : \tau}{\Gamma \vdash e \cong e' : \tau} \tag{48.5b}$$

$$\frac{\Gamma \vdash e \cong e' : \tau \quad \Gamma \vdash e' \cong e'' : \tau}{\Gamma \vdash e \cong e'' : \tau} \tag{48.5c}$$

Reflexivity is an instance of a more general principle, that all definitional equivalences are observational equivalences.

$$\frac{\Gamma \vdash e \equiv e' : \tau}{\Gamma \vdash e \cong e' : \tau} \tag{48.6a}$$

This is called the *principle of symbolic evaluation*.

Observational equivalence is a congruence: we may replace equals by equals anywhere in an expression.

$$\frac{\Gamma \vdash e \cong e' : \tau \quad \mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')}{\Gamma' \vdash \mathcal{C}\{e\} \cong \mathcal{C}\{e'\} : \tau'} \tag{48.7a}$$

Equivalence is stable under substitution for free variables, and substituting equivalent expressions in an expression gives equivalent results.

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_2 \cong e_2' : \tau'}{\Gamma \vdash [e/x]e_2 \cong [e/x]e_2' : \tau'} \tag{48.8a}$$

$$\frac{\Gamma \vdash e_1 \cong e_1' : \tau \quad \Gamma, x : \tau \vdash e_2 \cong e_2' : \tau'}{\Gamma \vdash [e_1/x]e_2 \cong [e_1'/x]e_2' : \tau'} \tag{48.8b}$$

### 48.4.2 Equality Laws

Two functions are equal if they are equal on all arguments.

$$\frac{\Gamma, x : \tau_1 \vdash e(x) \cong e'(x) : \tau_2}{\Gamma \vdash e \cong e' : \tau_1 \to \tau_2} \tag{48.9}$$

Consequently, every expression of function type is equal to a $\lambda$-abstraction:

$$\frac{}{\Gamma \vdash e \cong \lambda\,(x{:}\tau_1)\,e(x) : \tau_1 \to \tau_2} \tag{48.10}$$

Preview

### 48.4.3 Induction Law

An equation involving a free variable, $x$, of type `nat` can be proved by induction on $x$.

$$\frac{\Gamma \vdash [\overline{n}/x]e \cong [\overline{n}/x]e' : \tau \text{ (for every } n \in \mathbb{N})}{\Gamma, x : \mathtt{nat} \vdash e \cong e' : \tau} \tag{48.11a}$$

To apply the induction rule, we proceed by mathematical induction on $n \in \mathbb{N}$, which reduces to showing:

1. $\Gamma \vdash [\mathtt{z}/x]e \cong [\mathtt{z}/x]e' : \tau$, and
2. $\Gamma \vdash [\mathtt{s}(\overline{n})/x]e \cong [\mathtt{s}(\overline{n})/x]e' : \tau$, if $\Gamma \vdash [\overline{n}/x]e \cong [\overline{n}/x]e' : \tau$.

## 48.5 Notes

The technique of *logical relations* interprets types as relations (here, equivalence relations) by associating with each type constructor a relational action that transforms the relation interpreting its arguments to the relation interpreting the constructed type. Logical relations (Statman, 1985) are a fundamental tool in proof theory and provide the foundation for the semantics of the NuPRL type theory (Constable, 1986; Allen, 1987; Harper, 1992). The use of logical relations to characterize observational equivalence is essentially an adaptation of the NuPRL semantics to the simpler setting of Gödel's System **T**.

Preview

# Chapter 49

# Equational Reasoning for PCF

In this Chapter we develop the theory of observational equivalence for **PCF**, with an eager interpretation of the type of natural numbers. The development proceeds along lines similar to those in Chapter 48, but is complicated by the presence of general recursion. The proof depends on the concept of an *admissible relation*, one that admits the principle of *proof by fixed point induction*.

## 49.1 Observational Equivalence

The definition of observational equivalence, along with the auxiliary notion of Kleene equivalence, are defined similarly to Chapter 48, but modified to account for the possibility of non-termination.

The collection of well-formed **PCF** contexts is inductively defined in a manner directly analogous to that in Chapter 48. Specifically, we define the judgment $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ by rules similar to rules (48.1), modified for **PCF**. (We leave the precise definition as an exercise for the reader.) When $\Gamma$ and $\Gamma'$ are empty, we write just $\mathcal{C} : \tau \rightsquigarrow \tau'$.

A *complete program* is a closed expression of type `nat`.

**Definition 49.1.** *We say that two complete programs, $e$ and $e'$, are* Kleene equal*, written $e \simeq e'$, iff for every $n \geq 0$, $e \longmapsto^* \overline{n}$ iff $e' \longmapsto^* \overline{n}$.*

Kleene equality is easily seen to be an equivalence relation and to be closed under converse evaluation. Moreover, $\overline{0} \not\simeq \overline{1}$, and, if $e$ and $e'$ are both divergent, then $e \simeq e'$.

Observational equivalence is defined just as it is in Chapter 48.

**Definition 49.2.** *We say that $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$ are* observationally*, or* contextually*,* equivalent *iff for every program context $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\varnothing \triangleright \mathtt{nat})$, $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$.*

**Theorem 49.3.** *Observational equivalence is the coarsest consistent congruence.*

*Proof.* See the proof of Theorem 48.6. □

**Lemma 49.4** (Substitution and Functionality)**.** *If $\Gamma \vdash e \cong e' : \tau$ and $\gamma : \Gamma$, then $\hat{\gamma}(e) \cong_\tau \hat{\gamma}(e')$. Moreover, if $\gamma \cong_\Gamma \gamma'$, then $\hat{\gamma}(e) \cong_\tau \hat{\gamma}'(e)$ and $\hat{\gamma}(e') \cong_\tau \hat{\gamma}'(e')$.*

Preview

*Proof.* See Lemma 48.7. □

## 49.2 Logical Equivalence

**Definition 49.5.** Logical equivalence, $e \sim_\tau e'$, *between closed expressions of type* $\tau$ *is defined by induction on* $\tau$ *as follows:*

$$e \sim_{\mathtt{nat}} e' \quad \textit{iff} \quad e \simeq e'$$

$$e \sim_{\tau_1 \to \tau_2} e' \quad \textit{iff} \quad e_1 \sim_{\tau_1} e_1' \textit{ implies } e(e_1) \sim_{\tau_2} e'(e_1')$$

Formally, logical equivalence is defined as in Chapter 48, except that the definition of Kleene equivalence is altered to account for non-termination. Logical equivalence is extended to open terms by substitution. Specifically, we define $\Gamma \vdash e \sim e' : \tau$ to mean that $\hat{\gamma}(e) \sim_\tau \hat{\gamma'}(e')$ whenever $\gamma \sim_\Gamma \gamma'$.

By the same argument as given in the proof of Lemma 48.9 logical equivalence is symmetric and transitive, as is its open extension.

**Lemma 49.6** (Strictness)**.** *If* $e : \tau$ *and* $e' : \tau$ *are both divergent, then* $e \sim_\tau e'$.

*Proof.* By induction on the structure of $\tau$. If $\tau = \mathtt{nat}$, then the result follows immediately from the definition of Kleene equivalence. If $\tau = \tau_1 \to \tau_2$, then $e(e_1)$ and $e'(e_1')$ diverge, so by induction $e(e_1) \sim_{\tau_2} e'(e_1')$, as required. □

**Lemma 49.7** (Converse Evaluation)**.** *Suppose that* $e \sim_\tau e'$. *If* $d \longmapsto e$, *then* $d \sim_\tau e'$, *and if* $d' \longmapsto e'$, *then* $e \sim_\tau d'$.

## 49.3 Logical and Observational Equivalence Coincide

The proof of coincidence of logical and observational equivalence relies on the concept of *bounded recursion*, which we define by induction on $m \geq 0$ as follows:

$$\mathtt{fix}^0\, x : \tau \,\mathtt{is}\, e \triangleq \mathtt{fix}\, x : \tau \,\mathtt{is}\, x$$
$$\mathtt{fix}^{m+1}\, x : \tau \,\mathtt{is}\, e \triangleq [\mathtt{fix}^m\, x : \tau \,\mathtt{is}\, e/x]e$$

When $m = 0$, bounded recursion is defined to be a divergent expression of type $\tau$. When $m > 0$, bounded recursion is defined by unrolling the recursion $m$ times by iterated substitution. Intuitively, the bounded recursive expression $\mathtt{fix}^m\, x : \tau \,\mathtt{is}\, e$ is as good as $\mathtt{fix}\, x : \tau \,\mathtt{is}\, e$ for up to $m$ unrollings, after which it is divergent.

It is easy to check that the follow rule is derivable for each $m \geq 0$:

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{fix}^m[\tau](x.e) : \tau}\ . \tag{49.1a}$$

The proof is by induction on $m \geq 0$, and amounts to an iteration of the substitution lemma for the statics of **PCF**.



Preview

The key property of bounded recursion is the principle of fixed point induction, which permits reasoning about a recursive computation by induction on the number of unrollings required to reach a value. The proof relies on *compactness*, which will be stated and proved in Section 49.4 below.

**Theorem 49.8** (Fixed Point Induction)**.** *Suppose that* $x : \tau \vdash e : \tau$*. If*

$$(\forall m \geq 0)\ \mathit{fix}^m\, x : \tau \,\mathtt{is}\, e \sim_\tau \mathit{fix}^m\, x : \tau \,\mathtt{is}\, e',$$

*then* $\mathtt{fix}\, x : \tau \,\mathtt{is}\, e \sim_\tau \mathtt{fix}\, x : \tau \,\mathtt{is}\, e'$*.*

*Proof.* Define an *applicative context*, $\mathcal{A}$, to be either a hole, $\circ$, or an application of the form $\mathcal{A}(e)$, where $\mathcal{A}$ is an applicative context. The typing judgment for applicative contexts, $\mathcal{A} : \tau_0 \rightsquigarrow \tau$, is a special case of the general typing judgment for contexts. Define logical equivalence of applicative contexts, $\mathcal{A} \sim \mathcal{A}' : \tau_0 \rightsquigarrow \tau$, by induction on the structure of $\mathcal{A}$ as follows:

1. $\circ \sim \circ : \tau_0 \rightsquigarrow \tau_0$;
2. if $\mathcal{A} \sim \mathcal{A}' : \tau_0 \rightsquigarrow \tau_2 \rightharpoonup \tau$ and $e_2 \sim_{\tau_2} e_2'$, then $\mathcal{A}(e_2) \sim \mathcal{A}'(e_2') : \tau_0 \rightsquigarrow \tau$.

We prove by induction on the structure of $\tau$, if $\mathcal{A} \sim \mathcal{A}' : \tau_0 \rightsquigarrow \tau$ and

$$\text{for every } m \geq 0,\ \mathcal{A}\{\mathtt{fix}^m\, x : \tau_0 \,\mathtt{is}\, e\} \sim_\tau \mathcal{A}'\{\mathtt{fix}^m\, x : \tau_0 \,\mathtt{is}\, e'\}, \tag{49.2}$$

then

$$\mathcal{A}\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e\} \sim_\tau \mathcal{A}'\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e'\}. \tag{49.3}$$

Choosing $\mathcal{A} = \mathcal{A}' = \circ$ with $\tau_0 = \tau$ completes the proof.

If $\tau = \mathtt{nat}$, then assume that $\mathcal{A} \sim \mathcal{A}' : \tau_0 \rightsquigarrow \mathtt{nat}$ and (49.2). By Definition 49.5, we are to show

$$\mathcal{A}\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e\} \simeq \mathcal{A}'\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e'\}.$$

By Corollary 49.17 there exists $m \geq 0$ such that

$$\mathcal{A}\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e\} \simeq \mathcal{A}\{\mathtt{fix}^m\, x : \tau_0 \,\mathtt{is}\, e\}.$$

By (49.2) we have

$$\mathcal{A}\{\mathtt{fix}^m\, x : \tau_0 \,\mathtt{is}\, e\} \simeq \mathcal{A}'\{\mathtt{fix}^m\, x : \tau_0 \,\mathtt{is}\, e'\}.$$

By Corollary 49.17

$$\mathcal{A}'\{\mathtt{fix}^m\, x : \tau_0 \,\mathtt{is}\, e'\} \simeq \mathcal{A}'\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e'\}.$$

The result follows by transitivity of Kleene equivalence.

If $\tau = \tau_1 \rightharpoonup \tau_2$, then by Definition 49.5, it is enough to show

$$\mathcal{A}\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e\}(e_1) \sim_{\tau_2} \mathcal{A}'\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e'\}(e_1')$$

whenever $e_1 \sim_{\tau_1} e_1'$. Let $\mathcal{A}_2 = \mathcal{A}(e_1)$ and $\mathcal{A}_2' = \mathcal{A}'(e_1')$. It follows from (49.2) that for every $m \geq 0$

$$\mathcal{A}_2\{\mathtt{fix}^m\, x : \tau_0 \,\mathtt{is}\, e\} \sim_{\tau_2} \mathcal{A}_2'\{\mathtt{fix}^m\, x : \tau_0 \,\mathtt{is}\, e'\}.$$

Noting that $\mathcal{A}_2 \sim \mathcal{A}_2' : \tau_0 \rightsquigarrow \tau_2$, we have by induction

$$\mathcal{A}_2\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e\} \sim_{\tau_2} \mathcal{A}_2'\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e'\},$$

as required. □

Preview

**Lemma 49.9** (Reflexivity). *If* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash e \sim e : \tau$.

*Proof.* The proof proceeds along the same lines as the proof of Theorem 48.13. The main difference is the treatment of general recursion, which is proved by fixed point induction. Consider rule (15.1g). Assuming $\gamma \sim_\Gamma \gamma'$, we are to show that

$$\mathtt{fix}\, x : \tau \,\mathtt{is}\, \hat{\gamma}(e) \sim_\tau \mathtt{fix}\, x : \tau \,\mathtt{is}\, \widehat{\gamma'}(e).$$

By Theorem 49.8 it is enough to show that, for every $m \geq 0$,

$$\mathtt{fix}^m\, x : \tau \,\mathtt{is}\, \hat{\gamma}(e) \sim_\tau \mathtt{fix}^m\, x : \tau \,\mathtt{is}\, \widehat{\gamma'}(e).$$

We proceed by an inner induction on $m$. When $m = 0$ the result is immediate, because both sides of the desired equivalence diverge. Assuming the result for $m$, and applying Lemma 49.7, it is enough to show that $\hat{\gamma}(e_1) \sim_\tau \widehat{\gamma'}(e_1)$, where

$$e_1 = [\mathtt{fix}^m\, x : \tau \,\mathtt{is}\, \hat{\gamma}(e)/x]\hat{\gamma}(e), \text{ and} \tag{49.4}$$

$$e_1' = [\mathtt{fix}^m\, x : \tau \,\mathtt{is}\, \widehat{\gamma'}(e)/x]\widehat{\gamma'}(e). \tag{49.5}$$

But this follows directly from the inner and outer inductive hypotheses. For by the outer inductive hypothesis, if

$$\mathtt{fix}^m\, x : \tau \,\mathtt{is}\, \hat{\gamma}(e) \sim_\tau \mathtt{fix}^m\, x : \tau \,\mathtt{is}\, \widehat{\gamma'}(e),$$

then

$$[\mathtt{fix}^m\, x : \tau \,\mathtt{is}\, \hat{\gamma}(e)/x]\hat{\gamma}(e) \sim_\tau [\mathtt{fix}^m\, x : \tau \,\mathtt{is}\, \widehat{\gamma'}(e)/x]\widehat{\gamma'}(e).$$

But the hypothesis holds by the inner inductive hypothesis, from which the result follows.

To handle the conditional $\mathtt{ifz}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}$, we proceed by cases on whether $e$ diverges, in which case the conditional is divergent and therefore self-related by Lemma 49.6, or $e$ converges, in which case we can proceed by an inner mathematical induction on its value, appealing to the inductive hypotheses governing the branches of the conditional to complete the argument. □

Symmetry and transitivity of eager logical equivalence are easily established by induction on types, noting that Kleene equivalence is symmetric and transitive. Eager logical equivalence is therefore an equivalence relation.

**Lemma 49.10** (Congruence). *If* $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_0)$, *and* $\Gamma \vdash e \sim e' : \tau$, *then* $\Gamma_0 \vdash \mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_0$.

*Proof.* By induction on the derivation of the typing of $\mathcal{C}_0$, following along similar lines to the proof of Lemma 49.9. □

Logical equivalence is consistent, by definition. Consequently, it is contained in observational equivalence.

**Theorem 49.11.** *If* $\Gamma \vdash e \sim e' : \tau$, *then* $\Gamma \vdash e \cong e' : \tau$.



Preview

*Proof.* By consistency and congruence of logical equivalence. □

**Lemma 49.12.** *If $e \cong_\tau e'$, then $e \sim_\tau e'$.*

*Proof.* By induction on the structure of $\tau$. If $\tau = \mathtt{nat}$, then the result is immediate, because the empty expression context is a program context. If $\tau = \tau_1 \rightharpoonup \tau_2$, then suppose that $e_1 \sim_{\tau_1} e_1'$. We are to show that $e(e_1) \sim_{\tau_2} e'(e_1')$. By Theorem 49.11 $e_1 \cong_{\tau_1} e_1'$, and hence by Lemma 49.4 $e(e_1) \cong_{\tau_2} e'(e_1')$, from which the result follows by induction. □

**Theorem 49.13.** *If $\Gamma \vdash e \cong e' : \tau$, then $\Gamma \vdash e \sim e' : \tau$.*

*Proof.* Assume that $\Gamma \vdash e \cong e' : \tau$. Suppose that $\gamma \sim_\Gamma \gamma'$. By Theorem 49.11 we have $\gamma \cong_\Gamma \gamma'$, and so by Lemma 49.4 we have

$$\hat{\gamma}(e) \cong_\tau \hat{\gamma'}(e').$$

Therefore by Lemma 49.12 we have

$$\hat{\gamma}(e) \sim_\tau \hat{\gamma'}(e').$$

□

**Corollary 49.14.** *$\Gamma \vdash e \cong e' : \tau$ iff $\Gamma \vdash e \sim e' : \tau$.*

## 49.4 Compactness

The principle of fixed point induction is derived from a critical property of **PCF**, called *compactness*. This property states that only finitely many unwindings of a fixed point expression are needed in a complete evaluation of a program. Although intuitively obvious (one cannot complete infinitely many recursive calls in a finite computation), it is rather tricky to state and prove rigorously.

The proof of compactness (Theorem 49.16) makes use of the stack machine for **PCF** defined in Chapter 28, augmented with the following transitions for bounded recursive expressions:

$$\frac{}{k \triangleright \mathtt{fix}^0\, x : \tau \,\mathtt{is}\, e \longmapsto k \triangleright \mathtt{fix}^0\, x : \tau \,\mathtt{is}\, e} \tag{49.6a}$$

$$\frac{}{k \triangleright \mathtt{fix}^{m+1}\, x : \tau \,\mathtt{is}\, e \longmapsto k \triangleright [\mathtt{fix}^m\, x : \tau \,\mathtt{is}\, e/x]e} \tag{49.6b}$$

It is not difficult to extend the proof of Corollary 28.4 to account for bounded recursion.

To get a feel for what is involved in the compactness proof, consider first the factorial function, $f$, in **PCF**:

$$\mathtt{fix}\, f : \mathtt{nat} \rightharpoonup \mathtt{nat} \,\mathtt{is}\, \lambda\, (x{:}\mathtt{nat})\, \mathtt{ifz}\, x\, \{\mathtt{z} \hookrightarrow \mathtt{s}(\mathtt{z}) \mid \mathtt{s}(x') \hookrightarrow x * f(x')\}.$$

Obviously evaluation of $f(\overline{n})$ requires $n$ recursive calls to the function itself. This means that, for a given input, $n$, we may place a *bound*, $m$, on the recursion that is sufficient to ensure termination of the computation. This can be expressed formally using the $m$-bounded form of general recursion,

$$\mathtt{fix}^m\, f : \mathtt{nat} \rightharpoonup \mathtt{nat} \,\mathtt{is}\, \lambda\, (x{:}\mathtt{nat})\, \mathtt{ifz}\, x\, \{\mathtt{z} \hookrightarrow \mathtt{s}(\mathtt{z}) \mid \mathtt{s}(x') \hookrightarrow x * f(x')\}.$$



Preview

Call this expression $f^{(m)}$. It follows from the definition of $f$ that if $f(\overline{n}) \longmapsto^* \overline{p}$, then $f^{(m)}(\overline{n}) \longmapsto^* \overline{p}$ for some $m \geq 0$ (in fact, $m = n$ suffices).

When considering expressions of higher type, we cannot expect to get the *same* result from the bounded recursion as from the unbounded. For example, consider the addition function, $a$, of type $\tau = \mathtt{nat} \rightharpoonup (\mathtt{nat} \rightharpoonup \mathtt{nat})$, given by the expression

$$\mathtt{fix}\, p : \tau\, \mathtt{is}\, \lambda\, (x{:}\mathtt{nat})\, \mathtt{ifz}\, x\, \{\mathtt{z} \hookrightarrow id \mid \mathtt{s}(x') \hookrightarrow s \circ (p(x'))\},$$

where $id = \lambda\, (y{:}\mathtt{nat})\, y$ is the identity, $e' \circ e = \lambda\, (x{:}\tau)\, e'(e(x))$ is composition, and $s = \lambda\, (x{:}\mathtt{nat})\, \mathtt{s}(x)$ is the successor function. The application $a(\overline{n})$ terminates after three transitions, regardless of the value of $n$, resulting in a $\lambda$-abstraction. When $n$ is positive, the result contains a *residual* copy of $a$ itself, which is applied to $n-1$ as a recursive call. The $m$-bounded version of $a$, written $a^{(m)}$, is also such that $a^{(m)}(\overline{n})$ terminates in three steps, provided that $m > 0$. But the result is not the same, because the residuals of $a$ appear as $a^{(m-1)}$, rather than as $a$ itself.

Turning now to the proof of compactness, it is helpful to introduce some notation. Suppose that $x : \tau \vdash e_x : \tau$ for some arbitrary abstractor $x.e_x$. Let $f^{(\omega)} = \mathtt{fix}\, x : \tau\, \mathtt{is}\, e_x$, and let $f^{(m)} = \mathtt{fix}^m\, x : \tau\, \mathtt{is}\, e_x$. Observe that $f^{(\omega)} : \tau$ and $f^{(m)} : \tau$ for any $m \geq 0$.

The following technical lemma governing the stack machine permits the bound on occurrences of a recursive expression to be raised without affecting the outcome of evaluation.

**Lemma 49.15.** *For every $m \geq 0$, if $[f^{(m)}/y]k \triangleright [f^{(m)}/y]e \longmapsto^* \epsilon \triangleleft \overline{n}$, then $[f^{(m+1)}/y]k \triangleright [f^{(m+1)}/y]e \longmapsto^* \epsilon \triangleleft \overline{n}$.*

*Proof.* By induction on $m \geq 0$, and then induction on transition. □

**Theorem 49.16** (Compactness)**.** *Suppose that $y : \tau \vdash e : \mathtt{nat}$ where $y \notin f^{(\omega)}$. If $[f^{(\omega)}/y]e \longmapsto^* \overline{n}$, then there exists $m \geq 0$ such that $[f^{(m)}/y]e \longmapsto^* \overline{n}$.*

*Proof.* We prove simultaneously the stronger statements that if

$$[f^{(\omega)}/y]k \triangleright [f^{(\omega)}/y]e \longmapsto^* \epsilon \triangleleft \overline{n},$$

then for some $m \geq 0$,

$$[f^{(m)}/y]k \triangleright [f^{(m)}/y]e \longmapsto^* \epsilon \triangleleft \overline{n},$$

and if

$$[f^{(\omega)}/y]k \triangleleft [f^{(\omega)}/y]e \longmapsto^* \epsilon \triangleleft \overline{n}$$

then for some $m \geq 0$,

$$[f^{(m)}/y]k \triangleleft [f^{(m)}/y]e \longmapsto^* \epsilon \triangleleft \overline{n}.$$

(Note that if $[f^{(\omega)}/y]e$ val, then $[f^{(m)}/y]e$ val for all $m \geq 0$.) The result then follows by the correctness of the stack machine (Corollary 28.4).

We proceed by induction on transition. Suppose that the initial state is

$$[f^{(\omega)}/y]k \triangleright f^{(\omega)},$$



Preview

which arises when $e = y$, and the transition sequence is as follows:

$$[f^{(\omega)}/y]k \triangleright f^{(\omega)} \longmapsto [f^{(\omega)}/y]k \triangleright [f^{(\omega)}/x]e_x \longmapsto^* \epsilon \triangleleft \overline{n}.$$

Noting that $[f^{(\omega)}/x]e_x = [f^{(\omega)}/y][y/x]e_x$, we have by induction that there exists $m \geq 0$ such that

$$[f^{(m)}/y]k \triangleright [f^{(m)}/x]e_x \longmapsto^* \epsilon \triangleleft \overline{n}.$$

By Lemma 49.15

$$[f^{(m+1)}/y]k \triangleright [f^{(m)}/x]e_x \longmapsto^* \epsilon \triangleleft \overline{n}$$

and we need only recall that

$$[f^{(m+1)}/y]k \triangleright f^{(m+1)} = [f^{(m+1)}/y]k \triangleright [f^{(m)}/x]e_x$$

to complete the proof. If, on the other hand, the initial step is an unrolling, but $e \neq y$, then we have for some $z \notin f^{(\omega)}$ and $z \neq y$

$$[f^{(\omega)}/y]k \triangleright \mathtt{fix}\, z : \tau \,\mathtt{is}\, d_\omega \longmapsto [f^{(\omega)}/y]k \triangleright [\mathtt{fix}\, z : \tau \,\mathtt{is}\, d_\omega / z]d_\omega \longmapsto^* \epsilon \triangleleft \overline{n}.$$

where $d_\omega = [f^{(\omega)}/y]d$. By induction there exists $m \geq 0$ such that

$$[f^{(m)}/y]k \triangleright [\mathtt{fix}\, z : \tau \,\mathtt{is}\, d_m / z]d_m \longmapsto^* \epsilon \triangleleft \overline{n},$$

where $d_m = [f^{(m)}/y]d$. But then by Lemma 49.15 we have

$$[f^{(m+1)}/y]k \triangleright [\mathtt{fix}\, z : \tau \,\mathtt{is}\, d_{m+1} / z]d_{m+1} \longmapsto^* \epsilon \triangleleft \overline{n},$$

where $d_{m+1} = [f^{(m+1)}/y]d$, from which the result follows directly. □

**Corollary 49.17.** *There exists $m \geq 0$ such that $[f^{(\omega)}/y]e \simeq [f^{(m)}/y]e$.*

*Proof.* If $[f^{(\omega)}/y]e$ diverges, then taking $m$ to be zero suffices. Otherwise, apply Theorem 49.16 to obtain $m$, and note that the required Kleene equivalence follows. □

## 49.5 Lazy Natural Numbers

Recall from Chapter 15 that if the successor is evaluated lazily, then the type `nat` changes its meaning to that of the lazy natural numbers, which we shall write `lnat` for emphasis. This type contains an "infinite number", $\omega$, which may be thought of as consisting of an endless stack of successors.

To account for the lazy successor the definition of logical equivalence must be reformulated. Rather than being defined *inductively* as the strongest relation closed under specified conditions, it is now defined *coinductively* as the weakest relation consistent with two analogous conditions. We may then show that two expressions are related using the principle of *proof by coinduction*.



Preview

The definition of Kleene equivalence must be altered to account for the lazily evaluated successor operation. To account for $\omega$, two computations are compared based solely on the outermost form of their values, if any. We define $e \simeq e'$ to hold iff (a) if $e \longmapsto^* \mathtt{z}$, then $e' \longmapsto^* \mathtt{z}$, and *vice versa*; and (b) if $e \longmapsto^* \mathtt{s}(e_1)$, then $e' \longmapsto^* \mathtt{s}(e_1')$, and *vice versa*.

Corollary 49.17 can be proved for the co-natural numbers by essentially the same argument as before.

The definition of logical equivalence at type `lnat` is defined to be the *weakest* equivalence relation, $\mathcal{E}$, between closed terms of type `lnat` satisfying the following *consistency conditions*: if $e \mathrel{\mathcal{E}} e' : \mathtt{lnat}$, then

1. If $e \longmapsto^* \mathtt{z}$, then $e' \longmapsto^* \mathtt{z}$, and *vice versa*.
2. If $e \longmapsto^* \mathtt{s}(e_1)$, then $e' \longmapsto^* \mathtt{s}(e_1')$ with $e_1 \mathrel{\mathcal{E}} e_1' : \mathtt{lnat}$, and *vice versa*.

It is immediate that if $e \sim_{\mathtt{lnat}} e'$, then $e \simeq e'$, and so logical equivalence is consistent. It is also strict in that if $e$ and $e'$ are both divergent expressions of type `lnat`, then $e \sim_{\mathtt{lnat}} e'$.

The principle of *proof by coinduction* states that to show $e \sim_{\mathtt{lnat}} e'$, it suffices to exhibit a relation, $\mathcal{E}$, such that

1. $e \mathrel{\mathcal{E}} e' : \mathtt{lnat}$, and
2. $\mathcal{E}$ satisfies the above consistency conditions.

If these requirements hold, then $\mathcal{E}$ is contained in logical equivalence at type `lnat`, and hence $e \sim_{\mathtt{lnat}} e'$, as required.

As an application of coinduction, let us consider the proof of Theorem 49.8. The overall argument remains as before, but the proof for the type `lnat` must be altered as follows. Suppose that $\mathcal{A} \sim \mathcal{A}' : \tau_0 \rightsquigarrow \mathtt{lnat}$, and let $a = \mathcal{A}\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e\}$ and $a' = \mathcal{A}'\{\mathtt{fix}\, x : \tau_0 \,\mathtt{is}\, e'\}$. Writing $a^{(m)} = \mathcal{A}\{\mathtt{fix}^m\, x : \tau_0 \,\mathtt{is}\, e\}$ and $a'^{(m)} = \mathcal{A}'\{\mathtt{fix}^m\, x : \tau_0 \,\mathtt{is}\, e'\}$, assume that

$$\text{for every } m \geq 0,\ a^{(m)} \sim_{\mathtt{lnat}} a'^{(m)}.$$

We are to show that

$$a \sim_{\mathtt{lnat}} a'.$$

Define the functions $p_n$ for $n \geq 0$ on closed terms of type `lnat` by the following equations:

$$p_0(d) = d$$

$$p_{(n+1)}(d) = \begin{cases} d' & \text{if } p_n(d) \longmapsto^* \mathtt{s}(d') \\ \textit{undefined} & \text{otherwise} \end{cases}$$

For $n \geq 0$, let $a_n = p_n(a)$ and $a_n' = p_n(a')$. Correspondingly, let $a_n^{(m)} = p_n(a^{(m)})$ and $a_n'^{(m)} = p_n(a'^{(m)})$. Define $\mathcal{E}$ to be the strongest relation such that $a_n \mathrel{\mathcal{E}} a_n' : \mathtt{lnat}$ for all $n \geq 0$. We will show that the relation $\mathcal{E}$ satisfies the consistency conditions, and so it is contained in logical equivalence. Because $a \mathrel{\mathcal{E}} a' : \mathtt{lnat}$ (by construction), the result follows immediately.



Preview

To show that $\mathcal{E}$ is consistent, suppose that $a_n \mathrel{\mathcal{E}} a'_n : \mathtt{lnat}$ for some $n \geq 0$. We have by Corollary 49.17 $a_n \simeq a_n^{(m)}$, for some $m \geq 0$, and hence, by the assumption, $a_n \simeq {a'_n}^{(m)}$, and so by Corollary 49.17 again, ${a'_n}^{(m)} \simeq a'_n$. Now if $a_n \longmapsto^* \mathtt{s}(b_n)$, then $a_n^{(m)} \longmapsto^* \mathtt{s}(b_n^{(m)})$ for some $b_n^{(m)}$, and hence there exists ${b'_n}^{(m)}$ such that ${a'_n}^{(m)} \longmapsto^* {b'_n}^{(m)}$, and so there exists $b'_n$ such that $a'_n \longmapsto^* \mathtt{s}(b'_n)$. But $b_n = p_{n+1}(a)$ and $b'_n = p_{n+1}(a')$, and we have $b_n \mathrel{\mathcal{E}} b'_n : \mathtt{lnat}$ by construction, as required.

## 49.6 Notes

The use of logical relations to characterize observational equivalence for **PCF** is inspired by the treatment of partiality in type theory by Constable and Smith (1987) and by the studies of observational equivalence by Pitts (2000). Although the technical details differ, the proof of compactness here is inspired by Pitts's structurally inductive characterization of termination using an abstract machine. It is critical to restrict attention to transition systems whose states are complete programs (closed expressions of observable type). Structural operational semantics usually does not fulfill this requirement, thereby requiring a considerably more complex argument than given here.

Preview

Preview

# Chapter 50

# Parametricity

The motivation for introducing polymorphism was to enable more programs to be written — those that are "generic" in one or more types, such as the composition function given in Chapter 20. Then if a program *does not* depend on the choice of types, we can code it using polymorphism. Moreover, if we wish to insist that a program *can not* depend on a choice of types, we demand that it be polymorphic. Thus polymorphism can be used both to expand the collection of programs we may write, and also to limit the collection of programs that are permissible in a given context.

The restrictions imposed by polymorphic typing give rise to the experience that in a polymorphic functional language, if the types are correct, then the program is correct. Roughly speaking, if a function has a polymorphic type, then the strictures of type genericity vastly cut down the set of programs with that type. Thus if you have written a program with this type, it is quite likely to be the one you intended!

The technical foundation for these remarks is called *parametricity*. The goal of this chapter is to give an account of parametricity for **F** under a call-by-name interpretation.

## 50.1 Overview

We will begin with an informal discussion of parametricity based on a "seat of the pants" understanding of the set of well-formed programs of a type.

Suppose that a function value $f$ has the type $\forall(t.t \to t)$. What function could it be? When instantiated at a type $\tau$ it should evaluate to a function $g$ of type $\tau \to \tau$ that, when further applied to a value $v$ of type $\tau$ returns a value $v'$ of type $\tau$. Because $f$ is polymorphic, $g$ cannot depend on $v$, so $v'$ must be $v$. In other words, $g$ must be the identity function at type $\tau$, and $f$ must therefore be the *polymorphic identity*.

Suppose that $f$ is a function of type $\forall(t.t)$. What function could it be? A moment's thought reveals that it cannot exist at all. For it must, when instantiated at a type $\tau$, return a value of that type. But not every type has a value (including this one), so this is an impossible assignment. The only conclusion is that $\forall(t.t)$ is an *empty* type.

Preview

Let $N$ be the type of polymorphic Church numerals introduced in Chapter 20, namely $\forall(t.t \to (t \to t) \to t)$. What are the values of this type? Given any type $\tau$, and values $z : \tau$ and $s : \tau \to \tau$, the expression

$$f[\tau](z)(s)$$

must yield a value of type $\tau$. Moreover, it must behave uniformly with respect to the choice of $\tau$. What values could it yield? The only way to build a value of type $\tau$ is by using the element $z$ and the function $s$ passed to it. A moment's thought reveals that the application must amount to the $n$-fold composition

$$s(s(\ldots s(z)\ldots)).$$

That is, the elements of $N$ are in one-to-one correspondence with the natural numbers.

## 50.2 Observational Equivalence

The definition of observational equivalence given in Chapters 48 and 49 is based on identifying a type of *answers* that are observable outcomes of complete programs. Values of function type are not regarded as answers, but are treated as "black boxes" with no internal structure, only input-output behavior. In **F**, however, there are no (closed) base types. Every type is either a function type or a polymorphic type, and hence no types suitable to serve as observable answers.

One way to manage this difficulty is to augment **F** with a base type of answers to serve as the observable outcomes of a computation. The only requirement is that this type have two elements that can be immediately distinguished from each other by evaluation. We may achieve this by enriching **F** with a base type, **2**, containing two constants, **tt** and **ff**, that serve as possible answers for a complete computation. A complete program is a closed expression of type **2**.

Kleene equality is defined for complete programs by requiring that $e \simeq e'$ iff either (a) $e \longmapsto^* \mathbf{tt}$ and $e' \longmapsto^* \mathbf{tt}$; or (b) $e \longmapsto^* \mathbf{ff}$ and $e' \longmapsto^* \mathbf{ff}$. This is obviously an equivalence relation, and it is immediate that $\mathbf{tt} \not\simeq \mathbf{ff}$, because these are two distinct constants. As before, we say that a type-indexed family of equivalence relations between closed expressions of the same type is *consistent* if it implies Kleene equality at the answer type, **2**.

To define observational equivalence, we must first define the concept of an expression context for **F** as an expression with a "hole" in it. More precisely, we may give an inductive definition of the judgment

$$\mathcal{C} : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow (\Delta'; \Gamma' \triangleright \tau'),$$

which states that $\mathcal{C}$ is an expression context that, when filled with an expression $\Delta; \Gamma \vdash e : \tau$ yields an expression $\Delta'; \Gamma' \vdash \mathcal{C}\{e\} : \tau'$. (We leave the precise definition of this judgment, and the verification of its properties, as an exercise for the reader.)

**Definition 50.1.** *Two expressions of the same type are* observationally equivalent, $\Delta; \Gamma \vdash e \cong e' : \tau$, *iff* $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$ *whenever* $\mathcal{C} : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow (\emptyset; \emptyset \triangleright \mathbf{2})$.

**Lemma 50.2.** *Observational equivalence is the coarsest consistent congruence.*

*Proof.* Essentially the same as the the proof of Theorem 48.6. □



Preview

**Lemma 50.3.**

1. *If* $\Delta, t; \Gamma \vdash e \cong e' : \tau$ *and* $\tau_0$ *type, then* $\Delta; [\tau_0/t]\Gamma \vdash [\tau_0/t]e \cong [\tau_0/t]e' : [\tau_0/t]\tau$.
2. *If* $\emptyset; \Gamma, x : \tau_0 \vdash e \cong e' : \tau$ *and* $d : \tau_0$, *then* $\emptyset; \Gamma \vdash [d/x]e \cong [d/x]e' : \tau$. *Moreover, if* $d \cong_{\tau_0} d'$, *then* $\emptyset; \Gamma \vdash [d/x]e \cong [d'/x]e : \tau$ *and* $\emptyset; \Gamma \vdash [d/x]e' \cong [d'/x]e' : \tau$.

*Proof.* 1. Let $\mathcal{C} : (\Delta; [\tau_0/t]\Gamma \triangleright [\tau_0/t]\tau) \rightsquigarrow (\emptyset \triangleright \mathbf{2})$ be a program context. We are to show that

$$\mathcal{C}\{[\tau_0/t]e\} \simeq \mathcal{C}\{[\tau_0/t]e'\}.$$

Because $\mathcal{C}$ is closed, this is equivalent to

$$[\tau_0/t]\mathcal{C}\{e\} \simeq [\tau_0/t]\mathcal{C}\{e'\}.$$

Let $\mathcal{C}'$ be the context $\Lambda(t.\mathcal{C}\{\circ\})[\tau_0]$, and observe that

$$\mathcal{C}' : (\Delta, t; \Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathbf{2}).$$

Therefore, from the assumption,

$$\mathcal{C}'\{e\} \simeq \mathcal{C}'\{e'\}.$$

But $\mathcal{C}'\{e\} \simeq [\tau_0/t]\mathcal{C}\{e\}$, and $\mathcal{C}'\{e'\} \simeq [\tau_0/t]\mathcal{C}\{e'\}$, from which the result follows.

2. By an argument essentially similar to that for Lemma 48.7.

□

## 50.3 Logical Equivalence

In this section we introduce a form of logical equivalence that captures the informal concept of parametricity, and also provides a characterization of observational equivalence. This will permit us to derive properties of observational equivalence of polymorphic programs of the kind suggested earlier.

The definition of logical equivalence for **F** is somewhat more complex than for **T**. The main idea is to define logical equivalence for a polymorphic type, $\forall(t.\tau)$ to satisfy a very strong condition that captures the essence of parametricity. As a first approximation, we might say that two expressions, $e$ and $e'$, of this type should be logically equivalent if they are logically equivalent for "all possible" interpretations of the type $t$. More precisely, we might require that $e[\rho]$ be related to $e'[\rho]$ at type $[\rho/t]\tau$, for any choice of type $\rho$. But this runs into two problems, one technical, the other conceptual. The same device will be used to solve both problems.

The technical problem stems from impredicativity. In Chapter 48 logical equivalence is defined by induction on the structure of types. But when polymorphism is impredicative, the type $[\rho/t]\tau$ might well be larger than $\forall(t.\tau)$. At the very least we would have to justify the definition of logical equivalence on some other grounds, but no criterion appears to be available. The conceptual problem is that, even if we could make sense of the definition of logical equivalence, it would be too restrictive. For such a definition amounts to saying that the unknown type $t$ is to be interpreted



Preview

as logical equivalence at whatever type it turns out to be when instantiated. To obtain useful parametricity results, we shall ask for much more than this. What we shall do is to consider *separately* instances of $e$ and $e'$ by types $\rho$ and $\rho'$, and treat the type variable $t$ as standing for *any relation* (of some form) between $\rho$ and $\rho'$. We may suspect that this is asking too much: perhaps logical equivalence is the *empty* relation. Surprisingly, this is not the case, and indeed it is this very feature of the definition that we shall exploit to derive parametricity results about the language.

To manage both of these problems we will consider a generalization of logical equivalence that is parameterized by a relational interpretation of the free type variables of its classifier. The parameters determine a separate binding for each free type variable in the classifier for each side of the equation, with the discrepancy being mediated by a specified relation between them. Thus related expressions need not have the same type, with the differences between them mediated by the given relation.

We will restrict attention to a certain collection of "admissible" binary relations between closed expressions. The conditions are imposed to ensure that logical equivalence and observational equivalence coincide.

**Definition 50.4** (Admissibility)**.** *A relation $R$ between expressions of types $\rho$ and $\rho'$ is* admissible*, written $R : \rho \leftrightarrow \rho'$, iff it satisfies two requirements:*

1. *Respect for observational equivalence: if $R(e, e')$ and $d \cong_\rho e$ and $d' \cong_{\rho'} e'$, then $R(d, d')$.*

2. *Closure under converse evaluation: if $R(e, e')$, then if $d \longmapsto e$, then $R(d, e')$ and if $d' \longmapsto e'$, then $R(e, d')$.*

Closure under converse evaluation will turn out to be a consequence of respect for observational equivalence, but we are not yet in a position to establish this fact.

The judgment $\delta : \Delta$ states that $\delta$ is a *type substitution* that assigns a closed type to each type variable $t \in \Delta$. A type substitution, $\delta$, induces a substitution function, $\hat{\delta}$, on types given by the equation

$$\hat{\delta}(\tau) = [\delta(t_1), \ldots, \delta(t_n)/t_1, \ldots, t_n]\tau,$$

and similarly for expressions. Substitution is extended to contexts pointwise by defining $\hat{\delta}(\Gamma)(x) = \hat{\delta}(\Gamma(x))$ for each $x \in dom(\Gamma)$.

Let $\delta$ and $\delta'$ be two type substitutions of closed types to the type variables in $\Delta$. An *admissible relation assignment*, $\eta$, between $\delta$ and $\delta'$ is an assignment of an admissible relation $\eta(t) : \delta(t) \leftrightarrow \delta'(t)$ to each $t \in \Delta$. The judgment $\eta : \delta \leftrightarrow \delta'$ states that $\eta$ is an admissible relation assignment between $\delta$ and $\delta'$.

Logical equivalence is defined in terms of its generalization, called *parametric logical equivalence*, written $e \sim_\tau e'\ [\eta : \delta \leftrightarrow \delta']$, defined as follows.

**Definition 50.5** (Parametric Logical Equivalence)**.** *The relation $e \sim_\tau e'\ [\eta : \delta \leftrightarrow \delta']$ is defined by*



Preview

*induction on the structure of $\tau$ by the following conditions:*

$$\begin{array}{lll} e \sim_t e' \ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & \eta(t)(e, e') \\ e \sim_{\mathbf{2}} e' \ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & e \simeq e' \\ e \sim_{\tau_1 \to \tau_2} e' \ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & e_1 \sim_{\tau_1} e_1' \ [\eta : \delta \leftrightarrow \delta'] \textit{ implies} \\ & & e(e_1) \sim_{\tau_2} e'(e_1') \ [\eta : \delta \leftrightarrow \delta'] \\ e \sim_{\forall(t.\tau)} e' \ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & \textit{for every } \rho,\ \rho', \textit{ and every admissible } R : \rho \leftrightarrow \rho', \\ & & e[\rho] \sim_\tau e'[\rho'] \ [\eta \otimes t \hookrightarrow R : \delta \otimes t \hookrightarrow \rho \leftrightarrow \delta' \otimes t \hookrightarrow \rho'] \end{array}$$

Logical equivalence is defined in terms of parametric logical equivalence by considering all possible interpretations of its free type- and expression variables. An *expression substitution*, $\gamma$, for a context $\Gamma$, written $\gamma : \Gamma$, is a substitution of a closed expression $\gamma(x) : \Gamma(x)$ to each variable $x \in dom(\Gamma)$. An expression substitution, $\gamma : \Gamma$, induces a substitution function, $\hat{\gamma}$, defined by the equation

$$\hat{\gamma}(e) = [\gamma(x_1), \ldots, \gamma(x_n)/x_1, \ldots, x_n]e,$$

where the domain of $\Gamma$ consists of the variables $x_1, \ldots, x_n$. The relation $\gamma \sim_\Gamma \gamma' \ [\eta : \delta \leftrightarrow \delta']$ is defined to hold iff $dom(\gamma) = dom(\gamma') = dom(\Gamma)$, and $\gamma(x) \sim_{\Gamma(x)} \gamma'(x) \ [\eta : \delta \leftrightarrow \delta']$ for every variable, $x$, in their common domain.

**Definition 50.6** (Logical Equivalence)**.** *The expressions $\Delta; \Gamma \vdash e : \tau$ and $\Delta; \Gamma \vdash e' : \tau$ are* logically equivalent*, written $\Delta; \Gamma \vdash e \sim e' : \tau$ iff, for every assignment $\delta$ and $\delta'$ of closed types to type variables in $\Delta$, and every admissible relation assignment $\eta : \delta \leftrightarrow \delta'$, if $\gamma \sim_\Gamma \gamma' \ [\eta : \delta \leftrightarrow \delta']$, then $\hat{\gamma}(\hat{\delta}(e)) \sim_\tau \widehat{\gamma'}(\widehat{\delta'}(e')) \ [\eta : \delta \leftrightarrow \delta']$.*

When $e$, $e'$, and $\tau$ are closed, this definition states that $e \sim_\tau e'$ iff $e \sim_\tau e' \ [\emptyset : \emptyset \leftrightarrow \emptyset]$, so that logical equivalence is indeed a special case of its generalization.

**Lemma 50.7** (Closure under Converse Evaluation)**.** *Suppose that $e \sim_\tau e' \ [\eta : \delta \leftrightarrow \delta']$. If $d \longmapsto e$, then $d \sim_\tau e'$, and if $d' \longmapsto e'$, then $e \sim_\tau d'$.*

*Proof.* By induction on the structure of $\tau$. When $\tau = t$, the result holds by the definition of admissibility. Otherwise the result follows by induction, making use of the definition of the transition relation for applications and type applications. □

**Lemma 50.8** (Respect for Observational Equivalence)**.** *Suppose that $e \sim_\tau e' \ [\eta : \delta \leftrightarrow \delta']$. If $d \cong_{\hat{\delta}(\tau)} e$ and $d' \cong_{\widehat{\delta'}(\tau)} e'$, then $d \sim_\tau d' \ [\eta : \delta \leftrightarrow \delta']$.*

*Proof.* By induction on the structure of $\tau$, relying on the definition of admissibility, and the congruence property of observational equivalence. For example, if $\tau = \forall(t.\tau_2)$, then we are to show that for every admissible $R : \rho \leftrightarrow \rho'$,

$$d[\rho] \sim_{\tau_2} d'[\rho'] \ [\eta \otimes t \hookrightarrow R : \delta \otimes t \hookrightarrow \rho \leftrightarrow \delta' \otimes t \hookrightarrow \rho'].$$

Because observational equivalence is a congruence, we have $d[\rho] \cong_{[\rho/t]\hat{\delta}(\tau_2)} e[\rho]$, and $d'[\rho'] \cong_{[\rho'/t]\widehat{\delta'}(\tau_2)} e'[\rho]$. It follows that

$$e[\rho] \sim_{\tau_2} e'[\rho'] \ [\eta \otimes t \hookrightarrow R : \delta \otimes t \hookrightarrow \rho \leftrightarrow \delta' \otimes t \hookrightarrow \rho'],$$

from which the result follows by induction. □



Preview

**Corollary 50.9.** *The relation $e \sim_\tau e' \ [\eta : \delta \leftrightarrow \delta']$ is an admissible relation between closed types $\hat{\delta}(\tau)$ and $\widehat{\delta'}(\tau)$.*

*Proof.* By Lemmas 50.7 and 50.8. □

**Corollary 50.10.** *If $\Delta; \Gamma \vdash e \sim e' : \tau$, and $\Delta; \Gamma \vdash d \cong e : \tau$ and $\Delta; \Gamma \vdash d' \cong e' : \tau$, then $\Delta; \Gamma \vdash d \sim d' : \tau$.*

*Proof.* By Lemma 50.3 and Corollary 50.9. □

**Lemma 50.11** (Compositionality)**.** *Let $R : \hat{\delta}(\rho) \leftrightarrow \widehat{\delta'}(\rho)$ be the relational interpretation of some type $\rho$, which is to say $R(d, d')$ holds iff $d \sim_\rho d' \ [\eta : \delta \leftrightarrow \delta']$. Then $e \sim_{[\rho/t]\tau} e' \ [\eta : \delta \leftrightarrow \delta']$ if, and only if,*

$$e \sim_\tau e' \ [\eta \otimes t \hookrightarrow R : \delta \otimes t \hookrightarrow \hat{\delta}(\rho) \leftrightarrow \delta' \otimes t \hookrightarrow \widehat{\delta'}(\rho)].$$

*Proof.* By induction on the structure of $\tau$. When $\tau = t$, the result is immediate from the choice of the relation $R$. When $\tau = t' \neq t$, the result follows directly from Definition 50.5. When $\tau = \tau_1 \to \tau_2$, the result follows by induction, using Definition 50.5. Similarly, when or $\tau = \forall(u.\tau_1)$, the result follows by induction, noting that we may assume, without loss of generality, that $u \neq t$ and $u \notin \rho$. □

Despite the strong conditions on polymorphic types, logical equivalence is not overly restrictive—every expression satisfies its constraints. This result is often called the *parametricity theorem* or the *abstraction theorem*:

**Theorem 50.12** (Parametricity)**.** *If $\Delta; \Gamma \vdash e : \tau$, then $\Delta; \Gamma \vdash e \sim e : \tau$.*

*Proof.* By rule induction on the statics of **F** given by rules (20.2).

We consider two representative cases here.

**Rule** (20.2d) Suppose $\delta : \Delta, \delta' : \Delta, \eta : \delta \leftrightarrow \delta'$, and $\gamma \sim_\Gamma \gamma' \ [\eta : \delta \leftrightarrow \delta']$. By induction we have that for all $\rho, \rho'$, and admissible $R : \rho \leftrightarrow \rho'$,

$$[\rho/t]\hat{\gamma}(\hat{\delta}(e)) \sim_\tau [\rho'/t]\widehat{\gamma'}(\widehat{\delta'}(e)) \ [\eta_* : \delta_* \leftrightarrow \delta'_*],$$

where $\eta_* = \eta \otimes t \hookrightarrow R$, $\delta_* = \delta \otimes t \hookrightarrow \rho$, and $\delta'_* = \delta' \otimes t \hookrightarrow \rho'$. Because

$$\Lambda(t.\hat{\gamma}(\hat{\delta}(e)))[\rho] \longmapsto^* [\rho/t]\hat{\gamma}(\hat{\delta}(e))$$

and

$$\Lambda(t.\widehat{\gamma'}(\widehat{\delta'}(e)))[\rho'] \longmapsto^* [\rho'/t]\widehat{\gamma'}(\widehat{\delta'}(e)),$$

the result follows by Lemma 50.7.

**Rule** (20.2e) Suppose $\delta : \Delta, \delta' : \Delta, \eta : \delta \leftrightarrow \delta'$, and $\gamma \sim_\Gamma \gamma' \ [\eta : \delta \leftrightarrow \delta']$. By induction we have

$$\hat{\gamma}(\hat{\delta}(e)) \sim_{\forall(t.\tau)} \widehat{\gamma'}(\widehat{\delta'}(e)) \ [\eta : \delta \leftrightarrow \delta']$$



Preview

Let $\hat{\rho} = \hat{\delta}(\rho)$ and $\hat{\rho}' = \widehat{\delta'}(\rho)$. Define the relation $R : \hat{\rho} \leftrightarrow \hat{\rho}'$ by $R(d, d')$ iff $d \sim_\rho d'$ $[\eta : \delta \leftrightarrow \delta']$. By Corollary 50.9, this relation is admissible.

By the definition of logical equivalence at polymorphic types, we obtain

$$\hat{\gamma}(\hat{\delta}(e))[\hat{\rho}] \sim_\tau \widehat{\gamma'}(\widehat{\delta'}(e))[\hat{\rho}'] \; [\eta \otimes t \hookrightarrow R : \delta \otimes t \hookrightarrow \hat{\rho} \leftrightarrow \delta' \otimes t \hookrightarrow \hat{\rho}'].$$

By Lemma 50.11

$$\hat{\gamma}(\hat{\delta}(e))[\hat{\rho}] \sim_{[\rho/t]\tau} \widehat{\gamma'}(\widehat{\delta'}(e))[\hat{\rho}'] \; [\eta : \delta \leftrightarrow \delta']$$

But

$$\begin{aligned} \hat{\gamma}(\hat{\delta}(e))[\hat{\rho}] &= \hat{\gamma}(\hat{\delta}(e))[\hat{\delta}(\rho)] && (50.1)\\ &= \hat{\gamma}(\hat{\delta}(e[\rho])), && (50.2) \end{aligned}$$

and similarly

$$\begin{aligned} \widehat{\gamma'}(\widehat{\delta'}(e))[\hat{\rho}'] &= \widehat{\gamma'}(\widehat{\delta'}(e))[\widehat{\delta'}(\rho)] && (50.3)\\ &= \widehat{\gamma'}(\widehat{\delta'}(e[\rho])), && (50.4) \end{aligned}$$

from which the result follows.

□

**Corollary 50.13.** *If* $\Delta; \Gamma \vdash e \cong e' : \tau$, *then* $\Delta; \Gamma \vdash e \sim e' : \tau$.

*Proof.* By Theorem 50.12 $\Delta; \Gamma \vdash e \sim e : \tau$, and hence by Corollary 50.10, $\Delta; \Gamma \vdash e \sim e' : \tau$. □

**Lemma 50.14** (Congruence)**.** *If* $\Delta; \Gamma \vdash e \sim e' : \tau$ *and* $\mathcal{C} : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow (\Delta'; \Gamma' \triangleright \tau')$, *then* $\Delta'; \Gamma' \vdash \mathcal{C}\{e\} \sim \mathcal{C}\{e'\} : \tau'$.

*Proof.* By induction on the structure of $\mathcal{C}$, following along very similar lines to the proof of Theorem 50.12. □

**Lemma 50.15** (Consistency)**.** *Logical equivalence is consistent.*

*Proof.* Follows immediately from the definition of logical equivalence. □

**Corollary 50.16.** *If* $\Delta; \Gamma \vdash e \sim e' : \tau$, *then* $\Delta; \Gamma \vdash e \cong e' : \tau$.

*Proof.* By Lemma 50.15 Logical equivalence is consistent, and by Lemma 50.14, it is a congruence, and hence is contained in observational equivalence. □

**Corollary 50.17.** *Logical and observational equivalence coincide.*

*Proof.* By Corollaries 50.13 and 50.16. □

If $d : \tau$ and $d \longmapsto e$, then $d \sim_\tau e$, and hence by Corollary 50.16, $d \cong_\tau e$. Therefore if a relation respects observational equivalence, it must also be closed under converse evaluation. This shows that the second condition on admissibility is redundant, now that we have established the coincidence of logical and observational equivalence.



Preview

**Corollary 50.18** (Extensionality)**.**

1. $e \cong_{\tau_1 \to \tau_2} e'$ *iff for all* $e_1 : \tau_1$, $e(e_1) \cong_{\tau_2} e'(e_1)$.
2. $e \cong_{\forall(t.\tau)} e'$ *iff for all* $\rho$, $e[\rho] \cong_{[\rho/t]\tau} e'[\rho]$.

*Proof.* The forward direction is immediate in both cases, because observational equivalence is a congruence, by definition. The backward direction is proved similarly in both cases, by appeal to Theorem 50.12. In the first case, by Corollary 50.17 it suffices to show that $e \sim_{\tau_1 \to \tau_2} e'$. To this end suppose that $e_1 \sim_{\tau_1} e_1'$. We are to show that $e(e_1) \sim_{\tau_2} e'(e_1')$. By the assumption we have $e(e_1') \cong_{\tau_2} e'(e_1')$. By parametricity we have $e \sim_{\tau_1 \to \tau_2} e$, and hence $e(e_1) \sim_{\tau_2} e(e_1')$. The result then follows by Lemma 50.8. In the second case, by Corollary 50.17 it is sufficient to show that $e \sim_{\forall(t.\tau)} e'$. Suppose that $R : \rho \leftrightarrow \rho'$ for some closed types $\rho$ and $\rho'$. It suffices to show that $e[\rho] \sim_\tau e'[\rho']$ $[\eta : \delta \leftrightarrow \delta']$, where $\eta(t) = R$, $\delta(t) = \rho$, and $\delta'(t) = \rho'$. By the assumption we have $e[\rho'] \cong_{[\rho'/t]\tau} e'[\rho']$. By parametricity $e \sim_{\forall(t.\tau)} e$, and hence $e[\rho] \sim_\tau e'[\rho']$ $[\eta : \delta \leftrightarrow \delta']$. The result then follows by Lemma 50.8. □

**Lemma 50.19** (Identity Extension)**.** *Let* $\eta : \delta \leftrightarrow \delta$ *be such that* $\eta(t)$ *is observational equivalence at type* $\delta(t)$ *for each* $t \in dom(\delta)$*. Then* $e \sim_\tau e'$ $[\eta : \delta \leftrightarrow \delta]$ *iff* $e \cong_{\delta(\tau)} e'$.

*Proof.* The backward direction follows immediately from Theorem 50.12 and respect for observational equivalence. The forward direction is proved by induction on the structure of $\tau$, appealing to Corollary 50.18 to establish observational equivalence at function and polymorphic types. □

## 50.4 Parametricity Properties

The parametricity theorem enables us to deduce properties of expressions of **F** that hold solely because of their type. The stringencies of parametricity ensure that a polymorphic type has very few inhabitants. For example, we may prove that *every* expression of type $\forall(t.t \to t)$ behaves like the identity function.

**Theorem 50.20.** *Let* $e : \forall(t.t \to t)$ *be arbitrary, and let id be* $\Lambda(t.\lambda\ (x{:}t)\ x)$*. Then* $e \cong_{\forall(t.t \to t)} id$.

*Proof.* By Corollary 50.17 it is sufficient to show that $e \sim_{\forall(t.t \to t)} id$. Let $\rho$ and $\rho'$ be arbitrary closed types, let $R : \rho \leftrightarrow \rho'$ be an admissible relation, and suppose that $e_0\ R\ e_0'$. We are to show

$$e[\rho](e_0)\ R\ id[\rho](e_0'),$$

which, given the definition of *id* and closure under converse evaluation, is to say

$$e[\rho](e_0)\ R\ e_0'.$$

It suffices to show that $e[\rho](e_0) \cong_\rho e_0$, for then the result follows by the admissibility of $R$ and the assumption $e_0\ R\ e_0'$.

By Theorem 50.12 we have $e \sim_{\forall(t.t \to t)} e$. Let the relation $S : \rho \leftrightarrow \rho$ be defined by $d\ S\ d'$ iff $d \cong_\rho e_0$ and $d' \cong_\rho e_0$. This is clearly admissible, and we have $e_0\ S\ e_0$. It follows that

$$e[\rho](e_0)\ S\ e[\rho](e_0),$$



Preview

and so, by the definition of the relation $S$, $e[\rho](e_0) \cong_\rho e_0$.

□

In Chapter 20 we showed that product, sum, and natural numbers types are all definable in **F**. The proof of definability in each case consisted of showing that the type and its associated introduction and elimination forms are encodable in **F**. The encodings are correct in the (weak) sense that the dynamics of these constructs as given in the earlier chapters is derivable from the dynamics of **F** via these definitions. By taking advantage of parametricity we may extend these results to obtain a strong correspondence between these types and their encodings.

As a first example, let us consider the representation of the unit type, `unit`, in **F**, as defined in Chapter 20 by the following equations:

$$\begin{aligned}\texttt{unit} &= \forall(r.r \to r)\\ \langle\rangle &= \Lambda(r.\lambda\,(x{:}r)\,x)\end{aligned}$$

It is easy to see that $\langle\rangle : \texttt{unit}$ according to these definitions. But this merely says that the type `unit` is inhabited (has an element). What we would like to know is that, up to observational equivalence, the expression $\langle\rangle$ is the *only* element of that type. But this is precisely the content of Theorem 50.20. We say that the type `unit` is *strongly definable* within **F**.

Continuing in this vein, let us examine the definition of the binary product type in **F**, also given in Chapter 20:

$$\begin{aligned}\tau_1 \times \tau_2 &= \forall(r.(\tau_1 \to \tau_2 \to r) \to r)\\ \langle e_1, e_2\rangle &= \Lambda(r.\lambda\,(x{:}\tau_1 \to \tau_2 \to r)\,x(e_1)(e_2))\\ e \cdot \texttt{l} &= e[\tau_1](\lambda\,(x{:}\tau_1)\,\lambda\,(y{:}\tau_2)\,x)\\ e \cdot \texttt{r} &= e[\tau_2](\lambda\,(x{:}\tau_1)\,\lambda\,(y{:}\tau_2)\,y)\end{aligned}$$

It is easy to check that $\langle e_1, e_2\rangle \cdot \texttt{l} \cong_{\tau_1} e_1$ and $\langle e_1, e_2\rangle \cdot \texttt{r} \cong_{\tau_2} e_2$ by a direct calculation.

We wish to show that the ordered pair, as defined above, is the unique such expression, and hence that Cartesian products are strongly definable in **F**. We will make use of a lemma governing the behavior of the elements of the product type whose proof relies on Theorem 50.12.

**Lemma 50.21.** *If $e : \tau_1 \times \tau_2$, then $e \cong_{\tau_1\times\tau_2} \langle e_1, e_2\rangle$ for some $e_1 : \tau_1$ and $e_2 : \tau_2$.*

*Proof.* Expanding the definitions of pairing and the product type, and applying Corollary 50.17, we let $\rho$ and $\rho'$ be arbitrary closed types, and let $R : \rho \leftrightarrow \rho'$ be an admissible relation between them. Suppose further that

$$h \sim_{\tau_1\to\tau_2\to t} h'\ [\eta : \delta \leftrightarrow \delta'],$$

where $\eta(t) = R$, $\delta(t) = \rho$, and $\delta'(t) = \rho'$ (and each is undefined on $t' \neq t$). We are to show that for some $e_1 : \tau_1$ and $e_2 : \tau_2$,

$$e[\rho](h) \sim_t h'(e_1)(e_2)\ [\eta : \delta \leftrightarrow \delta'],$$

which is to say

$$e[\rho](h)\ R\ h'(e_1)(e_2).$$

Now by Theorem 50.12 we have $e \sim_{\tau_1\times\tau_2} e$. Define the relation $S : \rho \leftrightarrow \rho'$ by $d\ S\ d'$ iff the following conditions are satisfied:



Preview

1. $d \cong_\rho h(d_1)(d_2)$ for some $d_1 : \tau_1$ and $d_2 : \tau_2$;
2. $d' \cong_{\rho'} h'(d'_1)(d'_2)$ for some $d'_1 : \tau_1$ and $d'_2 : \tau_2$;
3. $d \; R \; d'$.

This is clearly an admissible relation. Noting that

$$h \sim_{\tau_1 \to \tau_2 \to t} h' \; [\eta' : \delta \leftrightarrow \delta'],$$

where $\eta'(t) = S$ and $\eta'(t')$ is undefined for $t' \neq t$, we conclude that $e[\rho](h) \; S \; e[\rho'](h')$, and hence

$$e[\rho](h) \; R \; h'(d'_1)(d'_2),$$

as required. □

Now suppose that $e : \tau_1 \times \tau_2$ is such that $e \cdot \mathtt{l} \cong_{\tau_1} e_1$ and $e \cdot \mathtt{r} \cong_{\tau_2} e_2$. We wish to show that $e \cong_{\tau_1 \times \tau_2} \langle e_1, e_2 \rangle$. From Lemma 50.21 it follows that $e \cong_{\tau_1 \times \tau_2} \langle e \cdot \mathtt{l}, e \cdot \mathtt{r} \rangle$ by congruence and direct calculation. Hence, by congruence we have $e \cong_{\tau_1 \times \tau_2} \langle e_1, e_2 \rangle$.

By a similar line of reasoning we may show that the Church encoding of the natural numbers given in Chapter 20 strongly defines the natural numbers in that the following properties hold:

1. $\mathtt{iter}\,\mathtt{z}\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\} \cong_\rho e_0$.
2. $\mathtt{iter}\,\mathtt{s}(e)\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\} \cong_\rho [\mathtt{iter}\,e\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}/x]e_1$.
3. Suppose that $x : \mathtt{nat} \vdash r(x) : \rho$. If
   (a) $r(\mathtt{z}) \cong_\rho e_0$, and
   (b) $r(\mathtt{s}(e)) \cong_\rho [r(e)/x]e_1$,

   then for every $e : \mathtt{nat}$, $r(e) \cong_\rho \mathtt{iter}\,e\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}$.

The first two equations, which constitute weak definability, are easily established by calculation, using the definitions given in Chapter 20. The third property, the unicity of the iterator, is proved using parametricity by showing that every closed expression of type $\mathtt{nat}$ is observationally equivalent to a numeral $\overline{n}$. We then argue for unicity of the iterator by mathematical induction on $n \geq 0$.

**Lemma 50.22.** *If $e : \mathtt{nat}$, then either $e \cong_{\mathtt{nat}} \mathtt{z}$, or there exists $e' : \mathtt{nat}$ such that $e \cong_{\mathtt{nat}} \mathtt{s}(e')$. Consequently, there exists $n \geq 0$ such that $e \cong_{\mathtt{nat}} \overline{n}$.*

*Proof.* By Theorem 50.12 we have $e \sim_{\mathtt{nat}} e$. Define the relation $R : \mathtt{nat} \leftrightarrow \mathtt{nat}$ to be the strongest relation such that $d \; R \; d'$ iff either $d \cong_{\mathtt{nat}} \mathtt{z}$ and $d' \cong_{\mathtt{nat}} \mathtt{z}$, or $d \cong_{\mathtt{nat}} \mathtt{s}(d_1)$ and $d' \cong_{\mathtt{nat}} \mathtt{s}(d'_1)$ and $d_1 \; R \; d'_1$. It is easy to see that $\mathtt{z} \; R \; \mathtt{z}$, and if $e \; R \; e'$, then $\mathtt{s}(e) \; R \; \mathtt{s}(e')$. Letting $\mathtt{zero} = \mathtt{z}$ and $\mathtt{succ} = \lambda\,(x{:}\mathtt{nat})\,\mathtt{s}(x)$, we have

$$e[\mathtt{nat}](\mathtt{zero})(\mathtt{succ}) \; R \; e[\mathtt{nat}](\mathtt{zero})(\mathtt{succ}).$$

The result follows by the induction principle arising from the definition of $R$ as the strongest relation satisfying its defining conditions. □



Preview

## 50.5 Representation Independence, Revisited

In Section 21.4 we discussed the property of *representation independence* for abstract types. This property states that if two implementations of an abstract type are "similar", then the client behavior is not affected by replacing one for the other. The crux of the matter is the definition of similarity of two implementations. Informally, two implementations of an abstract type are similar if there is a relation, $R$, between their representation types that is *preserved* by the operations of the type. The relation $R$ may be thought of as expressing the "equivalence" of the two representations; checking that each operation preserves $R$ amounts to checking that the result of performing that operation on equivalent representations yields equivalent results.

As an example, we argued informally in Section 21.4 that two implementations of a queue abstraction are similar. The two representations of queues are related by a relation, $R$, such that $q\,R\,(b, f)$ iff $q$ is $b$ followed by the reversal of $f$. When then argued that the operations preserve this relationship, and then claimed, without proof, that the behavior of the client would not be disrupted by changing one implementation to the other.

The proof of this claim relies on parametricity, as may be seen by considering the definability of existential types in **F** given in Section 21.3. According to that definition, the client, $e$, of an abstract type $\exists(t.\tau)$ is a polymorphic function of type $\forall(t.\tau \to \tau_2)$, where $\tau_2$, the result type of the computation, does not involve the type variable $t$. Being polymorphic, the client enjoys the parametricity property given by Theorem 50.12. Specifically, suppose that $\rho_1$ and $\rho_2$ are two closed representation types and that $R : \rho_1 \leftrightarrow \rho_2$ is an admissible relation between them. For example, in the case of the queue abstraction, $\rho_1$ is the type of lists of elements of the queue, $\rho_2$ is the type of a pair of lists of elements, and $R$ is the relation given above. Suppose further that $e_1 : [\rho_1/t]\tau$ and $e_2 : [\rho_2/t]\tau$ are two implementations of the operations such that

$$e_1 \sim_\tau e_2\ [\eta : \delta_1 \leftrightarrow \delta_2], \tag{50.5}$$

where $\eta(t) = R$, $\delta_1(t) = \rho_1$, and $\delta_2(t) = \rho_2$. In the case of the queues example the expression $e_1$ is the implementation of the queue operations in terms of lists, and the $e_2$ is the implementation in terms of pairs of lists described earlier. Condition (50.5) states that the two implementations are similar in that they preserve the relation $R$ between the representation types. By Theorem 50.12 it follows that the client, $e$, satisfies

$$e \sim_{\tau_2} e\ [\eta : \delta_1 \leftrightarrow \delta_2].$$

But because $\tau_2$ is a closed type (in particular, does not involve $t$), this is equivalent to

$$e \sim_{\tau_2} e\ [\emptyset : \emptyset \leftrightarrow \emptyset].$$

But then by Lemma 50.19 we have

$$e[\rho_1](e_1) \cong_{\tau_2} e[\rho_2](e_2).$$

That is, the client behavior is not affected by the change of representation.



Preview

## 50.6 Notes

The concept of parametricity is latent in the proof of normalization for System **F** (Girard, 1972). Reynolds (1983), though technically flawed due to its reliance on a (non-existent) set-theoretic model of polymorphism, emphasizes the centrality of logical equivalence for characterizing equality of polymorphic programs. The application of parametricity to representation independence was suggested by Reynolds, and developed for existential types by Mitchell (1986) and Pitts (1998). The extension of System **F** with a positive (in the sense of Chapter 39) observable type appears to be needed to even define observational equivalence, but this point seems not to have been made elsewhere in the literature.

Preview

# Chapter 51

# Process Equivalence

As the name implies a process is an ongoing computation that may interact with other processes by sending and receiving messages. From this point of view a concurrent computation has no definite "final outcome" but rather affords an opportunity for interaction that may well continue indefinitely. The notion of equivalence of processes must therefore be based on their potential for interaction, rather than on the "answer" that they may compute. Let $P$ and $Q$ be such that $\vdash_\Sigma P$ proc and $\vdash_\Sigma Q$ proc. We say that $P$ and $Q$ are *equivalent*, written $P \approx_\Sigma Q$, iff there is a bisimulation, $\mathcal{R}$, such that $P\,\mathcal{R}_\Sigma\,Q$. A family of relations $\mathcal{R} = \{\,\mathcal{R}_\Sigma\,\}_\Sigma$ is a *bisimulation* iff whenever $P$ may evolve to $P'$ taking the action $\alpha$, then $Q$ may also evolve to some process $Q'$ taking the same action such that $P'\ \mathcal{R}_\Sigma\ Q'$, and, conversely, if $Q$ may evolve to $Q'$ taking action $\alpha$, then $P$ may evolve to $P'$ taking the same action, and $P'\ \mathcal{R}_\Sigma\ Q'$. This captures the idea that the two processes afford the same opportunities for interaction in that they each simulate each other's behavior with respect to their ability to interact with their environment.

## 51.1 Process Calculus

We will consider a process calculus that consolidates the main ideas explored in Chapters 42 and 43. We assume as given an ambient language of expressions that includes the type `clsfd` of classified values (see Chapter 35). Channels are treated as dynamically generated classes with which to build messages, as described in Chapter 43.

The syntax of the process calculus is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Proc | $P$ | ::= | `stop` | $\mathbf{1}$ | inert |
| | | | `par`$(P_1; P_2)$ | $P_1 \parallel P_2$ | composition |
| | | | `await`$(E)$ | $\$\,E$ | synchronize |
| | | | `new`$[\tau](a.P)$ | $\nu\, a \sim \tau.P$ | allocation |
| | | | `emit`$(e)$ | $!\,e$ | broadcast |
| Evt | $E$ | ::= | `null` | $\mathbf{0}$ | null |
| | | | `or`$(E_1; E_2)$ | $E_1 + E_2$ | choice |
| | | | `acc`$(x.P)$ | $?\,(x.P)$ | acceptance |

Preview

The statics is given by the judgments $\Gamma \vdash_\Sigma P$ proc and $\Gamma \vdash_\Sigma E$ event defined by the following rules. We assume as given a judgment $\Gamma \vdash_\Sigma e : \tau$ for $\tau$ a type including the type `clsfd` of classified values.

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{1} \text{ proc}} \tag{51.1a}$$

$$\frac{\Gamma \vdash_\Sigma P_1 \text{ proc} \quad \Gamma \vdash_\Sigma P_2 \text{ proc}}{\Gamma \vdash_\Sigma P_1 \parallel P_2 \text{ proc}} \tag{51.1b}$$

$$\frac{\Gamma \vdash_\Sigma E \text{ event}}{\Gamma \vdash_\Sigma \$\, E \text{ proc}} \tag{51.1c}$$

$$\frac{\Gamma \vdash_{\Sigma, a \sim \tau} P \text{ proc}}{\Gamma \vdash_\Sigma \nu\, a \sim \tau . P \text{ proc}} \tag{51.1d}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{clsfd}}{\Gamma \vdash_\Sigma !\, e \text{ proc}} \tag{51.1e}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{0} \text{ event}} \tag{51.1f}$$

$$\frac{\Gamma \vdash_\Sigma E_1 \text{ event} \quad \Gamma \vdash_\Sigma E_2 \text{ event}}{\Gamma \vdash_\Sigma E_1 + E_2 \text{ event}} \tag{51.1g}$$

$$\frac{\Gamma, x : \texttt{clsfd} \vdash_\Sigma P \text{ proc}}{\Gamma \vdash_\Sigma ?\,(x.P) \text{ event}} \tag{51.1h}$$

The dynamics is given by the judgments $P \overset{\alpha}{\underset{\Sigma}{\mapsto}} P'$ and $E \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} P$, defined as in Chapter 42. We assume as given the judgments $e \underset{\Sigma}{\mapsto} e'$ and $e \text{ val}_\Sigma$ for expressions. Processes and events are identified up to structural congruence, as described in Chapter 42.

$$\frac{P_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}} P_1'}{P_1 \parallel P_2 \overset{\alpha}{\underset{\Sigma}{\mapsto}} P_1' \parallel P_2} \tag{51.2a}$$

$$\frac{P_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}} P_1' \quad P_2 \overset{\bar{\alpha}}{\underset{\Sigma}{\mapsto}} P_2'}{P_1 \parallel P_2 \overset{\varepsilon}{\underset{\Sigma}{\mapsto}} P_1' \parallel P_2'} \tag{51.2b}$$

$$\frac{E \overset{\alpha}{\underset{\Sigma}{\Rightarrow}} P}{\$\, E \overset{\alpha}{\underset{\Sigma}{\mapsto}} P} \tag{51.2c}$$

$$\frac{P \overset{\alpha}{\underset{\Sigma, a \sim \tau}{\longmapsto}} P' \quad \vdash_\Sigma \alpha \text{ action}}{\nu\, a \sim \tau . P \overset{\alpha}{\underset{\Sigma}{\mapsto}} \nu\, a \sim \tau . P'} \tag{51.2d}$$

Preview

$$\frac{e\ \mathsf{val}_\Sigma \quad \vdash_\Sigma e : \mathtt{clsfd}}{!\,e \overset{e!}{\underset{\Sigma}{\longmapsto}} \mathbf{1}} \tag{51.2e}$$

$$\frac{E_1 \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} P}{E_1 + E_2 \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} P} \tag{51.2f}$$

$$\frac{e\ \mathsf{val}_\Sigma}{?\,(x.P) \overset{e?}{\underset{\Sigma}{\Longrightarrow}} [e/x]P} \tag{51.2g}$$

Assuming that substitution is valid for expressions, it is also valid for processes and events.

**Lemma 51.1.**

1. *If* $\Gamma, x : \tau \vdash_\Sigma P$ *proc and* $\Gamma \vdash_\Sigma e : \tau$, *then* $\Gamma \vdash_\Sigma [e/x]P$ *proc.*
2. *If* $\Gamma, x : \tau \vdash_\Sigma E$ *event and* $\Gamma \vdash_\Sigma e : \tau$, *then* $\Gamma \vdash_\Sigma [e/x]E$ *event.*

Transitions preserve well-formedness of processes and events.

**Lemma 51.2.**

1. *If* $\vdash_\Sigma P$ *proc and* $P \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P'$, *then* $\vdash_\Sigma P'$ *proc.*
2. *If* $\vdash_\Sigma E$ *event and* $E \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} P$, *then* $\vdash_\Sigma P$ *proc.*

## 51.2 Strong Equivalence

Bisimilarity makes precise the informal idea that two processes are equivalent if they each can take the same actions and, in doing so, evolve into equivalent processes. A *process relation*, $\mathcal{P}$, is a family $\{\mathcal{P}_\Sigma\}$ of binary relations between processes $P$ and $Q$ such that $\vdash_\Sigma P$ proc and $\vdash_\Sigma Q$ proc, and an *event relation*, $\mathcal{E}$, is a family $\{\mathcal{E}_\Sigma\}$ of binary relations between events $E$ and $F$ such that $\vdash_\Sigma E$ event and $\vdash_\Sigma F$ event. A *(strong) bisimulation* is a pair $(\mathcal{P}, \mathcal{E})$ consisting of a process relation, $\mathcal{P}$, and an event relation, $\mathcal{E}$, satisfying the following conditions:

1. If $P\ \mathcal{P}_\Sigma\ Q$, then
   (a) if $P \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P'$, then there exists $Q'$ such that $Q \overset{\alpha}{\underset{\Sigma}{\longmapsto}} Q'$ with $P'\ \mathcal{P}_\Sigma\ Q'$, and
   (b) if $Q \overset{\alpha}{\underset{\Sigma}{\longmapsto}} Q'$, then there exists $P'$ such that $P \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P'$ with $P'\ \mathcal{P}_\Sigma\ Q'$.
2. If $E\ \mathcal{E}_\Sigma\ F$, then
   (a) if $E \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} P$, then there exists $Q$ such that $F \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} Q$ with $P\ \mathcal{P}_\Sigma\ Q$, and
   (b) if $F \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} Q$, then there exists $P$ such that $E \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} P$ with $P\ \mathcal{P}_\Sigma\ Q$.

Preview

The qualifier "strong" refers to the fact that the action, $\alpha$, in the conditions on being a bisimulation include the silent action, $\varepsilon$. (In Section 51.3 we discuss another notion of bisimulation in which the silent actions are treated specially.)

*(Strong) equivalence* is the pair $(\approx, \approx)$ of process and event relations such that $P \approx_\Sigma Q$ and $E \approx_\Sigma F$ iff there exists a strong bisimulation $(\mathcal{P}, \mathcal{E})$ such that $P \; \mathcal{P}_\Sigma \; Q$, and $E \; \mathcal{E}_\Sigma \; F$.

**Lemma 51.3.** *Strong equivalence is a strong bisimulation.*

*Proof.* Follows immediately from the definition. □

The definition of strong equivalence gives rise to the principle of *proof by coinduction*. To show that $P \approx_\Sigma Q$, it is enough to give a bisimulation $(\mathcal{P}, \mathcal{E})$ such that $P \; \mathcal{P}_\Sigma \; Q$ (and similarly for events). An instance of coinduction that arises fairly frequently is to choose $(\mathcal{P}, \mathcal{E})$ to be $(\approx \cup \mathcal{P}_0, \approx \cup \mathcal{E}_0)$ for some $\mathcal{P}_0$ and $\mathcal{E}_0$ such that $P \; \mathcal{P}_0 \; Q$, and show that this expansion is a bisimulation. Because strong equivalence is itself a bisimulation, this reduces to show that if $P' \; \mathcal{P}_0 \; Q'$ and $P' \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P''$, then $Q' \overset{\alpha}{\underset{\Sigma}{\longmapsto}} Q''$ for some $Q''$ such that either $P'' \approx_\Sigma Q''$ or $P'' \; \mathcal{P}_0 \; Q''$ (and analogously for transitions from $Q'$, and similarly for event transitions). This proof method amounts to *assuming what we are trying to prove* and showing that this assumption is tenable. The proof that the expanded relation is a bisimulation may make use of the assumptions $\mathcal{P}_0$ and $\mathcal{E}_0$; in this sense "circular reasoning" is a perfectly valid method of proof.

**Lemma 51.4.** *Strong equivalence is an equivalence relation.*

*Proof.* For reflexivity and symmetry, it suffices to note that the identity relation is a bisimulation, as is the converse of a bisimulation. For transitivity we need that the composition of two bisimulations is again a bisimulation, which follows directly from the definition. □

It remains to verify that strong equivalence is a congruence, which means that each of the process- and event-forming constructs respects strong equivalence. To show this we require the *open extension* of strong equivalence to processes and events with free variables. The relation $\Gamma \vdash_\Sigma P \approx Q$ is defined for processes $P$ and $Q$ such that $\Gamma \vdash_\Sigma P \; \mathsf{proc}$ and $\Gamma \vdash_\Sigma Q \; \mathsf{proc}$ to mean that $\hat{\gamma}(P) \approx_\Sigma \hat{\gamma}(Q)$ for every substitution, $\gamma$, of closed values of appropriate type for the variables $\Gamma$.

**Lemma 51.5.** *If* $\Gamma, x : \mathtt{clsfd} \vdash_\Sigma P \approx Q$, *then* $\Gamma \vdash_\Sigma ?\,(x.P) \approx ?\,(x.Q)$.

*Proof.* Fix a closing substitution, $\gamma$, for $\Gamma$, and let $\hat{P} = \hat{\gamma}(P)$ and $\hat{Q} = \hat{\gamma}(Q)$. By assumption we have $x : \mathtt{clsfd} \vdash_\Sigma \hat{P} \approx \hat{Q}$. We are to show that $?\,(x.\hat{P}) \approx_\Sigma ?\,(x.\hat{Q})$. The proof is by coinduction, taking $\mathcal{P} = \approx$ and $\mathcal{E} = \approx \cup \mathcal{E}_0$, where

$$\mathcal{E}_0 = \{\, (?\,(x.P'), ?\,(x.Q')) \mid x : \mathtt{clsfd} \vdash_\Sigma P' \approx Q' \,\}.$$

Clearly $?\,(x.\hat{P}) \; \mathcal{E}_0 \; ?\,(x.\hat{Q})$. Suppose that $?\,(x.P') \; \mathcal{E}_0 \; ?\,(x.Q')$. By inspection of rules (51.2), if $?\,(x.P') \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} P''$, then $\alpha = v?$ and $P'' = [v/x]P'$ for some $v \; \mathsf{val}_\Sigma$ such that $\vdash_\Sigma v : \mathtt{clsfd}$. But $?\,(x.Q') \overset{v?}{\underset{\Sigma}{\Longrightarrow}} [v/x]Q'$, and we have that $[v/x]P' \approx_\Sigma [v/x]Q'$ by the definition of $\mathcal{E}_0$, and hence $[v/x]P' \; \mathcal{E}_0 \; [v/x]Q'$, as required. The symmetric case follows symmetrically, completing the proof. □

Preview

**Lemma 51.6.** *If* $\Gamma \vdash_{\Sigma,a\sim\tau} P \approx Q$, *then* $\Gamma \vdash_\Sigma \nu\, a\sim\tau.P \approx \nu\, a\sim\tau.Q$.

*Proof.* Fix a closing value substitution, $\gamma$, for $\Gamma$, and let $\hat{P} = \hat{\gamma}(P)$ and $\hat{Q} = \hat{\gamma}(Q)$. Assuming that $\hat{P} \approx_{\Sigma,a\sim\tau} \hat{Q}$, we are to show that $\nu\, a\sim\tau.\hat{P} \approx_\Sigma \nu\, a\sim\tau.\hat{Q}$. The proof is by coinduction, taking $\mathcal{P} = \approx \cup\, \mathcal{P}_0$ and $\mathcal{E} = \approx$, where

$$\mathcal{P}_0 = \{\, (\nu\, a\sim\tau.P', \nu\, a\sim\tau.Q') \mid P' \approx_{\Sigma,a\sim\tau} Q' \,\}.$$

Clearly $\nu\, a\sim\tau.\hat{P}\ \mathcal{P}_0\ \nu\, a\sim\tau.\hat{Q}$. Suppose that $\nu\, a\sim\tau.P'\ \mathcal{P}_0\ \nu\, a\sim\tau.Q'$, and that $\nu\, a\sim\tau.P' \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P''$. By inspection of rules (51.2), we see that $\vdash_\Sigma \alpha$ action and that $P'' = \nu\, a\sim\tau.P'''$ for some $P'''$ such that $P' \overset{\alpha}{\underset{\Sigma,a\sim\tau}{\longmapsto}} P'''$. But by definition of $\mathcal{P}_0$ we have $P' \approx_{\Sigma,a\sim\tau} Q'$, and hence $Q' \overset{\alpha}{\underset{\Sigma,a\sim\tau}{\longmapsto}} Q'''$ with $P''' \approx_{\Sigma,a\sim\tau} Q'''$. Letting $Q'' = \nu\, a\sim\tau.Q'$, we have that $\nu\, a\sim\tau.Q' \overset{\alpha}{\underset{\Sigma,a\sim\tau}{\longmapsto}} Q''$ and by definition of $\mathcal{P}_0$ we have $P''\ \mathcal{P}_0\ Q''$, as required. The symmetric case is proved symmetrically, completing the proof. □

Lemmas 51.5 and 51.6 capture two different cases of binding, the former of variables, and the latter of classes. The hypothesis of Lemma 51.5 relates all substitutions for the variable $x$ in the recipient processes, whereas the hypothesis of Lemma 51.6 relates the constituent processes schematically in the class name, $a$. This makes all the difference, for if we were to consider all substitution instances of a class name by another class name, then a class would no longer be "new" within its scope, because we could identify it with an "old" class by substitution. On the other hand we *must* consider substitution instances for variables, because the meaning of a variable is given in such terms. This shows that classes and variables must be distinct concepts. (See Chapter 35 for an example of what goes wrong when the two concepts are confused.)

**Lemma 51.7.** *If* $\Gamma \vdash_\Sigma P_1 \approx Q_1$ *and* $\Gamma \vdash_\Sigma P_2 \approx Q_2$, *then* $\Gamma \vdash_\Sigma P_1 \parallel P_2 \approx Q_1 \parallel Q_2$.

*Proof.* Let $\gamma$ be a closing value substitution for $\Gamma$, and let $\hat{P}_i = \hat{\gamma}(P_i)$ and $\hat{Q}_i = \hat{\gamma}(Q_i)$ for $i = 1, 2$. The proof is by coinduction, considering the relation $\mathcal{P} = \approx \cup\, \mathcal{P}_0$ and $\mathcal{E} = \approx$, where

$$\mathcal{P}_0 = \{\, (P_1' \parallel P_2', Q_1' \parallel Q_2') \mid P_1' \approx_\Sigma Q_1' \text{ and } P_2' \approx_\Sigma Q_2' \,\}.$$

Suppose that $P_1' \parallel P_2'\ \mathcal{P}_0\ Q_1' \parallel Q_2'$, and that $P_1' \parallel P_2' \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P''$. There are two cases to consider, the interesting one being rule (51.2b). In this case we have $P'' = P_1'' \parallel P_2''$ with $P_1' \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P_1''$ and $P_2' \overset{\overline{\alpha}}{\underset{\Sigma}{\longmapsto}} P_2''$. By definition of $\mathcal{P}_0$ we have that $Q_1' \overset{\alpha}{\underset{\Sigma}{\longmapsto}} Q_1''$ and $Q_2' \overset{\overline{\alpha}}{\underset{\Sigma}{\longmapsto}} Q_2''$ with $P_1'' \approx_\Sigma Q_1''$ and $P_2'' \approx_\Sigma Q_2''$. Letting $Q'' = Q_1'' \parallel Q_2''$, we have that $P''\ \mathcal{P}_0\ Q''$, as required. The symmetric case is handled symmetrically, and rule (51.2a) is handled similarly. □

**Lemma 51.8.** *If* $\Gamma \vdash_\Sigma E_1 \approx F_1$ *and* $\Gamma \vdash_\Sigma E_2 \approx F_2$, *then* $\Gamma \vdash_\Sigma E_1 + E_2 \approx F_1 + F_2$.

*Proof.* Follows immediately from rules (51.2) and the definition of bisimulation. □

**Lemma 51.9.** *If* $\Gamma \vdash_\Sigma E \approx F$, *then* $\Gamma \vdash_\Sigma \$\, E \approx \$\, F$.

Preview

*Proof.* Follows immediately from rules (51.2) and the definition of bisimulation. □

**Lemma 51.10.** *If* $\Gamma \vdash_\Sigma d \cong e : \mathtt{clsfd}$, *then* $\Gamma \vdash_\Sigma !\,d \approx !\,e$.

*Proof.* The process calculus introduces no new observations on expressions, so that $d$ and $e$ remain indistinguishable as actions. □

**Theorem 51.11.** *Strong equivalence is a congruence.*

*Proof.* Follows immediately from the preceding lemmas, which cover each case separately. □

## 51.3 Weak Equivalence

Strong equivalence expresses the idea that two processes are equivalent if they simulate each other step-by-step. Every action taken by one process is matched by a corresponding action taken by the other. This seems natural for the non-trivial actions $e!$ and $e?$, but is arguably overly restrictive for the silent action, $\varepsilon$. Silent actions correspond to the actual steps of computation, whereas the send and receive actions express the potential to interact with another process. Silent steps are therefore of a very different flavor than the other forms of action, and therefore might usefully be treated differently from them. Weak equivalence seeks to do just that.

Silent actions arise within the process calculus itself (when two processes communicate), but they play an even more important role when the dynamics of expressions is considered explicitly (as in Chapter 43). For then each step $e \underset{\Sigma}{\longmapsto} e'$ of evaluation of an expression corresponds to a silent transition for any process in which it is embedded. In particular, $!\,e \overset{\varepsilon}{\underset{\Sigma}{\longmapsto}} !\,e'$ whenever $e \underset{\Sigma}{\longmapsto} e'$. We may also consider atomic processes of the form $\mathtt{proc}(m)$ consisting of a command to be executed in accordance with the rules of some underlying dynamics. Here again we would expect that each step of command execution induces a silent transition from one atomic process to another.

From the point of view of equivalence, it therefore seems sensible to allow that a silent action by one process may be mimicked by one or more silent actions by another. For example, there appears to be little to be gained by distinguishing, say, $\mathtt{proc}(\mathtt{ret}\ 3{+}4)$ from $\mathtt{proc}(\mathtt{ret}\ (1{+}2){+}(2{+}2))$ merely because the latter takes more steps to compute the same value than the former! The purpose of weak equivalence is precisely to disregard such trivial distinctions by allowing a transition to be matched by a matching transition, possibly preceded by any number of silent transitions.

A *weak bisimulation* is a pair $(\mathcal{P}, \mathcal{E})$ consisting of a process relation, $\mathcal{P}$, and an event relation, $\mathcal{E}$, satisfying the following conditions:

1. If $P\ \mathcal{P}_\Sigma\ Q$, then
   (a) if $P \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P'$, where $\alpha \neq \varepsilon$, then there exists $Q''$ and $Q'$ such that $Q \overset{\varepsilon}{\underset{\Sigma}{\longmapsto}}^* Q'' \overset{\alpha}{\underset{\Sigma}{\longmapsto}} Q'$ with $P'\ \mathcal{P}_\Sigma\ Q'$, and if $P \overset{\varepsilon}{\underset{\Sigma}{\longmapsto}} P'$, then $Q \overset{\varepsilon}{\underset{\Sigma}{\longmapsto}}^* Q'$ with $P'\ \mathcal{P}_\Sigma\ Q'$;
   (b) if $Q \overset{\alpha}{\underset{\Sigma}{\longmapsto}} Q'$, where $\alpha \neq \varepsilon$, then there exists $P''$ and $P'$ such that $P \overset{\varepsilon}{\underset{\Sigma}{\longmapsto}}^* P'' \overset{\alpha}{\underset{\Sigma}{\longmapsto}} P'$ with $P'\ \mathcal{P}_\Sigma\ Q'$, and if $Q \overset{\varepsilon}{\underset{\Sigma}{\longmapsto}} Q'$, then $P \overset{\varepsilon}{\underset{\Sigma}{\longmapsto}}^* P'$ with $P'\ \mathcal{P}_\Sigma\ Q'$;

PREVIEW

2. If $E \; \mathcal{E}_\Sigma \; F$, then

   (a) if $E \stackrel{\alpha}{\underset{\Sigma}{\Rightarrow}} P$, then there exists $Q$ such that $F \stackrel{\alpha}{\underset{\Sigma}{\Rightarrow}} Q$ with $P \; \mathcal{P}_\Sigma \; Q$, and

   (b) if $F \stackrel{\alpha}{\underset{\Sigma}{\Rightarrow}} Q$, then there exists $P$ such that $E \stackrel{\alpha}{\underset{\Sigma}{\Rightarrow}} P$ with $P \; \mathcal{P}_\Sigma \; Q$.

(The conditions on the event relation are the same as for strong bisimilarity because there are, in this calculus, no silent actions on events.)

*Weak equivalence* is the pair $(\sim, \sim)$ of process and event relations defined by $P \sim_\Sigma Q$ and $E \sim_\Sigma F$ iff there exists a weak bisimulation $(\mathcal{P}, \mathcal{E})$ such that $P \; \mathcal{P}_\Sigma \; Q$, and $E \; \mathcal{E}_\Sigma \; F$. The open extension of weak equivalence, written $\Gamma \vdash_\Sigma P \sim Q$ and $\Gamma \vdash_\Sigma E \sim F$, is defined exactly as is the open extension of strong equivalence.

**Theorem 51.12.** *Weak equivalence is an equivalence relation and a congruence.*

*Proof.* The proof proceeds along similar lines to that of Theorem 51.11. □

## 51.4 Notes

The literature on process equivalence is extensive. Numerous variations have been considered for an equally numerous array of formalisms. Milner recounts the history and development of the concept of bisimilarity in his monograph on the $\pi$-calculus (Milner, 1999), crediting David Park with its original conception (Park, 1981). The development in this chapter is inspired by Milner, and by a proof of congruence of strong bisimilarity given by Bernardo Toninho for the process calculus considered in Chapter 42.

Preview

Preview

# Part XX

# Appendices

Preview

Preview

# Appendix A

# Finite Sets and Finite Functions

We make frequent use of the concepts of a *finite set* of *discrete objects* and of *finite functions* between them. A set $X$ is *discrete* iff equality of its elements is decidable: for every $x, y \in X$, either $x = y \in X$ or $x \neq y \in X$. This condition is to be understood constructively as stating that we may effectively determine whether any two elements of the set $X$ are equal or not. Perhaps the most basic example of a discrete set is the set, $\omega$, of natural numbers. A set $X$ is *countable* iff there is a bijection, $f : X \cong \omega$, between $X$ and the set of natural numbers, and it is *finite* iff there is a bijection, $f : X \cong \{ 0, \ldots, n-1 \}$, where $n \in \omega$, between it and some inital segment of the natural numbers. This condition is again to be understood constructively in terms of computable mappings, so that countable and finite sets are computably enumerable and, in the finite case, have a computable size.

Given countable sets, $U$ and $V$, a *finite function* is a computable partial function $\phi : U \to V$ between them. The *domain*, $dom(\phi)$, of $\phi$ is the set $\{ u \in U \mid \phi(u) \downarrow \}$, of objects $u \in U$ such that $\phi(u) = v$ for some $v \in V$. Two finite functions, $\phi$ and $\psi$, between $U$ and $V$ are *disjoint* iff $dom(\phi) \cap dom(\psi) = \emptyset$. The *empty* finite function, $\emptyset$, between $U$ and $V$ is the totally undefined partial function between them. If $u \in U$ and $v \in V$, the finite function, $u \hookrightarrow v$, between $U$ and $V$ sends $u$ to $v$, and is undefined otherwise; its domain is therefore the singleton set $\{ u \}$. In some situations we write $u \sim v$ for the finite function $u \hookrightarrow v$.

If $\phi$ and $\psi$ are two disjoint finite functions from $U$ to $V$, then $\phi \otimes \psi$ is the finite function from $U$ to $V$ defined by the equation

$$(\phi \otimes \psi)(u) = \begin{cases} \phi(u) & \text{if } u \in dom(\phi) \\ \psi(v) & \text{if } v \in dom(\psi) \\ \text{undefined} & \text{otherwise} \end{cases}$$

If $u_1, \ldots, u_n \in U$ are pairwise distinct, and $v_1, \ldots, v_n \in V$, then we sometimes write $u_1 \hookrightarrow v_1, \ldots, u_n \hookrightarrow v_n$, or $u_1 \sim v_1, \ldots, u_n \sim v_n$, for the finite function

$$u_1 \hookrightarrow v_1 \otimes \ldots \otimes u_n \hookrightarrow v_n.$$

Preview

Preview

# Appendix B

# Solutions to Exercises

## Chapter 1

**1.1**. Because $\mathcal{X} \subseteq \mathcal{Y}$, any variable in $\mathcal{A}[\mathcal{X}]$ is also a variable in $\mathcal{A}[\mathcal{Y}]$. Inductively, if $a_i \in \mathcal{A}[\mathcal{X}]$, then $a_i \in \mathcal{A}[\mathcal{Y}]$, and therefore $o(a_1, \ldots, a_n) \in \mathcal{A}[\mathcal{Y}]$.

**1.2**. Extending the solution to the preceding exercise, we need only account for abstractors. Suppose that $\vec{x}.a \in \mathcal{A}[\mathcal{X}]$, and we are to show that $\vec{x}.a \in \mathcal{A}[\mathcal{Y}]$. Pick any $\pi : \vec{x} \leftrightarrow \vec{x}'$ such that $\vec{x}' \notin \mathcal{Y}$. Noting that $\vec{x}' \notin \mathcal{X}$, because $\mathcal{X} \subseteq \mathcal{Y}$, we have that $\pi \cdot a \in \mathcal{A}[\mathcal{X}, \vec{x}']$, and hence inductively in $\mathcal{A}[\mathcal{Y}, \vec{x}']$ as well, which suffices for the result.

**1.3**. A standard solution is to represent back edges by *de Bruijn indices*: a bound variable occurrence is represented by $bv[i]$, where $i$ is a positive natural number designating the $i$th enclosing abstractor. An abstractor for abg's has the form $.g$, where the "dot" indicates the introductorion of an (unnamed) bound variable. Letting $\mathcal{G}[\mathcal{X}]_n$ stand for the abg's with $n$ enclosing abstractors, we may say that $.g$ is in $\mathcal{G}[\mathcal{X}]_{n+1}$ if $g \in \mathcal{G}[\mathcal{X}]_n$, and that $bv[i] \in \mathcal{G}[\mathcal{X}]_n$ if $1 \leq i \leq n$.

## Chapter 2

**2.1**. One possible definition of the judgment $\mathsf{max}(m;n;p)$ is given by the following rules:

$$\frac{}{\mathsf{max}(m;\mathtt{zero};m)} \tag{B.1a}$$

$$\frac{}{\mathsf{max}(\mathtt{zero};\mathtt{succ}(n);\mathtt{succ}(n))} \tag{B.1b}$$

Preview

$$\frac{\mathsf{max}(m;n;p)}{\mathsf{max}(\mathtt{succ}(m);\mathtt{succ}(n);\mathtt{succ}(p))} \tag{B.1c}$$

One may show by rule induction that this judgement has mode $(\forall, \forall, \exists)$ over the natural numbers, which is to say that to every two natural number inputs there corresponds a natural number output. A second nested pair of rule inductions shows that if $\mathsf{max}(m;n;p)$ and $\mathsf{max}(m;n;q)$, then $p$ is $q$. This establishes that the mode is $(\forall, \forall, \exists!)$, which is to say that the three-place relation defines a total function of its first two arguments.

**2.2**. Assume that $t$ tree and $n$ nat. As in Solution **2.1** it is easiest to prove first that $\mathsf{hgt}(t;n)$ has mode $(\forall, \exists)$, showing that every tree has at least one height. Then one may prove by rule induction that if $\mathsf{hgt}(t;m)$ and $\mathsf{hgt}(t;n)$, then $m$ is $n$.

**2.3**. The judgments $t$ tree and $f$ forest may be simultaneously defined by the following rules:

$$\frac{f \text{ forest}}{\mathtt{node}(f) \text{ tree}} \tag{B.2a}$$

$$\frac{}{\mathtt{nil} \text{ forest}} \tag{B.2b}$$

$$\frac{t \text{ tree} \quad f \text{ forest}}{\mathtt{cons}(t;f) \text{ forest}} \tag{B.2c}$$

The empty tree may be thought of as $\mathtt{node}(\mathtt{nil})$, the node with no children.

**2.4**. The judgments $\mathsf{hgtT}(t;n)$, stating that the variadic tree $t$ has height $n$, and $\mathsf{hgtF}(f;n)$, stating that the variadic forest $f$ has height $n$, may be simultaneously inductively defined by the following rules:

$$\frac{\mathsf{hgtF}(f;n)}{\mathsf{hgtT}(\mathtt{node}(f);\mathtt{succ}(n))} \tag{B.3a}$$

$$\frac{}{\mathsf{hgtF}(\mathtt{nil};\mathtt{zero})} \tag{B.3b}$$

$$\frac{\mathsf{hgtT}(t;m) \quad \mathsf{hgtF}(f;n) \quad \mathsf{max}(m;n;p)}{\mathsf{hgtF}(\mathtt{cons}(t;f);p)} \tag{B.3c}$$

The required modes may be proved as outlined in the preceding exercises, albeit by a simultaneous rule induction for each case.

**2.5**. The judgment $n$ bin stating that $n$ is a natural number in binary may be defined by the following rules:

$$\frac{}{\mathtt{zero} \text{ bin}} \tag{B.4a}$$

$$\frac{n \text{ bin}}{\mathtt{twice}(n) \text{ bin}} \tag{B.4b}$$



Preview

$$\frac{n\ \mathsf{bin}}{\mathtt{twiceplus1}(n)\ \mathsf{bin}} \tag{B.4c}$$

Clearly `zero` represents the number 0, and there are two forms of "successor", corresponding to doubling, and to doubling and adding one. The (unique) representation of the number $5 = 2 \times (2 \times 1) + 1$ is therefore

$$\mathtt{twiceplus1}(\mathtt{twice}(\mathtt{twiceplus1}(\mathtt{zero}))).$$

**2.6**. The sum of two numbers in binary, represented in binary, is given by the judgment $\mathsf{sum}(m;n;p)$ defined in terms of an auxiliary judgment $\mathsf{succ}(m;n)$, as follows:

$$\frac{}{\mathsf{sum}(\mathtt{zero};\mathtt{zero};\mathtt{zero})} \tag{B.5a}$$

$$\frac{\mathsf{sum}(m;n;p)}{\mathsf{sum}(\mathtt{twice}(m);\mathtt{twice}(n);\mathtt{twice}(p))} \tag{B.5b}$$

$$\frac{\mathsf{sum}(m;n;p)}{\mathsf{sum}(\mathtt{twice}(m);\mathtt{twiceplus1}(n);\mathtt{twiceplus1}(p))} \tag{B.5c}$$

$$\frac{\mathsf{sum}(m;n;p)}{\mathsf{sum}(\mathtt{twiceplus1}(m);\mathtt{twice}(n);\mathtt{twiceplus1}(p))} \tag{B.5d}$$

$$\frac{\mathsf{sum}(m;n;p) \quad \mathsf{succ}(p;q)}{\mathsf{sum}(\mathtt{twiceplus1}(m);\mathtt{twiceplus1}(n);\mathtt{twiceplus1}(q))} \tag{B.5e}$$

The auxiliary computation of the successor is defined as follows:

$$\frac{}{\mathsf{succ}(\mathtt{zero};\mathtt{twiceplus1}(\mathtt{zero}))} \tag{B.6a}$$

$$\frac{}{\mathsf{succ}(\mathtt{twice}(n);\mathtt{twiceplus1}(n))} \tag{B.6b}$$

$$\frac{\mathsf{succ}(n;p)}{\mathsf{succ}(\mathtt{twiceplus1}(n);\mathtt{twice}(p))} \tag{B.6c}$$

For correctness we must check that if $n$ bin then there exists $p$ bin such that $\mathsf{succ}(n;p)$, where $p$ is the representation of the successor of $n$ in binary, and that if $m$ bin and $n$ bin then there exists $p$ bin such that $\mathsf{sum}(m;n;p)$ and $p$ is the representation of the sum of $m$ and $n$ in binary. These may both be proved by induction on the foregoing rules, making use of such elementary facts as

$$(2 \times m + 1) + (2 \times n + 1) = 2 \times (m + n) + 2 = 2 \times (m + n + 1).$$

The sample solution above uses the successor to compute one more than the sum of $m$ and $n$, which is obtained recursively. This suggests the alternative solution in which one defines simultaneously both the sum of $m$ and $n$ and one more than the sum of $m$ and $n$. Each calls the other because

$$(2 \times m + 1) + (2 \times n) + 1 = 2 \times (m + n) + 2 = 2 \times (m + n + 1).$$

Preview

# Chapter 3

**3.1**. As usual, give an inductive definition of the two-place judgment $\mathsf{len}(a;n)$, where $a$ comb and $n$ nat, and show that it has mode $(\forall, \exists!)$ by induction on the given rules $\mathcal{C}$ defining $a$ comb.

**3.2**. Pick a renaming $x'$ of $x$, and extend rules $\mathcal{C}$ with the axiom $x'$ comb. Proceed by rule induction on this extended rule set, replacing $x'$ comb by $a_1$ comb at the base case, and otherwise proceeding inductively.

**3.3**. The required derivation is suggested by the following equivalences:

$$\begin{aligned} \mathtt{s}\,\mathtt{k}\,\mathtt{k}\,x &\equiv (\mathtt{k}\,x)\,(\mathtt{k}\,x) \\ &\equiv x \end{aligned}$$

The first is justified by the `S` axiom, the second by the `K` axiom.

**3.4**. The formulation of the question suggests to fix $x$ and define a judgment $\mathsf{abs}_x\ a\ \mathtt{is}\ a'$ and show that it has mode $(\forall, \exists!)$:

$$\frac{}{\mathsf{abs}_x\ x\ \mathtt{is}\ \mathtt{s}\,\mathtt{k}\,\mathtt{k}} \tag{B.7a}$$

$$\frac{a\ \mathsf{comb}}{\mathsf{abs}_x\ a\ \mathtt{is}\ \mathtt{k}\,a} \tag{B.7b}$$

$$\frac{\mathsf{abs}_x\ a_1\ \mathtt{is}\ a_1' \quad \mathsf{abs}_x\ a_2\ \mathtt{is}\ a_2'}{\mathsf{abs}_x\ a_1\,a_2\ \mathtt{is}\ \mathtt{s}\,a_1'\,a_2'} \tag{B.7c}$$

Rule (B.7b) specifies that $a$ is a combinator independent of the distinguished variable $x$. It is easy to check that the required equivalence holds, noting that the axioms governing `k` and `s` have been chosen precisely to make the proof go through without complication.

**3.5**. Simply redefine bracket abstraction so that $[x]\,a \triangleq \mathtt{ap}(\mathtt{k};a)$ when $x \notin a$. This generalizes the original case, where $a = y \neq x$, to avoid altering any combinator in which $x$ does not occur. Then prove that $[a/y][x]\,b = [x]\,[a/y]b$ under the stated conditions by induction on the derivation of $x\,y \mid x\ \mathsf{comb}\ y\ \mathsf{comb} \vdash b\ \mathsf{comb}$.

# Chapter 4

**4.1**. Many variations are possible. Here is an illustrative fragment of a solution that incorporates some of the suggestions given in Exercise 4.2.

$$\frac{}{\Gamma\,x \uparrow \tau \vdash x \uparrow \tau} \tag{B.8a}$$



Preview

$$\frac{\Gamma \vdash e \uparrow \tau}{\Gamma \vdash e \downarrow \tau} \tag{B.8b}$$

$$\frac{\Gamma \vdash e \downarrow \tau}{\Gamma \vdash \mathtt{cast}[\tau](e) \uparrow \tau} \tag{B.8c}$$

$$\frac{}{\Gamma \vdash \mathtt{num}[n] \downarrow \mathtt{num}} \tag{B.8d}$$

$$\frac{\Gamma \vdash e_1 \downarrow \mathtt{num} \quad \Gamma \vdash e_2 \downarrow \mathtt{num}}{\Gamma \vdash \mathtt{plus}(e_1; e_2) \uparrow \mathtt{num}} \tag{B.8e}$$

$$\frac{}{\Gamma \vdash \mathtt{str}[s] \downarrow \mathtt{str}} \tag{B.8f}$$

$$\frac{\Gamma \vdash e_1 \uparrow \tau_1 \quad \Gamma, x \uparrow \tau_1 \vdash e_2 \downarrow \tau_2}{\Gamma \vdash \mathtt{let}(e_1; x.e_2) \downarrow \tau_2} \tag{B.8g}$$

The separation of synthetic from analytic typing resolves the difficulty with the type of the defined term in a definition expression.

**4.2**. The main difficulty is to ensure that you do not preclude programs that ought to be allowed, or that can only be expressed very awkwardly. Within these constraints there are many possible variations on Solution **4.1**.

## Chapter 5

**5.1**. Proceed by rule induction on the first premise.

**5.2**. The definitions of multi-step and $k$-step transition are chosen so as to make this proof a routine induction as indicated. Because it is obvious that if $s \longmapsto^{k} s'$ and $s' \longmapsto^{k'} s''$, then $s \longmapsto^{k+k'} s''$, Solution **5.1** may be obtained as a corollary of this solution.

**5.3**. Proceed by rule induction on rules (5.10). The suggested strengthening ensures that rule (5.10f) can be proved without complication. The assumptions on $e_i$ and $e_i'$ are preserved when passing to the premises of rule (5.10f), and these assumptions are needed when considering reflexivity for a variable $x_i$. The rest of the proof is routine.

## Chapter 6

**6.1**. The remaining cases follow along the same lines as those given in the proof of Theorem 6.2.

**6.2**. The remaining cases follow along the same lines as those given in the proof of Theorem 6.4.

**6.3**. The suggested case analysis ensures that errors are propagated properly by each construct. The proof as a whole ensures that there are no well-typed "stuck" expressions other than values and checked errors.

Preview

# Chapter 7

**7.1**. Proceed by a simultaneous rule induction on rules (7.1).

**7.2**. Proceed along the same lines as those steps already given.

**7.3**. The second part proceeds by a rule induction on rules (5.1), appealing to the lemma in the inductive step.

**7.4**. The difficulty is that the progress theorem would allow an unchecked, as well as a checked, error in its statement. Moreover, Theorem 7.5 is no longer valid in the presence of a checked error, so safety is no longer a corollary of progress. The most obvious alternative is to introduce *two* forms of error checks, one for unchecked errors (solely to express safety), and one for checked errors (to allow for run-time errors arising from well-typed expressions). Such a formulation becomes rather baroque.

**7.5**. Besides the given rule for variables, the rule for definitions should be given as follows:

$$\frac{\Delta \vdash e_1 \Downarrow v_1 \quad \Delta, x_1 \Downarrow v_1 \vdash e_2 \Downarrow v_2}{\Delta \vdash \mathtt{let}(e_1; x.e_2) \Downarrow v_2}$$

The remaining rules are self-evident.

The left-to-right direction of the correctness proof is proved by induction on the rules defining the environmental evaluation dynamics. The right-to-left direction must be proved by induction on the structure of $e$, rather than on the derivation of $[v_1, \ldots, v_n / x_1, \ldots, x_n]e \Downarrow v$ so that it is clear when a variable is to be evaluated.

# Chapter 8

**8.1**. Introduce a new judgment form, $f \Downarrow' x.e$, and allow judgments of this form as hypotheses of evaluation. The evaluation rule for (call-by-name) application takes the form

$$\frac{\Delta \vdash [e'/x]e \Downarrow e''}{\Delta, f \Downarrow' x.e \vdash \mathtt{apply}[f](e') \Downarrow e''} \tag{B.9}$$

More provocatively, the atomic judgment $f \Downarrow' x.e$ can be understood instead as the generic judgment

$$x \mid \mathtt{apply}[f](x) \Downarrow e.$$

Admitting such a judgment as an assumption extends the framework given in Chapter 3 to admit *higher-order* judgment forms. Doing so requires some additional machinery that we do not develop further in this book.



Preview

**8.2**. The difficulty is how to specify the evaluation of a $\lambda$-abstraction, which may contain free variables governed by the hypotheses:

$$\frac{}{\Delta \vdash \mathtt{lam}[\tau](x.e) \Downarrow \textbf{???}}$$

The value of the $\lambda$ cannot be itself, as would be the case in a substitution-based evaluation semantics; to do so would be to lose the connection between the binding of a variable and its subsequent usage, amounting to a form of dynamic scope.

One natural solution is to replace each free variable in the $\lambda$-abstraction by its binding in $\Delta$ at the time that the $\lambda$ is evaluated:

$$\frac{}{x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash \mathtt{lam}[\tau](x.e) \Downarrow \mathtt{lam}[\tau](x.[\vec{v}/\vec{x}]e)}\ .$$

(We assume, without loss of generality, that $x$ is not already governed by an assumption in $\Delta$, so that no confusion of distinct variables may occur.) But doing so defeats the purpose of the environmental dynamics; we may use the substitutional evaluation dynamics instead.

A variation on this approach is to regard $[\vec{v}/\vec{x}]e$ as a form of expression, called an *explicit substitution*, or *closure*. At application we must perform a "context switch" from the ambient hypotheses to the hypotheses encoded in the closure:

$$\frac{\Delta \vdash e_1 \Downarrow [\vec{v}/\vec{x}]e \quad x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v}{\Delta \vdash \mathtt{ap}(e_1; e_2) \Downarrow v}\ .$$

Both approaches suffer from an abuse of the framework of inductive definitions in Chapter 3. Check, for example, that the formulation using closures does not admit weakening, a basic requirement for a well-defined a hypothetical judgment.

**8.3**. The example in Section 8.4 provides the basis for a solution to this problem. Other solutions may be readily generated based on this example.

# Chapter 9

**9.1**. Proceed by rule induction on the statics of **T**.

**9.2**. Decompose the safety theorem into a preservation and progress lemma, which are proved along standard lines, appealing to Lemma 9.2 in the progress proof.

**9.3**. The proof breaks down more or less immediately. For example, even if $e_1(e_2)$ : `nat`, the expression $e_1$ is of function type, and the theorem as stated provides no inductive hypothesis for it. But the termination of the application clearly depends on the termination of the function being applied!



Preview

**9.4**. The proof breaks down at application, for even if $e_1 : \tau_2 \to \tau$ is terminating and $e_2 : \tau_2$ is terminating, it does not follow directly that the application $e_1(e_2)$ is terminating. For example, $e_1$ might evaluate to a function that, when applied, fails to terminate. The inductive hypothesis provides no information with which to rule out this possibility.

**9.5**. The stronger inductive hypothesis is sufficient to handle applications: if $e_1 : \tau_2 \to \tau$ is hereditarily terminating, and $e_2 : \tau_2$ is hereditarily terminating, then so is $e_1(e_2)$, by definition. But how are we to show that $\lambda\,(x{:}\tau_1)\,e_2$ is hereditarily terminating at type $\tau_1 \to \tau_2$? We must show that if $e_1$ is hereditarily terminating at type $\tau_1$, then the application $(\lambda\,(x{:}\tau_1)\,e_2)(e_1)$ is hereditarily terminating at type $\tau_2$. The restriction to closed terms prevents us from applying the inductive hypothesis to $e_2$, because, in general, it has a free variable $x$ occurring within it. There is as yet no way to proceed.

**9.6**. Proceed by induction on the structure of $\tau$. If $\tau = \mathtt{nat}$, then the result is immediate by definition of hereditary termination at $\mathtt{nat}$. If $\tau = \tau_1 \to \tau_2$, let $e_1$ be hereditarily terminating at type $\tau_1$, and observe that $e'(e_1) \longmapsto e(e_1)$. But the latter expression is hereditarily terminating, and so, by induction, is the former.

Returning to Solution **9.5**, it suffices to show that $[e_1/x]e_2$ is hereditarily terminating at type $\tau_2$. This term is closed if $\lambda\,(x{:}\tau_1)\,e_2$ is closed, but we still do not have justification to conclude that the latter expression is hereditarily terminating, because the inductive hypothesis does not apply to open terms.

**9.7**. The final strengthening given in the exercise is now sufficient to show that every well-typed open term is open hereditarily terminating in the stated sense. The original result follows by considering a closed term of type $\mathtt{nat}$, which is thereby shown to be hereditary terminating, and hence terminating.

# Chapter 10

**10.1**. Let $\sigma = \prod_{i \in I} \tau_i$ be a database schema. A database on this schema may be considered to be a value of type $\mathtt{nat} \times (\mathtt{nat} \to \sigma)$ whose elements consist of pairs $\langle n, s\rangle$ such that the sequence $s$ is defined on all natural numbers less than $n$, and is undefined otherwise. Using this representation, the *project* function sending the database $\langle n, s\rangle$ onto $I' \subseteq I$ is given by the pair $\langle n, s'\rangle$, where $s'$ is the sequence

$$\lambda\,(k{:}\mathtt{nat})\,\langle s(k) \cdot i'\rangle_{i' \in I'}$$

that selects the columns specified by $I'$ from each row of the given database. The standard *select* and *join* operations on databases are similarly defined.

**10.2**. Negative in terms of positive:

$$\begin{aligned}
\langle e_1, e_2\rangle &\triangleq (\lambda\,(_{:}\mathtt{unit})\,e_1) \otimes (\lambda\,(_{:}\mathtt{unit})\,e_2)\\
e \cdot \mathtt{l} &\triangleq \mathtt{split}\,e\,\mathtt{as}\,x_1 \otimes _\,\mathtt{in}\,x_1(\langle\rangle)\\
e \cdot \mathtt{r} &\triangleq \mathtt{split}\,e\,\mathtt{as}\,_ \otimes x_2\,\mathtt{in}\,x_2(\langle\rangle)
\end{aligned}$$

Preview

Positive in terms of negative:

$$e_1 \otimes e_2 \triangleq \texttt{let}\, x_1\, \texttt{be}\, e_1\, \texttt{in}\, \texttt{let}\, x_2\, \texttt{be}\, e_2\, \texttt{in}\, x_1 \otimes x_2$$
$$\texttt{split}\, e_0\, \texttt{as}\, x_1 \otimes x_2\, \texttt{in}\, e \triangleq \texttt{let}\, x_1\, \texttt{be}\, e_0 \cdot \texttt{l}\, \texttt{in}\, \texttt{let}\, x_2\, \texttt{be}\, e_0 \cdot \texttt{r}\, \texttt{in}\, e$$

**10.3**. The introductory form would remain the same; the eliminatory form would be a degenerate form of decomposition:

$$\frac{\Gamma \vdash e_0 : \langle\rangle \quad \Gamma \vdash e : \tau}{\Gamma \vdash \texttt{check}(e_0; e) : \tau}$$

The principal argument, $e_0$, should always be evaluated. By the canonical forms lemma, it must be $\langle\rangle$, and hence we may continue by evaluating $e$.

There would be little point in formulating a positive unit type apart from the desire to achieve uniformity among all finite positive products.

# Chapter 11

**11.1**. Follow the example of the Booleans given in Section 11.3.2, which are just a finite enumeration type with two elements.

**11.2**. The option type cannot be simulated in the manner described. Here is a reasonable attempt that corresponds to Hoare's intended practice:

$$\texttt{null} \triangleq \langle \texttt{false}, \texttt{null} \rangle$$
$$\texttt{just}(e) \triangleq \langle \texttt{true}, e \rangle$$
$$\texttt{check}\, e\, \{\texttt{null} \hookrightarrow e_1 \mid \texttt{just}(x) \hookrightarrow e_2\} \triangleq \texttt{if}\, e \cdot \texttt{l}\, \texttt{then}\, [e \cdot \texttt{r}/x]e_2\, \texttt{else}\, e_1$$

The solution makes use of `null` as the "null" inhabitant of type $\tau$. This immediately conflicts with the existence of empty types, such as `void`, that do not have a value. Worse, regardless of the setting of the flag, the second component of the pair is always accessible and may be used in a computation. It is a matter of convention not to do this, but experience shows that, whether by mistake or by malice, it is often used inappropriately. By contrast the option type requires no special "null" value, and precludes the abuses just mentioned.

**11.3**. It would be considerably more flexbile to generalize schemas from product types to, at least, *products of sums* of atomic types. Null values are naturally represented using options, and heterogeneous values are just homogeneous values of a sum type. More generally, one might wish to consider admitting, for example, *nested* databases, in which an attribute of a tuple might be a database itself.

Preview

# Chapter 12

**12.1**. Multi-argument functions and keyword parameter passing are just functions whose domain is a finite product type $\langle \tau_i \rangle_{i \in I}$ for some finite set $I$. When thought of as a multi-argument function, a value of this type typically has the form

$$\lambda\,(x{:}\langle \tau_i \rangle_{i \in I})\,\texttt{match}\,x\,\{\langle x_i \rangle_{i \in I} \hookrightarrow e\},$$

where $x$ itself does not occur in $e$, because the arguments are only referenced individually, never as a whole. In such cases the composite form can be abbreviated to the form

$$\lambda\,(\langle x_i \rangle_{i \in I}{:}\langle \tau_i \rangle_{i \in I})\,e,$$

mimicking the syntax used by many languages.

**12.2**. The highlighted examples suggest that the key decisions are when, and in what order, to evaluate the components of a tuple, and whether to evaluate an expression matched by either a wildcard or a variable. In an attempt to be maximally lazy, remove rule (12.8a), and remove the $e$ val premises from rules (12.8b) to (12.8d), so that the decision when to evaluate is left to rules (12.6) and (12.7) (from which we remove the presupposition that the candidate expression be a value). It is clear that rules (12.6c) to (12.6f) must first evaluate the expression $e$ in order to check whether it matches the pattern $p$. But what should be done about rules (12.6a) and (12.6b)? Because neither pattern makes any demand on the form of $e$, it makes the most sense to just succeed, binding $x$ to $e$ in unevaluated form in the former. On the other hand rules (12.7) must evaluate the expression $e$ in every case in order to ensure that it does not match the given pattern $p$. It is easy to see that the mismatch judgment holds iff the corresponding match judgment fails to hold, so a failing match need not induce useless evaluation just to check for a mismatch.

**12.3**. Any solution must take essentially the following form:

```
match ⟨e1, ⟨e2, e3⟩⟩ {
  ⟨true, ⟨true, _⟩⟩ ↪ true
| ⟨true, ⟨_, true⟩⟩ ↪ true
| ⟨_, ⟨true, true⟩⟩ ↪ true
| _ ↪ false
}.
```

For any *fixed* ordering of the clauses, there is some input for which the output is other incorrect (if, say, the last rule were evaluated first), or else will require evaluation of a component that need not have been evaluated in order to determine the output. Thus "lazy evaluation" does *not* ensure that the least amount of work is performed in order to determine the value of an expression.

Preview

# Chapter 13

**13.1**. Closure under substitution may be shown by structural induction on $\tau'$.

**13.2**. Proceed by induction on the structure of $\tau$. Observe that, for example, if $\tau = \tau_1 \times \tau_2$, then $e$ will be transformed into $\langle e \cdot \mathtt{l}, e \cdot \mathtt{r} \rangle$. But by the canonical forms property for closed values of product type, $e$ will itself be a pair $\langle e_1, e_2 \rangle$, where $e_1$ and $e_2$ are closed values of type $\tau_1$ and $\tau_2$, respectively. Thus $\langle e \cdot \mathtt{l}, e \cdot \mathtt{r} \rangle$ evaluates (under an eager dynamics) to $\langle e_1, e_2 \rangle$, which is $e$ itself. The complication with function types $\tau = \tau_1 \to \tau_2$ is that the transformation will yield $\lambda\,(x{:}\tau_1)\,e(x)$, which is a value that is not identical with $e$, but only interchangeable with it in the sense of Chapter 49.

**13.3**. Let the database schema, $\sigma$, be a finite product type $\prod_{i \in I} \tau_i$, where $\mathtt{first}$ and $\mathtt{last}$ are elements of $I$, and for which $\tau_{\mathtt{first}}$ and $\tau_{\mathtt{last}}$ are both $\mathtt{str}$. Let $\sigma'$ be the finite product type $\prod_{i \in I} \tau'_i$ that agrees with $\sigma$ on each each attribute, except that $\tau'_{\mathtt{first}}$ and $\tau'_{\mathtt{last}}$ are both chosen to be the type variable $t$, indicating the positions of the intended transformation. We have, by construction, that $[\mathtt{str}/t]\sigma'$ is $\sigma$, the database schema.

Let $d$ be a database on the schema $\sigma$, which, according to Solution 10.1, is a value of type

$$\mathtt{nat} \times (\mathtt{nat} \to \sigma).$$

To perform the required transformation, it suffices to use the generic extension of $t.\sigma'$ applied to the capitalization function, $c$, and the database, $d$:

$$\mathtt{map}_{t.\mathtt{nat} \times (\mathtt{nat} \to \sigma')}(x.c(x); d).$$

Keeping in mind Exericse 13.2, we may see at a glance that the size of the database remains fixed, that the only columns that are transformed are those specified by the occurrences of $t$ in $\sigma'$, and that the resulting database replaces the value $v$ of each row at these columns with $c(v)$, as required.

**13.4**. The judgments $t.\tau$ non-neg and $t.\tau$ neg are defined simultaneously on the structure of $\tau$. The key clauses of the definitions are as follows:

$$\frac{}{t.t \text{ non-neg}} \tag{B.10a}$$

$$\frac{t.\tau_1 \text{ neg} \quad t.\tau_2 \text{ non-neg}}{t.\tau_1 \to \tau_2 \text{ non-neg}} \tag{B.10b}$$

$$\frac{t.\tau_1 \text{ non-neg} \quad t.\tau_2 \text{ neg}}{t.\tau_1 \to \tau_2 \text{ neg}} \tag{B.10c}$$

Observe that the parameter of the type operator cannot be judged to occur negatively, and the definitions for function types swaps polarities in the domain, and preserves them in the range. The remaining cases are defined similarly, preserving the polarity in all positions.

It is easy to give a derivation of $t.(t \to \mathtt{bool}) \to \mathtt{bool}$ non-neg according to the above rules.

Preview

**13.5**. The dynamics of the two forms of generic extension are given by the following key rules:

$$\frac{}{\mathtt{map}^{--}[t.t](x.e';e) \longmapsto [e/x]e'} \tag{B.11a}$$

$$\frac{}{\mathtt{map}^{--}[t.\tau_1 \to \tau_2](x.e';e) \longmapsto \lambda\,(x_1{:}\tau_1)\,\mathtt{map}^{--}[t.\tau_2](x.e';e(\mathtt{map}^{-}[t.\tau_1](x.e';x_1)))} \tag{B.11b}$$

$$\frac{}{\mathtt{map}^{-}[t.\tau_1 \to \tau_2](x.e';e) \longmapsto \lambda\,(x_1{:}\tau_1)\,\mathtt{map}^{-}[t.\tau_2](x.e';e(\mathtt{map}^{--}[t.\tau_1](x.e';x_1)))} \tag{B.11c}$$

The non-negative generic extension of the non-negative operator $t.(t \to \mathtt{bool}) \to \mathtt{bool}$ on $x.e'$ sends a function $f$ of type $(\rho \to \mathtt{bool}) \to \mathtt{bool}$ to the function

$$\lambda\,(g{:}\rho' \to \mathtt{bool})\,f(\lambda\,(x{:}\rho)\,g(e'))$$

of type $(\rho' \to \mathtt{bool}) \to \mathtt{bool}$.

# Chapter 14

**14.1**. Define $\mathtt{iter}\,e\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}$ to be the expression

$$\mathtt{rec}_{\mathtt{nat}}(e.y; \mathtt{case}\,y\,\{\mathtt{l} \cdot _ \hookrightarrow e_0 \mid \mathtt{r} \cdot x \hookrightarrow e_1\}).$$

Then check that the dynamics of the iterator given in Chapter 9 is derivable from this definition.

**14.2**. Define $\mathtt{strgen}\,x\,\mathtt{is}\,e\,\mathtt{in}\,\langle\mathtt{hd} \hookrightarrow e_1, \mathtt{tl} \hookrightarrow e_2\rangle$ to be the expression

$$\mathtt{gen}_{\mathtt{stream}}(e.x; \langle e_1, e_2\rangle).$$

Then check that the dynamics, as given in Section 14.1 is derivable from this definition.

**14.3**. The required transformation is given by the function

$$\lambda\,(q{:}\mathtt{seq})\,\mathtt{strgen}\,x\,\mathtt{is}\,\mathtt{z}\,\mathtt{in}\,\langle\mathtt{hd} \hookrightarrow q(x), \mathtt{tl} \hookrightarrow \mathtt{s}(x)\rangle.$$

The $n$th tail of the stream associated to $q$ is the stream

$$\mathtt{strgen}\,x\,\mathtt{is}\,\overline{n}\,\mathtt{in}\,\langle\mathtt{hd} \hookrightarrow q(x), \mathtt{tl} \hookrightarrow \overline{n+1}\rangle.$$

It's head is therefore $q(\overline{n})$, as required. The two transformation are, informally, mutually inverse, showing that `stream` and `seq` are isomorphic types.

Preview

**14.4**. Define the lists as follows:

$$\begin{aligned}
\texttt{natlist} &\triangleq \mu(t.\texttt{unit} + (\texttt{nat} \times t)) \\
\texttt{nil} &\triangleq \texttt{fold}(\texttt{l} \cdot \langle\rangle) \\
\texttt{cons}(e_1; e_2) &\triangleq \texttt{fold}(\texttt{r} \cdot \langle e_1, e_2\rangle) \\
\texttt{rec}\, e\, \{\texttt{nil} \hookrightarrow e_0 \mid \texttt{cons}(x; y) \hookrightarrow e_1\} &\triangleq \texttt{rec}(z.\texttt{case}\, z\, \{\texttt{l} \cdot _ \hookrightarrow e_0 \mid \texttt{r} \cdot u \hookrightarrow [u \cdot \texttt{l}, u \cdot \texttt{r}/x, y]e_1\}; e)
\end{aligned}$$

Then check that the requisite statics and dynamics are derivable under these definitions.

**14.5**. The key rule of the dynamics is the inversion principle given by the rule

$$\frac{}{\texttt{view}(\texttt{itgen}\, x\, \texttt{is}\, e\, \texttt{in}\, e') \longmapsto \texttt{map}_{t.(t\times t)\,\texttt{opt}}(y.\texttt{itgen}\, x\, \texttt{is}\, z\, \texttt{in}\, e'; [e/x]e')} \tag{B.12}$$

The generic extension operation makes it convenient to apply the recursive calls, as necessary, to the result of the state transformation.

The type `itree` may be coinductively defined to be the type $\nu(t.(t \times t)\,\texttt{opt})$. The derivation of the introduction and elimination forms from this definition follows directly from this characterization.

# Chapter 15

**15.1**. We first define $e_{\texttt{eo}}$ of type

$$\tau_{\texttt{eo}} \triangleq \langle \texttt{even} \hookrightarrow (\texttt{nat} \rightharpoonup \texttt{nat}), \texttt{odd} \hookrightarrow (\texttt{nat} \rightharpoonup \texttt{nat})\rangle$$

from which we obtain the desired functions by projection, $e_{\texttt{eo}} \cdot \texttt{even}$ and $e_{\texttt{eo}} \cdot \texttt{odd}$, respectively.

The expression $e_{\texttt{eo}}$ is defined by general recursion to be

$$\texttt{fix}\, \texttt{this} : \tau_{\texttt{eo}}\, \texttt{is}\, \langle \texttt{even} \hookrightarrow e_{\texttt{ev}}, \texttt{odd} \hookrightarrow e_{\texttt{od}}\rangle,$$

where $e_{\texttt{ev}}$ is the expression

$$\lambda\, (\texttt{x:nat})\, \texttt{ifz}\, \texttt{x}\, \{\texttt{z} \hookrightarrow \texttt{s}(\texttt{z}) \mid \texttt{s}(\texttt{y}) \hookrightarrow \texttt{this} \cdot \texttt{odd}(\texttt{y})\},$$

and $e_{\texttt{od}}$ is the expression

$$\lambda\, (\texttt{x:nat})\, \texttt{ifz}\, \texttt{x}\, \{\texttt{z} \hookrightarrow \texttt{z} \mid \texttt{s}(\texttt{y}) \hookrightarrow \texttt{this} \cdot \texttt{even}(\texttt{y})\}.$$

Preview

**15.2**. Using the general fixed point operator, define a search function that repeatedly tests $\phi(m,n)$ on successive values of $m$, starting with zero, until either $\phi(m,n)$ evaluates to zero. The resulting computation diverges if no such $m$ exists.

**15.3**. Suppose that $e_{halts}$ were a definition of $\phi_{halts}$ in **PCF**. Define $e_{diag}$ to be the function

$$\lambda\,(n{:}\mathtt{nat})\;\mathtt{ifz}\,e_{halts}(n)\,\{\mathtt{z} \hookrightarrow e_{diverge} \mid \mathtt{s}(_) \hookrightarrow \mathtt{z}\},$$

where $e_{diverge}$ diverges always. Let $d$ be $\ulcorner e_{diag} \urcorner$, the Gödel-number of $e_{diag}$. By the assumption either $e_{halts}(d)$ evaluates to zero or to one. If the former, then by the definition of $e_{diag}$ we have that $e_{diag}(d)$ converges, which means that $e_{halts}(d)$ evaluates to one, which means that $e_{diag}(d)$ diverges, a contradiction. If the latter, then $e_{diag}(d)$ diverges, which by the definition of $e_{diag}$ means that $e_{halts}(d)$ evaluates to zero, which means that $e_{diag}(d)$ converges, also a contradiction. Therefore $\phi_{halts}$ is not definable in **PCF**.

**15.4**. The difficulty is that $\phi$ might be undefined for certain values of $m$ prior to the first one for whcih $\phi(m,n)$ is zero. The search process described in Solution **15.2** would diverge prior to finding the first zero of $\phi$, violating the specification. One can show that this form of minimization, if definable, could be used to solve the halting problem, in contradiction to Solution **15.3**. To see this consider the function $\phi_{step}(m)(\ulcorner e \urcorner)$, which converges iff $e(\overline{\ulcorner e \urcorner})$ converges in fewer than $m$ steps, and diverges otherwise.

**15.5**. The "parallel or" function is not definable in **PCF**, yet is, intuitively, computable. The problem is that the dynamics of **PCF** is *sequential* in the sense that it evaluates the arguments of a two-argument function in a definite order, first committing to one, then to the other. This precludes defining the function described in the exercise. One solution is to enrich **PCF** with a form of parallelism, called *dovetailing*, that interleaves the evaluation of two expressions, returning the result of the first to converge (or, say, the leftmost one if both converge simultaneously).

**15.6**. First, define a notion of Gödel-numbering that respects $\alpha$-equivalence, for example by using de Bruijn indices as described in Solution **1.3**. Second, provide operations that allow one to build (the Gödel numbers of) expressions from (the Gödel numbers of) their components, and to decompose (the Gödel numbers of) expressions into (the Gödel numbers of) their components. Third, define the universal function in terms of these primitives, using general recursion to define the interpretation of general recursion, and functions to define the interpretation of functions, and so forth. Such a function is called a *metacircular interpreter* because it defines the interpretation of the constructs of a language in terms of those constructs themselves!

# Chapter 16

Preview

**16.1**. The following definitions suffice:

$$\begin{aligned}
\mathtt{k} &\triangleq \mathtt{fold}(\lambda\,(x{:}\mathtt{D})\,\mathtt{fold}(\lambda\,(y{:}\mathtt{D})\,x))\\
\mathtt{s} &\triangleq \mathtt{fold}(\lambda\,(x{:}\mathtt{D})\,\mathtt{fold}(\lambda\,(y{:}\mathtt{D})\,\mathtt{fold}(\lambda\,(z{:}\mathtt{D})\,(x\cdot z)\cdot(y\cdot z))))\\
x\cdot y &\triangleq (\mathtt{unfold}(x))(y).
\end{aligned}$$

Surprisingly, this structure is sufficient to represent every partial computable function on the natural numbers as an element of $\mathtt{D}$!

**16.2**. Let $\rho$ be the result type of the recursor and the state type of the generator in the chart below:

$$\begin{aligned}
\mathtt{fold}_{t.\tau'}(e) &\triangleq \mathtt{fold}(e)\\
\mathtt{rec}(x.e';e) &\triangleq (\mathtt{fix}\,r\,\mathtt{is}\,\lambda\,(u{:}\tau)\,e_{rec})(e),\ \text{where}\\
e_{rec} &\triangleq [\mathtt{map}_{t.\tau'}(x.r(x);\mathtt{unfold}(u))/x]e'\\
\mathtt{unfold}_{t.\tau'}(e) &\triangleq \mathtt{unfold}(e)\\
\mathtt{gen}(x.e';e) &\triangleq (\mathtt{fix}\,g\,\mathtt{is}\,\lambda\,(u{:}\rho)\,e_{gen})(e),\ \text{where}\\
e_{gen} &\triangleq \mathtt{fold}(\mathtt{map}_{t.\tau'}(x.g(x);[u/x]e'))
\end{aligned}$$

The dual symmetry of the definitions is striking. Check that the statics of the recursor and generator are derivable under these definitions.

However, the dynamics of these operations is ill-behaved. Under an eager interpretation the generator may not converge, depending on the choice of $e'$, and under a lazy interpretation the recursor may not converge, again depending on the choice of $e'$. These outcomes are a reflection of the fact that in the eager case the recursive type is inductive, not coinductive, whereas in the lazy case the recursive type is coinductive, not inductive.

**16.3**. Under a lazy dynamics we may define the type $\mathtt{signal}$ of signals to be the recursive type $\nu(t.\mathtt{bool}\times t)$, whereas under an eager dynamics one may use instead $\nu(t.\mathtt{unit}\rightharpoonup\mathtt{bool}\times t)$. For simplicity, assume a lazy dynamics. A NOR gate may be defined as the function of type $(\mathtt{signal}\times\mathtt{signal})\rightharpoonup\mathtt{signal}$ given by

$$\lambda\,(ab)\,\mathtt{gen}(\langle a,b\rangle.\langle e_{nor}(\langle\mathtt{hd}(a),\mathtt{hd}(b)\rangle),\langle\mathtt{tl}(a),\mathtt{tl}(b)\rangle\rangle;ab),$$

wherein we have used the definition of the generator from Exercise **16.2**. The internal state of the gate consists of the two input signals, $a$ and $b$. Whenever an output is required, the heads of $a$ and $b$ are nor'd together, and the new state consists of the tails of $a$ and $b$, one bit from each having been consumed.

Using the NOR gate just defined, one may then use general recursion to define an RS latch by "cross-feeding" the outputs of each of the NOR gates back into one of the inputs of the other. The remaining inputs are then the $r$ and $s$ signals that reset and set the latch, respectively.

**16.4**. Define the internal state type, $\rho$, of the stream to be the type

$$\langle \mathtt{X}\hookrightarrow\mathtt{bool},\mathtt{Q}\hookrightarrow\mathtt{bool}\rangle.$$



Preview

Define an RS latch with internal state of this type as follows:

$$e_{rsl} \triangleq \mathtt{gen}(\langle \mathtt{X} \hookrightarrow x, \mathtt{Q} \hookrightarrow q \rangle . e'_{rsl}; \langle \mathtt{X} \hookrightarrow \mathtt{false}, \mathtt{Q} \hookrightarrow \mathtt{false} \rangle),$$

where

$$e'_{rsl} \triangleq \langle \mathtt{X} \hookrightarrow x, \mathtt{Q} \hookrightarrow q, \mathtt{N} \hookrightarrow \langle \mathtt{X} \hookrightarrow e_{nor}(\langle s, q \rangle), \mathtt{Q} \hookrightarrow e_{nor}(\langle x, r \rangle) \rangle \rangle.$$

The state is arbitrarily initialized with both `X` and `Q` being `false`, and these are the initial outputs of the latch. Then, the next state of the latch is computed using $e_{nor}$ applied to the fixed $r$ and $s$ values and to the current $x$ and $q$ values.

When the generator is expanded according to Solution **16.2**, and the result simplified for clarity, we obtain the following formulation of $e_{rsl}$:

$$\mathtt{fix}\, g\, \mathtt{is}\, \lambda\, (\langle \mathtt{X} \hookrightarrow x, \mathtt{Q} \hookrightarrow q \rangle)\, \mathtt{fold}(e''_{rsl}),$$

where $e''_{rsl}$ is given by

$$\langle \mathtt{X} \hookrightarrow x, \mathtt{Q} \hookrightarrow q, \mathtt{N} \hookrightarrow g(\langle \mathtt{X} \hookrightarrow e_{nor}(\langle s, q \rangle), \mathtt{Q} \hookrightarrow e_{nor}(\langle x, r \rangle) \rangle) \rangle.$$

Notice that the state is maintained by the recursive self-reference, much as in Section 16.4.

# Chapter 17

**17.1**. The encoding of finite products is given by the following equations:

$$\begin{aligned} \langle\rangle &\triangleq \lambda\, (x)\, x \\ \langle u_1, u_2 \rangle &\triangleq \lambda\, (u)\, u(u_1)(u_2) \\ u \cdot \mathtt{l} &\triangleq u((\lambda\, (x)\, \lambda\, (y)\, x)) \\ u \cdot \mathtt{r} &\triangleq u((\lambda\, (x)\, \lambda\, (y)\, y)) \end{aligned}$$

**17.2**. Using only primitive recursion as provided by the Church numerals:

$$\lambda\, (x)\, x(\overline{1})(\lambda\, (y)\, (\mathtt{times}(x)(y))),$$

where `times` is the Church encoding of multiplication. Using the fixed point combinator we may define factorial as follows:

$$\mathtt{Y}(\lambda\, (f)\, \lambda\, (x)\, x(\overline{1})(f(\mathtt{pred}(x)))).$$

The required equations may be proved by induction on $n$.

Preview

**17.3**. As the specification implies,

$$\begin{aligned}\texttt{true} &\triangleq \lambda\,(x)\,\lambda\,(y)\,x\\ \texttt{false} &\triangleq \lambda\,(x)\,\lambda\,(y)\,y\\ \texttt{if}\,u\,\texttt{then}\,u_1\,\texttt{else}\,u_2 &\triangleq u(u_1)(u_2)\end{aligned}$$

**17.4**. Define sums along similar lines as follows:

$$\begin{aligned}\texttt{l}\cdot u &\triangleq \lambda\,(l)\,\lambda\,(r)\,l(u)\\ \texttt{r}\cdot u &\triangleq \lambda\,(l)\,\lambda\,(r)\,r(u)\\ \texttt{case}\,u\,\{\texttt{l}\cdot x_1 \hookrightarrow u_1 \mid \texttt{r}\cdot x_2 \hookrightarrow u_2\} &\triangleq u(\lambda\,(x_1)\,u_1)(\lambda\,(x_2)\,u_2)\end{aligned}$$

The booleans are a special case in which $u_1$ and $u_2$ are always $\lambda\,(x)\,x$.

**17.5**. Define lists by their eliminatory form as follows:

$$\begin{aligned}\texttt{nil} &\triangleq \lambda\,(n)\,\lambda\,(c)\,n\\ \texttt{cons}(u_1;u_2) &\triangleq \lambda\,(n)\,\lambda\,(c)\,c(u_1)(u_2)\\ \texttt{rec}\,u\,\{\texttt{nil} \hookrightarrow u_0 \mid \texttt{cons}(x_1;x_2) \hookrightarrow u_1\} &\triangleq u(u_0)(\lambda\,(x_1)\,\lambda\,(x_2)\,u_1).\end{aligned}$$

**17.6**. Dually to lists, define streams by their introductory form:

$$\begin{aligned}\texttt{strgen}\,u\,\texttt{is}\,x\,\texttt{in}\,\langle\texttt{hd} \hookrightarrow u_1,\texttt{tl} \hookrightarrow u_2\rangle &\triangleq (\lambda\,(x)\,\langle u_1,u_2\rangle)(u)\\ \texttt{hd}(u) &\triangleq u\cdot\texttt{l}\\ \texttt{tl}(u) &\triangleq (\lambda\,(x)\,\langle u_1,u_2\rangle)(u\cdot\texttt{r}).\end{aligned}$$

The encoding relies on binary products as defined in Solution **17.1**.

**17.7**. The translation is given as follows:

$$\begin{aligned}x^* &\triangleq x\\ (u_1(u_2))^* &\triangleq \texttt{ap}(u_1^*;u_2^*)\\ (\lambda\,(x)\,u)^* &\triangleq [x]\,u^*\end{aligned}$$

Compositionality of the translation follows directly from Solution **3.5**.

Then proceed by induction on rules (17.2). The only difficult case is rule (17.2f), stating that

$$(\lambda\,(x)\,u_1)(u_2) \equiv [u_2/x]u_1.$$

Preview

This is handled as follows:

$$\begin{aligned}((\lambda\,(x)\,u)(u_2))^* &= (\lambda\,(x)\,u)^*(u_2^*)\\ &= ([x]\,u^*)(u_2^*)\\ &\equiv [u_2^*/x]u^*\\ &= ([u_2/x]u)^*.\end{aligned}$$

The second-to-last equation is Exercise 3.4.

# Chapter 18

**18.1**. Here is one possible definition of `plus` in **DPCF**:

$$\lambda\,(a)\,\texttt{fix}\,p\,\texttt{is}\,\lambda\,(b)\,\texttt{ifz}\,b\,\{\texttt{zero} \hookrightarrow a \mid \texttt{succ}(x) \hookrightarrow \texttt{succ}(p(x))\}.$$

Examine the transition sequence $\texttt{plus}(\overline{5})(\overline{7})$ carefully, and observe these points:

(a) Each recursive call to `plus` requires a run-time check to ensure that it is, in fact, a function, even though it cannot fail to be so because of the definiton of `plus`.

(b) Each iteration requires examination and removal of the numeric tag from the argument $b$ to determine whether or not it is zero.

(c) Each iteration but the last involves computation of the successor, which requires checking, removing, and reattaching the numeric tag from its argument.

None of this takes place in the dynamics of **PCF** for the analogous definition of addition.

**18.2**. Follow the pattern for the natural numbers given in Section 18.1, but with `nil` and `cons` forming separate classes of values. It is not clear whether `cons` should impose any class restrictions on its arguments; conventionally, it does not. The behavior of `append` suffers from similar deficiencies to those outlined in Solution 18.1, for largely the same reasons.

**18.3**. Follow the same pattern as the treatment of the class of numbers in **DPCF**, with `nil` and `cons` playing the roles of zero and successor, respectively. In this case `cons` should require its second argument to be of the class `list`, which introduces additional overhead to the dynamics of `append`. Appending a list to a non-list will result in run-time failure at the last step, after the first list has been traversed. Using dynamic classes to express static invariants is a losing proposition.

**18.4**. Consideration of multiple arguments and multiple results amounts to an admission that more than one type is necessary in a practical language. Many issues arise, with no fully satisfactory solutions.

Preview

(a) There is no way to express the restriction to a particular number of arguments in a dynamic language, which has only one type. At bottom the arguments must be considered to be a tuple whose components are accessed by projections that may fail at run-time if the call site provides too few arguments. What happems with too many arguments is highly dependent on the implementation of tuples and projections.

(b) Multi-argument functions are often "optimized" by means that amount to a very special case of pattern matching. Match failure can still occur at run-time, rather than be caught at compile time. The core issue is not efficiency, but rather expressiveness—precise expression of invariants admits efficient implementation, but no amount of implementation tricks can make up for lack of expressive power.

(c) The arguments must be processed as a list of unbounded size. Access to the arguments requires a recursive traversal of this list. Many languages provide *ad hoc* forms of pattern matching to, say, allow one to name the first $k \geq 0$ elements of this list. Argument mismatch fails at run-time.

(d) Keyword parameter passing is simply a mode of use of pattern matching for labelled product types as described in Exercise 12.2. Here again in a dynamic setting any mismatches would result in run-time, rather than compile-time, failures.

(e) Multiple results are notoriously problematic. In a dynamic setting it is a matter of indifference whether the number of results is fixed or varying because the only choice is to return a value whose structure is determined dynamically. The caller must know the structure of the result, and arrange to access its parts by an unstated convention. Sometimes special syntax is introduced to cover common cases such as finite tuples of results, but without static typing it remains error-prone and difficult to remember the prevalent convention.

(f) Defining composition in such a setting is a mess, because there is no way to express the "impedance matching" restrictions that ensure that the output of one function matches the input of the other.

Needless to say, none of these issues arise in a statically typed language with pattern matching. Problems with multiple arguments and multiple results in dynamic languages are prime examples of iatrogenic language disorders.

# Chapter 19

**19.1**. Add two new classes, `nil` and `cons`, and extend the statics **HPCF** as follows:

$$\frac{\Gamma \vdash e : \mathtt{unit}}{\Gamma \vdash \mathtt{new}[\mathtt{nil}](e) : \mathtt{dyn}} \tag{B.13a}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn} \times \mathtt{dyn}}{\Gamma \vdash \mathtt{new}[\mathtt{cons}](e) : \mathtt{dyn}} \tag{B.13b}$$



Preview

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{nil}](e) : \mathtt{unit}} \tag{B.13c}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{cons}](e) : \mathtt{dyn} \times \mathtt{dyn}} \tag{B.13d}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{inst}[\mathtt{nil}](e) : \mathtt{bool}} \tag{B.13e}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{inst}[\mathtt{cons}](e) : \mathtt{bool}} \tag{B.13f}$$

The dynamics may be extended by following the pattern in Chapter 19.

Using these extensions we may make the following definitions of the null and pairing primitives of **DPCF**:

$$\begin{aligned}
\mathtt{nil}^\dagger &\triangleq \mathtt{nil}\,!\,\langle\rangle \\
\mathtt{cons}(d_1;d_2)^\dagger &\triangleq \mathtt{cons}\,!\,\langle d_1^\dagger, d_2^\dagger\rangle \\
\mathtt{car}(d)^\dagger &\triangleq (d^\dagger \mathbin{@} \mathtt{cons}) \cdot \mathtt{l} \\
\mathtt{cdr}(d)^\dagger &\triangleq (d^\dagger \mathbin{@} \mathtt{cons}) \cdot \mathtt{r} \\
\mathtt{nil?}(d)^\dagger &\triangleq \mathtt{if}\,\mathtt{nil}\,?\,d^\dagger\,\mathtt{then}\,\mathtt{cons}(\mathtt{nil};\mathtt{nil})^\dagger\,\mathtt{else}\,\mathtt{nil}^\dagger \\
\mathtt{cons?}(d)^\dagger &\triangleq \mathtt{if}\,\mathtt{cons}\,?\,d^\dagger\,\mathtt{then}\,\mathtt{cons}(\mathtt{nil};\mathtt{nil})^\dagger\,\mathtt{else}\,\mathtt{nil}^\dagger \\
\mathtt{cond}(d;d_0;d_1) &\triangleq \mathtt{if}\,\mathtt{nil}\,?\,d\,\mathtt{then}\,d_1\,\mathtt{else}\,d_0
\end{aligned}$$

The choice of value $\mathtt{cons}(\mathtt{nil};\mathtt{nil})^\dagger$ is arbitrary; it can be anything other than $\mathtt{nil}^\dagger$.

**19.2**. Define dyn in **FPC** to be the recursive type

$$\mathtt{rec}\,t\,\mathtt{is}\,[\mathtt{num} \hookrightarrow \mathtt{nat}, \mathtt{fun} \hookrightarrow t \rightharpoonup t, \mathtt{nil} \hookrightarrow \mathtt{unit}, \mathtt{cons} \hookrightarrow t \times t].$$

The null and pairing primitives given in Section 18.2 may then be defined directly in **FPC**. For example, $\mathtt{cond}(d;d_0;d_1)$ may be defined as

$$\mathtt{case}\,\mathtt{unfold}(d)\,\{\mathtt{nil} \cdot _ \hookrightarrow d_0 \mid \mathtt{cons} \cdot _ \hookrightarrow d_1 \mid \mathtt{num} \cdot _ \hookrightarrow d_1 \mid \mathtt{fun} \cdot _ \hookrightarrow d_1\}.$$

It is apparent from this definition that $\mathtt{cond}(d;d_0;d_1)$ throws away useful information in dispatching to either $d_0$ or $d_1$ without passing any other information to either branch.

**19.3**. The translation of the append function of **DPCF** into **HPCF** is as follows:

$$\mathtt{fix}\,a : \mathtt{dyn}\,\mathtt{is}\,\mathtt{fun}\,!\,\lambda\,(x{:}\mathtt{dyn})\,\mathtt{fun}\,!\,\lambda\,(y{:}\mathtt{dyn})\,e_{a,x,y},$$

where

$$a : \mathtt{dyn}, x : \mathtt{dyn}, y : \mathtt{dyn} \vdash e_{a,x,y} : \mathtt{dyn}$$



Preview

is the expression

$$\texttt{if nil ? } x \texttt{ then cons ! } \langle (x \texttt{ @ cons}) \cdot \texttt{l}, e'_{a,x,y} \rangle \texttt{ else } y,$$

and where

$$a : \texttt{dyn}, x : \texttt{dyn}, y : \texttt{dyn} \vdash e'_{a,x,y} : \texttt{dyn}$$

is the expression

$$(a \texttt{ @ fun})((x \texttt{ @ cons}) \cdot \texttt{r}).$$

This code may be optimized by a process similar to that outlined in Section 19.3.

# Chapter 20

**20.1**. Define `bool` to be the type $\forall(t.t \to t \to t)$. Then define the introduction and elimination forms as follows:

$$\begin{aligned} \texttt{true} &\triangleq \Lambda(t.\lambda\,(x{:}t)\,\lambda\,(y{:}t)\,x) \\ \texttt{false} &\triangleq \Lambda(t.\lambda\,(x{:}t)\,\lambda\,(y{:}t)\,y) \\ \texttt{if}\,e\,\texttt{then}\,e_0\,\texttt{else}\,e_1 &\triangleq e[\rho](e_0)(e_1), \end{aligned}$$

where $\rho$ is the result type of the conditional. Check that the statics and dynamics of these operations are derivable according to these definitions.

**20.2**. The type `natlist` may be defined in **F** as follows:

$$\forall(t.t \to (\texttt{nat} \to t \to t) \to t).$$

The introduction and elimination forms may then be defined as follows:

$$\begin{aligned} \texttt{nil} &\triangleq \Lambda(t.\lambda\,(n{:}t)\,\lambda\,(c{:}\texttt{nat} \to t \to t)\,n) \\ \texttt{cons}(e_h; e_t) &\triangleq \Lambda(t.\lambda\,(n{:}t)\,\lambda\,(c{:}\texttt{nat} \to t \to t)\,c(e_h)(e_t(n)(c))) \\ \texttt{rec}\,e\,\{\texttt{nil} \hookrightarrow e_0 \mid \texttt{cons}(x;y) \hookrightarrow e_1\} &\triangleq e[\rho](e_0)((\lambda\,(x{:}\rho)\,\lambda\,(y{:}\texttt{nat} \to \rho \to \rho)\,e_1)). \end{aligned}$$

Check that the statics and dynamics are derivable according to these definitions.

**20.3**. The inductive type $\mu(t.\tau)$, where $t.\tau$ pos, may be defined in **F** by the equation

$$\mu(t.\tau) \triangleq \forall(t.(\tau \to t) \to t).$$

The introduction and elimination forms are defined as follows:

$$\begin{aligned} \texttt{fold}(e) &\triangleq \Lambda(t.\lambda\,(f{:}\tau \to t)\,f(\texttt{map}_{t.\tau}(y.y[t](f);e))) \\ \texttt{rec}(x.e';e) &\triangleq e[\rho](\lambda\,(x{:}\tau)\,e'), \end{aligned}$$

wherein $\rho$ is the result type of the recursor. One may check that the statics and dynamics are derivable from these definitions. It is very instructive to check that this definition essentially coincides with the definition of the natural numbers given in Chapter 20 under the identification of `nat` with $\mu(t.\texttt{unit} + t)$, and the definition of sum types within **F**.

Preview

**20.4**. Fix $\tau$ and $l_0 : \tau\,\mathtt{list}$. Define $\mathcal{P}_{l_0}$ to hold of $z : \tau$ iff $z$ is among the elements of $l_0$. By parametricity the function $f$ must preserve $\mathcal{P}$, which means that if its input has elements among those of $l_0$, then so must its output. But $l_0$ has just such elements, so $f[\tau](l_0)$ must have elements among those of $l_0$ as well. Thus, among other things, $f$ could be the constantly `nil` function, or the list reversal function, or the function that drops every other element of its input. But it cannot, for example, transform the elements of its input in any way.

**20.5**. Even to compute one natural number from another requires that the polymorphic type variable be instantiated to the polymorphic type itself, which requires impredicativity.

# Chapter 21

**21.1**. Define the type `stream` as the following existential type:

$$\mathtt{stream} \triangleq \exists(t.(t \to (\mathtt{nat} \times t)) \times t).$$

The introduction and elimination forms for streams are defined as follows:

$$\begin{aligned}
&\mathtt{strgen}\,x\,\mathtt{is}\,e\,\mathtt{in}\,{<}\mathtt{hd} \hookrightarrow e_0, \mathtt{tl} \hookrightarrow e_1{>} \triangleq \\
&\quad \mathtt{pack}\,\tau\,\mathtt{with}\,\langle \lambda\,(x{:}\tau)\,\langle e_0, e_1\rangle, e\rangle\,\mathtt{as}\,\mathtt{stream} \\
&\mathtt{hd}(e) \triangleq \mathtt{open}\,e\,\mathtt{as}\,t\,\mathtt{with}\,\langle f : t \to (\mathtt{nat} \times t), x : t\rangle\,\mathtt{in}\,f(x) \cdot \mathtt{l} \\
&\mathtt{tl}(e) \triangleq \mathtt{open}\,e\,\mathtt{as}\,t\,\mathtt{with}\,\langle f : t \to (\mathtt{nat} \times t), x : t\rangle\,\mathtt{in}\,e', \text{ where} \\
&\quad e' \triangleq \mathtt{strgen}\,x\,\mathtt{is}\,f(x) \cdot \mathtt{r}\,\mathtt{in}\,{<}\mathtt{hd} \hookrightarrow e_0, \mathtt{tl} \hookrightarrow e_1{>}
\end{aligned}$$

**21.2**. The coinductive type $\nu(t.\tau)$, where $t.\tau$ pos, may be defined in **FE** by the type

$$\exists(t.(t \to \tau) \times t).$$

The associated introduction and elimination forms may be defined as follows:

$$\begin{aligned}
\mathtt{gen}(x.e'; e) &\triangleq \mathtt{pack}\,\sigma\,\mathtt{with}\,\langle \lambda\,(x{:}t)\,e', e\rangle\,\mathtt{as}\,\nu(t.\tau) \\
\mathtt{unfold}(e) &\triangleq \mathtt{open}\,e\,\mathtt{as}\,t\,\mathtt{with}\,\langle g : t \to \tau, x : t\rangle\,\mathtt{in}\ldots,
\end{aligned}$$

wherein $\sigma$ is the type of the internal state of the generator. One may check that Solution **21.1** is a special case of this representation under the identification of `stream` with the coinductive type $\nu(t.\mathtt{nat} \times t)$. It is fascinating, if a bit unnerving, to expand the definition of the existential type, and its associated operations, to obtain a representation of coninductive types in **F**.

**21.3**. Recalling the definition of $\exists(t.\tau)$ in **FE** as the type $\forall(u.\forall(t.\tau \to u) \to u)$ in **F**, the abstract type of queues becomes the polymorphic type

$$\forall(u.\forall(t.\tau_{queue} \to u) \to u),$$



Preview

where $\tau_{queue}$ is the type

$$\langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow \mathtt{nat} \times t \to t, \mathtt{rem} \hookrightarrow t \to (\mathtt{nat} \times t)\,\mathtt{opt} \rangle.$$

For any choice, $\rho$, of result type, the client of the abstraction is therefore of polymorphic type $\forall(t.\tau_{queue} \to \rho)$, where $\rho$ is fixed. Now spell out the relational interpretation of this quantified type with the binary relation, $R$, given in Section 21.4, and check that it coincides with the conditions given there. Assigning the identity relation to $\rho$ yields the desired result that the two implementations of queues are observably indistinguishable in **FE**.

# Chapter 23

**23.1**. Neither, it is invariant. Formally, the variance rules for function types imply that in the first component $\tau$ is covariant, yet in the second component $\tau$ is contravariant. It cannot be both, so it is neither. If the composite type were deemed covariant in $\tau$, then one could construct a counterexample to type safety by exploiting the second component, and, conversely, if the composite were deemed contravariant in $\tau$, then one could construct a counterexample to safety by exploiting the first component.

**23.2**. Attempting to show that $\rho_2 <: \rho_1$ requires showing that

$$t_2 <: t_1 \vdash \langle \mathtt{eq} \hookrightarrow (t_2 \rightharpoonup \mathtt{bool}), \mathtt{f} \hookrightarrow \mathtt{bool} \rangle <: \langle \mathtt{eq} \hookrightarrow (t_1 \rightharpoonup \mathtt{bool}) \rangle.$$

But this requires showing that $t_2 \rightharpoonup \mathtt{bool} <: t_1 \rightharpoonup \mathtt{bool}$, which requires showing that $t_1 <: t_2$, which is the opposite of what we have assumed. This suggests that the suggested subtyping relation is false. Suppose that $\rho_2 <: \rho_1$, for the sake of a contradiction. This means that if $e : \rho_2$, then $e : \rho_1$ by subsumption, and hence that $\mathtt{unfold}(e) \cdot \mathtt{eq} : \rho_1 \rightharpoonup \mathtt{bool}$. Now choose $e : \rho_2$ to be

$$\mathtt{fold}(\langle \mathtt{eq} \hookrightarrow (\lambda\,(x{:}\rho_2)\,x \cdot \mathtt{f}), \mathtt{f} \hookrightarrow \mathtt{true} \rangle).$$

Consider the application

$$(\mathtt{unfold}(e) \cdot \mathtt{eq})(e_1) : \mathtt{bool},$$

where

$$e_1 \triangleq \mathtt{fold}(\langle \mathtt{eq} \hookrightarrow \lambda\,(_{:}\rho_1)\,\mathtt{true} \rangle).$$

The expression $e_1$ is chosen to not contain an `f` field; the choice of `eq` field is immaterial and can be any function of type $\rho_1 \rightharpoonup \mathtt{bool}$. Evaluation of the application "gets stuck" attempting to access the `f` component of $e_1$, a violation of type safety. Thus the assumed subtyping $\rho_2 <: \rho_1$ cannot be valid.

The relevance of this example is that it is sometimes thought that if one extends a tuple with a new field, then the type of the extension is always a subtype of the type being extended. But this fails when the tuples are of recursive type, and hence can be self-referential. Extant programming languages nevertheless postulate the incorrect subtyping relationship, and are consequently not type safe.

Preview

**23.3**. One may give an inductive definition of the judgment $\chi : \tau <: \tau'$ specifying that $\chi$ witnesses that $\tau$ is a subtype of $\tau'$. The following incomplete set of rules illustrate the main ideas:

$$\frac{}{\lambda\,(x{:}\tau)\,x : \tau <: \tau} \tag{B.14a}$$

$$\frac{\chi_1 : \tau_1 <: \tau_2 \quad \chi_2 : \tau_2 <: \tau_3}{\lambda\,(x_1{:}\tau_1)\,\chi_2(\chi_1(x_1)) : \tau_1 <: \tau_3} \tag{B.14b}$$

$$\frac{J \subseteq I}{\lambda\,(x{:}\textstyle\prod_{i\in I}\tau_i)\,\langle x\cdot j\rangle_{j\in J} : \prod_{i\in I}\tau_i <: \prod_{j\in J}\tau_j} \tag{B.14c}$$

Reflexivity is witnessed by the identity function, transitivity by function composition, and width subtyping by creating a narrower tuple on passage to a supertype of a tuple type. To complete the definition fill in the rules that express the variance principles for product, sum, and function types, and check that the coercion is indeed an expression of function type mapping the subtype to the supertype.

The full coercion interpretation of product subtyping is coherent, but the proof requires an extension of the methods developed in Chapter 49 and will not be given here.

# Chapter 24

**24.1**. Suppose that $\phi_1 \leq \phi_1'$ and $\phi_2 \leq \phi_2'$. We are to show that $\phi_1 \wedge \phi_2 \leq \phi_1' \wedge \phi_2'$. By rule (24.2f) it suffices to show that $\phi_1 \wedge \phi_2 \leq \phi_1'$ and that $\phi_1 \wedge \phi_2 \leq \phi_2'$. By rules (24.2d) and (24.2e) we have $\phi_1 \wedge \phi_2 \leq \phi_1$ and $\phi_1 \wedge \phi_2 \leq \phi_2$, respectively. But then by the assumptions and transitivity (rule (24.2b)), the result follows.

**24.2**. The forward direction is an immediate consequence of transivity of entailment (rule (24.2b)). Suppose that $\phi'' \leq \phi$ implies $\phi'' \leq \phi'$ for every $\phi''$. In particular we may take $\phi''$ to be $\phi$, since by reflexivity of entailment (rule (24.2a)) we have $\phi \leq \phi$. But then $\phi \leq \phi'$, as desired. An alternative to Solution **24.1** making use of the Yoneda Lemma is as follows. Assuming $\phi_1 \leq \phi_1'$ and $\phi_2 \leq \phi_2'$, let us further assume that $\phi \leq \phi_1 \wedge \phi_2$. Then $\phi \leq \phi_1$ and $\phi \leq \phi_2$ by transitivity, and hence $\phi \leq \phi_1'$ and $\phi \leq \phi_2'$ by the assumptions, so $\phi \leq \phi_1' \wedge \phi_2'$, from which the result follows by the Yoneda Lemma.

**24.3**. The refinement $\mathtt{fold}(\phi)$ is defined to refine a recursive type by the rule

$$\frac{\phi \sqsubseteq [\mathtt{rec}\,t\,\mathtt{is}\,\tau/t]\tau}{\mathtt{fold}(\phi) \sqsubseteq \mathtt{rec}\,t\,\mathtt{is}\,\tau} \tag{B.15}$$

Satisfaction of this refinement may be defined by the rules

$$\frac{\Phi \vdash e \in_{[\mathtt{rec}\,t\,\mathtt{is}\,\tau/t]\tau} \phi}{\Phi \vdash \mathtt{fold}(e) \in_{\mathtt{rec}\,t\,\mathtt{is}\,\tau} \mathtt{fold}(\phi)} \tag{B.16a}$$



Preview

$$\frac{\Phi \vdash e \in_{[\texttt{rec}\, t\, \texttt{is}\, \tau / t]\tau} \texttt{fold}(\phi)}{\Phi \vdash \texttt{unfold}(e) \in_{\texttt{rec}\, t\, \texttt{is}\, \tau} \phi} \tag{B.16b}$$

**24.4**. Summand refinements may be used to improve the expressiveness of type refinements by extending the satisfaction rules for refinements.

(a) Summand refinements are stronger than general sum refinements, are contradictory with one another, and are covariant:

$$\frac{}{\texttt{l} \cdot \phi_1 \leq \phi_1 + \phi_2} \tag{B.17a}$$

$$\frac{}{\texttt{r} \cdot \phi_2 \leq \phi_1 + \phi_2} \tag{B.17b}$$

$$\frac{}{\texttt{l} \cdot \phi_1 \wedge \texttt{r} \cdot \phi_2 \leq \phi} \tag{B.17c}$$

$$\frac{\phi_1 \leq \phi_1'}{\texttt{l} \cdot \phi_1 \leq \texttt{l} \cdot \phi_1'} \tag{B.17d}$$

$$\frac{\phi_2 \leq \phi_2'}{\texttt{r} \cdot \phi_2 \leq \texttt{l} \cdot \phi_2'} \tag{B.17e}$$

(b) The introductory forms may be given summand refinements:

$$\frac{\Phi \vdash e_1 \in_{\tau_1} \phi_1}{\Phi \vdash \texttt{l} \cdot e_1 \in_{\tau_1 + \tau_2} \texttt{l} \cdot \phi_1} \tag{B.18a}$$

$$\frac{\Phi \vdash e_2 \in_{\tau_2} \phi_2}{\Phi \vdash \texttt{r} \cdot e_2 \in_{\tau_1 + \tau_2} \texttt{r} \cdot \phi_2} \tag{B.18b}$$

(c) Unreachable branches of `case` may be ignored:

$$\frac{\Phi \vdash e \in_{\tau_1 + \tau_2} \texttt{l} \cdot \phi_1 \quad \Phi, x_1 \in_{\tau_1} \phi_1 \vdash e_1 \in_{\tau} \phi}{\Phi \vdash \texttt{case}\, e\, \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \in_{\tau} \phi} \tag{B.19a}$$

$$\frac{\Phi \vdash e \in_{\tau_1 + \tau_2} \texttt{r} \cdot \phi_2 \quad \Phi, x_2 \in_{\tau_2} \phi_2 \vdash e_2 \in_{\tau} \phi}{\Phi \vdash \texttt{case}\, e\, \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \in_{\tau} \phi} \tag{B.19b}$$

(d) The learned information may be propagated by adding the hypothesis $e \in_{\tau_1 + \tau_2} \texttt{l} \cdot \phi_1$ while refinement checking $e_1$, and correspondingly adding $e \in_{\tau_1 + \tau_2} \texttt{r} \cdot \phi_2$ while refinement checking $e_2$. In the event of a further case analysis on $e$ within either the branch, one of the preceding two rules would apply.

Preview

The relevance to Boolean blindness stems from the identification of $\texttt{bool}$ with $\texttt{unit} + \texttt{unit}$, and the application of the foregoing rules. Propagating $e \in_{\texttt{bool}} \texttt{true}$ and $e \in_{\texttt{bool}} \texttt{false}$ into the `then` and `else` branches is weaker than one might expect because doing so records information about the expression $e$ itself, but not about any expression equivalent to $e$.

**24.5**. Recall from Chapter 19 the definition

$$\texttt{dyn} \triangleq \texttt{rec}\, t\, \texttt{is}\, [\texttt{num} \hookrightarrow \texttt{nat}, \texttt{fun} \hookrightarrow t \rightharpoonup t]$$

of the type `dyn` as a recursive type. With this in mind we may make the following definitions of the refinements of type `dyn`:

$$\texttt{num}\,!\,\phi \triangleq \texttt{fold}(\texttt{num} \cdot \phi)$$
$$\texttt{fun}\,!\,\phi \triangleq \texttt{fold}(\texttt{fun} \cdot \phi),$$

using a labelled form of summand refinements.

**24.6**. The verification consists of exhibiting a derivation composed of the rules of refinement satisfaction given in Chapter 24. The description of the behavior of the addition function is absurdly complicated, but accurately reflects the behavior of even so simple a function as addition when formulated using dynamic typing. The surface syntax of **DPCF**, in particular, is highly deceptive because it obscures the needless complexity of the underlying code that is exposed by type refinements (and by the optimization process carried out in detail in Chapter 19).

**24.7**. To establish that the dynamic addition function satisfies the stated refinement necessitates showing that the interior general recursion satisfies the refinement $\phi \triangleq \texttt{fun}\,!\,\texttt{num}\,!\,\top \rightharpoonup \texttt{num}\,!\,\top$. This is achieved by assuming that $p \in_{\texttt{dyn}} \phi$ and showing that its body also satisfies $\phi$. The assumption that $p \in_{\texttt{dyn}} \phi$ amounts to a loop invariant that guarantees that the cast of $p$ to the function class cannot fail (by the safety theorem for type refinements), and hence can be safely eliminated, exactly as described in Section 19.3. Similarly, the assumptions that $x \in_{\texttt{dyn}} \texttt{num}\,!\,\top$ and $y \in_{\texttt{dyn}} \texttt{num}\,!\,\top$ suffice to underwrite the other optimizations described in that section.

# Chapter 26

**26.1**. Disregarding self-reference, the type of the dispatch matrix allows for some entries to be absent by redefining its type as follows:

$$\tau_{\mathsf{dm}} \triangleq \prod_{c \in C} \prod_{d \in D} (\tau^c \rightharpoonup \rho_d)\, \texttt{opt}.$$

Thus an entry in the dispatch matrix may either be absent, represented by `null`, or be present, represented by by $\texttt{just}(e_d^c)$, where $e_d^c$ is the behavior of method $d$ on instances of class $c$, as before.

Preview

For the class-based implementation the object type, $\rho$, must be redefined to reflect the possibility of a "not understood" error:

$$\rho \triangleq \prod_{d \in D} \rho_d \, \mathtt{opt}.$$

Message send is defined as before by projection, $e \Leftarrow d \triangleq e \cdot d$, but now has the type $\rho_d \, \mathtt{opt}$ to reflect the possibility that the method $d$ is undefined for the object $e$. The class vector is defined to be the tuple $e_{\mathsf{cv}} = \langle c \hookrightarrow e^c \rangle_{c \in C}$, where

$$e^c \triangleq \lambda\,(u{:}\tau^c)\,\langle d \hookrightarrow \mathtt{check}\, e_{\mathsf{dm}} \cdot c \cdot d\,\{\mathtt{null} \hookrightarrow \mathtt{null} \mid \mathtt{just}(f) \hookrightarrow \mathtt{just}(f(u))\}\rangle_{d \in D}.$$

As before, the class vector has type

$$\tau_{\mathsf{cv}} \triangleq \prod_{c \in C} \tau^c \rightharpoonup \rho,$$

albeit for the modified definition of $\tau_{\mathsf{obj}}$ given above. Instantiation cannot fail, but sending a message to the instantiated object may signal a "not understood" error by returning `null`, rather than $\mathtt{just}(e)$ for some value $e : \rho_d$.

For the method-based implementation, the object type, $\tau$, remains as before, but the type of the method vector is altered to

$$\tau_{\mathsf{mv}} \triangleq \prod_{d \in D} \tau_{\mathsf{obj}} \rightharpoonup \rho_d \, \mathtt{opt},$$

reflecting the possibility that a message send may fail. The implementation of the method vector is given by the tuple $\langle d \hookrightarrow e_d \rangle_{d \in D}$, where

$$e_d \triangleq \lambda\,(\mathit{this}{:}\tau)\,\mathtt{case}\,\mathit{this}\,\{c \cdot u \hookrightarrow \mathtt{check}\, e_{\mathsf{dm}} \cdot c \cdot d\,\{\mathtt{null} \hookrightarrow \mathtt{null} \mid \mathtt{just}(f) \hookrightarrow \mathtt{just}(f(u))\}\}_{c \in C}.$$

Here again accessing the dispatch matrix checks whether the required entry is present or not, with the result type reflecting the outcome accordingly.

To account for self-reference we must now allow for the possibility that the behavior assigned by the dispatch matrix for a particular class and method may, when present and called, incur a "not understood" error by sending a message to an instance for which it is not defined. The type of the dispatch matrix changes to the following more complex type:

$$\tau_{\mathsf{dm}} \triangleq \prod_{c \in C} \prod_{d \in D} (\forall(t_{\mathsf{obj}}.\tau_{\mathsf{cv}} \rightharpoonup \tau_{\mathsf{mv}} \rightharpoonup \tau^c \rightharpoonup \rho_d \, \mathtt{opt}))\, \mathtt{opt}.$$

The outermost option in each entry represents, as before, the presence or absence of a behavior for class $c$ and method $d$. The option at the end reflects the possibility that the behavior may incur a "not understood" error when applied, as is now possible by creating instances and sending them messages using the given class- and method vectors.

Preview

The types $\tau_{\mathsf{cv}}$ and $\tau_{\mathsf{mv}}$ are given in terms of the abstract type, $t$, of objects as follows:

$$\tau_{\mathsf{cv}} \triangleq \prod_{c \in C} \tau^c \rightharpoonup t_{\mathsf{obj}}$$
$$\tau_{\mathsf{mv}} \triangleq \prod_{d \in D} t_{\mathsf{obj}} \rightharpoonup \rho_d\, \mathtt{opt},$$

the latter reflecting the possibility of a "not understood" error upon message send. Within the entries of the dispatch matrix, class instantiation is mediated by the given class vector, without the possibility of error, and message send by the given method vector, with the possibility of error. The implementations of the class- and method vectors for specific choices of object type are given along the lines sketched above.

**26.2**. (a) The refinement $\mathtt{inst}[c]$ of $\tau_{\mathsf{obj}}$ is the (generalized) summand refinement $c \hookrightarrow \top_{\tau^c}$ corresponding to $c \in C$, imposing no condition on its instance data. The refinement $\mathtt{admits}[d]$ is defined to be the (generalized) sum refinement $\sum_{c \in C_d} \top_{\tau^c}$, which holds of any value of type $\tau_{\mathsf{obj}}$, provided that it is an instance of a class that admits method $d$. It is immediate that $\mathtt{inst}[c] \leq \mathtt{admits}[d]$ when $c \in C_d$, and hence when $d \in D_c$.

(b) The class- and method-vector refinements are chosen as follows:

$$\phi_{\mathsf{cv}} \triangleq \prod_{c \in C} \top_{\tau^c} \rightharpoonup \mathtt{inst}[c]$$
$$\phi_{\mathsf{mv}} \triangleq \prod_{d \in D} \mathtt{admits}[d] \rightharpoonup \mathtt{just}(\top_{\rho_d})$$

Check that $\phi_{\mathsf{cv}} \sqsubseteq \tau_{\mathsf{cv}}$ and $\phi_{\mathsf{mv}} \sqsubseteq \tau_{\mathsf{mv}}$ in the sense of Solution **26.1** with $\tau_{\mathsf{obj}}$ chosen to be $\sum_{c \in C} \tau^c$. It is immediate that $e_{\mathsf{cv}} \in_{\tau_{\mathsf{cv}}} \phi_{\mathsf{cv}}$ and not much harder to check that $e_{\mathsf{mv}} \in_{\tau_{\mathsf{mv}}} \phi_{\mathsf{mv}}$, bearing in mind the assumption that $e_{\mathsf{dm}} \in_{\tau_{\mathsf{dm}}} \phi_{\mathsf{dm}}$, which ensures that the dispatch yields a non-error result.

(c) Because $\mathtt{new}[c](e) \in_{\tau_{\mathsf{obj}}} \mathtt{inst}[c]$ and $\mathtt{inst}[c] \leq \mathtt{admits}[d]$ whenever $d \in D_c$, it follows that $\mathtt{new}[c](e) \in_{\tau_{\mathsf{obj}}} \mathtt{admits}[d]$ whenever $d \in D_c$. Consequently, $\mathtt{new}[c](e) \Leftarrow d \in_{\rho_d} \mathtt{just}(\top_{\rho_d})$ whenever $d \in D_c$, which is to say that a "not understood" error does not arise for well-refined message send operations.

**26.3**. The problem is to define the entries $e^{\mathtt{num}}_{\mathtt{ev}}$ and $e^{\mathtt{num}}_{\mathtt{od}}$ with the specified behavior. These have similar overall form, with different method bodies:

$$e^{\mathtt{num}}_{\mathtt{ev}} \triangleq \Lambda(t_{\mathsf{obj}}.\lambda\,(cv{:}\tau_{\mathsf{cv}})\,\lambda\,(mv{:}\tau_{\mathsf{mv}})\,\lambda\,(u{:}\tau^{\mathtt{num}})\,e_{\mathtt{ev}})$$
$$e^{\mathtt{num}}_{\mathtt{od}} \triangleq \Lambda(t_{\mathsf{obj}}.\lambda\,(cv{:}\tau_{\mathsf{cv}})\,\lambda\,(mv{:}\tau_{\mathsf{mv}})\,\lambda\,(u{:}\tau^{\mathtt{num}})\,e_{\mathtt{od}})$$

Their respective method bodies are defined as follows:

$$e_{\mathtt{ev}} \triangleq \mathtt{ifz}\,u\,\{\mathtt{z} \hookrightarrow \mathtt{true} \mid \mathtt{s}(u') \hookrightarrow mv \cdot \mathtt{od}(cv \cdot c(u'))\}$$
$$\mathtt{od} \triangleq \mathtt{ifz}\,u\,\{\mathtt{z} \hookrightarrow \mathtt{false} \mid \mathtt{s}(u') \hookrightarrow mv \cdot \mathtt{ev}(cv \cdot c(u'))\}.$$

Message send is effected by projection from $mv$ with the argument being a new instance of $c$ obtained by projection from $cv$.

Preview

**26.4**. Suppose that the abstract object type, $t$, is permitted to occur in the instance type, $\tau^c$, of some class, or in the result type, $\rho_d$, of some method. The alterations required depend on whether we are considering the method-based or the class-based organization.

In the method-based organization the concrete object type, $\tau$, becomes the recursive sum type

$$\tau \triangleq \mathtt{rec}\, t\, \mathtt{is} \sum_{c \in C} \tau^c.$$

Correspondingly, the definition of the method vector, $e_{\mathsf{mv}}$, of type $[\tau/t]\tau_{\mathsf{mv}}$ `self` becomes

$$\mathtt{self}\ mv\ \mathtt{is}\ \langle d \hookrightarrow \lambda\,(this{:}\tau)\ \mathtt{case}\ \mathtt{unfold}(this)\ \{c \cdot u \hookrightarrow e_{\mathsf{dm}} \cdot c \cdot d[\tau](e''_{\mathsf{cv}})(e'_{\mathsf{mv}})(u)\}_{c \in C}\rangle_{d \in D},$$

wherein $e'_{\mathsf{cv}}$ is revised to become

$$e'_{\mathsf{cv}} \triangleq \langle c \hookrightarrow \lambda\,(u{:}[\tau/t]\tau^c)\ \mathtt{fold}(c \cdot u)\rangle_{c \in C} : [\tau/t]\tau_{\mathsf{cv}}$$

and $e'_{\mathsf{mv}}$ remains as in the simple self-referential case. Object creation is redefined similarly, with instance type $[\tau/t]\tau^c$ possibly involving the object type,

$$\mathtt{new}[c](e) \triangleq \mathtt{fold}(c \cdot e) : \tau.$$

Message send remains as before, but with a result type that may include the object type:

$$e \Leftarrow d \triangleq \mathtt{unroll}(e_{\mathsf{mv}}) \cdot d(e) : [\tau/t]\rho_d.$$

In the class-based organization the concrete object type, $\rho$, becomes the recursive product type

$$\rho \triangleq \mathtt{rec}\, t\, \mathtt{is} \prod_{d \in D} \rho_d.$$

Correspondingly, the class vector, $e_{\mathsf{cv}}$, of type $[\rho/t]\tau_{\mathsf{cv}}$ `self`, becomes

$$\mathtt{self}\ cv\ \mathtt{is}\ \langle c \hookrightarrow \lambda\,(u{:}[\rho/t]\tau^c)\ \mathtt{fold}(\langle d \hookrightarrow e_{\mathsf{dm}} \cdot c \cdot d[\rho](e''_{\mathsf{cv}})(e''_{\mathsf{mv}})(u)\rangle_{d \in D})\rangle_{c \in C},$$

wherein $e''_{\mathsf{mv}}$ is revised to become

$$e''_{\mathsf{mv}} \triangleq \langle d \hookrightarrow \lambda\,(this{:}\rho)\ \mathtt{unfold}(this) \cdot d\rangle_{d \in D} : [\rho/t]\tau_{\mathsf{mv}}$$

and $e''_{\mathsf{cv}}$ remains as in the chapter. Message send is redefined similarly:

$$e \Leftarrow d \triangleq \mathtt{unfold}(e) \cdot d : [\rho/t]\rho_d.$$

Object creation remains as before, but taking an argument of type $[\rho/t]\tau^c$:

$$\mathtt{new}[c](e) \triangleq \mathtt{unroll}(e_{\mathsf{cv}}) \cdot c(e) : \rho,$$

Preview

# Chapter 27

**27.1**. We wish to extend $C$ with a new class $c^*$, and to define the dispatch matrix entry $e_d^{c^*}$ by inheritance to be $e_d^c$. The chief difficulty is that the entries are parameterized by the abstract class- and method vectors. Adding a new class, $c^*$, extends $C$, so that the type of the extended class vector is (up to a reordering isomorphism) the product

$$\tau_{\mathsf{cv}}^* \triangleq (\prod_{c \in C} \tau^c \rightharpoonup t_{\mathsf{obj}}) \times (\tau^{c^*} \rightharpoonup t_{\mathsf{obj}}).$$

By product subtyping we may regard $\tau_{\mathsf{cv}}^* <: \tau_{\mathsf{cv}}$ so that $e_d^c$ is applicable when supplied with the extended class vector. In any case we demand that $\tau^{c^*} <: \tau^c$, as before, to ensure that the inherited method may be applied to the instance data of the new class, $c^*$.

The analysis of method extension with inheritance is dual to that of class extension, with covariance on the result type and method vector.

**27.2**. No, even though the `ev` method is invoked on an instance of `nat`$*$, the inherited definition of `ev` will send an `od` message to an instance of `num`, and the recursion will continue on `num` instances thereafter. If the revised version of `od` differed in behavior, this fact would not be reflected in the behavior of `ev` on instances of `num`$*$.

**27.3**. Work in the context of Solution **26.4**, which permits instance data and results to be objects.

(a) Choose $\tau^{\mathtt{num}}$ to be $t_{\mathsf{obj}}$, the abstract type of objects. The intention is that this object be of a class that supports the `ev` and `od` methods; this can be enforced using refinements in the manner of Solution **26.2**. The method bodies for `ev` and `od` are defined as follows:[1]

$$e_{\mathtt{ev}}^{\mathtt{num}} \triangleq \ldots mv \cdot \mathtt{ev}(u)$$
$$e_{\mathtt{od}}^{\mathtt{num}} \triangleq \ldots mv \cdot \mathtt{ev}(u).$$

(b) Choose $\tau^{\mathtt{zero}}$ to be `unit`, there being no useful instance data. Define `ev` and `od` as follows:

$$e_{\mathtt{ev}}^{\mathtt{zero}} \triangleq \ldots \mathtt{true}$$
$$e_{\mathtt{od}}^{\mathtt{zero}} \triangleq \ldots \mathtt{false}.$$

Choose $\tau^{\mathtt{succ}}$ to be $t_{\mathsf{obj}}$, with the intention that it be an object of class `num`, an invariant that may be enforced with refinements in the manner of Solution **26.2**. The `ev` and `od` methods are implemented as follows:

$$e_{\mathtt{ev}}^{\mathtt{zero}} \triangleq \ldots mv \cdot \mathtt{od}(u)$$
$$e_{\mathtt{od}}^{\mathtt{zero}} \triangleq \ldots mv \cdot \mathtt{ev}(u).$$

The instance data of the `succ` class is the predecessor, which is assumed to implement the `ev` and `od` methods.

[1] The prefixing abstractions over $t_{\mathsf{obj}}$, $cv$, $mv$, and $u$ are elided for clarity.

Preview

(c) Now introduce a subclass succ* of succ that overrides the od method. Observe that the semantics of dynamic dispatch ensures that sending ev or od to any instance of succ* will invoke the overridden od method.

# Chapter 28

**28.1**. The preservation proof consists of three parts:

(a) Rule (28.4a): We have $s = k \triangleright \mathtt{ifz}(e; e_0; x.e_1)$ and $s' = k;\mathtt{ifz}(-; e_0; x.e_1) \triangleright e$. By inversion of Rules (28.9a) and (28.9a) we have $k \triangleleft: \tau$ and $\mathtt{ifz}(e; e_0; x.e_1) : \tau$. Therefore by Rules (28.8b), (28.7b), we have $s'$ ok, as required.

(b) Rule (28.4b): We have $s = k;\mathtt{ifz}(-; e_0; x.e_1) \triangleleft \mathtt{z}$ and $s' = k \triangleright e$. By inversion of Rules (28.9b), (28.7b) and (28.8b), we have $k \triangleleft: \tau$, $\mathtt{ifz}(-; e_0; x.e_1) : \mathtt{nat} \rightsquigarrow \tau$, $e : \mathtt{nat}$, and $e_0 : \tau$. But then we have $s'$ ok by Rule (28.9a).

(c) Rule (28.4c): We have $s = k;\mathtt{ifz}(-; e_0; x.e_1) \triangleleft \mathtt{s}(e)$ and $s' = k \triangleright [e/x]e_1$. By inversion of Rules (28.9b), (28.7b) and (28.8b), we have $k \triangleleft: \tau$, $\mathtt{ifz}(-; e_0; x.e_1) : \mathtt{nat} \rightsquigarrow \tau$, $\mathtt{s}(e) : \mathtt{nat}$, and hence by inversion and substitution for **PCF**, we have $[e/x]e_1 : \tau$. But then the result follows from Rule (28.9a).

The progress proof proceeds as follows. Suppose that $s$ ok. Either $s = k \triangleright e$ or $e = k \triangleleft e$. In either case by inversion of Rules (28.9) we have $k \triangleleft: \tau$ and $e : \tau$ for some type $\tau$. In addition $e$ val in the case that $e = k \triangleleft e$. By analyzing the statics and canonical forms for **PCF**, we may verify in each case that the state is either final or may make progress.

**28.2**. First, we need an additional form of frame representing argument evaluation:

$$\frac{e_1 \ \mathsf{val}}{\mathtt{ap}(e_1; -) \ \mathsf{frame}} \tag{B.20}$$

Second, we must replace rule (28.5c) by these rules:

$$\overline{k;\mathtt{ap}(-; e_2) \triangleleft e_1 \longmapsto k;\mathtt{ap}(e_1; -) \triangleright e_2} \tag{B.21a}$$

$$\overline{k;\mathtt{ap}(\mathtt{lam}[\tau](x.e); -) \triangleleft e_2 \longmapsto k \triangleright [e_2/x]e} \tag{B.21b}$$

Corresponding modifications are required of the proofs of safety and correctness, following along similar lines to those given in Chapter 28.

**28.3**. Each step of the dynamics involves either extending the stack by one frame, and descending into a sub-expression, or analyzing the outermost form of a value and the topmost frame of a stack to determine how to proceed. These may all be performed in constant time with a suitable representation of stacks as linked data structures and expressions as trees. Determining



Preview

whether an expression may take time proportional to the size of that value, so determining whether the machine is finished requires time proportional to the size of the resulting value (that is, to "read" the result). If answers are limited to natural numbers, this takes time proportional to the value of the number, because we are using unary representations. This obviously can be reduced to logarithmic time by using binary representations. Substitution takes time proportional to the size of the expression into which we are substituting, a significant cost.

**28.4**. The number of transitions taken by the **PCF** machine to compute the value, $v$, of an expression, $e$, is proportional to the size of the derivation of $e \longmapsto^* v$, taking into account the sub-derivations that check whether an expression is a value. For example, the machine always fully explores a numeral to compute its value, whereas the the value judgment makes this traversal without involving any transitions.

# Chapter 29

**29.1**. The proof proceeds along standard lines, by induction on the machine dynamics for preservation, and by induction on the well-formation of states for the proof of progress. The proof of preservation relies on there being one universal type $\tau_{\mathsf{exn}}$ of exception values. Otherwise an exception value may be passed to an exception handler expecting a different type. The proof of progress must take into account the transitions that propagate an exception through the frames of the control stack until either a handler is reached, or the stack is exhausted. But in that case the state is final, according to the revised definition given by Rules (29.6).

**29.2**. To encapsulate exceptions within a pure expression, simply introduce a sequentialization expression of the form $\mathtt{let}(e; x_1.e_1; x_2.e_2)$, which behaves similarly to the sequentialization command, except that the branches are required to be pure expressions. Any exception raised by the command given by $e$ will be intercepted by the sequentialization, yielding an exception-free result. The typing rule for the new construct is given by

$$\frac{\Gamma \vdash e : \mathtt{cmd}(\tau) \quad \Gamma, x_1 : \tau \vdash e_1 : \tau' \quad \Gamma, x_2 : \tau_{\mathsf{exn}} \vdash e_2 : \tau'}{\Gamma \vdash \mathtt{let}(e; x_1.e_1; x_2.e_2) : \tau'} \tag{B.22}$$

The **MPCF** machine is readily extended to account for this encapsulation construct.

**29.3**. The command refinement judgment is defined by these rules:

$$\frac{\Phi \vdash e \in_\tau \phi}{\Phi \vdash \mathtt{ret}(e) \in_\tau \phi\ \mathtt{ow}\ \chi} \tag{B.23a}$$

$$\frac{\Phi \vdash e \in_{\tau_{\mathsf{exn}}} \chi}{\Phi \vdash \mathtt{raise}[\tau](e) \in_\tau \phi\ \mathtt{ow}\ \chi} \tag{B.23b}$$

$$\frac{\Phi \vdash e \in_{\mathtt{cmd}(\tau)} \mathtt{cmd}(\phi_0; \chi_0) \quad \Phi, x_1 \in_\tau \phi_0 \vdash m_1 \in_{\tau'} \phi\ \mathtt{ow}\ \chi \quad \Phi, x_2 \in_\tau \chi_0 \vdash m_2 \in_{\tau'} \phi\ \mathtt{ow}\ \chi}{\Phi \vdash \mathtt{bnd}(e; x_1.e_1; x_2.e_2) \in_{\tau'} \phi\ \mathtt{ow}\ \chi} \tag{B.23c}$$

Preview

Rule (B.23a) admits an arbitrary $\chi \sqsubseteq \tau_{\mathsf{exn}}$ to allow for over-approximation of the exception values that may be raised; it is valid to choose $\chi$ to be $\emptyset$, because pure expressions cannot raise exceptions. Rule (B.23b) specifies that the raised value satisfies the condition that is derived for it at the expression level. Rule (B.23c) specifies that the exceptions that may be raised by executing the command $e$ are absorbed by the handler. The only exceptions that may arise from it arise from either the normal or the exceptional return commands. Their exception refinements must, in general, be weakened to a common refinement, thereby attenuating the precision with which exceptions may be tracked.

The command refinement $(\phi;\chi)\,\mathtt{cmd}$ is defined to be monotone in each position,

$$\frac{\phi \leq \phi' \quad \chi \leq \chi'}{\mathtt{cmd}(\phi;\chi) \leq \mathtt{cmd}(\phi';\chi')}\ . \tag{B.24}$$

The expression satisfaction rule for this refinement is given as follows:

$$\frac{\Phi \vdash m \in_\tau \phi\ \mathtt{ow}\ \chi}{\Phi \vdash \mathtt{cmd}(m) \in_{\mathtt{cmd}(\tau)} \mathtt{cmd}(\phi;\chi)} \tag{B.25}$$

**29.4**. The key parts of the expression and command translation are defined as follows:

$$\widehat{\mathtt{cmd}(m)} \triangleq \widehat{m}$$
$$\widehat{\mathtt{ret}(e)} \triangleq \mathtt{l} \cdot \widehat{e}$$
$$\widehat{\mathtt{raise}[\tau](e)} \triangleq \mathtt{r} \cdot \widehat{e}$$
$$\mathtt{bnd}(e; \widehat{x_1.m_1; x_2.m_2}) \triangleq \mathtt{case}\,\widehat{e}\,\{\mathtt{l} \cdot x_1 \hookrightarrow \widehat{m_1} \mid \mathtt{r} \cdot x_2 \hookrightarrow \widehat{m_2}\}$$

The expressions of **MPCF** must be translated to propagate exceptions. For example, application is translated as follows:

$$\widehat{e_1(e_2)} \triangleq \mathtt{case}\,\widehat{e_1}\,\{\mathtt{l} \cdot x_1 \hookrightarrow x_1(\widehat{e_2}) \mid \mathtt{r} \cdot x_1' \hookrightarrow \mathtt{r} \cdot x_1'\}.$$

The case analysis corresponds to propagation of an exception across the stack frame for a pending application.

# Chapter 30

**30.1**. A closed value of type $\tau\ \mathtt{cont}$ has the form $\mathtt{cont}(k)$, where $k$ is a control stack such that $k \lhd : \tau$. This is enough to ensure progress. Preservation is assured because if $k \rhd \mathtt{letcc}\ x\ \mathtt{in}\ e\ \mathtt{ok}$, then $k \lhd : \tau$ and $x : \tau\ \mathtt{cont} \vdash e : \tau$ for some type $\tau$. Consequently, $[\mathtt{cont}(k)/x]e : \tau$ by substitution, which is enough for preservation.

**30.2**. (a) $\lambda\,(x{:}\tau\ \mathtt{cont}\ \mathtt{cont})\ \mathtt{letcc}\ k\ \mathtt{in}\ \mathtt{throw}\ k\ \mathtt{to}\ x$.



Preview

(b) $\texttt{letcc}\ \mathit{ret}\ \texttt{in}\ \texttt{r} \cdot (\texttt{letcc}\ r\ \texttt{in}\ \texttt{throw}\ \texttt{l} \cdot (\texttt{letcc}\ l\ \texttt{in}\ \texttt{throw}\ l\ \texttt{to}\ r)\ \texttt{to}\ \mathit{ret})$.

(c) $\lambda\,(x{:}\tau_2\ \texttt{cont} \to \tau_1\ \texttt{cont})\ \lambda\,(x_1{:}\tau_1)\ \texttt{letcc}\ k_2\ \texttt{in}\ \texttt{throw}\ x_1\ \texttt{to}\ x(k_2)$.

(d) $\lambda\,(k{:}(\tau_1 + \tau_2)\ \texttt{cont})\ \langle e_1, e_2\rangle$, where

$$e_1 \triangleq \texttt{letcc}\ r_1\ \texttt{in}\ \texttt{throw}\ \texttt{l} \cdot (\texttt{letcc}\ k_1\ \texttt{in}\ \texttt{throw}\ k_1\ \texttt{to}\ r_1)\ \texttt{to}\ k, \text{ and}$$
$$e_2 \triangleq \texttt{letcc}\ r_2\ \texttt{in}\ \texttt{throw}\ \texttt{r} \cdot (\texttt{letcc}\ k_2\ \texttt{in}\ \texttt{throw}\ k_2\ \texttt{to}\ r_2)\ \texttt{to}\ k.$$

**30.3**. Define $\texttt{stream} \triangleq \texttt{rec}\ t\ \texttt{is}\ (\texttt{nat} \times t)\ \texttt{cont}\ \texttt{cont}$, so that fold and unfold provide the required isomorphism. The eliminatory operations, $\texttt{hd}(e)$ and $\texttt{tl}(e)$, on streams are defined by projection from the client expression

$$\texttt{letcc}\ c\ \texttt{in}\ \texttt{throw}\ c\ \texttt{to}\ \texttt{unfold}(e).$$

The client evaluates to a pair consisting of the first number and the rest of the given stream by passing a return continuation to the stream generator, which provides it with the head and tail of the stream. The introductory generator $\texttt{strgen}\ x\ \texttt{is}\ e\ \texttt{in} <\texttt{hd} \hookrightarrow e_1, \texttt{tl} \hookrightarrow e_2>$ is defined as the following producer expression:

$$(\texttt{fix}\ g\ \texttt{is}\ \lambda\,(x{:}\tau)\ \texttt{letcc}\ \mathit{ret}\ \texttt{in}\ \texttt{throw}\ \langle e_1, g(e_2)\rangle\ \texttt{to}\ (\texttt{letcc}\ p\ \texttt{in}\ \texttt{throw}\ \texttt{fold}(p)\ \texttt{to}\ \mathit{ret}))(e),$$

where $\tau$ is the state type of the stream. The innermost parenthesized sub-expression creates a new stream (essentially a continuation), and returns it as the result of the generating function, $g$, which takes as argument the current state of the generator. When a client continuation is thrown to that stream, the next number, paired with the remaining numbers, is thrown to it.

# Chapter 31

**31.1**. Informally, to prove $\neg\neg(\phi \vee \neg\phi)$ true, assume $\neg(\phi \vee \neg\phi)$ true and derive a contradiction. To do so, we prove $\phi \vee \neg\phi$. (Why is this plausible, given that LEM cannot be expected to hold for a general $\phi$?) To prove a disjunction, it suffices to prove one of its disjuncts; in this case, prove $\neg\phi$ true. To do so, assume $\phi$ true and derive a contradiction. You now have two assumptions, $\neg(\phi \vee \neg\phi)$ true, and $\phi$ true. From the latter it follows that $\phi \vee \neg\phi$ true, which contradicts the former. By discharging the second assumption derive $\neg\phi$ true. But then $\phi \vee \neg\phi$ true, again contradicting the first assumption. Discharging the first assumption, derive $\neg\neg(\phi \vee \neg\phi)$. Formally, the proof term $\lambda\,(x)\,x(\texttt{r} \cdot \lambda\,(y)\,x(\texttt{l} \cdot y))$ has the required type.

**31.2**. Informally, suppose that LEM holds universally, let $\phi$ be an arbitrary proposition, and assume $\neg\neg\phi$ true with the intent to derive $\phi$ true. By LEM instantiated with $\phi$ we have $\phi \vee \neg\phi$ true. In the former case we have $\phi$ true by assumption; in the latter we have a contradiction of the assumption $\phi$ true, and hence by false elimination we have $\phi$ true as well. Formally, the proof term $\lambda\,(y)\,\texttt{case}\,\mathrm{LEM}_\phi\,\{\texttt{l} \cdot y_1 \hookrightarrow y_1 \mid \texttt{r} \cdot y_2 \hookrightarrow \texttt{abort}(y_2(y))\}$ has the required type.



Preview

**31.3**. The required properties all follow more or less directly from the definition of entailment in constructive logic. First, check that $A_1$ true, $\ldots$, $A_n$ true $\vdash$ $A$ true iff $A_1 \wedge \ldots A_n$ true $\vdash$ $A$ true. Second, check the required properties of conjunction and truth, and of disjunction and falsehood, respectively. Third, check that $\phi \supset \psi$ is such that $\phi \wedge \phi \supset \psi \leq \psi$. Suppose that $\phi \wedge \rho \leq \psi$; we are to show that $\rho \leq \phi \supset \psi$. It is convenient to appeal to the Yoneda Lemma (Exercise **24.2**). It is enough to show that if $\gamma \leq \rho$, then $\gamma \leq \phi \supset \psi$. Given $\gamma \leq \rho$, it follows from the assumption that $\phi \wedge \gamma \leq \psi$. But then $\gamma \leq \phi \supset \psi$, by implication introduction.

Consider the equivalence $\phi \wedge (\psi_1 \vee \psi_2) \equiv (\phi \wedge \psi_1) \vee (\phi \wedge \psi_2)$. Let $\lambda$ stand for the left-hand side, and $\rho$ stand for the right-hand side. Because $\lambda$ is a meet, it suffices to show $\rho \leq \phi$ and $\rho \leq \psi_1 \vee \psi_2$. The former is immediate, the latter almost immediate, and so $\rho \leq \lambda$. To show that $\lambda \leq \rho$, use the exponential property to reduce the problem to showing $\phi \leq \rho^{\psi_1 \vee \psi_2}$, which is to say that $\phi \leq \rho^{\psi_1}$ and $\phi \leq \rho_2^{\psi}$. But for these we need only show $\phi \wedge \psi_1 \leq \rho$ and $\phi \wedge \psi_2$, both of which are immediate. The "dual duality" is proved dually, and is left as an exercise.

It makes sense that the exponential is used in the preceding argument, because not all lattices are distributive.

**31.4**. It is elementary to check that the truth tables define a Boolean algebra with the exponential $\phi \supset \psi$ given by $\neg\phi \vee \phi$. The first de Morgan duality law may be proved for any Heyting algebra, but the second requires LEM (or one of its many equivalents).

# Chapter 32

**32.1**. It is helpful to derive the negation and implication elimination forms from constructive logic as follows. For negation, if $p' : \neg\phi$ true and $p : \phi$ true, then $p'(p) : \bot$, where $p'(p) \triangleq \mathtt{ccr}(u.(\mathtt{not}(p) \# p'))$. For implication, a similar derivation shows that if $\phi \supset \psi$ true and $\phi$ true, then $\psi$ true.

(a) $\lambda\,(x)\,\mathtt{ccr}(u.(\mathtt{abort} \# x(\mathtt{not}(u)))) : (\neg\neg\phi) \supset \phi$.

(b) $\lambda\,(x)\,\lambda\,(x_1)\,\mathtt{ccr}(u_2.(\mathtt{abort} \# x(\mathtt{not}(u_2))(x_1))) : (\neg\phi_2 \supset \neg\phi_1) \supset (\phi_1 \supset \phi_2)$.

(c) $\lambda\,(x)\,\langle \mathtt{not}(\mathtt{ccp}(x_1.(\mathtt{abort} \# x(\mathtt{l} \cdot x_1)))), \mathtt{not}(\mathtt{ccp}(x_2.(\mathtt{abort} \# x(\mathtt{r} \cdot x_2))))\rangle$ is a proof of $\neg(\phi_1 \vee \phi_2) \supset (\neg\phi_1 \wedge \neg\phi_2)$.

Compare these proof terms to Solution **30.2**, which amount to the same thing, under the identification of the proof $\mathtt{not}(k)$, where $k \div \phi$, with the continuation $\mathtt{cont}(k)$, where $k \triangleleft \tau$. The ability to create a machine state directly avoids the ruses used in Solution **30.2** to throw a continuation for use elsewhere in the computation.

**32.2**. Section 32.5 sketches the main ideas of the proof. It is left to you to compare the "compiled" and "hand-written" proof of the doubly negated LEM; it depends on the details of your particular formulation of the translation.

Preview

# Chapter 33

**33.1**. Simply add a pritimive equality test $\mathtt{eq} : \rho\,\mathtt{sym} \times \rho\,\mathtt{sym} \rightharpoonup \mathtt{bool}$, and observe that it is applicable only if the two symbol references have the same associated type.

**33.2**. It is not difficult to program a linear search in **SPCF** using Solution 33.1 to test equality.

**33.3**. A natural ordering of symbols in a deterministic language such as **SPCF** would be their order of allocation represented by their order of declaration in the signature. The difficulty is that the ordering is not invariant under renaming, because $\alpha$-equivalence need not respect ordering. One solution involves mutable state, which will be discussed in Chapter 37: maintain a global counter and associate a unique number with each symbol that is used to determine a linear ordering among them.

**33.4**. The main idea is to define take $\mathtt{'}a$ to be the primitive symbol reference, and to extend it to all s-expressions by defining $\mathtt{'nil} \triangleq \mathtt{nil}$ and $\mathtt{'cons}(e_1;e_2) \triangleq \mathtt{cons}(\mathtt{'}e_1;\mathtt{'}e_2)$. So, in particular, $\mathtt{'}(e_0,\ldots,e_{n-1}) \triangleq (\mathtt{'}e_0,\ldots,\mathtt{'}e_{n-1})$. For example, $\mathtt{'}(a,b,c,d)$ is the list $(\mathtt{'}a,\mathtt{'}b,\mathtt{'}c,\mathtt{'}d)$. Notice that $\mathtt{'nil}$ is $\mathtt{nil}$, the injection into the recursive sum, and is not a symbol! If numbers are included among s-expressions, then $\mathtt{'}n$ would be defined to be $n$, and not a symbol that happens to have a numeric representation. These, and other related cases, have led to controversy among various Lisp dialects and implementations, in part because of the absence of a rigorous mathematical foundation, leaving only opinion and authority as the determinative criteria.

**33.5**. Following the structure of **MPCF**, introduce a command $\mathit{new}\,a \sim \tau$ that allocates a fresh symbol $a$ with associated type $\tau$. In the scoped case the rest of the command constitutes the scope of the symbol, and the overall return type must be mobile in that its values must not contain symbol references. In the scope-free case no mobility restriction is required.

# Chapter 34

**34.1**. Enrich the **FPCF** machine with transitions corresponding to allocating a symbol, putting a binding for a symbol, and getting a binding for a symbol as follows:

$$\frac{}{k \triangleright \mathtt{new}\,a \sim \rho\,\mathtt{in}\,e \longmapsto k;\mathtt{new}\,a \sim \rho\,\mathtt{in} - \triangleright e} \tag{B.26a}$$

$$\frac{}{k;\mathtt{new}\,a \sim \rho\,\mathtt{in} - \triangleleft e \longmapsto k \triangleleft e} \tag{B.26b}$$

$$\frac{}{k;\mathtt{new}\,a \sim \rho\,\mathtt{in} - \blacktriangleleft \longmapsto k \blacktriangleleft} \tag{B.26c}$$

Preview

$$\frac{}{k \triangleright \mathtt{put}\, e_1\, \mathtt{for}\, a\, \mathtt{in}\, e_2 \longmapsto k;\mathtt{put} - \mathtt{for}\, a\, \mathtt{in}\, e_2 \triangleright e_1} \tag{B.26d}$$

$$\frac{}{k;\mathtt{put} - \mathtt{for}\, a\, \mathtt{in}\, e_2 \triangleleft e_1 \longmapsto k;\mathtt{put}\, e_1\, \mathtt{for}\, a\, \mathtt{in} - \triangleright e_2} \tag{B.26e}$$

$$\frac{}{k;\mathtt{put}\, e_1\, \mathtt{for}\, a\, \mathtt{in} - \triangleleft e_2 \longmapsto k \triangleleft e_2} \tag{B.26f}$$

$$\frac{}{k;\mathtt{put}\, e_1\, \mathtt{for}\, a\, \mathtt{in} - \blacktriangleleft \longmapsto k \blacktriangleleft} \tag{B.26g}$$

$$\frac{}{k \triangleright \mathtt{get}\, a \longmapsto k \geq k\ ?\ a} \tag{B.26h}$$

$$\frac{}{k \geq k';\mathtt{put}\, e\, \mathtt{for}\, a\, \mathtt{in} - ?\ a \longmapsto k \triangleleft e} \tag{B.26i}$$

$$\frac{}{k \geq k';\mathtt{new}\, a \sim \rho\, \mathtt{in} - ?\ a \longmapsto k \blacktriangleleft} \tag{B.26j}$$

$$\frac{(f \neq \mathtt{put}\, e_1\, \mathtt{for}\, a\, \mathtt{in} -, f \neq \mathtt{new}\, a \sim \rho\, \mathtt{in} -)}{k \geq k'; f\ ?\ a \longmapsto k \geq k'\ ?\ a} \tag{B.26k}$$

Rules (B.26a) to (B.26c) mark the allocation of a new symbol by pushing a frame on the stack, and propagate normal and failure return through it. Rules (B.26d) to (B.26g) implement the stack semantics of `put`. In particular, rule (B.26f) reveals the underlying problem discussed in Section 34.4: the value $e$ may depend on the symbol $a$ whose binding is being dropped on the transition. Rule (B.26h) initiates the lookup of a binding for $a$ in the stack. The linear search of the stack for the most recent binding of a symbol $a$ is implemented by Rules (B.26i) to (B.26k).

**34.2**. The following rules implement shallow binding:

$$\frac{}{k \parallel \mu \triangleright \mathtt{new}\, a \sim \rho\, \mathtt{in}\, e \longmapsto k;\mathtt{new}\, a \sim \rho\, \mathtt{in} - \parallel \mu \otimes \{a \hookrightarrow \epsilon\} \triangleright e} \tag{B.27a}$$

$$\frac{}{k;\mathtt{new}\, a \sim \rho\, \mathtt{in} - \parallel \mu \otimes \{a \hookrightarrow \epsilon\} \triangleleft e \longmapsto k \parallel \mu \triangleleft e} \tag{B.27b}$$

$$\frac{}{k;\mathtt{new}\, a \sim \rho\, \mathtt{in} - \parallel \mu \otimes \{a \hookrightarrow \epsilon\} \blacktriangleleft\!\!\longmapsto k \parallel \mu \blacktriangleleft} \tag{B.27c}$$

$$\frac{}{k \parallel \mu \triangleright \mathtt{put}\, e_1\, \mathtt{for}\, a\, \mathtt{in}\, e_2 \longmapsto k;\mathtt{put} - \mathtt{for}\, a\, \mathtt{in}\, e_2 \parallel \mu \triangleright e_1} \tag{B.27d}$$

Preview

$$\overline{k;\texttt{put}-\texttt{for}\,a\,\texttt{in}\,e_2 \parallel \mu \otimes \{a \hookrightarrow s\} \triangleleft e_1 \longmapsto k;\texttt{put}\,a\,\texttt{for}\,\rho\,\texttt{in}--\parallel \mu \otimes \{a \hookrightarrow s;e_1\} \triangleright e_2} \tag{B.27e}$$

$$\overline{k;\texttt{put}-\texttt{for}\,a\,\texttt{in}-\parallel \mu \otimes \{a \hookrightarrow s;e_1\} \triangleleft e_2 \longmapsto k \parallel \mu \triangleleft e_2} \tag{B.27f}$$

$$\overline{k;\texttt{put}-\texttt{for}\,a\,\texttt{in}-\parallel \mu \otimes \{a \hookrightarrow s;e_1\} \blacktriangleleft\!\!\longmapsto k \parallel \mu \blacktriangleleft} \tag{B.27g}$$

$$\overline{k \parallel \mu \otimes \{a \hookrightarrow s;e\} \triangleright \texttt{get}\,a \longmapsto k \parallel \mu \otimes \{a \hookrightarrow s;e\} \triangleleft e} \tag{B.27h}$$

$$\overline{k \parallel \mu \otimes \{a \hookrightarrow \epsilon\} \triangleright \texttt{get}\,a \longmapsto k \parallel \mu \otimes \{a \hookrightarrow \epsilon\} \blacktriangleleft} \tag{B.27i}$$

Obtaining the current binding of a symbol is now immediate (rules (B.27h) and (B.27i)), at the expense of maintaining synchrony between the binding stacks and the control stack.

**34.3**. Preliminarily, it is important to think carefully about the interaction between the dynamics of fluids and of continuations. The correct behavior is defind by the deep dynamics given by Solution **34.1**; the bindings of the fluids are determined by the `put` frames present on the target control stack. So, for example, if one seizes a continuation $k$, then puts a binding for $a$ as an extension to $k$, then throws back to $k$, the binding for $a$ is implicitly restored to its state prior to the `put`. The putatively more efficient shallow dynamics given in Solution **34.2** requires a fairly complicated protocol to ensure that when a seized continuation is reactivated, the binding stacks of the active fluids are restored to their proper state.

The implementation of exception handling in terms of fluids and continuations is given as follows:

$$\begin{aligned}\texttt{handle}\,e_1\,\texttt{ow}\,x \hookrightarrow e_2 &\triangleq \texttt{letcc}\,\mathit{ret}\,\texttt{in}\,\texttt{let}\,x\,\texttt{be}\,(\texttt{letcc}\,k\,\texttt{in}\,\texttt{put}\,k\,\texttt{for}\,\mathit{hdlr}\,\texttt{in}\,\texttt{throw}\,e_1\,\texttt{to}\,\mathit{ret})\,\texttt{in}\,e_2\\ \texttt{raise}(e) &\triangleq \texttt{throw}\,e\,\texttt{to}\,(\texttt{get}\,\mathit{hdlr})\end{aligned}$$

The associated type of the fluid *hdlr* is $\tau_{\mathsf{exn}}\,\texttt{cont}$. The implementation of `handle` restores the proper handler on both normal and exceptional return precisely because fluid binding is defined to behave as specified by the deep dynamics even in the presence of first-class continuations.

Ironically, the implementation of exceptions in terms of fluids is useful only insofar as a shallow dynamics is used, so that *hdlr* may be accessed directly, but the overhead that would be required to implement shallow binding in the presence of continuations is likely sufficient to render it ineffective as a means of implementing exceptions. As is often the case, the semantics serves here as a useful guide to implementation.

Preview

# Chapter 35

**35.1**. The implementation of named exception handling is given as follows:

$$\texttt{exception}\, a\, \texttt{of}\, \tau\, \texttt{in}\, e \triangleq \texttt{new}\, a \sim \tau\, \texttt{in}\, e$$
$$\texttt{raise}\, a\, \texttt{with}\, e \triangleq \texttt{raise}(a \cdot e)$$
$$\texttt{try}\, e\, \texttt{ow}\, a_1(x_1) \hookrightarrow e_1 \mid \ldots \mid a_n(x_n) \hookrightarrow e_n \mid x \hookrightarrow e' \triangleq$$
$$\texttt{handle}\, e\, \texttt{ow}\, x \hookrightarrow \texttt{match}\, x\, \texttt{as}\, a_1 \cdot x_1 \hookrightarrow e_1\, \texttt{ow} \hookrightarrow \texttt{match}\, x\, \texttt{as}\, a_n \cdot x_n \hookrightarrow e_n\, \texttt{ow} \hookrightarrow e'$$

Observe that what are called exceptions in a named exception mechanism are just dynamically allocated classes.

**35.2**. (a) One may readily reduce the halting problem to the question of whether a given expression does or does not raise a particular exception. Exact exception tracking is therefore unachievable by computable means.

(b) Consider the expression

$$\texttt{exception}\, a\, \texttt{of}\, \texttt{unit}\, \texttt{in}\, \texttt{raise}\, a\, \texttt{with}\, \langle\rangle,$$

which allocates an exception name, then raises that exception name out of its scope. No finite set of exceptions for the whole expression can involve the name $a$, which is scoped within that expression. Yet the expression can raise that very exception, so the stated upper bound is not accurate.

# Chapter 36

**36.1**. To add array assignables to **RMA** proceed as follows:

- Introduce a command to declare an array $A$, $\texttt{dcl}\, A[e_1] := e_2\, \texttt{in}\, m$, where $e_1 : \texttt{nat}$, $e_2 : \texttt{nat} \rightharpoonup \tau$, and $m \div \rho$. The function $e_2$ provides the initial values for the elements of the array.
- Introduce a family of commands $|A|$ indexed by array assignables $A$ that returns the number of elements of $A$.
- Introduce families of commands $A[i]$ and $A[i] := e$ that, respectively, retrieve the $i$th element of $A$ and update the $i$th element of $A$ with a new value given by $e$. It is, of course, a fatal error to exceed the size bound of an array assignable.
- The memory $\mu$ maps scalar assignables to scalar values, and it maps array assignables to a size $n$ and a length-$n$ sequence $s$ of scalar values.

Preview

**36.2**. By the miracle of $\alpha$-conversion each recursive call allocates a *fresh* version of $a$, which is implicitly renamed to, say, $a'$, before being entered into the signature as a new assignable. The body of the procedure is correspondingly renamed so that $a$ becomes $a'$ wherever it occurs, so that there can be no confusion among the multiple active instances of the "same" assignable declaration.

**36.3**. The procedure declaration with own assignables

$$\texttt{proc}\,p(x:\tau):\rho\,\texttt{is}\,\{\texttt{own}\,a:=e\,\texttt{in}\,m\}\,\texttt{in}\,m'$$

is short-hand for the composite command

$$\texttt{dcl}\,a:=e\,\texttt{in}\,\texttt{proc}\,p(x:\tau):\rho\,\texttt{is}\,m\,\texttt{in}\,m',$$

where

$$\texttt{proc}\,p(x:\tau):\rho\,\texttt{is}\,m\,\texttt{in}\,m'$$

is itself short-hand for

$$\texttt{bnd}\,p\hookleftarrow\texttt{cmd}(\texttt{ret}(\texttt{fix}\,p\,\texttt{is}\,\lambda\,(x{:}\tau)\,\texttt{cmd}(m)))\,;m'.$$

Thus, within the scope of $p$, the assignable $a$, which is private to the body of $p$, maintains its state across calls. Were the `own` declaration replaced by an ordinary declaration, each call would create a fresh instance of $a$, with no retention of state across calls.

**36.4**. Besides the tedium of threading the memory through each step of expression evaluation, you need only add one rule to account for assignables as values:

$$\frac{}{a\parallel\mu\otimes a\hookleftarrow e\xmapsto[\Sigma,a\sim\tau]{}\texttt{ret}(e)\parallel\mu\otimes a\hookleftarrow e} \tag{B.28}$$

Final states are defined by the rule

$$\frac{e\ \mathsf{val}}{e\parallel\mu\ \mathsf{final}} \tag{B.29}$$

You may then prove memory invariance by induction on the revised rules, noting that no rule, include this one, alters the memory across a transition.

The class of passive commands may be isolated by a judgment $m$ passive whose definition need not be given here. The statics for a passive block is given by the following rule:

$$\frac{\Gamma\vdash_\Sigma m\div\tau}{\Gamma\vdash_\Sigma\texttt{do}\,\{m\}:\tau}\ . \tag{B.30}$$

The dynamics of a passive block is as follows:

$$\frac{m\parallel\mu\underset{\Sigma}{\mapsto}m'\parallel\mu'}{\texttt{do}\,\{m\}\parallel\mu\underset{\Sigma}{\mapsto}m\parallel\mu'}\ . \tag{B.31}$$



Preview

$$\frac{e \parallel \mu \text{ final}}{\texttt{do}\,\{\texttt{ret}\,e\} \parallel \mu \underset{\Sigma}{\mapsto} e \parallel \mu}\,. \tag{B.32}$$

It should be clear that passive commands enjoy memory invariance, so this property extends to expressions that involve passive blocks.

The possibility of rendering an expression involving assignables in many different ways exposes its fundamental weakness as a language design principle: it is not predictable by the programmer when or how often the memory will be accessed during evaluation of an expression. For **MA** itself there is no significant difference, apart from the overhead required to access memory, but in a concurrent setting (see Chapter 43), the difference is vitally important, because the contents of an assignable may change between accesses in a single passive block, let alone across such blocks.

**36.5**. First, introduce a *class* declaration command that takes an argument that is used to initialize the shared private state. Second, introduce a command to instantiate a class by providing the instance data as argument, and obtaining a tuple of procedures sharing private state in the manner of Solution **36.3**. The tuple of procedures is an object whose components may be called as ordinary procedures as described in the Chapter. The moral of the story is that with the right analysis of the fundamentals embedded in a sufficiently general substrate, one can create numerous useful programming methods without introducing a cult of $x$-oriented programming for various values of $x$.

**36.6**. To use a consolidated stack $k$, include frames of the form $\texttt{dcl}\,a := e\,\texttt{in} - $ in which $e$ represents the current contents of $a$, taking account of any modifications that may have been made to it since its declaration. To set $a$ to another value you must update the appropriate frame in $k$ to obtain a new stack $k'$ with which to proceed. To get the contents of $a$ you must traverse the stack looking for its declaration and its associated contents. The update and traversal can be optimized using imperative programming methods, at the expense of rendering the control stack to be an ephemeral data structure, creating complications for seizing stacks as continuations.

It is advantageous to separate the control stack from the memory, because it provides direct access to the current contents of an assignable, much as does the shallow binding dynamics of fluids. The control stack frame corresponding to an assignable declaration has the form $\texttt{dcl}\,a := -\,\texttt{in} - $, which has a hole for the binding as well as the body, so that it records the declaration of $a$, but not its contents. The contents of the active assignables is maintained in a separate memory, much as in the structural dynamics. The critical rules for managing the correlation between the declaration and the memory are as follows:

$$\frac{e \longmapsto^* e' \quad e' \text{ val}}{\mu \parallel k \rhd_\Sigma \texttt{dcl}\,a := e\,\texttt{in}\,m \longmapsto \mu \otimes \{a \hookrightarrow e'\} \parallel k;\texttt{dcl}\,a := -\,\texttt{in} - \rhd_{\Sigma,a\sim\tau} m} \tag{B.33a}$$

$$\frac{}{\mu \otimes \{a \hookrightarrow e'\} \parallel k;\texttt{dcl}\,a := -\,\texttt{in} - \lhd_\Sigma e \longmapsto \mu \parallel k \lhd_\Sigma e} \tag{B.33b}$$

Preview

On entry to a declaration of $a$ the memory is extended with a new location, $a$, whose contents is initialized to the value of $e$. On exit from the declaration that location is deallocated from the memory. This behavior exemplifies the stack-allocation of assignables in **MA**. Indeed, we may view the memory as a stack onto which are pushed and popped bindings for assignables that are not otherwise declared in the memory.

# Chapter 37

**37.1**. Define commands $*\,e_1[e_2]$ and $e_1[e_2] *= e_3$ by the following dynamics:

$$\frac{}{*\,(\&A)[i] \parallel \mu \longmapsto A[i] \parallel \mu} \tag{B.34a}$$

$$\frac{e\ \mathsf{val}}{(\&A)[i] *= e \parallel \mu \longmapsto A[i] := e \parallel \mu} \tag{B.34b}$$

**37.2**. The following recursive types each in turn satisfy the stated requirements:

(a) $(\mathtt{rec}\,t\,\mathtt{is}\,(\mathtt{nat} \times t)\,\mathtt{opt})\,\mathtt{ref}$.

(b) $\mathtt{rec}\,t\,\mathtt{is}\,((\mathtt{nat} \times t)\,\mathtt{opt})\,\mathtt{ref}$.

(c) $\mathtt{rec}\,t\,\mathtt{is}\,((\mathtt{nat} \times t)\,\mathtt{ref})\,\mathtt{opt}$.

(d) $\mathtt{rec}\,t\,\mathtt{is}\,(\mathtt{nat} \times (t\,\mathtt{ref}))\,\mathtt{opt}$.

(e) $\mathtt{rec}\,t\,\mathtt{is}\,((\mathtt{nat}\,\mathtt{ref}) \times (t\,\mathtt{ref}))\,\mathtt{opt}$.

(f) $\mathtt{rec}\,t\,\mathtt{is}\,(((\mathtt{nat}\,\mathtt{ref}) \times (t\,\mathtt{ref}))\,\mathtt{opt})\,\mathtt{ref}$.

As you can see, each of these definitions has legitimate claim to being a form of mutable linked list, but it is not at all obvious which is the canonical choice.

**37.3**. In the absence of sum types, other than their approximation using the "null pointer", one is left with the following representation of mutable linked lists:

$$\mathtt{rec}\,t\,\mathtt{is}\,\langle \mathtt{isempty} \hookrightarrow \mathtt{bool}\,\mathtt{ref}\,\mathtt{opt}, \mathtt{data} \hookrightarrow \mathtt{nat}\,\mathtt{ref}\,\mathtt{opt}, \mathtt{next} \hookrightarrow t\,\mathtt{ref}\,\mathtt{opt}\rangle\,\mathtt{ref}\,\mathtt{opt}.$$

The optional reference types account for the possibility of the "null pointer" wherever a reference is used. The `isempty` field is assignable independently of the `data` and `next` fields, which means that there is no inherent coordination between the value of this field and the presence or meaning of the other two. In particular one would expect that if the `isempty` field is `false` then the other two fields should not be `null`, but the type does not express this invariant. Notice, in particular, that the above type is not equivalent to any of the six candidates for the type of mutable linked lists given in Solution **37.2**!

Preview

# Chapter 38

**38.1**. Under the by-name dynamics given in Chapter 15 the value of $\omega$ is $\mathtt{s}(\omega)$, so that evaluation of $\mathtt{ifz}\,\omega\,\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}$ strips off the successor and evaluates $[\omega/x]e_1$. Should $\omega$ be evaluated again, the same process repeats, with each such evaluation recreating the successor of $\omega$. Under the by-need interpretation the value of $\omega$ is $\mathtt{s}(@\,a)$ for some cell $a$ containing $\mathtt{s}(@\,a)$. A conditional analysis once again strips the successor, and then evaluates $[@\,a/x]e_1$. Any further evaluation of the expression $\omega$ will once again result in the stored value $\mathtt{s}(@\,a)$, without recreating it. The memo table is self-referential, or circular, in that it contains a value that refers to its own memo cell, sharing the result once for all uses of $\omega$. This behavior is one of the attractions of by-need evaluation; suspension types provide the same benefts in an eager setting.

**38.2**. The expression

$$(\mathtt{fix}\,l\,\mathtt{is}\,\lambda\,(x{:}\mathtt{nat})\,\mathtt{fold}(\mathtt{cons}\cdot\langle x, l(\mathtt{s}(x))\rangle))(\mathtt{z})$$

has the required type

$$\mathtt{rec}\,t\,\mathtt{is}\,[\mathtt{nil} \hookrightarrow \mathtt{unit}, \mathtt{cons} \hookrightarrow \mathtt{nat} \times t].$$

**38.3**. The lazy interpretation in terms of suspensions is given by the following equations:

$$\begin{aligned}
\mathtt{unit} &\triangleq \mathtt{unit}\\
\tau_1 \times \tau_2 &\triangleq \widehat{\tau}_1 \times \widehat{\tau}_2\\
\mathtt{void} &\triangleq \mathtt{void}\,\mathtt{susp}\\
\tau_1 + \tau_2 &\triangleq (\widehat{\tau}_1 + \widehat{\tau}_2)\,\mathtt{susp}\\
\mathtt{rec}\,t\,\mathtt{is}\,\tau &\triangleq \mathtt{rec}\,t\,\mathtt{is}\,(\tau\,\mathtt{susp}).
\end{aligned}$$

Notice the additional level of suspensions required for sum types that is not required for product types. This is justified by the requirement that, in the case of binary products, the summand not be determined until the value is required. The nullary case is similar: the suspended expression must diverge, but only when its value is requested. It is for this reason that sums are ill-behaved in lazy languages such as **LFPC**.

Preview

Preview

# Appendix C

# Errata

Apart from minor typographical errors, which have been corrected in place, the following errors have also been called to the author's attention:

1. The definition of the local transition judgment in Section 40.4 oversimplifies the treatment of the individual **PCF** instruction steps, which must account for the possibility of parallelism arising within a compound expression such as an application or case analysis. The simplest correction is to introduce a "unary" form of join to enforce sequencing of evaluation of subexpressions in **PCF**.

   For example, the rules for (by-name) function application would be expressed as the following local transitions:

$$\frac{}{\nu\, a \sim \tau\, \{\, a \hookrightarrow (\lambda\, (x{:}\tau_2)\, e)(e_2)\,\} \longmapsto_{loc} \nu\, a \sim \tau\, \{\, a \hookrightarrow [e_2/x]e\,\}} \tag{C.1a}$$

$$\frac{\neg\ e_1\ \mathsf{val}}{\left\{\begin{array}{c} \nu\, a \sim \tau\, \{\, a \hookrightarrow e_1(e_2)\,\} \\ \longmapsto_{loc} \\ \nu\, a_1 \sim \tau_2 \rightharpoonup \tau\, a \sim \tau\, \{\, a_1 \hookrightarrow e_1 \otimes a \hookrightarrow \mathtt{join}[a_1](x_1.x_1(e_2))\,\} \end{array}\right\}} \tag{C.1b}$$

   A "unary join" rule is required to implement the sequencing:

$$\frac{e_1\ \mathsf{val}}{\left\{\begin{array}{c} \nu\, a_1 \sim \tau_2 \rightharpoonup \tau\, a \sim \tau\, \{\, a_1 \hookrightarrow e_1 \otimes a \hookrightarrow \mathtt{join}[a_1](x_1.e)\,\} \\ \longmapsto_{loc} \\ \nu\, a \sim \tau\, \{\, a \hookrightarrow [e_1/x_1]e\,\} \end{array}\right\}} \tag{C.2}$$

2. In the printed edition of the book rule (42.10d) was inadvertently omitted.

Preview

3. Theorem 43.3 is incorrect as stated. A process of the form

$$\nu\, a \sim \mathtt{unit.proc}(\mathtt{emit}(a \cdot \langle\rangle))$$

is well-formed, but cannot make progress by even a labeled transition, because of the narrow scope of the channel (class) $a$. The statement must be corrected to allow for extrusion of the scopes of channels:

**Theorem C.1** (Progress)**.** *If $\vdash_\Sigma p$ proc, then either $p \equiv \mathbf{1}$, or $p \equiv \nu\, \Sigma'\{p'\}$ such that $p' \xmapsto[\Sigma\,\Sigma']{\alpha} p''$ for some $\vdash_{\Sigma\,\Sigma'} p''$ and some $\vdash_{\Sigma\,\Sigma'} \alpha$ action.*

4. In Chapters 48 and 50, if termination is to be derived as a consequence of the relational interpretation, then Kleene equivalence must be defined as in Chapter 49 to ensure that it is reflexive without assuming termination. Alternatively, termination may be proved separately using a predicate (unary relation) interpretation following along the same lines as the binary interpretation so that Kleene equivalence is ensured to be reflexive as defined in the two chapters mentioned.

5. The suggested strengthening of the inductive hypothesis in the proof sketch for Theorem 5.5 in the printed edition is not sufficient to establish the result. Instead one must assume, for each variable, if two expressions that evaluate to the same value are substituted on either side, then the stated equation is derivable. This ensures that, for the case of variables, the strengthened assumption covers the intended result directly, and the rest follows as described.

6. The definition of canonization given in Section 22.4 has been corrected to perform simplification at the root before neutralization. The terms "atomic" and "neutral" were used interchangeably; these have been consolidated to use just the term "neutral."

7. Chapter 19 lacked an account of class discriminators required to implement the extensions of **DPCF** discussed in Chapter 18.

8. The treatment of exceptions as encapsulatable effects given in Section 29.4 was needlessly confusing and overly complicated. It has been rewritten entirely.

Preview

# Bibliography


Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996. 143, 252, 258

Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 104–115, 2001. 423

Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, chapter C.7, pages 783–818. North-Holland, 1977. 21

John Allen. *Anatomy of LISP*. Computer Science Series. McGraw-Hill, 1978. 10, 329

S.F. Allen, M. Bickford, R.L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006. ISSN 1570-8683. doi: 10.1016/j.jal.2005.10.005. 87

Stuart Allen. A non-type-theoretic definition of Martin-Löf's types. In *LICS*, pages 215–221, 1987. 482

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3): 297–347, 1992. 383

Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3): 265–301, 1997. 375

Arvind, Rishiyur S. Nikhil, and Keshav Pingali. I-structures: Data structures for parallel computing. In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 336–369. Springer, 1986. ISBN 3-540-18420-1. 405

Arnon Avron. Simple consequence relations. *Information and Computation*, 92:105–139, 1991. 30

Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984. 153

Henk Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, *Computational Structures*. Oxford University Press, 1992. 39


Preview

Nick Benton and Andrew Kennedy. Exceptional syntax. *Journal of Functional Programming*, 11(4): 395–410, July 2001. 278

Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin, editors. *From Semantics to Computer Science: Essays in Honor of Gilles Kahn*. Cambridge University Press, 2009. 567

Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990. ISBN 0-262-02313-X. 398

Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *FPCA*, pages 226–237, 1995. 398

Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ICFP*, pages 213–225, 1996. 61, 398

Manuel Blum. On the size of machines. *Information and Control*, 11(3):257–265, September 1967.

Stephen D. Brookes. The essence of parallel algol. *Inf. Comput.*, 179(1):118–149, 2002. 434

Rod M. Burstall, David B. MacQueen, and Donald Sannella. Hope: An experimental applicative language. In *LISP Conference*, pages 136–143, 1980. 105

Samuel R. Buss, editor. *Handbook of Proof Theory*. Elsevier, Amsterdam, 1998. 564

Luca Cardelli. Structural subtyping and the notion of power type. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 70–79, 1988. 216

Luca Cardelli. Program fragments, linking, and modularization. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 266–277, 1997. 448

Giuseppe Castagna and Benjamin C. Pierce. Decidable bounded quantification. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 151–162, 1994. 459

Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941. 10, 71, 153

Alain Colmerauer and Philippe Roussel. The birth of Prolog. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, pages 37–52, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4. doi: http://doi.acm.org/10.1145/154766.155362. URL http://doi.acm.org/10.1145/154766.155362. 105

R. L. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986. iii, 229, 482

Robert L. Constable. Types in logic, mathematics, and programming. In Buss (1998), chapter X. iii

Robert L. Constable and Scott F. Smith. Partial objects in constructive type theory. In *LICS*, pages 183–193. IEEE Computer Society, 1987. 491

William R. Cook. On understanding data abstraction, revisited. In *OOPSLA*, pages 557–572, 2009. 143

Preview

Guy Cousineau and Michel Mauny. *The functional approach to programming*. Cambridge University Press, 1998. ISBN 978-0-521-57183-8. 105

Karl Crary. A syntactic account of singleton types via hereditary substitution. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '09, pages 21–29, New York, NY, USA, 2009. ACM. 241

Rowan Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University School of Computer Science, May 2005. Available as Technical Report CMU–CS–05–110. 229

Rowan Davies and Frank Pfenning. Intersection types and computational effects. In Martin Odersky and Philip Wadler, editors, *ICFP*, pages 198–208. ACM, 2000. ISBN 1-58113-202-6. 229

Ewen Denney. Refinement types for specification. In David Gries and Willem P. de Roever, editors, *PROCOMET*, volume 125 of *IFIP Conference Proceedings*, pages 148–166. Chapman & Hall, 1998. ISBN 0-412-83760-9. 229

Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 2005. 459

Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In Andrew D. Gordon, editor, *FoSSaCS*, volume 2620 of *Lecture Notes in Computer Science*, pages 250–266. Springer, 2003. ISBN 3-540-00897-7. 229

Uffe Engberg and Mogens Nielsen. A calculus of communicating systems with label passing - ten years after. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 599–622. The MIT Press, 2000. 423

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *TCS: Theoretical Computer Science*, 103, 1992. 48, 288

Tim Freeman and Frank Pfenning. Refinement types for ml. In David S. Wise, editor, *PLDI*, pages 268–277. ACM, 1991. ISBN 0-89791-428-7. 229

Daniel Friedman and David Wise. The impact of applicative programming on multiprocessing. In *International Conference on Parallel Processing*, 1976. 405

David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985. 434

Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–213. North-Holland, Amsterdam, 1969. 39

J.-Y. Girard. *Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur*. These d'etat, Universite Paris VII, 1972. 186, 504

Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1989. Translated by Paul Taylor and Yves Lafont. iii, 448



Preview

Kurt Gödel. On a hitherto unexploited extension of the finitary standpoint. *Journal of Philosphical Logic*, 9:133–142, 1980. Translated by Wilfrid Hodges and Bruce Watson. 79

Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979. 10, 278

John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Trans. Program. Lang. Syst.*, 21(2):240–285, 1999. 405

Timothy Griffin. A formulae-as-types notion of control. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 47–58, 1990. 311

Carl Gunter. *Semantics of Programming Languages*. Foundations of Computing Series. MIT Press, 1992. 143

Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985. 405

Robert Harper. Constructing type systems over an operational semantics. *J. Symb. Comput.*, 14(1): 71–84, 1992. 482

Robert Harper. A simplified account of polymorphic references. *Inf. Process. Lett.*, 51(4):201–206, 1994. 362

Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 123–137, 1994. 241, 459, 470

Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 341–354, 1990. 459

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40:194–204, 1993. 10, 30

Ralf Hinze and Johan Jeuring. Generic haskell: Practice and theory. In Roland Carl Backhouse and Jeremy Gibbons, editors, *Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2003. ISBN 3-540-20194-7. 113

C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. 423

Tony Hoare. Null references: The billion dollar mistake. Presentation at QCon 2009, August 2009. 95

Simon L. Peyton Jones. Haskell 98: Introduction. *J. Funct. Program.*, 13(1):0–6, 2003. 105

S. C. Kleene. *Introduction to Metamathematics*. van Nostrand, 1952. 10



Preview

Robert A. Kowalski. The early years of logic programming. *Commun. ACM*, 31:38–43, January 1988. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/35043.35046. URL http://doi.acm.org/10.1145/35043.35046. 105

Imre Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press, 1976. 311

P. J. Landin. A correspondence between Algol 60 and Church's lambda notation. *CACM*, 8:89–101; 158–165, 1965. 48, 269

Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ml. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 173–184, 2007. 459

Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 109–122, 1994. 241, 459, 470

Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 142–153, 1995. 470

Daniel R. Licata and Robert Harper. A monadic formalization of ml5. In Karl Crary and Marino Miculan, editors, *LFMTP*, volume 34 of *EPTCS*, pages 69–83, 2010. 442

Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, May 1997. 459

Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994. 258

Saunders MacLane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer-Verlag, second edition, 1998. 113, 121

David B. MacQueen. Using dependent types to express modular structure. In *Proc. ACM Symposium on Principles of Programming Languges*, pages 277–286, 1986. 459

David B. MacQueen. Kahn networks at the dawn of functional programming. In Bertot et al. (2009), chapter 5. 143

Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In Runciman and Shivers (2003), pages 213–225. ISBN 1-58113-756-7. 229

Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science IV*, pages 153–175. North-Holland, 1980. 39, 55

Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Unpublished Lecture Notes, 1983. 21, 30

Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, Naples, Italy, 1984. iii, 39, 448



Preview

Per Martin-Löf. Truth of a proposition, evidence of a judgement, validity of a proof. *Synthese*, 73 (3):407–420, 1987. 21, 30

John McCarthy. *LISP 1.5 Programmer's Manual*. MIT Press, 1965. 10, 162, 320

N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *LICS*, pages 30–36, 1987. 121

Robin Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978. 55

Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999. ISBN 978-0-521-65869-0. 320, 336, 423, 511

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. 39, 61, 105, 216, 241, 336, 459, 470

John C. Mitchell. Coercion and type inference. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 175–185, 1984. 216

John C. Mitchell. Representation independence and data abstraction. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 263–276, 1986. 194, 504

John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996. iii

John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988. 193, 470

Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23. IEEE Computer Society, 1989. ISBN 0-8186-1954-6. 278

Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *LICS*, pages 286–295, 2004. 350, 442

Chetan R. Murthy. An evaluation semantics for classical proofs. In *LICS*, pages 96–107. IEEE Computer Society, 1991. 311

Aleksandar Nanevski. From dynamic binding to state via modal possibility. In *PPDP*, pages 207–218. ACM, 2003. ISBN 1-58113-705-2. 329

R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994. 10, 30

OCaml. Ocaml, 2012. URL `http://caml.inria.fr/ocaml/`. 470

David Michael Ritchie Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981. ISBN 3-540-10576-X. 511

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001. 278



Preview

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. iii, 87, 216, 252, 258

Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. iii

Andrew M. Pitts. Existential types: Logical relations and operational equivalence. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer, 1998. ISBN 3-540-64781-3. 504

Andrew M. Pitts. Operational semantics and program equivalence. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 378–412. Springer, 2000. ISBN 3-540-44044-5. 491

Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or what's new? In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *MFCS*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer, 1993. ISBN 3-540-57182-5. 320

G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University Computer Science Department, 1981. 48, 269

Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977. 135

Gordon D. Plotkin. The origins of structural operational semantics. *J. of Logic and Algebraic Programming*, 60:3–15, 2004. 48

John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999. 424, 434

J. C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, Amsterdam, 1983. 186, 504

John C. Reynolds. Towards a theory of type structure. In Bernard Robinet, editor, *Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974. ISBN 3-540-06859-7. 186, 193

John C. Reynolds. Using category theory to design implicit conversions and generic operators. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258. Springer, 1980. ISBN 3-540-10250-7. 216

John C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981. 349, 362

John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993. 288

John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, Cambridge, England, 1998. iii



Preview

Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In Andrew Kennedy and Nick Benton, editors, *TLDI*, pages 89–102. ACM, 2010. ISBN 978-1-60558-891-9. 459

Colin Runciman and Olin Shivers, editors. *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, 2003. ACM. ISBN 1-58113-756-7. 567, 570

Dana Scott. Lambda calculus: Some models, some philosophy. In J. Barwise, H. J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, pages 223–265. North Holland, Amsterdam, 1980. 153, 171

Dana S. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3):522–587, 1976. 143

Dana S. Scott. Domains for denotational semantics. In Mogens Nielsen and Erik Meineche Schmidt, editors, *ICALP*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer, 1982. ISBN 3-540-11576-5. 143

Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4):761–783, 1982. 143

Richard Statman. Logical relations and the typed lambda-calculus. *Information and Control*, 65 (2/3):85–97, 1985. 482

Guy L. Steele. *Common Lisp: The Language*. Digital Press, 2nd edition edition, 1990. 278

Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Trans. Comput. Log.*, 7(4):676–722, 2006. 241, 459

Paul Taylor. *Practical Foundations of Mathematics*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, Cambridge, 1999. 121

P.W. Trinder, K. Hammond, H.-W. Loidl, and S.L. Peyton Jones. Algorithm + strategy = parallelism. *JOURNAL OF FUNCTIONAL PROGRAMMING*, 8:23–60, 1998. 405

David Turner. An overview of Miranda. *Bulletin of the EATCS*, 33:103–114, 1987. 105

Jaap van Oosten. Realizability: A historical essay. *Mathematical Structures in Computer Science*, 12 (3):239–263, 2002. 229

Philip Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989. 186

Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992. 278

Philip Wadler. Call-by-value is dual to call-by-name. In Runciman and Shivers (2003), pages 189–201. ISBN 1-58113-756-7. 311

Mitchell Wand. Fixed-point constructions in order-enriched categories. *Theor. Comput. Sci.*, 8:13–30, 1979. 143



Preview

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. Specifying properties of concurrent computations in clf. *Electr. Notes Theor. Comput. Sci.*, 199:67–87, 2008. 203

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994. 55, 362

Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Jack W. Davidson, Keith D. Cooper, and A. Michael Berman, editors, *PLDI*, pages 249–257. ACM, 1998. ISBN 0-89791-987-4. 229

Noam Zeilberger. On the unity of duality. *Ann. Pure Appl. Logic*, 153(1-3):66–96, 2008. 383

Preview

# Index


**FPC**, *see* recursive types
**M**, *see* inductive types, coinductive types
**Λ**, *see* untyped $\lambda$-calculus
**F**, *see* universal types
**MA**, *see* Modernized Algol
**PCF**, *see* Plotkin's **PCF**
**T**, *see* Gödel's **T**

abstract binding tree, 3, 6
  abstractor, 7
    valence, 7
  $\alpha$-equivalence, 8
  bound variable, 8
  capture, 9
  de Bruijn form, 517
  free variable, 8
  identification convention, 9
  operator, 7
    arity, 7
    parameter, 10
  structural induction, 8
  substitution, 9
abstract syntax tree, 3, 4
  operator, 3
    arity, 3
    index, 5
    parameter, 5
  structural induction, 5
  substitution, 5
  variable, 3
abstract types, *see* existential types, *see also* signatures
abt, *see* abstract binding tree
assignables, *see* Modernized Algol
ast, *see* abstract syntax tree

backpatching, *see* references
benign effects, *see* references
bidirectional typing, 39, 520
boolean blindness, *see* type refinements
boolean type, 92

call-by-need, *see* laziness
capabilities, 353
channel types, *see* Concurrent Algol
class types, 333
  definability, 334
  dynamics, 333
  statics, 333
classes, *see* dynamic dispatch
classical logic, 301
  classical laws, 312, 551
  contradiction, 302, 304
  derivability of elimination forms, 306
  double-negation translation, 310, 312, 551
  dynamics, 307
  excluded middle, 309
  judgments, 302
  proof, 304
    conjunction, 305
    disjunction, 305
    implication, 305
    negation, 305
    truth, 305
    variable, 305
  provability, 302
    conjunction, 303
    disjunction, 303
    hypothesis, 302
    implication, 303
    negation, 303


Preview

truth, 303
refutability, 302
conjunction, 303
disjunction, 303
falsehood, 303
hypothesis, 302
implication, 303
negation, 303
refutation, 304
conjunction, 305
disjunction, 305
falsehood, 305
implication, 305
negation, 305
variable, 305
safety, 308
classified type, 331
applications, 335
confidentiality, 335
dynamics, 332
exception mechanism, 555
exception type, 555
exception values, 336
integrity, 335
named exceptions, 337
safety, 333
statics, 332
coinductive types
dynamics, 120
infinite trees, 122, 529
statics, 119
streams, 116
combinators, 30, 520
command types, *see* Modernized Algol
compactness, *see* equality
Concurrent Algol, 425
broadcast communication, 428
dynamics, 428
safety, 429
statics, 428
definability of free assignables, 433
dynamics, 426
selective communication, 430
dynamics, 432
statics, 430, 431
statics, 426
constructive logic, 291
Boolean algebra, 299, 551
conservation of proof, 296
decidable proposition, 298, 550
double-negation elimination, 298, 550
Gentzen's Principle, 296
Heyting algebra, 298, 551
judgment forms, 292
law of the excluded middle, 298, 550
proof, 295
conjunction, 295
disjunction, 296
falsehood, 296
implication, 295
truth, 295
proofs-as-programs, 297
propositions-as-types, 297
provability, 293
conjunction, 294
disjunction, 294
falsehood, 294
implication, 294
negation, 294
truth, 293
reversability of proof, 296
semantics, 292
constructors, 195, 196
canonical, 196, 197
canonization, 201, 202
canonizing substitution, 196, 199
formation rules, 197
general form, 201
neutral, 196, 197
neutralization, 201, 202
simplification, 201, 202
continuation types, 281
coroutines, 284
dynamics, 283
safety, 284
statics, 283
syntax, 283
contravariance, *see* subtyping


Preview

covariance, *see* subtyping

definitional equality, *see* equality
Distributed Algol, 435
  dynamics, 437
  safety, 438
  situated types, 439
    mobility, 441
    statics, 439, 441
  statics, 435
dynamc types
  destructors, 160
dynamic binding, *see* fluids
dynamic classification, *see* classified type
dynamic dispatch, 245, 246
  class-based, 247
    class vector, 247
    instance, 248
    message send, 248
    object type, 247, 248
    self-reference, 251
  dispatch matrix, 246
  message not understood, 252, 542, 544
  method-based, 247
    message send, 250
    method vector, 249
    object type, 249
    self-reference, 251
  self-reference, 250, 253, 544
    instances, 253, 545
    results, 253, 545
dynamic types, 156
  arithmetic, 534
  as static types, 167
  class dispatch, 160
  cons, 159
  critique, 161
  dynamics, 156
  lists, 162, 534
  multiple arguments, 162, 534
  multiple results, 162, 534
  nil, 159
  numeric classes, 159
  pairs, 162, 534
  predicates, 160
  safety, 158
  statics, 156
  subtyping, 210
dynamic typing
  *vs* static typing, 170
dynamics, 35, 41
  checked errors, 53
  contextual rules, 44
  cost rules, 60
  definitional equality, 46
  determinacy, 44
  environemental evaluation rules, 62
  equational rules, 46
  equivalence theorem, 45
  evaluation context, 44
  evaluation rules, 57
    equivalence to transition rules, 58
  induction on transition, 42
  inversion principle, 44
  structural rules, 42
  transition system, 41
  unchecked errors, 53
dynamics types
  arithmetic, 162

enumeration types, 93
equality, 473
  coinduction, 476, 489
  compactness, 484, 487, 488
  congruence, 476
  contexts, 474, 483
  definitional, 46, 131, 147, 179, 234, 473
  equational laws, 480
  fixed point induction, 484
  Kleene equality, 483
  Kleene equivalence, 475
  logical equivalence, 473, 477, 478, 484
    closed, 477
  observation, 474
  observational equivalence, 473, 475, 478, 483
  symbolic evaluation, 48
event types, *see* Concurrent Algol
exceptions, 271, 273

Preview

dynamics, 273
encapsulation, 279
exception type, 274, 276, 337
modal separation, 276
safety, 274, 279
statics, 273
syntax, 273
type refinements, 279
existential types, 187
coinductive types, 194, 538
definability from universals, 191
dynamics, 188
modeling data abstraction, 189
parametricity, 194, 538
representation independence, 503
representation independence, 191, 194, 538
safety, 189
statics, 188
streams, 194, 538
subtyping, 214

failures, *see also* exceptions, 271
dynamics, 272
failure-passing style, 279
safety, 273
statics, 271
finite function, 515
fixed point induction, *see* equality
fluid binding, *see* fluids
fluid reference types, 327
dynamics, 328
statics, 328
fluids, 323
deep binding, 329, 552
dynamics, 324
exception handling, 329, 554
safety, 325
shallow binding, 329, 553
statics, 323
subtleties, 326
function types
closures, 523
definitions, 66
dynamic binding, 70
explicit substitution, 523
first order, 66
dynamics, 66
safety, 67
statics, 66
first-class functions, 67
higher order, 67
dynamics, 68
safety, 68
statics, 67
second class functions, 66
static binding, 70
subtyping, 211
functors, *see* signatures
future types, 399, 400
parallel dynamics, 402
pipelining, 404
sequential dynamics, 400
sparks, 405
statics, 400
futures, *see* future types

Gödel's **T**, 73
definability, 76
definitional equality, 76
dynamics, 74
hereditary termination, 79, 524
iterator, 74
recursor, 73
safety, 75
statics, 74
undefinability, 77
general judgment, 23, 27
generic derivability, 28
proliferation, 28
structurality, 28
substitution, 28
parametric derivability, 28
general recursion, 129
generic inductive definition, 29
formal generic judgment, 29
rule, 29
rule induction, 29
structurality, 29


Preview

Girard's System **F**, *see* universal types

hybrid types, 165
  as recursive types, 167
  dynamics, 166
  optimization of dynamic types, 168
  safety, 166
  statics, 165
hypothetical inductive definition, 26
  formal derivability, 27
  rule, 26
    uniform, 27
  rule induction, 27
hypothetical judgment, 23
  admissibility, 25
    reflexivity, 26
    structurality, 26
    transitivity, 26
    weakening, 26
  derivability, 23
    reflexivity, 24
    stability, 24
    structurality, 24
    transitivity, 24
    weakening, 24

inductive definition, 13, 14
  backward chaining, 16
  derivation, 15
  forward chaining, 16
  function, 19
  iterated, 18
  rule, 14
    admissible, 25
    axiom, 14
    conclusion, 14
    derivable, 23
    premise, 14
  rule induction, 15, 16
  rule scheme, 14
    instance, 14
  simultaneous, 18
inductive types
  dynamics, 120
  lists, 122, 529
  natural numbers, 115
  statics, 119
inheritance, 255
  class extension, 255
  class-based, 256
  method extension, 256
  method specialization, 259, 546
  method-based, 257
  self-reference, 259, 546
  simple method override, 259, 546
  subclass, 255
  submethod, 255
  superclass, 255
  supermethod, 255
interface, *see* separate compilation

judgment, 13
  mode, 20
judgment form, 13
  predicate, 13
  subject, 13

kinds, 195, 196
  dependent, *see* singleton kinds, Σ kinds, Π kinds
  function kinds, 196
  higher kinds, 198
  product kinds, 196
  type kind, 196
Kleene equality, *see* equality

laziness, 367
  by-need general recursion, 370
  data structures, 376
  dovetailing, 530
  Gustav's function, 105, 526
  lists, 376, 559
  parallel or, 530
  parallel or, 136
  pattern matching, 105, 526
  recursion, 375, 559
  safety, 371
  sequentiality, 530

Preview

suspension interpretation, 376, 559
suspension types, 373
linking, *see* separate compilation
logical equivalence, *see* equality

methods, *see* dynamic dispatch
mobile types, 348
mobility condition, 348
rules, 348
Modernized Algol
multiple declaration instances, 556
separated and consolidated stacks, 352
Modernized Algol, 341
arrays, 350, 555
assignables, 341, 354
block structure, 344
classes and objects, 352, 557
command types, 347
commands, 341, 347
control stack, 352
control stack, 557
data stack, 352, 557
expressions, 341
free assignables, 356
free dynamics, 357
idioms
conditionals, 346
iteration, 346
procedures, 346
sequential composition, 346
multiple declaration instances, 350
own assignables, 351, 556
passive commands, 350, 556
recursive procedures, 350, 556
scoped dynamics, 343
scoped safety, 345
separated and consolidated stacks, 557
stack discipline, 344
stack machine, 352, 557
statics, 342, 348
modules, *see* signatures
mutual primitive recursion, 86

nested parallelism, *see* parallelism
null, *see* option types

objects, *see* dynamic dispatch
observational equivalence, *see* equality
option types, 94

parallelism, 387
binary fork-join, 387
Brent's Theorem, 395
cost dynamics, 390
cost dynamics *vs.* transition dynamics, 392
cost graphs, 390
implicit parallelism theorem, 390
multiple fork-join, 393
parallel complexity, 391
parallelizability, 395
provably efficient implementation, 395
sequence types, 393
cost dynamics, 394
statics, 394
sequential and parallel dynamics, 388
sequential complexity, 391
task dynamics, 396
work *vs.* depth, 391
parameterized modules, *see* signatures
parametricity, *see* equality
patterns, 98
constraints, 101
dual, 101
entailment, 102
satisfaction, 102
dynamics, 99
exhaustiveness, 101, 102
Gustav's function, 105, 526
laziness, 105, 526
natural numbers, 105
redundancy, 101, 102
statics, 98
phase distinction, 35, 195, *see also* signatures
Π kinds, 234, 237
elimination rules, 239
equivalence, 239
formation rules, 239
introduction rules, 239


Preview

subkinding, 239
Plotkin's **PCF**, 127
Blum size theorem, 135
bounded recursion, 484
definability, 132
definitional equality, 131
dynamics, 130
eager natural numbers, 133
eagerness and laziness, 133
halting problem, 135, 530
induction, 133
metacircular interpreter, 530
mutual recursion, 135, 529
safety, 130
statics, 129
totality and partiality, 134
polarization, 87, 377, 524
dynamics, 381
focusing, 379
negative types, 87, 378, 524
positive types, 87, 378, 524
safety, 382
statics, 379
polymorphic types, *see* universal types
primitive recursion, 86
process calculus, 409, 505
actions, 409
asynchronous communication, 419
bisimilarity, 505
channel types, 420
dynamics, 422
statics, 421
channels, 414, 417
coinduction, *see* strong and weak bisimilarity
concurrent composition, 411
dynamics, 412, 416, 418, 506
equivalence, *see* bisimilarity
events, 409
replication, 413
statics, 415, 417, 506
strong bisimilarity, 507, 508
strong bisumulation, 507
structural congruence, 410, 411
synchronization, 411
synchronous communication, 417
syntax, 505
universality, 422
weak bisimilarity, 510
coinduction, 511
weak bisimulation, 510
process equivalence, *see* process calculus
product types, 83
dynamics, 84
finite, 85
safety, 84
statics, 83
subtyping, 209, 211

recursive types, *see* type recursion, 137
dynamics, 138
eager data structures, 139
eager lists, 139
eager natural numbers, 139
lazy data structures, 139
lazy lists, 140
lazy natural numbers, 139
RS latch, 144
self-reference, 140
signals, 144
statics, 138
subtyping, 212, 217, 539
reference types, 353
aliasing, 356
free dynamics, 357
safety, 355, 358
scoped dynamics, 355
statics, 355
references
arrays, 362, 558
backpatching, 361
benign effects, 360
linked lists, 363, 558
mutable data structures, 362, 558
refinement types, *see* type refinements
refinements, *see* type refinements
representation independence, *see* existential types


Preview

representation indepndence, *see also* parametricity
Reynolds's Algol, *see* Modernized Algol

safety
  evaluation rules, 59, 60
scoped assignables, *see* Modernized Algol
self types, 141
  as recursive types, 141
  deriving general recursion, 142
  self-reference, 141
  unrolling, 141
separate compilation, 445
  initialization, 447
  interface, 445
  linking, 445
  units, 445
Σ kinds, 234, 237
  elimination rules, 238
  equivalence, 238
  formation rules, 238
  introduction rules, 238
  subkinding, 238
signatures, 449
  applicative functor, 469
  ascription, *see* sealing
  avoidance problem, 456
  dynamic part, 451
  dynamics, 457
  first- *vs* second-class, 458
  functors, 464, 465
  generative functor, 467
  hierarchies, 461, 462
  instances, 451
  modification, 465
  opacity, 450
  parameterization, 464
  parameterized modules, *see* functors
  principal signature, 452
  revelation, 450
  sealing, 450
  self-recognition, 457, 468
  sharing propagation, 462
  sharing specification, 462
  static part, 451
  statics, 455, 467
  structures, 450
  submodule, 463
  subsignature, 451, 453, 454
  syntax, 454, 467
  translucency, 450
  transparency, 450
  type abstractions, 449, 450
  type classes, 449, 451
  views, 451
singleton kinds, 234, 235
  as type definitions, 236
  constructor equivalence, 235
  higher singletons, 234, 240
  kind equivalence, 235
  kind formation, 235
  subkinding, 235
situated types, *see* Distributed Algol
sparks, *see* future types
speculation types, 401
  parallel dynamics, 402
  sequential dynamics, 401
  statics, 401
speculations, *see* speculation types
stack machine, 263
  correctness, 266
    completeness, 267
    soundness, 268
    unraveling, 268
  dynamics, 264
  frame, 263
  safety, 265
  stack, 263
  state, 263
state, 142
  from recursion, 142
  RS latch, 142
statics, 35
  canonical forms, 38
  decomposition, 38
  induction on typing, 37
  introduction and elimination, 38
  structurality, 37

Preview

substitution, 38
type system, 36
unicity, 37
weakening, 37
subkinding, 234
Π kinds, 239
Σ kinds, 238
singleton kinds, 235
submodules, *see* signatures
subtyping, 207
behavioral *vs* structural, 219
bounded quantification, 214
class types, 210
coercion, 217, 540
coherence, 217
dynamic types, 210
dynamics, 215
function types, 211
numeric types, 208
product types, 209, 211
quantified types, 214
recursive types, 212, 217, 539
safety, 215
subsumption, 207
sum types, 209, 211
variance, 211, 217, 539
sum types, 89
dynamics, 90
finite, 91
statics, 89
subtyping, 209, 211
symbol reference, *see* symbol types
symbol types, 318
association lists, 320, 552
dynamics, 319
equality, 320
safety, 319
statics, 318
symbolic expressions, 321, 552
symbols, 315
equality, 552
mobility, 317
modal separation, 321, 552
safety, 317, 318
scope-free dynamics, 317
scoped dynamics, 316
statics, 316
total ordering, 320, 552
syntax, 3
abstract, 3
binding, 3
chart, 35
concrete, 3
structural, 3
surface, 3
System **F**, *see* universal types

type abstractions, *see also* existential types,signatures
type classes, *see* signatures
type constructors, *see* constructors
type operators, 110
composition, 113, 527
database programming, 113, 527
generic extension, 110, 112
polynomial, 110
positive, 112
identity, 113, 527
non-negative, 114, 527
generic extension, 114, 528
polynomial, 110
positive, 112
type recursion, *see* recursive types
type refinements, 219
addition refinement, 230, 542
boolean blindness, 226
booleans *vs.* propositions, 226
conjunction covariance, 230, 540
dynamic type, 223, 230, 542
entailment, 221
error-freedom, 229
examples, 225
function types, 225
general satisfaction, 222
natural numbers, 225
optimization of dynamic typing, 231
pre-order, 221
preservation, 229
product types, 224


Preview

recursive types, 230, 540
refined canonical forms, 228
refinements *vs.* types, 219
safety, 228
safety theorem, 229
satisfaction, 221, 222
sum types, 224, 230, 541
summand refinements, 230, 541
syntax, 220
types *vs.* refinements, 229
underlying type, 220
Yoneda Lemma, 230, 540
type safety, 51
canonical forms, 52
checked errors, 54
preservation, 51
progress, 52

unit
dynamics, 84
statics, 83
unit type, 83
*vs* void type, 92
units, *see* separate compilation
unityped $\lambda$-calculus, 152
as untyped, 152
universal types, 176
abstraction theorem, 498
admissible relation, 496
Church numerals, 181
definability, 179
booleans, 186, 537
inductive types, 186, 537
lists, 186, 537
natural numbers, 180
products, 179
sums, 179
definitional equality, 179
dynamics, 178
equivalence candidate, *see* admissible relation
function extensionality, 499
impredicativity, 183
Kleene equality, 494
logical equivalence, 495
closed, 496
compositionality, 498
open, 497
observational equivalence, 494
parametricity, 181, 186, 493, *see* equality, 500, 503, 538
parametricity theorem, 498
predicative fragment, 182
predicativity, 183
prenex fragment, 183
rank-restricted Church numerals, 186, 538
rank-restricted fragments, 185
safety, 178
statics, 176
subtyping, 214
untyped $\lambda$-calculus, 147
Y combinator, 150
as unityped, 152
booleans, 153, 533
bracket abstraction, 154
Church numerals, 149
definability, 148
definitional equality, 147
dynamics, 147
lists, 154, 533
products, 153, 532
Scott's Theorem, 150
statics, 147
streams, 154, 533
sums, 154, 533

variance, *see* subtyping
void type, 89
*vs* unit type, 92
dynamics, 90
statics, 89


Preview

Preview

# Revision History

| Revision | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | 21.01.11 | RH | Created |
| 1.1 | 21.01.11 | RH | Concurrent and Distributed Algol |
| 1.2 | 03.02.11 | RH | Clarified discussion of rule induction. |
| 1.3 | 06.23.11 | RH | Revamped notation in equational reasoning chapter; miscellaneous corrections and improvements throughout. |
| 1.4 | 06.28.2011 | RH | Minor corrections to inductive definitions and type safety chapters. |
| 1.5 | 07.19.2011 | RH | Revamped treatment of syntactic objects. |
| 1.6 | 07.20.2011 | RH | Reorganized Part I. |
| 1.7 | 07.21.2011 | RH | Moved strings to Part I from Part II. |
| 1.8 | 07.22.2011 | RH | Remove exercises, add end notes. |
| 1.9 | 08.09.2011 | RH | Revise formulation and presentation of Concurrent and Distributed Algol. |
| 1.10 | 08.12.2011 | RH | Added chapter on process equivalence. |
| 1.11 | 08.15.2011 | RH | Disequality of classes for dynamic classification. |
| 1.12 | 08.18.2011 | RH | Revised distributed Algol chapter. |
| 1.13 | 08.22.2011 | RH | Units and linking. |
| 1.14 | 08.23.2011 | RH | Canonization of constructors. |
| 1.15 | 08.24.2011 | RH | Singleton kinds, bounded quantification. |
| 1.16 | 08.27.2011 | RH | Modules. |
| 1.17 | 09.13.2011 | RH | Citations and bibiography. |
| 1.18 | 09.19.2011 | RH | Module hierarchies and parameterization. |
| 1.19 | 10.03.2011 | RH | Polarization, classical logic. |
| 1.20 | 10.15.2011 | RH | Revise dynamic typing. |
| 1.21 | 11.04.2011 | RH | Index |
| 1.22 | 12.25.2011 | RH | Revised modularity chapters. |
| 1.23 | 01.03.2012 | RH | Appendix. |
| 1.24 | 01.18.2012 | RH | Revised notational conventions for dynamic languages. |

Preview

| | | | |
|---|---|---|---|
| 1.25 | 01.30.2012 | RH | Dynamic and hybrid languages. |
| 1.26 | 02.06.2012 | RH | Inheritance. |
| 1.27 | 02.10.2012 | RH | Typographical errors. |
| 1.28 | 02.29.2012 | RH | Inheritance rewrite. |
| 1.29 | 03.03.2012 | RH | Representation independence. |
| 1.30 | 03.15.2012 | RH | Recursive types. |
| 1.31 | 04.30.2012 | RH | Release. |
| 1.32 | 05.15.2012 | RH | Typographical errors. |
| 1.33 | 05.07.2013 | RH | Errata. |
| 1.34 | 11.10.2013 | RH | Errata. |
| 1.35 | 01.09.2014 | RH | Correct canonization error. |
| 1.36 | 01.10.2014 | RH | Dynamic PCF notational changes. |
| 1.37 | 01.11.2014 | RH | Hybrid PCF notational changes and discriminators. |
| 1.38 | 01.20.2014 | RH | Discussion of subtyping for dynamic types. |
| 1.39 | 01.21.2014 | RH | New chapter on refinement types. Miscellaneous clarifications. |
| 1.40 | 02.05.2014 | RH | Add term “abstraction theorem” |
| 1.41 | 02.26.2014 | RH | Add exercises and solutions |
| 1.42 | 03.17.2014 | RH | Re-organize T/PCF, rename langs for second edition. |
| 1.43 | 04.15.2014 | RH | More exercises |

Preview