Practical Foundations for Programming Languages

Robert Harper
Carnegie Mellon University

Spring, 2011

[Version 1.2 of 03.02.11.]

Preface

This is a working draft of a book on the foundations of programming languages. The central organizing principle of the book is that programming language features may be seen as manifestations of an underlying type structure that governs its syntax and semantics. The emphasis, therefore, is on the concept of *type*, which codifies and organizes the computational universe in much the same way that the concept of *set* may be seen as an organizing principle for the mathematical universe. The purpose of this book is to explain this remark.

This is very much a work in progress, with major revisions made nearly every day. This means that there may be internal inconsistencies as revisions to one part of the book invalidate material at another part. Please bear this in mind!

Corrections, comments, and suggestions are most welcome, and should be sent to the author at `rwh@cs.cmu.edu`. I am grateful to the following people for their comments, corrections, and suggestions to various versions of this book: Arbob Ahmad, Andrew Appel, Zena Ariola, Guy E. Blelloch, William Byrd, Luca Cardelli, Iliano Cervesato, Manuel Chakravarti, Richard C. Cobbe, Karl Crary, Daniel Dantas, Anupam Datta, Jake Donham, Derek Dreyer, Matthias Felleisen, Frank Pfenning, Dan Friedman, Maia Ginsburg, Kevin Hely, Cao Jing, Gabriele Keller, Danielle Kramer, Akiva Leffert, Ruy Ley-Wild, Dan Licata, Karen Liu, Dave MacQueen, Chris Martens, Greg Morrisett, Tom Murphy, Aleksandar Nanevski, Georg Neis, David Neville, Doug Perkins, Frank Pfenning, Benjamin C. Pierce, Andrew M. Pitts, Gordon D. Plotkin, David Renshaw, John C. Reynolds, Carter T. Schonwald, Dale Schumacher, Dana Scott, Zhong Shao, Robert Simmons, Pawel Sobocinski, Daniel Spoonhower, Paulo Tanimoto, Michael Tschantz, Kami Vaniea, Carsten Varming, David Walker, Dan Wang, Jack Wileden, Todd Wilson, Roger Wolff, Luke Zarko, Yu Zhang.

This material is based upon work supported by the National Science Foundation under Grant Nos. 0702381 and 0716469. Any opinions, find-

ings, 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.

The support of the Max Planck Institute for Software Systems in Saarbrücken, Germany is gratefully acknowledged.

Contents

Preface iii

I Judgements and Rules 1

1 Inductive Definitions 3
1.1 Judgements . . . 3
1.2 Inference Rules . . . 4
1.3 Derivations . . . 5
1.4 Rule Induction . . . 7
1.5 Iterated and Simultaneous Inductive Definitions . . . 9
1.6 Defining Functions by Rules . . . 11
1.7 Modes . . . 12
1.8 Exercises . . . 13

2 Hypothetical Judgements 15
2.1 Derivability . . . 15
2.2 Admissibility . . . 17
2.3 Hypothetical Inductive Definitions . . . 19
2.4 Exercises . . . 21

3 Syntax Trees 23
3.1 Abstract Syntax Trees . . . 23
3.2 Abstract Binding Trees . . . 25
3.3 Parameterization . . . 30
3.4 Exercises . . . 32

4 Generic Judgements 33
4.1 Rule Schemes . . . 33
4.2 Generic Derivability . . . 34

4.3 Generic Inductive Definitions 35
4.4 Parametric Derivability 36
4.5 Exercises 37

II Levels of Syntax 39

5 Concrete Syntax 41
5.1 Strings Over An Alphabet 41
5.2 Lexical Structure 42
5.3 Context-Free Grammars 46
5.4 Grammatical Structure 47
5.5 Ambiguity 48
5.6 Exercises 50

6 Abstract Syntax 51
6.1 Hierarchical and Binding Structure 51
6.2 Parsing Into Abstract Syntax Trees 53
6.3 Parsing Into Abstract Binding Trees 55
6.4 Exercises 57

III Statics and Dynamics 59

7 Statics 61
7.1 Syntax 61
7.2 Type System 62
7.3 Structural Properties 64
7.4 Exercises 66

8 Dynamics 67
8.1 Transition Systems 67
8.2 Structural Dynamics 68
8.3 Contextual Dynamics 71
8.4 Equational Dynamics 73
8.5 Exercises 76

9 Type Safety 77
9.1 Preservation 78
9.2 Progress 78
9.3 Run-Time Errors 80

9.4 Exercises 81

10 Evaluation Dynamics 83

10.1 Evaluation Dynamics 83
10.2 Relating Structural and Evaluation Dynamics 84
10.3 Type Safety, Revisited 85
10.4 Cost Dynamics 87
10.5 Exercises 88

IV Function Types 89

11 Function Definitions and Values 91

11.1 First-Order Functions 92
11.2 Higher-Order Functions 93
11.3 Evaluation Dynamics and Definitional Equivalence 95
11.4 Dynamic Scope 97
11.5 Exercises 98

12 Gödel's System T 99

12.1 Statics 100
12.2 Dynamics 101
12.3 Definability 102
12.4 Non-Definability 104
12.5 Exercises 106

13 Plotkin's PCF 107

13.1 Statics 109
13.2 Dynamics 110
13.3 Definability 112
13.4 Co-Natural Numbers 114
13.5 Exercises 114

V Finite Data Types 115

14 Product Types 117

14.1 Nullary and Binary Products 118
14.2 Finite Products 119
14.3 Primitive and Mutual Recursion 121
14.4 Exercises 122

15 Sum Types **123**
15.1 Binary and Nullary Sums 123
15.2 Finite Sums 125
15.3 Applications of Sum Types 126
15.3.1 Void and Unit 126
15.3.2 Booleans 127
15.3.3 Enumerations 127
15.3.4 Options 128
15.4 Exercises 129

16 Pattern Matching **131**
16.1 A Pattern Language 132
16.2 Statics 132
16.3 Dynamics 134
16.4 Exhaustiveness and Redundancy 136
16.4.1 Match Constraints 136
16.4.2 Enforcing Exhaustiveness and Redundancy 138
16.4.3 Checking Exhaustiveness and Redundancy 139
16.5 Exercises 140

17 Generic Programming **141**
17.1 Introduction 141
17.2 Type Operators 142
17.3 Generic Extension 142
17.4 Exercises 145

VI Infinite Data Types **147**

18 Inductive and Co-Inductive Types **149**
18.1 Motivating Examples 149
18.2 Statics 153
18.2.1 Types 153
18.2.2 Expressions 154
18.3 Dynamics 154
18.4 Exercises 155

19 Recursive Types **157**
19.1 Solving Type Isomorphisms 158
19.2 Recursive Data Structures 159

19.3 Self-Reference 161
19.4 Exercises 163

VII Dynamic Types 165

20 The Untyped λ-Calculus 167
20.1 The λ-Calculus 167
20.2 Definability 169
20.3 Scott's Theorem 171
20.4 Untyped Means Uni-Typed 173
20.5 Exercises 175

21 Dynamic Typing 177
21.1 Dynamically Typed PCF 177
21.2 Variations and Extensions 180
21.3 Critique of Dynamic Typing 183
21.4 Exercises 184

22 Hybrid Typing 185
22.1 A Hybrid Language 185
22.2 Optimization of Dynamic Typing 187
22.3 Static "Versus" Dynamic Typing 189
22.4 Reduction to Recursive Types 190

VIII Variable Types 191

23 Girard's System F 193
23.1 System **F** 194
23.2 Polymorphic Definability 197
23.2.1 Products and Sums 197
23.2.2 Natural Numbers 198
23.3 Parametricity Overview 199
23.4 Restricted Forms of Polymorphism 201
23.4.1 Predicative Fragment 201
23.4.2 Prenex Fragment 202
23.4.3 Rank-Restricted Fragments 204
23.5 Exercises 205

24 Abstract Types 207
24.1 Existential Types . . . 208
24.1.1 Statics . . . 208
24.1.2 Dynamics . . . 209
24.1.3 Safety . . . 210
24.2 Data Abstraction Via Existentials . . . 210
24.3 Definability of Existentials . . . 212
24.4 Representation Independence . . . 213
24.5 Exercises . . . 215

25 Constructors and Kinds 217
25.1 Statics . . . 218
25.2 Adding Constructors and Kinds . . . 220
25.3 Substitution . . . 222
25.4 Exercises . . . 225

26 Indexed Families of Types 227
26.1 Type Families . . . 227
26.2 Exercises . . . 227

IX Subtyping 229

27 Subtyping 231
27.1 Subsumption . . . 232
27.2 Varieties of Subtyping . . . 232
27.2.1 Numeric Types . . . 232
27.2.2 Product Types . . . 233
27.2.3 Sum Types . . . 235
27.3 Variance . . . 236
27.3.1 Product Types . . . 236
27.3.2 Sum Types . . . 236
27.3.3 Function Types . . . 237
27.3.4 Recursive Types . . . 238
27.4 Safety for Subtyping . . . 240
27.5 Exercises . . . 242

28 Singleton and Dependent Kinds 243
28.1 Informal Overview . . . 244

X Classes and Methods 247

29 Dynamic Dispatch 249
29.1 The Dispatch Matrix 251
29.2 Method-Based Organization 253
29.3 Class-Based Organization 254
29.4 Self-Reference 255
29.5 Exercises 257

30 Inheritance 259
30.1 Subclassing 260
30.2 Exercises 263

XI Control Effects 265

31 Control Stacks 267
31.1 Machine Definition 267
31.2 Safety 269
31.3 Correctness of the Control Machine 270
31.3.1 Completeness 272
31.3.2 Soundness 272
31.4 Exercises 273

32 Exceptions 275
32.1 Failures 275
32.2 Exceptions 277
32.3 Exception Type 278
32.4 Encapsulation 280
32.5 Exercises 282

33 Continuations 283
33.1 Informal Overview 283
33.2 Semantics of Continuations 285
33.3 Coroutines 287
33.4 Exercises 291

XII Types and Propositions 293

34 Constructive Logic 295

34.1 Constructive Semantics 296
34.2 Constructive Logic 297
34.2.1 Rules of Provability 298
34.2.2 Rules of Proof 300
34.3 Propositions as Types 301
34.4 Exercises 302

35 Classical Logic **303**
35.1 Classical Logic 304
35.1.1 Provability and Refutability 304
35.1.2 Proofs and Refutations 306
35.2 Deriving Elimination Forms 308
35.3 Proof Dynamics 310
35.4 Law of the Excluded Middle 311
35.5 Exercises 313

XIII Symbols **315**

36 Symbols **317**
36.1 Symbol Declaration 318
36.1.1 Scoped Dynamics 318
36.1.2 Scope-Free Dynamics 319
36.2 Symbolic References 321
36.2.1 Statics 322
36.2.2 Dynamics 322
36.2.3 Safety 323
36.3 Exercises 323

37 Fluid Binding **325**
37.1 Statics 325
37.2 Dynamics 326
37.3 Type Safety 327
37.4 Some Subtleties 328
37.5 Fluid References 331
37.6 Exercises 332

38 Dynamic Classification **333**
38.1 Dynamic Classes 334
38.1.1 Statics 334

38.1.2 Dynamics 335
38.1.3 Safety 335
38.2 Defining Dynamic Classes 336
38.3 Classifying Secrets 337
38.4 Exercises 338

XIV Storage Effects 339

39 Modernized Algol 341
39.1 Basic Commands 341
39.1.1 Statics 342
39.1.2 Dynamics 343
39.1.3 Safety 345
39.2 Some Programming Idioms 347
39.3 Typed Commands and Typed Assignables 349
39.4 Capabilities and References 351
39.5 Exercises 355

40 Mutable Data Structures 357
40.1 Free Assignables 358
40.2 Free References 359
40.3 Safety 360
40.4 Integrating Commands and Expressions 362
40.5 Exercises 365

XV Laziness 367

41 Lazy Evaluation 369
41.1 Need Dynamics 370
41.2 Safety 373
41.3 Lazy Data Structures 376
41.4 Suspensions 377
41.5 Exercises 379

42 Polarization 381
42.1 Polarization 382
42.2 Focusing 383
42.3 Statics 384
42.4 Dynamics 387

42.5 Safety 388
42.6 Definability 389
42.7 Exercises 389

XVI Parallelism 391

43 Nested Parallelism 393
43.1 Binary Fork-Join 394
43.2 Cost Dynamics 397
43.3 Multiple Fork-Join 400
43.4 Provably Efficient Implementations 402
43.5 Exercises 406

44 Futures and Speculation 407
44.1 Futures 408
44.1.1 Statics 408
44.1.2 Sequential Dynamics 409
44.2 Suspensions 409
44.2.1 Statics 409
44.2.2 Sequential Dynamics 410
44.3 Parallel Dynamics 410
44.4 Applications of Futures 413
44.5 Exercises 415

XVII Concurrency 417

45 Process Calculus 419
45.1 Actions and Events 419
45.2 Interaction 421
45.3 Replication 423
45.4 Allocating Channels 425
45.5 Communication 428
45.6 Channel Passing 432
45.7 Universality 434
45.8 Exercises 436

46 Concurrent Algol 437
46.1 Concurrent Algol 437
46.2 Broadcast Communication 440
46.3 Selective Communication 442
46.4 Free Assignables as Processes 445
46.5 Exercises 446

47 Distributed Algol 447
47.1 Statics 447
47.2 Dynamics 450
47.3 Safety 451
47.4 Situated Types 452
47.5 Exercises 455

XVIII Modularity 457

48 Separate Compilation and Linking 459
48.1 Linking and Substitution 459
48.2 Exercises 459

49 Basic Modules 461

50 Parameterized Modules 463

XIX Equivalence 465

51 Equational Reasoning for T 467
51.1 Observational Equivalence 468
51.2 Extensional Equivalence 472
51.3 Extensional and Observational Equivalence Coincide 473
51.4 Some Laws of Equivalence 476
51.4.1 General Laws 476
51.4.2 Extensionality Laws 477
51.4.3 Induction Law 477
51.5 Exercises 478

52 Equational Reasoning for PCF 479
52.1 Observational Equivalence 479
52.2 Extensional Equivalence 480

52.3 Extensional and Observational Equivalence Coincide 481
52.4 Compactness 484
52.5 Co-Natural Numbers 487
52.6 Exercises 489

53 Parametricity 491
53.1 Overview 491
53.2 Observational Equivalence 492
53.3 Logical Equivalence 494
53.4 Parametricity Properties 500
53.5 Representation Independence, Revisited 503
53.6 Exercises 505

XX Appendices 507

A Mathematical Preliminaries 509
A.1 Finite Sets and Maps 509
A.2 Families of Sets 509

Part I

Judgements and Rules

Chapter 1

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.

1.1 Judgements

We start with the notion of a *judgement*, or *assertion*, about a *syntactic object*.[1] We shall make use of many forms of judgement, including examples such as these:

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

A judgement 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 *judgement form*, and the judgement that an object or objects have that property or stand in that relation is said to be an *instance* of that judgement form. A judgement form is also called a *predicate*, and the syntactic objects constituting an instance are its *subjects*.

We will use the meta-variable J to stand for an unspecified judgement form, and the meta-variables a, b, and c to stand for syntactic objects. We write $a\ \mathsf{J}$ for the judgement asserting that J holds of a. When it is not

[1] We will defer a precise treatment of syntactic objects to Chapter 3. For the present purposes the meaning should be self-evident.

important to stress the subject of the judgement, we write J to stand for an unspecified judgement. For particular judgement forms, we freely use prefix, infix, or mixfix notation, as illustrated by the above examples, in order to enhance readability.

1.2 Inference Rules

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

$$\frac{J_1 \quad \dots \quad J_k}{J} \tag{1.1}$$

in which J and $J_1, \dots, J_k$ are all judgements of the form being defined. The judgements above the horizontal line are called the *premises* of the rule, and the judgement 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 $J_1, \dots, J_k$. 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 judgement a nat:

$$\frac{}{\texttt{zero}\ \mathsf{nat}} \tag{1.2a}$$

$$\frac{a\ \mathsf{nat}}{\texttt{succ}(a)\ \mathsf{nat}} \tag{1.2b}$$

These rules specify that a nat holds whenever either a is zero, or a is succ(b) where b nat. Taking these rules to be exhaustive, it follows that a nat iff a is a natural number written in unary.

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

$$\frac{}{\texttt{empty}\ \mathsf{tree}} \tag{1.3a}$$

$$\frac{a_1\ \mathsf{tree} \quad a_2\ \mathsf{tree}}{\texttt{node}(a_1;a_2)\ \mathsf{tree}} \tag{1.3b}$$

These rules specify that a tree holds if either a is empty, or a is 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 judgement $a = b$ nat defining equality of a nat and b nat is inductively defined by the following rules:

$$\frac{}{\texttt{zero} = \texttt{zero}\ \texttt{nat}} \tag{1.4a}$$

$$\frac{a = b\ \texttt{nat}}{\texttt{succ}(a) = \texttt{succ}(b)\ \texttt{nat}} \tag{1.4b}$$

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 (1.2b) 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* syntactic object, a, or is instead intended as a rule scheme specifying a rule for *each choice* of syntactic objects in the rule.

A collection of rules is considered to define the *strongest* judgement 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 judgement: J holds *if* there is a way to obtain it using the given rules. To be the *strongest* judgement 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*.

1.3 Derivations

To show that an inductively defined judgement holds, it is enough to exhibit a *derivation* of it. A derivation of a judgement is a finite composition of rules, starting with axioms and ending with that judgement. 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 judgement 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

$$\frac{\nabla_1 \quad \dots \quad \nabla_k}{J} \tag{1.5}$$

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

For example, this is a derivation of `succ(succ(succ(zero))) nat`:

$$\frac{\frac{\frac{\frac{}{\texttt{zero nat}}}{\texttt{succ(zero) nat}}}{\texttt{succ(succ(zero)) nat}}}{\texttt{succ(succ(succ(zero))) nat}} . \tag{1.6}$$

Similarly, here is a derivation of `node(node(empty;empty);empty) tree`:

$$\frac{\frac{\frac{}{\texttt{empty tree}} \quad \frac{}{\texttt{empty tree}}}{\texttt{node(empty;empty) tree}} \quad \frac{}{\texttt{empty tree}}}{\texttt{node(node(empty;empty);empty) tree}} . \tag{1.7}$$

To show that an inductively defined judgement 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 judgements, 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 judgement occurs in the set. Assuming that all rules are considered at every stage, forward chaining will eventually find a derivation of any derivable judgement, but it is impossible (in general) to decide algorithmically when to stop extending the set and conclude that the desired judgement is not derivable. We may go on and on adding more judgements 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 judgement 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, judgements whose derivations are to be sought. Initially, this set consists solely of the judgement we wish to derive. At each stage, we remove a judgement from the queue, and consider all rules whose conclusion is that judgement. 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 judgement, 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 judgements to the goal set, never reaching a point at which all goals have been satisfied.

1.4 Rule Induction

Since an inductive definition specifies the *strongest* judgement 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 judgement 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 judgement 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 judgement form as the *strongest* judgement form closed under the rules comprising the definition. This means that the judgement form defined by a set of rules is both (a) closed under those 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 judgement; the latter means that we may reason about an inductively defined judgement form by rule induction.

To show that $\mathcal{P}(J)$ holds whenever J is derivable, it is often helpful to show that the conjunction of $\mathcal{P}$ with J itself is closed under the rules. This is because one gets as an inductive hypotheses not only that $\mathcal{P}(J_i)$ holds for each premise J_i, but also that J_i is derivable. Of course the inductive step must correspondingly show both that $\mathcal{P}(J)$, but also that J is derivable. But the latter requirement is immediately satisfied by an application of the rule in question, whose premises are derivable by the inductive hypothesis. Having shown that the conjunction of $\mathcal{P}$ with J is closed under each of the rules, it follows that if J is derivable, then we have both that $\mathcal{P}(J)$, and also that J is derivable, and we can just ignore the redundant auxiliary result. In practice we shall often tacitly employ this maneuver, only showing that a property is closed when restricted to derivable judgements.

When specialized to Rules (1.2), the principle of rule induction states that to show $\mathcal{P}(a\ \mathsf{nat})$ whenever a nat, it is enough to show:

1. $\mathcal{P}(\texttt{zero}\ \mathsf{nat})$.

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

This is just the familiar principle of *mathematical induction* arising as a special case of rule induction. The first condition is called the *basis* of the induction, and the second is called the *inductive step*.

Similarly, rule induction for Rules (1.3) states that to show $\mathcal{P}(a\ \mathsf{tree})$ whenever a tree, it is enough to show

1. $\mathcal{P}(\texttt{empty}\ \mathsf{tree})$.

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

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

As a simple example of a proof by rule induction, let us prove that natural number equality as defined by Rules (1.4) is reflexive:

Lemma 1.1. *If* a nat, *then* $a = a$ nat.

Proof. By rule induction on Rules (1.2):

Rule (1.2a) Applying Rule (1.4a) we obtain $\texttt{zero} = \texttt{zero}\ \mathsf{nat}$.

Rule (1.2b) Assume that $a = a\ \mathsf{nat}$. It follows that $\texttt{succ}(a) = \texttt{succ}(a)\ \mathsf{nat}$ by an application of Rule (1.4b).

□

As another example of the use of rule induction, we may show that the predecessor of a natural number is also a natural number. While this may seem self-evident, the point of the example is to show how to derive this from first principles.

Lemma 1.2. *If $\texttt{succ}(a)\ \mathsf{nat}$, then $a\ \mathsf{nat}$.*

Proof. It suffices to show that the property, $\mathcal{P}(a\ \mathsf{nat})$ stating that $a\ \mathsf{nat}$ and that $a = \texttt{succ}(b)$ implies $b\ \mathsf{nat}$ is closed under Rules (1.2).

Rule (1.2a) Clearly $\texttt{zero}\ \mathsf{nat}$, and the second condition holds vacuously, since $\texttt{zero}$ is not of the form $\texttt{succ}(-)$.

Rule (1.2b) Inductively we know that $a\ \mathsf{nat}$ and that if a is of the form $\texttt{succ}(b)$, then $b\ \mathsf{nat}$. We are to show that $\texttt{succ}(a)\ \mathsf{nat}$, which is immediate, and that if $\texttt{succ}(a)$ is of the form $\texttt{succ}(b)$, then $b\ \mathsf{nat}$, and we have $b\ \mathsf{nat}$ by the inductive hypothesis.

This completes the proof. □

Similarly, let us show that the successor operation is injective.

Lemma 1.3. *If $\texttt{succ}(a_1) = \texttt{succ}(a_2)\ \mathsf{nat}$, then $a_1 = a_2\ \mathsf{nat}$.*

Proof. Similar to the proof of Lemma 1.2. □

1.5 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 judgement form, or the judgement form being defined. For example, the following rules define the judgement a list stating that a is a list of natural numbers.

$$\frac{}{\texttt{nil}\ \mathsf{list}} \tag{1.8a}$$

$$\frac{a\ \mathsf{nat} \quad b\ \mathsf{list}}{\texttt{cons}(a;b)\ \mathsf{list}} \tag{1.8b}$$

The first premise of Rule (1.8b) is an instance of the judgement form a nat, which was defined previously, whereas the premise b list is an instance of the judgement form being defined by these rules.

Frequently two or more judgements 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 judgement forms, any of which may appear as the premise of any rule. Since the rules defining each judgement form may involve any of the others, none of the judgement forms may be taken to be defined prior to the others. Instead one must understand that all of the judgement forms are being defined at once by the entire collection of rules. The judgement forms defined by these rules are, as before, the strongest judgement 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 judgement forms simultaneously.

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

$$\frac{}{\texttt{zero}\ \mathsf{even}} \tag{1.9a}$$

$$\frac{a\ \mathsf{odd}}{\texttt{succ}(a)\ \mathsf{even}} \tag{1.9b}$$

$$\frac{a\ \mathsf{even}}{\texttt{succ}(a)\ \mathsf{odd}} \tag{1.9c}$$

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:

1. $\mathcal{P}(\texttt{zero}\ \mathsf{even})$;

2. if $\mathcal{P}(a\ \mathsf{odd})$, then $\mathcal{P}(\texttt{succ}(a)\ \mathsf{even})$;

3. if $\mathcal{P}(a\ \mathsf{even})$, then $\mathcal{P}(\texttt{succ}(a)\ \mathsf{odd})$.

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 $a\ \mathsf{nat}$, and (2) $\mathcal{P}(a\ \mathsf{odd})$ iff $a\ \mathsf{nat}$. The principle of rule induction for Rules (1.9) states that it is sufficient to show the following facts:

1. $\texttt{zero}\ \mathsf{nat}$, which is derivable by Rule (1.2a).

2. If $a\ \mathsf{nat}$, then $\texttt{succ}(a)\ \mathsf{nat}$, which is derivable by Rule (1.2b).

3. If $a\ \mathsf{nat}$, then $\texttt{succ}(a)\ \mathsf{nat}$, which is also derivable by Rule (1.2b).

1.6 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:

$$\frac{b\ \mathsf{nat}}{\mathsf{sum}(\texttt{zero};b;b)} \tag{1.10a}$$

$$\frac{\mathsf{sum}(a;b;c)}{\mathsf{sum}(\texttt{succ}(a);b;\texttt{succ}(c))} \tag{1.10b}$$

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

Proof. The proof decomposes into two parts:

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

2. (Uniqueness) If $a\ \mathsf{nat}$, $b\ \mathsf{nat}$, $c\ \mathsf{nat}$, $c'\ \mathsf{nat}$, $\mathsf{sum}(a;b;c)$, and $\mathsf{sum}(a;b;c')$, then $c = c'\ \mathsf{nat}$.

For existence, let $\mathcal{P}(a\ \mathsf{nat})$ be the proposition *if* $b\ \mathsf{nat}$ *then there exists* $c\ \mathsf{nat}$ *such that* $\mathsf{sum}(a;b;c)$. We prove that if $a\ \mathsf{nat}$ then $\mathcal{P}(a\ \mathsf{nat})$ by rule induction on Rules (1.2). We have two cases to consider:

Rule (1.2a) We are to show $\mathcal{P}(\mathtt{zero}\ \mathsf{nat})$. Assuming $b\ \mathsf{nat}$ and taking c to be b, we obtain $\mathsf{sum}(\mathtt{zero};b;c)$ by Rule (1.10a).

Rule (1.2b) Assuming $\mathcal{P}(a\ \mathsf{nat})$, we are to show $\mathcal{P}(\mathtt{succ}(a)\ \mathsf{nat})$. That is, we assume that if $b\ \mathsf{nat}$ then there exists c such that $\mathsf{sum}(a;b;c)$, and are to show that if $b'\ \mathsf{nat}$, then there exists c' such that $\mathsf{sum}(\mathtt{succ}(a);b';c')$. To this end, suppose that $b'\ \mathsf{nat}$. Then by induction there exists c such that $\mathsf{sum}(a;b';c)$. Taking $c' = \mathtt{succ}(c)$, and applying Rule (1.10b), 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* $c_1 = c_2\ \mathsf{nat}$ by rule induction based on Rules (1.10).

Rule (1.10a) 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 1.1 on page 8 we obtain $b = b\ \mathsf{nat}$.

Rule (1.10b) 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 $c_2 = \mathtt{succ}(c_2')\ \mathsf{nat}$ where $\mathsf{sum}(a';b;c_2')$. By the outer inductive hypothesis $c_1' = c_2'\ \mathsf{nat}$ and so $c_1 = c_2\ \mathsf{nat}$.

□

1.7 Modes

The statement that one or more arguments of a judgement is (perhaps uniquely) determined by its other arguments is called a *mode specification* for that judgement. For example, we have shown that every two natural numbers have a sum according to Rules (1.10). This fact may be restated as a mode specification by saying that the judgement $\mathsf{sum}(a;b;c)$ has *mode* $(\forall,\forall,\exists)$. The notation arises from the form of the proposition it expresses: *for all* $a\ \mathsf{nat}$ *and for all* $b\ \mathsf{nat}$, *there exists* $c\ \mathsf{nat}$ *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 judgement $\mathsf{sum}(a;b;c)$ has mode $(\forall,\forall,\exists!)$, corresponding to the proposition *for all* $a\ \mathsf{nat}$ *and for all* $b\ \mathsf{nat}$, *there exists a unique* $c\ \mathsf{nat}$ *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 judgement has mode $(\forall, \forall, \exists^{\leq 1})$, corresponding to the proposition *for all* a $\mathtt{nat}$ *and for all* b $\mathtt{nat}$ *there exists at most one* c $\mathtt{nat}$ *such that* $\mathtt{sum}(a;b;c)$.

As these examples illustrate, a given judgement may satisfy several different mode specifications. In general the universally quantified arguments are to be thought of as the *inputs* of the judgement, 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 judgement, 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 (1.10)) using equations:

$$\frac{a\ \mathtt{nat}}{a + \mathtt{zero} = a\ \mathtt{nat}} \tag{1.11a}$$

$$\frac{a + b = c\ \mathtt{nat}}{a + \mathtt{succ}(b) = \mathtt{succ}(c)\ \mathtt{nat}} \tag{1.11b}$$

When using this notation we tacitly incur the obligation to prove that the mode of the judgement 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 $a + b = c$ $\mathtt{nat}$.

1.8 Exercises

1. Give an inductive definition of the judgement $\mathtt{max}(a;b;c)$, where a $\mathtt{nat}$, b $\mathtt{nat}$, and c $\mathtt{nat}$, with the meaning that c is the larger of a and b. Prove that this judgement has the mode $(\forall, \forall, \exists!)$.

2. Consider the following rules, which define the height of a binary tree as the judgement $\mathtt{hgt}(a;b)$.

$$\frac{}{\mathtt{hgt}(\mathtt{empty};\mathtt{zero})} \tag{1.12a}$$

$$\frac{\mathsf{hgt}(a_1;b_1) \quad \mathsf{hgt}(a_2;b_2) \quad \mathsf{max}(b_1;b_2;b)}{\mathsf{hgt}(\texttt{node}(a_1;a_2);\texttt{succ}(b))} \tag{1.12b}$$

Prove by tree induction that the judgement `hgt` has the mode $(\forall, \exists)$, with inputs being binary trees and outputs being natural numbers.

3. Give an inductive definition of the judgement "∇ is a derivation of J" for an inductively defined judgement J of your choice.

4. Give an inductive definition of the forward-chaining and backward-chaining search strategies.

Chapter 2

Hypothetical Judgements

A *hypothetical judgement* expresses an *entailment* between one or more *hypotheses* and a *conclusion*. We will consider two notions of entailment, called *derivability* and *admissibility*. Derivability expresses the stronger of the two forms of entailment, namely that the conclusion may be deduced directly from the hypotheses by composing rules. Admissibility expresses the weaker form, that the conclusion is derivable from the rules whenever the hypotheses are also derivable. Both forms of entailment enjoy the same *structural* properties that characterize conditional reasoning. One consequence of these properties is that derivability is stronger than admissibility (but the converse fails, in general). We then generalize the concept of an inductive definition to admit rules that have hypothetical judgements as premises. Using these we may enrich the rules with new axioms that are available for use within a specified premise of a rule.

2.1 Derivability

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

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

That is, we treat the *hypotheses*, or *antecedents*, of the judgement, $J_1, \ldots, J_n$ as *temporary axioms*, and derive the *conclusion*, or *consequent*, by composing rules in $\mathcal{R}$. That is, evidence for a hypothetical judgement consists of a derivation of the conclusion from the hypotheses using the rules in $\mathcal{R}$.

We use capital Greek letters, frequently Γ or Δ, to stand for a finite collection of basic judgements, and write $\mathcal{R}[\Gamma]$ for the expansion of $\mathcal{R}$ with an axiom corresponding to each judgement in Γ. The judgement $\Gamma \vdash_{\mathcal{R}} K$ means that K is derivable from rules $\mathcal{R}[\Gamma]$. We sometimes write $\vdash_{\mathcal{R}} \Gamma$ to mean that $\vdash_{\mathcal{R}} J$ for each judgement J in Γ. The derivability judgement $J_1, \ldots, J_n \vdash_{\mathcal{R}} J$ is sometimes expressed by saying that the rule

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

is *derivable* from the rules $\mathcal{R}$.

For example, consider the derivability judgement

$$a \ \mathsf{nat} \vdash_{(1.2)} \mathtt{succ}(\mathtt{succ}(a)) \ \mathsf{nat} \tag{2.2}$$

relative to Rules (1.2). This judgement is valid for *any* choice of object a, as evidenced by the derivation

$$\cfrac{\cfrac{a \ \mathsf{nat}}{\mathtt{succ}(a) \ \mathsf{nat}}}{\mathtt{succ}(\mathtt{succ}(a)) \ \mathsf{nat}}\ , \tag{2.3}$$

which composes Rules (1.2), starting with a nat as an axiom, and ending with $\mathtt{succ}(\mathtt{succ}(a))$ nat. Equivalently, the validity of (2.2) may also be expressed by stating that the rule

$$\frac{a \ \mathsf{nat}}{\mathtt{succ}(\mathtt{succ}(a)) \ \mathsf{nat}} \tag{2.4}$$

is derivable from Rules (1.2).

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

Theorem 2.1 (Stability). *If* $\Gamma \vdash_{\mathcal{R}} J$*, then* $\Gamma \vdash_{\mathcal{R} \cup \mathcal{R}'} J$*.*

Proof. Any derivation of J from $\mathcal{R}[\Gamma]$ is also a derivation from $(\mathcal{R} \cup \mathcal{R}')[\Gamma]$, since 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 judgement 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.

2.2 Admissibility

Admissibility, written $\Gamma \models_{\mathcal{R}} J$, is a weaker form of hypothetical judgement 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 Γ are all derivable from rules $\mathcal{R}$. In particular if any of the hypotheses are *not* derivable relative to $\mathcal{R}$, then the judgement is vacuously true. The admissibility judgement $J_1, \ldots, J_n \models_{\mathcal{R}} J$ is sometimes expressed by stating that the rule,

$$\frac{J_1 \quad \ldots \quad J_n}{J}\,, \tag{2.5}$$

is *admissible* relative to the rules in $\mathcal{R}$.

For example, the admissibility judgement

$$\texttt{succ}(a)\ \texttt{nat} \models_{(1.2)} a\ \texttt{nat} \tag{2.6}$$

is valid, because any derivation of $\texttt{succ}(a)\ \texttt{nat}$ from Rules (1.2) must contain a sub-derivation of $a\ \texttt{nat}$ from the same rules, which justifies the conclusion. The validity of (2.6) may equivalently be expressed by stating that the rule

$$\frac{\texttt{succ}(a)\ \texttt{nat}}{a\ \texttt{nat}} \tag{2.7}$$

is admissible for Rules (1.2).

In contrast to derivability the admissibility judgement is *not* stable under extension to the rules. For example, if we enrich Rules (1.2) with the axiom

$$\frac{}{\texttt{succ(junk)}\ \texttt{nat}} \tag{2.8}$$

(where `junk` is some object for which `junk nat` is not derivable), then the admissibility (2.6) is *invalid*. This is because Rule (2.8) 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 2.2. *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

$$\texttt{succ(junk) nat} \vdash_{(1.2)} \texttt{junk nat},$$

yet the admissibility judgement

$$\texttt{succ(junk) nat} \models_{(1.2)} \texttt{junk nat}$$

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 ∇ of the consequent. Therefore, the admissibility judgement enjoys the same structural properties as derivability, and hence is a form of hypothetical judgement:

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 judgements in Γ are derivable from these rules, then J must also be derivable assuming that Γ 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 judgements in Γ are derivable, so is K, by assumption, and hence so are the judgements in Γ, K, and hence so is J.

Theorem 2.3. *The admissibility judgement $\Gamma \models_{\mathcal{R}} J$ is structural.*

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 $\mathcal{R}$. Admissiblity of a rule, r, of the form (2.5) means that any derivations of $J_1,\ldots,J_n$ with respect to rules $\mathcal{R}$ may be transformed into a derivation of J with respect to the same set of rules. 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 $\mathcal{P}(J)$.

2.3 Hypothetical Inductive Definitions

It is useful to enrich the concept of an inductive definition to permit rules with derivability judgements 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:

$$\frac{\Gamma\,\Gamma_1 \vdash J_1 \quad \ldots \quad \Gamma\,\Gamma_n \vdash J_n}{\Gamma \vdash J} . \tag{2.9}$$

The hypotheses Γ are the *global hypotheses* of the rule, and the hypotheses Γ_i are the *local hypotheses* of the ith premise of the rule. Informally, this rule states that J is a derivable consequence of Γ whenever each J_i is a derivable consequence of Γ, augmented with the additional hypotheses Γ_i. Thus, one way to show that J is derivable from Γ 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 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

$$\frac{\Gamma_1 \vdash J_1 \quad \dots \quad \Gamma_n \vdash J_n}{J}\ , \tag{2.10}$$

which stands for the collection of all rules of the form (2.9) 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 judgement* $\Gamma \vdash J$ consisting of a finite set of basic judgements, Γ, and a basic judgement, J. A collection of hypothetical rules, $\mathcal{R}$, defines the strongest formal derivability judgement that is *structural* and *closed* under rules $\mathcal{R}$. Structurality means that the formal derivability judgement must be closed under the following rules:

$$\frac{}{\Gamma, J \vdash J} \tag{2.11a}$$

$$\frac{\Gamma \vdash J}{\Gamma, K \vdash J} \tag{2.11b}$$

$$\frac{\Gamma \vdash K \quad \Gamma, K \vdash J}{\Gamma \vdash J} \tag{2.11c}$$

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

The principal of *hypothetical rule induction* is just the principal of rule induction applied to the formal hypothetical judgement. 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 (2.10), whether structural or in $\mathcal{R}$, we must show that

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

This is just a restatement of the principle of rule induction given in Chapter 1, specialized to the formal derivability judgement $\Gamma \vdash J$.

In practice we usually dispense with the structural rules by the method described at the end of Section 2.2 on page 17. 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 (2.11b) and (2.11c) are readily seen to be admissible. However, it is typically necessary to include Rule (2.11a) explicitly to ensure reflexivity.

2.4 Exercises

1. Define $\Gamma' \vdash \Gamma$ to mean that $\Gamma' \vdash J_i$ for each J_i in Γ. Show that $\Gamma \vdash J$ iff whenever $\Gamma' \vdash \Gamma$, it follows that $\Gamma' \vdash J$. *Hint*: from left to right, appeal to transitivity of entailment; from right to left, consider the case of $\Gamma' = \Gamma$.

2. Show that it is dangerous to permit admissibility judgements in the premise of a rule. *Hint*: show that using such rules one may "define" an inconsistent judgement form J for which we have $a\ J$ iff it is *not* the case that $a\ J$.

Chapter 3

Syntax Trees

Throughout this book we shall have need of a variety of syntactic objects with which to model programming language concepts. We will use a very general framework for these that accounts for three crucial concepts: (1) hierarchical structure, (2) binding and scope, and (3) parameterization. *Abstract syntax trees* account for hierarchical structure; these form the foundation of the framework. *Abstract binding trees* enrich abstract syntax trees with *variable binding and scope*.

3.1 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. Abstract syntax trees are classified into *sorts*. Each variables is assigned a sort. Each operator is assigned both a sort and an *arity*, a sequence of sorts specifying the number and sort of each of its arguments. A *nullary* operator is one that takes no arguments, a *unary* operator takes one, a *binary* operator two, and so forth.

For example, a sort, Num, of numerals is generated by the nullary operator, $\mathtt{zero}$, of sort Num, and the unary operator, $\mathtt{succ}$, of sort Num whose argument is also of sort Num. The tree $\mathtt{succ(succ(zero()))}$ is then an ast of sort Num. Moreover, if x is a variable of sort Num, then $\mathtt{succ(succ(}x\mathtt{))}$ is also an ast of sort Num. As another example, the sort Expr, of arithmetic expressions consisting of numerals, sums, and products by specifying that (1) $\mathtt{num}[n]$ is a nullary operator of sort Expr whenever $n \in \mathbb{N}$; (2) $\mathtt{plus}$ and $\mathtt{times}$ are binary operators of sort Expr whose arguments are both of sort

Expr. Then

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

is an ast of sort Expr, assuming that x is a variable of the same sort.

In general a collection of abstract syntax trees is specified by two pieces of data. First, we must specify a finite set, $\mathcal{S}$, of sorts. This determines the categories of syntax in the language under consideration. Second, for each sort $s \in \mathcal{S}$, we specify a set, $\mathcal{O}_s$, of operators of sort s, and define for each such operator, o, its arity, $\mathsf{ar}(o) = (s_1, \ldots, s_n)$, which determines the number and sorts of its arguments. We also assume that for each sort $s \in \mathcal{S}$ there is an infinite set, $\mathcal{X}_s$, of variables of sort s that are distinct from any of the operators (so that there can be no ambiguity about what is a variable).

Given this data, the $\mathcal{S}$-indexed family $\mathcal{A}[\mathcal{X}] = \{ \mathcal{A}[\mathcal{X}]_s \}_{s \in \mathcal{S}}$ of abstract syntax trees is inductively defined by 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. Compound ast's are formed by operators: if $\mathsf{ar}(o) = (s_1, \ldots, s_n)$, and $a_1 \in \mathcal{A}[\mathcal{X}]_{s_1}, \ldots, a_n \in \mathcal{A}[\mathcal{X}]_{s_n}$, then $o(a_1; \ldots; a_n) \in \mathcal{A}[\mathcal{X}]_s$.

We will often use notational conventions to identify the variables of a sort, and speak loosely of an "ast of sort s" without precisely specifying the sets of variables of each sort. When specifying the variables, we often write $\mathcal{X}, x$, where x is a variable of sort s such that $x \notin \mathcal{X}_s$, to mean 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 of sort s, is said to be the family obtained by *adjoining* the variable x to the family $\mathcal{X}$.

It is easy to see that if $\mathcal{X} \subseteq \mathcal{Y}$, then $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{A}[\mathcal{Y}]$.[1] A $\mathcal{S}$-indexed family of bijections $\pi : \mathcal{X} \leftrightarrow \mathcal{Y}$ between variables[2] induces a *renaming*, $\pi \cdot a$, on $a \in \mathcal{A}[\mathcal{X}]$ yielding an ast in $\mathcal{A}[\mathcal{Y}]$ obtained by replacing $x \in \mathcal{X}_s$ by $\pi_s(x)$ everywhere in a. (Renamings will play an important role in the generalization of ast's to account for binding and scope to be developed in Section 3.2 on the next page.)

Variables are given meaning by *substitution*: a variable stands for an ast to be determined later. More precisely, if $a \in \mathcal{A}[\mathcal{X}, x]$ and $b \in \mathcal{A}[\mathcal{X}]$, then $[b/x]a \in \mathcal{A}[\mathcal{X}]$, where $[b/x]a$ is the result of *substituting* b for every occurrence of x in a. The ast a is sometimes called the *target*, and x is called the *subject*, of the substitution. Substitution is defined by the following conditions:

[1] Relations on sets extend to relations on families of sets element-wise, so that $\mathcal{X} \subseteq \mathcal{Y}$ means that for every $s \in \mathcal{S}$, $\mathcal{X}_s \subseteq \mathcal{Y}_s$.

[2] That is, $\pi_s : \mathcal{X}_s \leftrightarrow \mathcal{Y}_s$ for each $s \in \mathcal{S}$.

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 readily check that

$$[\texttt{succ(zero())}/x]\texttt{succ(succ(}x\texttt{))} = \texttt{succ(succ(succ(zero())))}.$$

Substitution simply "plugs in" the given subject for a specified variable in the target.

To prove that a property, $\mathcal{P}$, holds of a class of abstract syntax trees we may use the principle of *structural induction*, or *induction on syntax*. To show that $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{P}$, it is enough to show:

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

That is, to show that every ast of sort s has property $\mathcal{P}_s$, it is enough to show that every variable of sort s has the property $\mathcal{P}_s$ and that for every operator o of sort s whose arguments have sorts $s_1,\ldots,s_n$, respectively, if a_1 has property $\mathcal{P}_{s_1}$, and ... and a_n has property $\mathcal{P}_{s_n}$, then $o(a_1;\ldots;a_n)$ has property $\mathcal{P}_s$.

For example, we may prove by structural induction that substitution on ast's is well-defined.

Theorem 3.1. *If $b \in \mathcal{A}[\mathcal{X}]$, 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 ... $[b/x]a_n = c_n$, and so c is $c = o(c_1;\ldots;c_n)$, by definition of substitution. □

3.2 Abstract Binding Trees

Abstract syntax goes a long way towards separating objective issues of syntax (the hierarchical structure of expressions) from subjective issues (their layout on the page). This can be usefully pushed a bit further by enriching abstract syntax to account for *binding* and *scope*.

All languages have facilities for introducing an identifier with a specified range of significance. For example, we may define a variable, x, to stand for an expression, e_1, so that we may conveniently refer to it within another expression, e_2, by writing $\texttt{let}\,x\,\texttt{be}\,e_1\,\texttt{in}\,e_2$. The intention is that x stands for e_1 *inside of* the expression e_2, but has no meaning whatsoever *outside of* that expression. The variable x is said to be *bound* within e_2 by the definition; equivalently, the *scope* of the variable x is the expression e_2.

Moreover, the name x has no intrinsic significance; we may just as well use any variable y for the same purpose, provided that we rename x to y within e_2. Such a renaming is always possible, provided only that there can be no *confusion* between two different definitions. So, for example, there is no difference between the expressions

$$\texttt{let}\,x\,\texttt{be}\,\texttt{succ(succ(zero))}\,\texttt{in}\,\texttt{succ(succ(}x\texttt{))}$$

and

$$\texttt{let}\,y\,\texttt{be}\,\texttt{succ(succ(zero))}\,\texttt{in}\,\texttt{succ(succ(}y\texttt{))}.$$

But we must be careful when nesting definitions, since

$$\texttt{let}\,x\,\texttt{be}\,\texttt{succ(succ(zero))}\,\texttt{in}\,\texttt{let}\,y\,\texttt{be}\,\texttt{succ(zero)}\,\texttt{in}\,\texttt{succ(}x\texttt{)}$$

is entirely different from

$$\texttt{let}\,y\,\texttt{be}\,\texttt{succ(succ(zero))}\,\texttt{in}\,\texttt{let}\,y\,\texttt{be}\,\texttt{succ(zero)}\,\texttt{in}\,\texttt{succ(}y\texttt{)}.$$

In this case we cannot rename x to y, nor can we rename y to x, because to do so would confuse two different definitions. The guiding principle is that bound variables are *pointers* to their binding sites, and that any renaming must preserve the pointer structure of the expression. Put in other terms, bound variables function as *pronouns*, which refer to objects separately introduced by a noun phrase (here, an expression). Renaming must preserve the pronoun structure, so that we cannot get confusions such as "which he do you mean?" that arise in less formal languages.

The concepts of binding and scope can be accounted by enriching abstract syntax trees with some additional structure. Such enriched abstract syntax trees are called *abstract binding trees*, or *abt's* for short. An operator on abt's may bind zero or more variables in each of its arguments independently of one another. Each argument is an *abstractor* of the form $x_1, \ldots, x_k\,.\,a$, where $x_1, \ldots, x_k$ are variables and a is an abt possibly mentioning those variables. Such an abstractor specifies that the variables $x_1, \ldots, x_k$

are bound within e_2. When k is zero, we usually elide the distinction between $.a$ and a itself. Thus, when written in the form of an abt, a definition has the form $\mathtt{let}(e_1;x.e_2)$. The abstractor $x.e_2$ in the second argument position makes clear that x is bound within e_2, and not within e_1.

Since an operator may bind variables in each of its arguments, the arity of an operator is generalized to be a finite sequence of *valences* of the form $(s_1,\ldots,s_k)s$ consisting of a finite sequence of sorts together with a sort. Such a valence specifies the overall sort of the argument, s, and the sorts $s_1,\ldots,s_k$ of the variables bound within that argument. Thus, for example, the arity of the operator `let` is $(\mathsf{Expr},(\mathsf{Expr})\mathsf{Expr})$, which indicates that it takes two arguments described as follows:

1. The first argument is of sort Expr, and binds no variables.

2. The second argument is also of sort Expr, and binds one variable of sort Expr.

A precise definition of abt's requires some care. As a first approximation let us naïvely define the $\mathcal{S}$-indexed family $\mathcal{B}[\mathcal{X}]$ of abt's over the $\mathcal{S}$-indexed variables $\mathcal{X}$ and $\mathcal{S}$-indexed family $\mathcal{O}$ of operators o of arity $\mathsf{ar}(o)$. To lighten the notation let us write $\vec{x}$ for a finite sequence $x_1,\ldots,x_n$ of n distinct variables, and $\vec{s}$ for a finite sequence $s_1,\ldots,s_n$ of n sorts. We say that $\vec{x}$ is a sequence of variables of sort $\vec{s}$ iff the two sequences have the same length, n, and for each $1 \leq i \leq n$ the variable x_i is of sort s_i. The following conditions would appear to suffice as the definition of the abt's of each sort:

1. Every variable is an abt: $\mathcal{X} \subseteq \mathcal{B}[\mathcal{X}]$.

2. Compound abt's are formed by operators: if arity $((\vec{s}_1)s_1,\ldots,(\vec{s}_n)s_n)$, and if $\vec{x}_1$ is of sort $\vec{s}_1$ and $a_1 \in \mathcal{B}[\mathcal{X},\vec{x}_1]_{s_1}$ and $\ldots$ and $\vec{x}_n$ is of sort $\vec{s}_n$ and $a_n \in \mathcal{B}[\mathcal{X},\vec{x}_n]_{s_n}$, 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. The problem is that it takes too literally the names of the bound variables in an ast. In particular an abt of the form $\mathtt{let}(e_1;x.\mathtt{let}(e_2;x.e_3))$ is always ill-formed according to this definition, because the first binding adjoins x to $\mathcal{X}$, which implies that the second cannot adjoin x to $\mathcal{X},x$ because it is already present.

To ensure that the names of bound variables do not matter, the second condition on formation of abt's is strengthened as follows:[3]

> *if* for every $1 \leq i \leq n$ and for every renaming $\pi_i : \vec{x}_i \leftrightarrow \vec{x}'_i$ such that $\vec{x}'_i \notin \mathcal{X}$ we have $\pi_i \cdot a_i \in \mathcal{B}[\mathcal{X}, \vec{x}'_i]$, *then*
>
> $$o(\vec{x}_1 \,.\, a_1 \,;\, \ldots \,;\, \vec{x}_n \,.\, a_n) \in \mathcal{B}[\mathcal{X}].$$

That is, we demand that an abstractor be well-formed with respect to *every* choice of variables that are not already active. This ensures, for example, that when nesting binders we rename bound variables to avoid collisions. This is called the *freshness condition on binders*, since it chooses the bound variable names to be "fresh" relative to any variables already in use in a given context.

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

1. $\mathcal{X} \subseteq \mathcal{P}[\mathcal{X}]$.

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

This means that in the inductive hypothesis we may assume that the property $\mathcal{P}$ holds *for all* renamings of the bound variables, provided only that no confusion arises by re-using varaible names.

As an example let us define by structural induction modulo renaming the relation $x \in a$, where $a \in \mathcal{B}[\mathcal{X}, x]$, to mean that x occurs free in a. Speaking somewhat loosely, we may say that this judgement is defined by the following conditions:

1. $x \in y$ if $x = y$.

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

More precisely, we are defining a family of relations $x \in a$ for each family $\mathcal{X}$ of variables such that $a \in \mathcal{B}[\mathcal{X}, x]$. The first condition condition states that

[3] 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.

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 regardless of the choice of bound variables, then it is free in the abt constructed by an operator. This implies, in particular, that x is *not* free in $\mathtt{let}(\mathtt{zero}; x.x)$, since x is not free in z for any *fresh* choice of z, which is necessarily distinct from x.

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 α-equivalent. As a matter of terminology, if $a =_\alpha b$, then b is said to be an α-variant of a (and *vice-versa*).

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 jth bound variable of the ith argument. In such cases *substitution is undefined* since 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 α-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 α-equivalence classes of abt's.

This motivates the following general policy:

> *Abstract binding trees are always to be identified up to α-equivalence.*

That is, we henceforth work with equivalence classes of abt's modulo α-equivalence. Whenever a particular abt is considered, we choose a convenient representative of its α-equivalence class so that its bound variables are disjoint from the finite set of active variables in a particular context. We tacitly assert that all operations and relations on abt's respect α-equivalence, so that they are properly defined on α-equivalence classes of abt's. Thus, in particular, it makes no sense to speak of a particular bound variable within an abt, because bound variables have no fixed identity. Whenever we examine an abt, we are choosing a representative of its α-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 α-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 α-equivalence!

3.3 Parameterization

It is often useful to consider *indexed families* of operators in which indices range over a fixed set given in advance. For example, earlier we considered the family of nullary operators $\texttt{num}n$ of sort Expr, where $n \in \mathbb{N}$ is a natural number. As in this example, we shall generally use square brackets to mark the index of a family of operators, and round brackets to designate its arguments. For example, consider the family of operators $\texttt{seq}[n]$ of sort Expr with arity

$$(\underbrace{\mathsf{Expr},\ldots,\mathsf{Expr}}_{n}).$$

Then $\texttt{seq}[n](a_1;\ldots;a_n)$ is an ast of sort Expr, provided that each a_i is an ast of the same sort.

Another form of indexing in which the index set of the family is not specified in advance, but rather depends on the context of occurrence of the operator, arises frequently. Such operators are indexed by *symbols* (also called *names* or *atoms*) that are introduced within a scope by a binder. Unlike variables, symbols are not given meaning by substitution; rather, they serve solely as names for operators. In particular there is no notion of substitution for symbols.

As an example, we might consider a family of nullary operators $\texttt{cls}[a]$ of sort Expr indexed by symbols a. (Such an operator might represent a class of data viewed as a form of expression; see Chapter 15 for further development of this idea.) Whether such an operator is available depends on whether it occurs within the scope of the declaration of a. As new symbols are introduced, new operators become available; as the scope of a symbol is exited, operators that depend on it may no longer be used.

To allow for scoped parameterization of operators, we enrich the concept of an abstract binding tree to include binding of symbols, as well as variables. In addition to its sort, s, an operator may also have a sort, r, of parameters, and has an arity of the form

$$((\vec{r}_1;\vec{s}_1)s_1,\ldots,(\vec{r}_n;\vec{s}_n)s_n)$$

specifying that it takes n arguments of sort $s_1,\ldots,s_n$, respectively, and binds parameters of sorts $\vec{r}_i$ and variables of sorts $\vec{s}_i$ in the ith argument position. The set $\mathcal{B}[\mathcal{U};\mathcal{X}]$ of parameterized abt's is inductively defined by the following conditions:

1. $\mathcal{X} \subseteq \mathcal{B}[\mathcal{U};\mathcal{X}]$.

2. If $\mathsf{ar}(o) = ((\vec{r}_1;\vec{s}_1)s_1,\ldots,(\vec{r}_n;\vec{s}_n)s_n)$, $u \in \mathcal{U}_r$, $a_1 \in \mathcal{B}[\mathcal{U},\vec{u}_1;\mathcal{X},\vec{x}_1]$ and $\ldots$ and $a_n \in \mathcal{B}[\mathcal{U},\vec{u}_n;\mathcal{X},\vec{x}_n]$, then $o[u](\vec{u}_1\,.\,\vec{x}_1\,.\,a_1;\ldots;\vec{u}_n\,.\,\vec{x}_n\,.\,a_n) \in \mathcal{B}[\mathcal{U};\mathcal{X}]$.[4]

[4] More precisely, we must consider all possible fresh renamings of the bound parameters in a parameterized abt, just as we considered all possible fresh renamings of the bound variables in the definition of an abt.

The principle of structural induction modulo renaming extends to parameterized abt's in such a way that the names of bound parameters may be chosen arbitrarily to be fresh, just as may the names of bound variables be chosen arbitrarily in the induction principle for abt's. The relation of α-equivalence extends to parameterized abt's in the evident manner, relating any two abt's that differ only in their choice of bound parameter names. As with abt's, we tacitly identify parameterized abt's up to this extended notion of α-equivalence, and demand that all properties and operations on parameterized abt's respect α-equivalence.

3.4 Exercises

1. Show that for every $a \in \mathcal{B}[\mathcal{X}, x]$, either $x \in a$ or $x \notin a$ by structural induction modulo renaming on a.

Chapter 4

Generic Judgements

Basic judgements express properties of objects of the universe of discourse. Hypothetical judgements express entailments between judgements, or reasoning under hypotheses. *Generic* and *parametric* judgements express generality with respect to variables and parameters, respectively. Generic judgements are given meaning by substitution, whereas parametric judgements express uniform dependence on parameters.

4.1 Rule Schemes

An inductive definition consists of a set, $\mathcal{R}$, of rules whose premises and conclusion are judgements involving syntactic objects generated by given sets of parameters and variables. We write $\Gamma \vdash_{\mathcal{R}}^{\mathcal{U};\mathcal{X}} J$ to indicate that J is derivable from rules $\mathcal{R}$ and hypotheses Γ over the universe $\mathcal{B}[\mathcal{U};\mathcal{X}]$. Thus, for example, if $a \in \mathcal{B}[\mathcal{U};\mathcal{X}]$, then the judgment $a\ \mathtt{nat} \vdash \mathtt{succ}(a)\ \mathtt{nat}$ is derivable from Rules (1.2) by applying Rule (1.2b) to the hypothesis $a\ \mathtt{nat}$.

This definition hides a subtle issue of the interpretation of rules. When working over a fixed universe of syntactic objects, one may understand a rule of the form

$$\frac{a\ \mathtt{nat}}{\mathtt{succ}(a)\ \mathtt{nat}} \tag{4.1}$$

as standing for an infinite set of rules, one for each choice of object a in the universe. However, when considering the same rule over many different universes (for example, by expanding the set of variables), this rough-and-ready interpretation must be refined.

To allow for variation in the universe we regard (4.1) as a *rule scheme* in which the *meta-variable*, a, stands for a syntactic object in any expansion of the universe. So, for example, if the variable x is adjoined to the set of active variables, then (4.1) has as an instance the rule

$$\frac{x\ \mathsf{nat}}{\mathtt{succ}(x)\ \mathsf{nat}} \tag{4.2}$$

in which we have taken a to be the parameter, x. If we further adjoin another variable, y, then more instances of the rule are possible.

4.2 Generic Derivability

A *generic derivability* judgement expresses the *uniform* derivability of a judgement with respect to specified parameters and variables. Let us consider first variables, and expand out to accomodate parameters later. The generic derivability judgement $\vec{x} \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$ states that for every fresh renaming $\pi : \vec{x} \leftrightarrow \vec{x}'$, the judgement $\pi \cdot \Gamma \vdash^{\mathcal{X},\vec{x}'}_{\mathcal{R}} \pi \cdot J$ holds. The renaming ensures that the choice of variables, $\vec{x}$, does not affect the meaning of the judgement; variables are simply placeholders that have no intrinsic meaning of their own.

Evidence for a generic derivability judgement $\vec{x} \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$ consists of a *generic derivation*, $\nabla_{\vec{x}}$, such that for every fresh renaming $\pi : \vec{x} \leftrightarrow \vec{x}'$, the derivation $\nabla_{\vec{x}'}$ is evidence for $\pi \cdot \Gamma \vdash^{\mathcal{X},\vec{x}'}_{\mathcal{R}} \pi \cdot J$. For example, the derivation ∇_x given by

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

is evidence for the generic judgement

$$x \mid x\ \mathsf{nat} \vdash^{\mathcal{X}}_{(1.2)} \mathtt{succ}(\mathtt{succ}(x))\ \mathsf{nat}.$$

The generic derivability judgement enjoys the following *structural properties*:

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 $\vec{x} \mid [a/x]\Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} [a/x]J$.

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 judgement. Substitution follows from the meaning of a rule scheme, since a substitution instance of a rule instance is itself a rule instance.

4.3 Generic Inductive Definitions

A *generic inductive definition* admits generic hypothetical judgements 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

$$\frac{\vec{x}\,\vec{x}_1 \mid \Gamma\,\Gamma_1 \vdash J_1 \quad \dots \quad \vec{x}\,\vec{x}_n \mid \Gamma\,\Gamma_n \vdash J_n}{\vec{x} \mid \Gamma \vdash J}\ . \tag{4.3}$$

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 ith 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*,

$$\frac{\vec{x}_1 \mid \Gamma_1 \vdash J_1 \quad \dots \quad \vec{x}_n \mid \Gamma_n \vdash J_n}{J}\ . \tag{4.4}$$

A generic inductive definition is just an ordinary inductive definition of a family of *formal generic judgements* of the form $\vec{x} \mid \Gamma \vdash J$. Formal generic judgements are identified up to renaming of variables, so that the latter judgement is treated as identical to the judgement $\vec{x}' \mid \pi \cdot \Gamma \vdash \pi \cdot J$ 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 judgement $\vec{x} \mid \Gamma \vdash J$ 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 (4.3), we must show that

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

Because of the identification convention the property $\mathcal{P}$ must respect renamings of the variables in a formal generic judgement. It is common to use notations such as $\mathcal{P}_{\vec{x}}(\Gamma \vdash J)$ or $\mathcal{P}^{\Gamma}_{\vec{x}}(J)$ or similar variations to indicate that $\mathcal{P}$ holds of the judgement $\vec{x} \mid \Gamma \vdash J$.

To ensure that the formal generic judgement behaves like a generic judgement, we must always ensure that the following *structural rules* are admissible in any generic inductive definition:

$$\frac{}{\vec{x} \mid \Gamma, J \vdash J} \tag{4.5a}$$

$$\frac{\vec{x} \mid \Gamma \vdash J}{\vec{x} \mid \Gamma, J' \vdash J} \tag{4.5b}$$

$$\frac{\vec{x} \mid \Gamma \vdash J}{\vec{x}, x \mid \Gamma \vdash J} \tag{4.5c}$$

$$\frac{\vec{x}, x' \mid [x \leftrightarrow x'] \cdot \Gamma \vdash [x \leftrightarrow x'] \cdot J}{\vec{x}, x \mid \Gamma \vdash J} \tag{4.5d}$$

$$\frac{\vec{x} \mid \Gamma \vdash J \quad \vec{x} \mid \Gamma, J \vdash J'}{\vec{x} \mid \Gamma \vdash J'} \tag{4.5e}$$

$$\frac{\vec{x}, x \mid \Gamma \vdash J \quad a \in \mathcal{B}[\vec{x}]}{\vec{x} \mid [a/x]\Gamma \vdash [a/x]J} \tag{4.5f}$$

The admissibility of Rule (4.5a) is, in practice, ensured by explicitly including it. The admissibility of Rules (4.5b) and (4.5c) is assured if each of the generic rules is uniform, since we may assimilate the additional parameter, x, to the global parameters, and the additional hypothesis, J, to the global hypotheses. The admissibility of Rule (4.5d) is ensured by the *identification convention* for the formal generic judgement. The second premise of Rule (4.5f) is the local form of the requirement that $a \in \mathcal{B}[\mathcal{X}, \vec{x}]$, in which the global variables are made explicit.

4.4 Parametric Derivability

The *parametric derivability* judgement $\vec{u} \parallel \vec{x} \mid \Gamma \vdash_{\mathcal{R}}^{\mathcal{U};\mathcal{X}} J$ states that the generic judgement holds *uniformly* for all choices of parameters $\vec{u}$. That is, for all $\pi : \vec{u} \leftrightarrow \vec{u}'$ such that $\vec{u}' \cap \mathcal{U} = \emptyset$, the generic judgement $\vec{x} \mid \pi \cdot \Gamma \vdash_{\mathcal{R}}^{\mathcal{U},\vec{u}';\mathcal{X}} \pi \cdot J$ is derivable.

The parametric judgement satisfies the following *structural properties*:

Proliferation If $\vec{u} \parallel \vec{x} \mid \Gamma \vdash_{\mathcal{R}}^{\mathcal{U};\mathcal{X}} J$, then $\vec{u}, u \parallel \vec{x} \mid \Gamma \vdash_{\mathcal{R}}^{\mathcal{U};\mathcal{X}} J$.

Renaming If $\vec{u} \parallel \vec{x} \mid \Gamma \vdash_{\mathcal{R}}^{\mathcal{U};\mathcal{X}} J$ and $\pi : \vec{u} \leftrightarrow \vec{u}'$, then $\vec{u}' \parallel \vec{x} \mid \pi \cdot \Gamma \vdash_{\mathcal{R}}^{\mathcal{U};\mathcal{X}} \pi \cdot J$.

Proliferation states that parametric derivability is sensitive only to the presence, but not the absence, of parameters. Renaming states that parametric derivability is independent of the choice of parameters. (There is no analogue of the structural property of substitution for parameters.)

We may also extend the concept of a generic inductive definition to allow for local parameters, as well as local variables. To do so, rules are defined on formal parametric judgements of the form $\vec{u} \parallel \vec{x} \mid \Gamma \vdash J$, with parameters $\vec{u}$, as well as variables, $\vec{x}$. Such formal judgements are identified up to renaming of both its parameters and its variables to ensure that the meaning is independent of the choice of names. Usually we segregate the hypotheses into two *zones*, written $\vec{u} \parallel \vec{x} \mid \Sigma\ \Gamma \vdash J$, where Σ governs the parameters, $\vec{u}$, and Γ governs the variables, $\vec{x}$. Once separated into zones, it is natural to write this judgement in the form $\vec{x} \mid \Gamma \vdash_{\vec{u} \parallel \Sigma} J$, or even just $\Gamma \vdash_{\Sigma} J$, to reduce notational clutter.

4.5 Exercises

Part II

Levels of Syntax

Chapter 5

Concrete Syntax

The *concrete syntax* of a language is a means of representing expressions as strings that may be written on a page or entered using a keyboard. The concrete syntax usually is designed to enhance readability and to eliminate ambiguity. While there are good methods for eliminating ambiguity, improving readability is, to a large extent, a matter of taste.

In this chapter we introduce the main methods for specifying concrete syntax, using as an example an illustrative expression language, called $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, that supports elementary arithmetic on the natural numbers and simple computations on strings. In addition, $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ includes a construct for binding the value of an expression to a variable within a specified scope.

5.1 Strings Over An Alphabet

An *alphabet* is a (finite or infinite) collection of *characters*. We write c char to indicate that c is a character, and let Σ stand for a finite set of such judgements, which is sometimes called an *alphabet*. The judgement $\Sigma \vdash s\ \mathsf{str}$, defining the strings over the alphabet Σ, is inductively defined by the following rules:

$$\frac{}{\Sigma \vdash \epsilon\ \mathsf{str}} \tag{5.1a}$$

$$\frac{\Sigma \vdash c\ \mathsf{char} \quad \Sigma \vdash s\ \mathsf{str}}{\Sigma \vdash c \cdot s\ \mathsf{str}} \tag{5.1b}$$

Thus a string is essentially a list of characters, with the null string being the empty list. We often suppress explicit mention of Σ when it is clear from context.

When specialized to Rules (5.1), the principle of rule induction states that to show $s\ \mathcal{P}$ holds whenever s str, it is enough to show

1. $\epsilon\ \mathcal{P}$, and

2. if $s\ \mathcal{P}$ and c char, then $c \cdot s\ \mathcal{P}$.

This is sometimes called the principle of *string induction*. It is essentially equivalent to induction over the length of a string, except that there is no need to define the length of a string in order to use it.

The following rules constitute an inductive definition of the judgement $s_1 \mathbin{\hat{}} s_2 = s$ str, stating that s is the result of concatenating the strings s_1 and s_2.

$$\frac{}{\epsilon \mathbin{\hat{}} s = s \ \mathsf{str}} \tag{5.2a}$$

$$\frac{s_1 \mathbin{\hat{}} s_2 = s \ \mathsf{str}}{(c \cdot s_1) \mathbin{\hat{}} s_2 = c \cdot s \ \mathsf{str}} \tag{5.2b}$$

It is easy to prove by string induction on the first argument that this judgement has mode $(\forall, \forall, \exists!)$. Thus, it determines a total function of its first two arguments.

String concatenation is associative.

Lemma 5.1. *If* $s_1 \mathbin{\hat{}} s_2 = s_{12}$ str *and* $s_2 \mathbin{\hat{}} s_3 = s_{23}$ str, *then* $s_1 \mathbin{\hat{}} s_{23} = s$ str *and* $s_{12} \mathbin{\hat{}} s_3 = s$ str *for some (uniquely determined) string* s.

In Section 5.5 on page 48 we will see that this innocuous-seeming fact is responsible for most of the complications in defining the concrete syntax of a language.

Strings are usually written as juxtapositions of characters, writing just *abcd* for the four-letter string $a \cdot (b \cdot (c \cdot (d \cdot \epsilon)))$, for example. Concatentation is also written as juxtaposition, and individual characters are often identified with the corresponding unit-length string. This means that *abcd* can be thought of in many ways, for example as the concatenations *ab cd*, *a bcd*, or *abc d*, or even ϵ *abcd* or *abcd* ϵ, as may be convenient in a given situation.

5.2 Lexical Structure

The first phase of syntactic processing is to convert from a character-based representation to a symbol-based representation of the input. This is called *lexical analysis*, or *lexing*. The main idea is to aggregate characters into symbols that serve as tokens for subsequent phases of analysis. For example, the numeral `467` is written as a sequence of three consecutive characters, one for each digit, but is regarded as a single token, namely the number 467. Similarly, an identifier such as `temp` comprises four letters, but is treated as a single symbol representing the entire word. Moreover, many character-based representations include empty "white space" (spaces, tabs, newlines, and, perhaps, comments) that are discarded by the lexical analyzer.[1]

The lexical structure of a language is usually described using *regular expressions*. For example, the lexical structure of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ may be specified as follows:

$$\begin{array}{lrcl}
\text{Item} & \mathsf{itm} & ::= & \mathsf{kwd} \mid \mathsf{id} \mid \mathsf{num} \mid \mathsf{lit} \mid \mathsf{spl} \\
\text{Keyword} & \mathsf{kwd} & ::= & \mathtt{l}\cdot\mathtt{e}\cdot\mathtt{t}\cdot\epsilon \mid \mathtt{b}\cdot\mathtt{e}\cdot\epsilon \mid \mathtt{i}\cdot\mathtt{n}\cdot\epsilon \\
\text{Identifier} & \mathsf{id} & ::= & \mathsf{ltr}\,(\mathsf{ltr} \mid \mathsf{dig})^{*} \\
\text{Numeral} & \mathsf{num} & ::= & \mathsf{dig}\,\mathsf{dig}^{*} \\
\text{Literal} & \mathsf{lit} & ::= & \mathsf{qum}\,(\mathsf{ltr} \mid \mathsf{dig})^{*}\mathsf{qum} \\
\text{Special} & \mathsf{spl} & ::= & \mathtt{+} \mid \mathtt{*} \mid \mathtt{\hat{}} \mid \mathtt{(} \mid \mathtt{)} \mid \mathtt{|} \\
\text{Letter} & \mathsf{ltr} & ::= & \mathtt{a} \mid \mathtt{b} \mid \ldots \\
\text{Digit} & \mathsf{dig} & ::= & \mathtt{0} \mid \mathtt{1} \mid \ldots \\
\text{Quote} & \mathsf{qum} & ::= & \mathtt{"}
\end{array}$$

A lexical item is either a keyword, an identifier, a numeral, a string literal, or a special symbol. There are three keywords, specified as sequences of characters, for emphasis. Identifiers start with a letter and may involve subsequent letters or digits. Numerals are non-empty sequences of digits. String literals are sequences of letters or digits surrounded by quotes. The special symbols, letters, digits, and quote marks are as enumerated. (Observe that we tacitly identify a character with the unit-length string consisting of that character.)

The job of the lexical analyzer is to translate character strings into token strings using the above definitions as a guide. An input string is scanned, ignoring white space, and translating lexical items into tokens, which are specified by the following rules:

$$\frac{s\ \mathsf{str}}{\mathtt{ID[}s\mathtt{]}\ \mathsf{tok}} \tag{5.3a}$$

$$\frac{n\ \mathsf{nat}}{\mathtt{NUM[}n\mathtt{]}\ \mathsf{tok}} \tag{5.3b}$$

$$\frac{s\ \mathsf{str}}{\mathtt{LIT[}s\mathtt{]}\ \mathsf{tok}} \tag{5.3c}$$

[1] In some languages white space *is* significant, in which case it must be converted to symbolic form for subsequent processing.

$$\frac{}{\texttt{LET}\ \mathsf{tok}} \tag{5.3d}$$

$$\frac{}{\texttt{BE}\ \mathsf{tok}} \tag{5.3e}$$

$$\frac{}{\texttt{IN}\ \mathsf{tok}} \tag{5.3f}$$

$$\frac{}{\texttt{ADD}\ \mathsf{tok}} \tag{5.3g}$$

$$\frac{}{\texttt{MUL}\ \mathsf{tok}} \tag{5.3h}$$

$$\frac{}{\texttt{CAT}\ \mathsf{tok}} \tag{5.3i}$$

$$\frac{}{\texttt{LP}\ \mathsf{tok}} \tag{5.3j}$$

$$\frac{}{\texttt{RP}\ \mathsf{tok}} \tag{5.3k}$$

$$\frac{}{\texttt{VB}\ \mathsf{tok}} \tag{5.3l}$$

Rule (5.3a) admits any string as an identifier, even though only certain strings will be treated as identifiers by the lexical analyzer.

Lexical analysis is inductively defined by the following judgement forms:

$s\ \mathsf{charstr} \longleftrightarrow t\ \mathsf{tokstr}$	Scan input
$s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}$	Scan an item
$s\ \mathsf{kwd} \longleftrightarrow t\ \mathsf{tok}$	Scan a keyword
$s\ \mathsf{id} \longleftrightarrow t\ \mathsf{tok}$	Scan an identifier
$s\ \mathsf{num} \longleftrightarrow t\ \mathsf{tok}$	Scan a number
$s\ \mathsf{spl} \longleftrightarrow t\ \mathsf{tok}$	Scan a symbol
$s\ \mathsf{lit} \longleftrightarrow t\ \mathsf{tok}$	Scan a string literal

The definition of these forms, which follows, makes use of several auxiliary judgements corresponding to the classifications of characters in the lexical structure of the language. For example, s whs states that the string s consists only of "white space", s lord states that s is either an alphabetic letter or a digit, and s $\mathsf{non\text{-}lord}$ states that s does not begin with a letter or digit, and so forth.

$$\frac{}{\epsilon\ \mathsf{charstr} \longleftrightarrow \epsilon\ \mathsf{tokstr}} \tag{5.4a}$$

$$\frac{s = s_1\,\hat{}\,s_2\,\hat{}\,s_3\ \mathsf{str} \quad s_1\ \mathsf{whs} \quad s_2\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok} \quad s_3\ \mathsf{charstr} \longleftrightarrow ts\ \mathsf{tokstr}}{s\ \mathsf{charstr} \longleftrightarrow t \cdot ts\ \mathsf{tokstr}} \tag{5.4b}$$

$$\frac{s\ \mathsf{kwd} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{5.4c}$$

$$\frac{s\ \mathsf{id} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{5.4d}$$

$$\frac{s\ \mathsf{num} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{5.4e}$$

$$\frac{s\ \mathsf{lit} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{5.4f}$$

$$\frac{s\ \mathsf{spl} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{5.4g}$$

$$\frac{s = \mathtt{l} \cdot \mathtt{e} \cdot \mathtt{t} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{kwd} \longleftrightarrow \mathtt{LET}\ \mathsf{tok}} \tag{5.4h}$$

$$\frac{s = \mathtt{b} \cdot \mathtt{e} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{kwd} \longleftrightarrow \mathtt{BE}\ \mathsf{tok}} \tag{5.4i}$$

$$\frac{s = \mathtt{i} \cdot \mathtt{n} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{kwd} \longleftrightarrow \mathtt{IN}\ \mathsf{tok}} \tag{5.4j}$$

$$\frac{s = a \cdot s'\ \mathsf{str} \quad a\ \mathsf{ltr} \quad s'\ \mathsf{lds}}{s\ \mathsf{id} \longleftrightarrow \mathtt{ID}[s]\ \mathsf{tok}} \tag{5.4k}$$

$$\frac{s = s_1\,\hat{}\,s_2\ \mathsf{str} \quad s_1\ \mathsf{dig} \quad s_2\ \mathsf{dgs} \quad s\ \mathsf{num} \longleftrightarrow n\ \mathsf{nat}}{s\ \mathsf{num} \longleftrightarrow \mathtt{NUM}[n]\ \mathsf{tok}} \tag{5.4l}$$

$$\frac{s = s_1\,\hat{}\,s_2\,\hat{}\,s_3\ \mathsf{str} \quad s_1\ \mathsf{qum} \quad s_2\ \mathsf{lord} \quad s_3\ \mathsf{qum}}{s\ \mathsf{lit} \longleftrightarrow \mathtt{LIT}[s_2]\ \mathsf{tok}} \tag{5.4m}$$

$$\frac{s = \mathtt{+} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{spl} \longleftrightarrow \mathtt{ADD}\ \mathsf{tok}} \tag{5.4n}$$

$$\frac{s = \mathtt{*} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{spl} \longleftrightarrow \mathtt{MUL}\ \mathsf{tok}} \tag{5.4o}$$

$$\frac{s = \hat{}\, \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{spl} \longleftrightarrow \mathtt{CAT}\ \mathsf{tok}} \tag{5.4p}$$

$$\frac{s = \mathtt{(} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{spl} \longleftrightarrow \mathtt{LP}\ \mathsf{tok}} \tag{5.4q}$$

$$\frac{s = \mathtt{)} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{spl} \longleftrightarrow \mathtt{RP}\ \mathsf{tok}} \tag{5.4r}$$

$$\frac{s = \mathtt{|} \cdot \epsilon\ \mathsf{str}}{s\ \mathsf{spl} \longleftrightarrow \mathtt{VB}\ \mathsf{tok}} \tag{5.4s}$$

Rules (5.4) do not specify a *deterministic* algorithm. Rather, Rule (5.4b) applies whenever the input string may be partitioned into three parts, consisting of white space, a lexical item, and the rest of the input. However, the associativity of string concatenation implies that the partititioning is not unique. For example, the string insert may be partitioned as in^sert or insert^ϵ, and hence tokenized as either IN followed by ID[sert], or as ID[insert] (or, indeed, as two consecutive identifiers in several ways).

One solution to this problem is to impose some extrinsic control criteria on the rules to ensure that they have a unique interpretation. For example, one may insist that Rule (5.4b) apply only when the string s_2 is chosen to be as long as possible so as to ensure that the string `insert` is analyzed as the identifier `ID[insert]`, rather than as two consecutive identifiers, say `ID[ins]` and `ID[ert]`. Moreover, we may impose an ordering on the rules, so that so that Rule (5.4j) takes priority over Rule (5.4k) so as to avoid interpreting `in` as an identifier, rather than as a keyword. Another solution is to reformulate the rules so that they are completely deterministic, a technique that will be used in the next section to resolve a similar ambiguity at the level of the concrete syntax.

5.3 Context-Free Grammars

The standard method for defining concrete syntax is by giving a *context-free grammar* for the language. A grammar consists of three components:

1. The *tokens*, or *terminals*, over which the grammar is defined.

2. The *syntactic classes*, or *non-terminals*, which are disjoint from the terminals.

3. The *rules*, or *productions*, which have the form $A ::= \alpha$, where A is a non-terminal and α is a string of terminals and non-terminals.

Each syntactic class is a collection of token strings. The rules determine which strings belong to which syntactic classes.

When defining a grammar, we often abbreviate a set of productions,

$$\begin{aligned} A &::= \alpha_1 \\ &\vdots \\ A &::= \alpha_n, \end{aligned}$$

each with the same left-hand side, by the *compound* production

$$A ::= \alpha_1 \mid \ldots \mid \alpha_n,$$

which specifies a set of alternatives for the syntactic class A.

A context-free grammar determines a simultaneous inductive definition of its syntactic classes. Specifically, we regard each non-terminal, A, as a judgement form, $s\ A$, over strings of terminals. To each production of the form

$$A ::= s_1\, A_1\, s_2\, \ldots\, s_n\, A_n\, s_{n+1} \tag{5.5}$$

we associate an inference rule

$$\frac{s_1'\ A_1 \quad \dots \quad s_n'\ A_n}{s_1\, s_1'\, s_2\ \dots\ s_n\, s_n'\, s_{n+1}\ A}\ . \tag{5.6}$$

The collection of all such rules constitutes an inductive definition of the syntactic classes of the grammar.

Recalling that juxtaposition of strings is short-hand for their concatenation, we may re-write the preceding rule as follows:

$$\frac{s_1'\ A_1 \quad \dots \quad s_n'\ A_n \quad s = s_1\,\hat{}\,s_1'\,\hat{}\,s_2\,\hat{}\dots s_n\,\hat{}\,s_n'\,\hat{}\,s_{n+1}}{s\ A}\ . \tag{5.7}$$

This formulation makes clear that $s\ A$ holds whenever s can be partitioned as described so that $s_i'\ A$ for each $1 \leq i \leq n$. Since string concatenation is associative, the decomposition is not unique, and so there may be many different ways in which the rule applies.

5.4 Grammatical Structure

The concrete syntax of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ may be specified by a context-free grammar over the tokens defined in Section 5.2 on page 42. The grammar has only one syntactic class, exp, which is defined by the following compound production:

$$\begin{array}{lrcl}
\text{Expression} & \mathsf{exp} & ::= & \mathsf{num} \mid \mathsf{lit} \mid \mathsf{id} \mid \texttt{LP}\,\mathsf{exp}\,\texttt{RP} \mid \mathsf{exp}\,\texttt{ADD}\,\mathsf{exp} \mid \\
 & & & \mathsf{exp}\,\texttt{MUL}\,\mathsf{exp} \mid \mathsf{exp}\,\texttt{CAT}\,\mathsf{exp} \mid \texttt{VB}\,\mathsf{exp}\,\texttt{VB} \mid \\
 & & & \texttt{LET}\,\mathsf{id}\,\texttt{BE}\,\mathsf{exp}\,\texttt{IN}\,\mathsf{exp} \\
\text{Number} & \mathsf{num} & ::= & \texttt{NUM[}n\texttt{]} \qquad (n\ \mathsf{nat}) \\
\text{String} & \mathsf{lit} & ::= & \texttt{LIT[}s\texttt{]} \qquad (s\ \mathsf{str}) \\
\text{Identifier} & \mathsf{id} & ::= & \texttt{ID[}s\texttt{]} \qquad (s\ \mathsf{str})
\end{array}$$

This grammar makes use of some standard notational conventions to improve readability: we identify a token with the corresponding unit-length string, and we use juxtaposition to denote string concatenation.

Applying the interpretation of a grammar as an inductive definition, we obtain the following rules:

$$\frac{s\ \mathsf{num}}{s\ \mathsf{exp}} \tag{5.8a}$$

$$\frac{s\ \mathsf{lit}}{s\ \mathsf{exp}} \tag{5.8b}$$

$$\frac{s\ \mathsf{id}}{s\ \mathsf{exp}} \tag{5.8c}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\, \mathtt{ADD}\, s_2\ \mathsf{exp}} \tag{5.8d}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\, \mathtt{MUL}\, s_2\ \mathsf{exp}} \tag{5.8e}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\, \mathtt{CAT}\, s_2\ \mathsf{exp}} \tag{5.8f}$$

$$\frac{s\ \mathsf{exp}}{\mathtt{VB}\, s\, \mathtt{VB}\ \mathsf{exp}} \tag{5.8g}$$

$$\frac{s\ \mathsf{exp}}{\mathtt{LP}\, s\, \mathtt{RP}\ \mathsf{exp}} \tag{5.8h}$$

$$\frac{s_1\ \mathsf{id} \quad s_2\ \mathsf{exp} \quad s_3\ \mathsf{exp}}{\mathtt{LET}\, s_1\, \mathtt{BE}\, s_2\, \mathtt{IN}\, s_3\ \mathsf{exp}} \tag{5.8i}$$

$$\frac{n\ \mathsf{nat}}{\mathtt{NUM}[n]\ \mathsf{num}} \tag{5.8j}$$

$$\frac{s\ \mathsf{str}}{\mathtt{LIT}[s]\ \mathsf{lit}} \tag{5.8k}$$

$$\frac{s\ \mathsf{str}}{\mathtt{ID}[s]\ \mathsf{id}} \tag{5.8l}$$

To emphasize the role of string concatentation, we may rewrite Rule (5.8e), for example, as follows:

$$\frac{s = s_1\, \mathtt{MUL}\, s_2\ \mathsf{str} \quad s_1\ \mathsf{exp} \qquad s_2\ \mathsf{exp}}{s\ \mathsf{exp}}\ . \tag{5.9}$$

That is, s exp is derivable if s is the concatentation of s_1, the multiplication sign, and s_2, where s_1 exp and s_2 exp.

5.5 Ambiguity

Apart from subjective matters of readability, a principal goal of concrete syntax design is to avoid ambiguity. The grammar of arithmetic expressions given above is *ambiguous* in the sense that some token strings may be thought of as arising in several different ways. More precisely, there are token strings s for which there is more than one derivation ending with s exp according to Rules (5.8).

For example, consider the character string 1+2*3, which, after lexical analysis, is translated to the token string

$$\mathtt{NUM[1]\ ADD\ NUM[2]\ MUL\ NUM[3]}.$$

Since string concatenation is associative, this token string can be thought of as arising in several ways, including

$$\texttt{NUM[1] ADD}_{\wedge}\texttt{NUM[2] MUL NUM[3]}$$

and

$$\texttt{NUM[1] ADD NUM[2]}_{\wedge}\texttt{MUL NUM[3]},$$

where the caret indicates the concatenation point.

One consequence of this observation is that the same token string may be seen to be grammatical according to the rules given in Section 5.4 on page 47 in two different ways. According to the first reading, the expression is principally an addition, with the first argument being a number, and the second being a multiplication of two numbers. According to the second reading, the expression is principally a multiplication, with the first argument being the addition of two numbers, and the second being a number.

Ambiguity is a *purely syntactic* property of grammars; it has nothing to do with the "meaning" of a string. For example, the token string

$$\texttt{NUM[1] ADD NUM[2] ADD NUM[3]},$$

also admits two readings. It is immaterial that both readings have the same meaning under the usual interpretation of arithmetic expressions. Moreover, nothing prevents us from interpreting the token ADD to mean "division," in which case the two readings would hardly coincide! Nothing in the syntax itself precludes this interpretation, so we do not regard it as relevant to whether the grammar is ambiguous.

To avoid ambiguity the grammar of $\mathcal{L}\{\texttt{num str}\}$ given in Section 5.4 on page 47 must be re-structured to ensure that every grammatical string has at most one derivation according to the rules of the grammar. The main method for achieving this is to introduce precedence and associativity conventions that ensure there is only one reading of any token string. Parenthesization may be used to override these conventions, so there is no fundamental loss of expressive power in doing so.

Precedence relationships are introduced by *layering* the grammar, which is achieved by splitting syntactic classes into several subclasses.

Factor	fct	::=	num | lit | id | LP prg RP
Term	trm	::=	fct | fct MUL trm | VB fct VB
Expression	exp	::=	trm | trm ADD exp | trm CAT exp
Program	prg	::=	exp | LET id BE exp IN prg

The effect of this grammar is to ensure that `let` has the lowest precedence, addition and concatenation intermediate precedence, and multiplication and length the highest precedence. Moreover, all forms are right-associative. Other choices of rules are possible, according to taste; this grammar illustrates one way to resolve the ambiguities of the original expression grammar.

5.6 Exercises

Chapter 6

Abstract Syntax

The concrete syntax of a language is concerned with the linear representation of the phrases of a language as strings of symbols—the form in which we write them on paper, type them into a computer, and read them from a page. But languages are also the subjects of study, as well as the instruments of expression. As such the concrete syntax of a language is just a nuisance. When analyzing a language mathematically we are only interested in the deep structure of its phrases, not their surface representation. The abstract syntax of a language exposes the hierarchical and binding structure of the language. *Parsing* is the process of translation from concrete to abstract syntax. It consists of analyzing the linear representation of a phrase in terms of the grammar of the language and transforming it into an abstract syntax tree or an abstract binding tree that reveals the deep structure of the phrase. *Formatting* is the inverse process of generating a linear representation of a given piece of abstract syntax.

6.1 Hierarchical and Binding Structure

For the purposes of analysis the most important elements of the syntax of a language are its *hierarchical* and *binding* structure. Ignoring binding and scope, the hierarchical structure of a language may be expressed using abstract syntax trees. Accounting for these requires the additional structure of abstract binding trees. We will define both an ast and an abt representation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ in order to compare the two and show how they relate to the concrete syntax described in Chapter 5.

The purely hierarchical abstract syntax of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is generated by the following operators and their arities:

num[n]	()	(n nat)
str[s]	()	(s str)
id[s]	()	(s str)
times	(Expr, Expr)	
plus	(Expr, Expr)	
len	(Expr)	
cat	(Expr, Expr)	
let[s]	(Expr, Expr)	(s str)

There is one sort, Expr, generated by the above operators. For each n nat there is an operator num[n] of arity () representing the number n. Similarly, for each s str there is an operator str[s] of arity (), representing a string literal. There are several operators corresponding to functions on numbers and strings.

Most importantly, there are two operators related to identifiers. The first, id[s], where s str, represents the identifier with name s thought of as an operator of arity (). The second, let[s], is a family of operators indexed by s str with two arguments, the binding of the identifier id[s] and the scope of that binding. These characterizations, however, are *purely informal* in that there is nothing in the "plain" abstract syntax of the language that supports these interpretations. In particular, there is no connection between any occurrences of id[s] and any occurrence of let[s] within an expression.

To account for the binding and scope of identifiers requires the greater expressive power of abstract binding trees. An abt representation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is defined by the following operators and their arities:

num[n]	()	(n nat)
str[s]	()	(s str)
times	(Expr, Expr)	
plus	(Expr, Expr)	
len	(Expr)	
cat	(Expr, Expr)	
let	(Expr, (Expr)Expr)	

There is no longer an operator id[s]; we instead use a variable to refer to a binding site. Correspondingly, the family of operators let[s] is repalced replaced by a single operator, let, of arity (Expr, (Expr)Expr), which binds a variable in its second argument.

To illustrate the relationship between these two representations of the abstract syntax of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, we will first describe the translation from the concrete syntax, given in Chapter 5, to an abstract syntax tree. We will then alter this translation to account for binding and scope, yielding an abstract binding tree.

6.2 Parsing Into Abstract Syntax Trees

We will simultaneously define parsing and formatting as a binary judgement relating the concrete to the abstract syntax. This judgement will have the mode $(\forall, \exists^{\leq 1})$, which states that the parser is a partial function of its input, being undefined for ungrammatical token strings, but otherwise uniquely determining the abstract syntax tree representation of each well-formed input. It will also have the mode $(\exists, \forall)$, which states that each piece of abstract syntax has a (not necessarily unique) representation as a token string in the concrete syntax.

The parsing judgements for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ follow the unambiguous grammar given in Chapter 5:

$s\ \mathsf{prg} \longleftrightarrow e\ \mathsf{expr}$	Parse/format as a program
$s\ \mathsf{exp} \longleftrightarrow e\ \mathsf{expr}$	Parse/format as an expression
$s\ \mathsf{trm} \longleftrightarrow e\ \mathsf{expr}$	Parse/format as a term
$s\ \mathsf{fct} \longleftrightarrow e\ \mathsf{expr}$	Parse/format as a factor
$s\ \mathsf{num} \longleftrightarrow e\ \mathsf{expr}$	Parse/format as a number
$s\ \mathsf{lit} \longleftrightarrow e\ \mathsf{expr}$	Parse/format as a literal
$s\ \mathsf{id} \longleftrightarrow e\ \mathsf{expr}$	Parse/format as an identifier

These judgements relate a token string, s, to an expression, e, viewed as an abstract syntax tree.

These judgements are inductively defined simultaneously by the following rules:

$$\frac{n\ \mathsf{nat}}{\texttt{NUM}[n]\ \mathsf{num} \longleftrightarrow \texttt{num}[n]\ \mathsf{expr}} \tag{6.1a}$$

$$\frac{s\ \mathsf{str}}{\texttt{LIT}[s]\ \mathsf{lit} \longleftrightarrow \texttt{str}[s]\ \mathsf{expr}} \tag{6.1b}$$

$$\frac{s\ \mathsf{str}}{\texttt{ID}[s]\ \mathsf{id} \longleftrightarrow \texttt{id}[s]\ \mathsf{expr}} \tag{6.1c}$$

$$\frac{s\ \mathsf{num} \longleftrightarrow e\ \mathsf{expr}}{s\ \mathsf{fct} \longleftrightarrow e\ \mathsf{expr}} \tag{6.1d}$$

$$\frac{s\ \mathsf{lit} \longleftrightarrow e\ \mathsf{expr}}{s\ \mathsf{fct} \longleftrightarrow e\ \mathsf{expr}} \tag{6.1e}$$

$$\frac{s\ \mathsf{id} \longleftrightarrow e\ \mathsf{expr}}{s\ \mathsf{fct} \longleftrightarrow e\ \mathsf{expr}} \tag{6.1f}$$

$$\frac{s\ \mathsf{prg} \longleftrightarrow e\ \mathsf{expr}}{\mathtt{LP}\, s\, \mathtt{RP}\ \mathsf{fct} \longleftrightarrow e\ \mathsf{expr}} \tag{6.1g}$$

$$\frac{s\ \mathsf{fct} \longleftrightarrow e\ \mathsf{expr}}{s\ \mathsf{trm} \longleftrightarrow e\ \mathsf{expr}} \tag{6.1h}$$

$$\frac{s_1\ \mathsf{fct} \longleftrightarrow e_1\ \mathsf{expr} \quad s_2\ \mathsf{trm} \longleftrightarrow e_2\ \mathsf{expr}}{s_1\, \mathtt{MUL}\, s_2\ \mathsf{trm} \longleftrightarrow \mathtt{times}(e_1;e_2)\ \mathsf{expr}} \tag{6.1i}$$

$$\frac{s\ \mathsf{fct} \longleftrightarrow e\ \mathsf{expr}}{\mathtt{VB}\, s\, \mathtt{VB}\ \mathsf{trm} \longleftrightarrow \mathtt{len}(e)\ \mathsf{expr}} \tag{6.1j}$$

$$\frac{s\ \mathsf{trm} \longleftrightarrow e\ \mathsf{expr}}{s\ \mathsf{exp} \longleftrightarrow e\ \mathsf{expr}} \tag{6.1k}$$

$$\frac{s_1\ \mathsf{trm} \longleftrightarrow e_1\ \mathsf{expr} \quad s_2\ \mathsf{exp} \longleftrightarrow e_2\ \mathsf{expr}}{s_1\, \mathtt{ADD}\, s_2\ \mathsf{exp} \longleftrightarrow \mathtt{plus}(e_1;e_2)\ \mathsf{expr}} \tag{6.1l}$$

$$\frac{s_1\ \mathsf{trm} \longleftrightarrow e_1\ \mathsf{expr} \quad s_2\ \mathsf{exp} \longleftrightarrow e_2\ \mathsf{expr}}{s_1\, \mathtt{CAT}\, s_2\ \mathsf{exp} \longleftrightarrow \mathtt{cat}(e_1;e_2)\ \mathsf{expr}} \tag{6.1m}$$

$$\frac{s\ \mathsf{exp} \longleftrightarrow e\ \mathsf{expr}}{s\ \mathsf{prg} \longleftrightarrow e\ \mathsf{expr}} \tag{6.1n}$$

$$\frac{s_1\ \mathsf{id} \longleftrightarrow \mathtt{id}[s]\ \mathsf{expr} \quad s_2\ \mathsf{exp} \longleftrightarrow e_2\ \mathsf{expr} \quad s_3\ \mathsf{prg} \longleftrightarrow e_3\ \mathsf{expr}}{\mathtt{LET}\, s_1\, \mathtt{BE}\, s_2\, \mathtt{IN}\, s_3\ \mathsf{prg} \longleftrightarrow \mathtt{let}[s](e_2;e_3)\ \mathsf{expr}} \tag{6.1o}$$

A successful parse implies that the token string must have been derived according to the rules of the unambiguous grammar and that the result is a well-formed abstract syntax tree.

Theorem 6.1. *If $s\ \mathsf{prg} \longleftrightarrow e\ \mathsf{expr}$, then $s\ \mathsf{prg}$ and $e\ \mathsf{expr}$, and similarly for the other parsing judgements.*

Proof. By a straightforward induction on Rules (6.1). □

Moreover, if a string is generated according to the rules of the grammar, then it has a parse as an ast.

Theorem 6.2. *If $s\ \mathsf{prg}$, then there is a unique e such that $s\ \mathsf{prg} \longleftrightarrow e\ \mathsf{expr}$, and similarly for the other parsing judgements. That is, the parsing judgements have mode $(\forall, \exists!)$ over well-formed strings and abstract syntax trees.*

Proof. By rule induction on the rules determined by reading Grammar (5.5) as an inductive definition. □

Finally, any piece of abstract syntax may be formatted as a string that parses as the given ast.

Theorem 6.3. *If e expr, then there exists a (not necessarily unique) string s such that s prg and s $\mathsf{prg} \longleftrightarrow e$ expr. That is, the parsing judgement has mode $(\exists, \forall)$.*

Proof. By rule induction on Grammar (5.5). □

The string representation of an abstract syntax tree is not unique, since we may introduce parentheses at will around any sub-expression.

6.3 Parsing Into Abstract Binding Trees

In this section we revise the parser given in Section 6.2 on page 53 to translate from token strings to abstract binding trees to make explicit the binding and scope of identifiers in a program. The revised parsing judgement, s $\mathsf{prg} \longleftrightarrow e$ expr, between strings s and abt's e, is defined by a collection of rules similar to those given in Section 6.2 on page 53. These rules take the form of a generic inductive definition (see Chapter 2) in which the premises and conclusions of the rules involve hypothetical judgments of the form

$$\mathtt{ID[}s_1\mathtt{]}\ \mathsf{id} \longleftrightarrow x_1\ \mathsf{expr}, \ldots, \mathtt{ID[}s_n\mathtt{]}\ \mathsf{id} \longleftrightarrow x_n\ \mathsf{expr} \vdash s\ \mathsf{prg} \longleftrightarrow e\ \mathsf{expr},$$

where the x_i's are pairwise distinct variable names. The hypotheses of the judgement dictate how identifiers are to be parsed as variables, for it follows from the reflexivity of the hypothetical judgement that

$$\Gamma, \mathtt{ID[}s\mathtt{]}\ \mathsf{id} \longleftrightarrow x\ \mathsf{expr} \vdash \mathtt{ID[}s\mathtt{]}\ \mathsf{id} \longleftrightarrow x\ \mathsf{expr}.$$

To maintain the association between identifiers and variables when parsing a `let` expression, we update the hypotheses to record the association between the bound identifier and a corresponding variable:

$$\frac{\begin{array}{c}\Gamma \vdash s_1\ \mathsf{id} \longleftrightarrow x\ \mathsf{expr} \qquad \Gamma \vdash s_2\ \mathsf{exp} \longleftrightarrow e_2\ \mathsf{expr} \\ \Gamma, s_1\ \mathsf{id} \longleftrightarrow x\ \mathsf{expr} \vdash s_3\ \mathsf{prg} \longleftrightarrow e_3\ \mathsf{expr}\end{array}}{\Gamma \vdash \mathtt{LET}\, s_1\, \mathtt{BE}\, s_2\, \mathtt{IN}\, s_3\ \mathsf{prg} \longleftrightarrow \mathtt{let(}e_2\mathtt{;}x\mathtt{.}e_3\mathtt{)}\ \mathsf{expr}} \tag{6.2a}$$

Unfortunately, this approach does not quite work properly! If an inner `let` expression binds the same identifier as an outer `let` expression, there is an ambiguity in how to parse occurrences of that identifier. Parsing such nested `let`'s will introduce two hypotheses, say $\mathtt{ID[}s\mathtt{]}\ \mathsf{id} \longleftrightarrow x_1\ \mathsf{expr}$ and

$\mathtt{ID[}s\mathtt{]}$ $\mathsf{id} \longleftrightarrow x_2$ expr, for the same identifier $\mathtt{ID[}s\mathtt{]}$. By the structural property of exchange, we may choose arbitrarily which to apply to any particular occurrence of $\mathtt{ID[}s\mathtt{]}$, and hence we may parse different occurrences differently.

To rectify this we resort to less elegant methods. Rather than use hypotheses, we instead maintain an explicit *symbol table* to record the association between identifiers and variables. We must define explicitly the procedures for creating and extending symbol tables, and for looking up an identifier in the symbol table to determine its associated variable. This gives us the freedom to implement a *shadowing* policy for re-used identifiers, according to which the most recent binding of an identifier determines the corresponding variable.

The main change to the parsing judgement is that the hypothetical judgement

$$\Gamma \vdash s\ \mathsf{prg} \longleftrightarrow e\ \mathsf{expr}$$

is reduced to the basic judgement

$$s\ \mathsf{prg} \longleftrightarrow e\ \mathsf{expr}\ [\sigma],$$

where σ is a symbol table. (Analogous changes must be made to the other parsing judgements.) The symbol table is now an argument to the judgement form, rather than an implicit mechanism for performing inference under hypotheses.

The rule for parsing `let` expressions is then formulated as follows:

$$\frac{\begin{array}{cc} s_1\ \mathsf{id} \longleftrightarrow x\ [\sigma] & s_2\ \mathsf{exp} \longleftrightarrow e_2\ \mathsf{expr}\ [\sigma] \\ \sigma' = \sigma[s_1 \mapsto x] & s_3\ \mathsf{prg} \longleftrightarrow e_3\ \mathsf{expr}\ [\sigma'] \end{array}}{\mathtt{LET}\, s_1\, \mathtt{BE}\, s_2\, \mathtt{IN}\, s_3\ \mathsf{prg} \longleftrightarrow \mathtt{let}(e_2; x\,.\,e_3)\ \mathsf{expr}\ [\sigma]} \tag{6.3}$$

This rule is quite similar to the hypothetical form, the difference being that we must manage the symbol table explicitly. In particular, we must include a rule for parsing identifiers, rather than relying on the reflexivity of the hypothetical judgement to do it for us.

$$\frac{\sigma(\mathtt{ID[}s\mathtt{]}) = x}{\mathtt{ID[}s\mathtt{]}\ \mathsf{id} \longleftrightarrow x\ [\sigma]} \tag{6.4}$$

The premise of this rule states that σ maps the identifier $\mathtt{ID[}s\mathtt{]}$ to the variable x.

Symbol tables may be defined to be finite sequences of ordered pairs of the form $(\mathtt{ID[}s\mathtt{]}, x)$, where $\mathtt{ID[}s\mathtt{]}$ is an identifier and x is a variable name. Using this representation it is straightforward to define the following judgement forms:

σ symtab	well-formed symbol table
$\sigma' = \sigma[\texttt{ID[}s\texttt{]} \mapsto x]$	add new association
$\sigma(\texttt{ID[}s\texttt{]}) = x$	lookup identifier

We leave the precise definitions of these judgements as an exercise for the reader.

6.4 Exercises

Part III

Statics and Dynamics

Chapter 7

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 judgements* 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 the language $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ as an illustration of the methodology that we shall employ throughout this book.

7.1 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 perspicuity of examples, 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 illustrated by example. The following chart summarizes the abstract and concrete syntax of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$, which was analyzed in detail in Chapters 5 and 6.

Type	τ	$::=$	$\mathtt{num}$	$\mathtt{num}$	numbers
			$\mathtt{str}$	$\mathtt{str}$	strings
Expr	e	$::=$	x	x	variable
			$\mathtt{num}[n]$	n	numeral
			$\mathtt{str}[s]$	$''s''$	literal
			$\mathtt{plus}(e_1;e_2)$	$e_1 + e_2$	addition
			$\mathtt{times}(e_1;e_2)$	$e_1 * e_2$	multiplication
			$\mathtt{cat}(e_1;e_2)$	$e_1\,\hat{}\,e_2$	concatenation
			$\mathtt{len}(e)$	$\mathtt{\vert}e\mathtt{\vert}$	length
			$\mathtt{let}(e_1;x.e_2)$	$\mathtt{let}\,x\,\mathtt{be}\,e_1\,\mathtt{in}\,e_2$	definition

There are two sorts, Type ranged over by the meta-variable τ, and Expr, ranged over by the meta-variable e. The meta-variable x ranges over variables of sort Expr. The chart defines a number of operators and their arities. For example, the operator `let` has arity (Expr, (Expr)Expr), which specifies that it has two arguments of sort Expr, and binds a variable of sort Expr in the second argument.

7.2 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 $\mathtt{plus}(x;\mathtt{num}[n])$ is sensible depends on whether or not the variable x is declared 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 $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ consists of an inductive definition of generic hypothetical judgements of the form

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

where $\vec{x}$ is a finite set of variables, and Γ 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 for variables that serve as parameters of the typing judgement. We write $x \notin dom(\Gamma)$ to indicate that there is no assumption in Γ of the form $x : \tau$ for any type τ, in which case we say that the variable x is *fresh* for Γ.

The rules defining the statics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ are as follows:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{7.1a}$$

$$\frac{}{\Gamma \vdash \texttt{str[}s\texttt{]} : \texttt{str}} \tag{7.1b}$$

$$\frac{}{\Gamma \vdash \texttt{num[}n\texttt{]} : \texttt{num}} \tag{7.1c}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{plus(}e_1\texttt{;}e_2\texttt{)} : \texttt{num}} \tag{7.1d}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{times(}e_1\texttt{;}e_2\texttt{)} : \texttt{num}} \tag{7.1e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \quad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash \texttt{cat(}e_1\texttt{;}e_2\texttt{)} : \texttt{str}} \tag{7.1f}$$

$$\frac{\Gamma \vdash e : \texttt{str}}{\Gamma \vdash \texttt{len(}e\texttt{)} : \texttt{num}} \tag{7.1g}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} : \tau_2} \tag{7.1h}$$

In Rule (7.1h) we tacitly assume that the variable, x, is not already declared in Γ. This condition may always be met by choosing a suitable representative of the α-equivalence class of the `let` expression.

Rules (7.1) illustrate an important organizational principle, called the *principle of introduction and elimination*, for a type system. The constructs of the language may be classified into one of two forms associated with each type. The *introductory* forms of a type are the means by which values of that type are created, or introduced. In the case of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, the introductory forms for the type `num` are the numerals, $\texttt{num[}n\texttt{]}$, and for the type `str` are the literals, $\texttt{str[}s\texttt{]}$. The *eliminatory* forms of a type are the means by which we may compute with values of that type to obtain values of some (possibly different) type. In the present case the eliminatory forms for the type `num` are addition and multiplication, and for the type `str` are concatenation and length. Each eliminatory form has one or more *principal* arguments of associated type, and zero or more *non-principal* arguments. In the present case all arguments for each of the eliminatory forms is principal, but we shall later see examples in which there are also non-principal arguments for eliminatory forms.

It is easy to check that every expression has at most one type.

Lemma 7.1 (Unicity of Typing)**.** *For every typing context* Γ *and expression* e*, there exists at most one* τ *such that* $\Gamma \vdash e : \tau$*.*

Proof. By rule induction on Rules (7.1). □

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

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

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

7.3 Structural Properties

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

Lemma 7.3 (Weakening)**.** *If* $\Gamma \vdash e' : \tau'$*, then* $\Gamma, x : \tau \vdash e' : \tau'$ *for any* $x \notin dom(\Gamma)$ *and any type* τ*.*

Proof. By induction on the derivation of $\Gamma \vdash e' : \tau'$. We will give one case here, for rule (7.1h). We have that $e' = \texttt{let}(e_1;z.e_2)$, 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. $\Gamma, x : \tau, z : \tau_1 \vdash e_2 : \tau'$,

from which the result follows by Rule (7.1h). □

Lemma 7.4 (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 (7.1h). 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

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

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

It follows by Rule (7.1h) that $\Gamma \vdash [e/x]\mathtt{let}(e_1;z.e_2) : \tau$, as desired. □

From a programming point of view, Lemma 7.3 on the facing page allows us to use an expression in any context that binds its free variables: if e is well-typed in a context Γ, then we may "import" it into any context that includes the assumptions Γ. 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 7.4 on the preceding page 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 7.4 on the facing page 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 7.5 (Decomposition). *If $\Gamma \vdash [e/x]e' : \tau'$, then for every type τ 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. □

This lemma 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'.

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

7.4 Exercises

1. Show that the expression $e = \mathtt{plus(num[7];str[}\mathit{abc}\mathtt{])}$ is ill-typed in that there is no τ such that $e : \tau$.

Chapter 8

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 equivalent* to another.

8.1 Transition Systems

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

1. $s\ \mathsf{state}$, asserting that s is a *state* of the transition system.

2. $s\ \mathsf{final}$, where $s\ \mathsf{state}$, asserting that s is a *final* state.

3. $s\ \mathsf{initial}$, where $s\ \mathsf{state}$, asserting that s is an *initial* state.

4. $s \longmapsto s'$, where $s\ \mathsf{state}$ and $s'\ \mathsf{state}$, 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 $s\ \mathsf{final}$, then there is no $s'\ \mathsf{state}$ 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 $s \mapsto s'$, otherwise it is *non-deterministic*.

A *transition sequence* is a sequence of states $s_0, \ldots, s_n$ such that s_0 initial, and $s_i \mapsto s_{i+1}$ for every $0 \leq i < n$. A transition sequence is *maximal* iff there is no s such that $s_n \mapsto s$, 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 judgement $s \downarrow$ means that there is a complete transition sequence starting from s, which is to say that there exists s' final such that $s \mapsto^* s'$.

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

$$\frac{}{s \mapsto^* s} \tag{8.1a}$$

$$\frac{s \mapsto s' \quad s' \mapsto^* s''}{s \mapsto^* s''} \tag{8.1b}$$

It is easy to show that iterated transition is transitive: if $s \mapsto^* s'$ and $s' \mapsto^* s''$, then $s \mapsto^* s''$.

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

1. $P(s, s)$.
2. if $s \mapsto s'$ 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 *converse evaluation*. Using this principle, it is easy to prove that $\mapsto^*$ is reflexive and transitive.

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

$$\frac{}{s \mapsto^0 s} \tag{8.2a}$$

$$\frac{s \mapsto s' \quad s' \mapsto^n s''}{s \mapsto^{n+1} s''} \tag{8.2b}$$

Theorem 8.1. *For all states s and s', $s \mapsto^* s'$ iff $s \mapsto^k s'$ for some $k \geq 0$.*

8.2 Structural Dynamics

A structural dynamics for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ consists of a transition system whose states are closed expressions. All states are initial, but the final states are the *(closed) values*, which are inductively defined by the following rules:

$$\frac{}{\texttt{num[}n\texttt{]}\ \mathsf{val}} \tag{8.3a}$$

$$\frac{}{\texttt{str[}s\texttt{]}\ \mathsf{val}} \tag{8.3b}$$

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

$$\frac{n_1 + n_2 = n\ \mathsf{nat}}{\texttt{plus(num[}n_1\texttt{];num[}n_2\texttt{])} \mapsto \texttt{num[}n\texttt{]}} \tag{8.4a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{plus(}e_1'\texttt{;}e_2\texttt{)}} \tag{8.4b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{plus(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{plus(}e_1\texttt{;}e_2'\texttt{)}} \tag{8.4c}$$

$$\frac{s_1\,\hat{}\,s_2 = s\ \mathsf{str}}{\texttt{cat(str[}s_1\texttt{];str[}s_2\texttt{])} \mapsto \texttt{str[}s\texttt{]}} \tag{8.4d}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{cat(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{cat(}e_1'\texttt{;}e_2\texttt{)}} \tag{8.4e}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{cat(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{cat(}e_1\texttt{;}e_2'\texttt{)}} \tag{8.4f}$$

$$\frac{}{\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \mapsto [e_1/x]e_2} \tag{8.4g}$$

We have omitted rules for multiplication and computing the length of a string, which follow a similar pattern. Rules (8.4a), (8.4d), and (8.4g) are *instruction transitions*, since they correspond to the primitive steps of evaluation. The remaining rules are *search transitions* that determine the order in which instructions are executed.

Rules (8.4) exhibit structure arising from the principle of introduction and elimination discussed in Chapter 7. The instruction transitions express the *inversion principle*, which states that *eliminatory forms are inverse to introductory forms*. For example, Rule (8.4a) extracts the natural number from the introductory forms of its arguments, adds these two numbers, and yields the corresponding numeral as result. The search transitions specify that the principal arguments of each eliminatory form are to be evaluated. (When non-principal arguments are present, which is not the case here, there is discretion about whether to evaluate them or not.) This is essential, because it prepares for the instruction transitions, which expect their principal arguments to be introductory forms.

Rule (8.4g) specifies a *by-name* interpretation, in which the bound variable stands for the expression e_1 itself.[1] If x does not occur in e_2, the expression e_1 is never evaluated. If, on the other hand, it occurs more than once, then e_1 will be re-evaluated at each occurence. To avoid repeated work in the latter case, we may instead specify a *by-value* interpretation of binding by the following rules:

$$\frac{e_1\ \mathsf{val}}{\texttt{let}(e_1;x.e_2) \mapsto [e_1/x]e_2} \tag{8.5a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{let}(e_1;x.e_2) \mapsto \texttt{let}(e_1';x.e_2)} \tag{8.5b}$$

Rule (8.5b) is an additional search rule specifying that we may evaluate e_1 before e_2. Rule (8.5a) ensures that e_2 is not evaluated until evaluation of e_1 is complete.

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.

$$\begin{array}{ll}
\multicolumn{2}{l}{\texttt{let}(\texttt{plus}(\texttt{num}[1];\texttt{num}[2]);x.\texttt{plus}(\texttt{plus}(x;\texttt{num}[3]);\texttt{num}[4]))} \\
\mapsto & \texttt{let}(\texttt{num}[3];x.\texttt{plus}(\texttt{plus}(x;\texttt{num}[3]);\texttt{num}[4])) \\
\mapsto & \texttt{plus}(\texttt{plus}(\texttt{num}[3];\texttt{num}[3]);\texttt{num}[4]) \\
\mapsto & \texttt{plus}(\texttt{num}[6];\texttt{num}[4]) \\
\mapsto & \texttt{num}[10]
\end{array}$$

Each step in this sequence of transitions is justified by a derivation according to Rules (8.4). For example, the third transition in the preceding example is justified by the following derivation:

$$\frac{\dfrac{}{\texttt{plus}(\texttt{num}[3];\texttt{num}[3]) \mapsto \texttt{num}[6]}\ (8.4a)}{\texttt{plus}(\texttt{plus}(\texttt{num}[3];\texttt{num}[3]);\texttt{num}[4]) \mapsto \texttt{plus}(\texttt{num}[6];\texttt{num}[4])}\ (8.4b)$$

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

The principle of rule induction for the structural dynamics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ states that to show $\mathcal{P}(e \mapsto e')$ whenever $e \mapsto e'$, it is sufficient to show that $\mathcal{P}$ is closed under Rules (8.4). For example, we may show by rule induction that structural dynamics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is *determinate*.

[1] The justification for the terminology "by name" is obscure, but as it is very well-established we will stick with it.

Lemma 8.2 (Determinacy). *If $e \mapsto e'$ and $e \mapsto e''$, then e' and e'' are α-equivalent.*

Proof. By rule induction on the premises $e \mapsto e'$ and $e \mapsto e''$, carried out either simultaneously or in either order. Since only one rule applies to each form of expression, e, the result follows directly in each case. □

8.3 Contextual Dynamics

A variant of structural dynamics, called *contextual dynamics*, is sometimes useful. There is no fundamental difference between the two approaches, only a difference in the style of presentation. The main idea is to isolate instruction steps as a special form of judgement, called *instruction transition*, and to formalize the process of locating the next instruction using a device called an *evaluation context*. The judgement, e val, defining whether an expression is a value, remains unchanged.

The instruction transition judgement, $e_1 \leadsto e_2$, for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is defined by the following rules, together with similar rules for multiplication of numbers and the length of a string.

$$\frac{m + n = p \ \mathsf{nat}}{\texttt{plus(num[}m\texttt{];num[}n\texttt{])} \leadsto \texttt{num[}p\texttt{]}} \tag{8.6a}$$

$$\frac{s\,\hat{}\,t = u \ \mathsf{str}}{\texttt{cat(str[}s\texttt{];str[}t\texttt{])} \leadsto \texttt{str[}u\texttt{]}} \tag{8.6b}$$

$$\frac{}{\texttt{let(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \leadsto [e_1/x]e_2} \tag{8.6c}$$

The judgement $\mathcal{E}$ 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 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.)

$$\frac{}{\circ \ \mathsf{ectxt}} \tag{8.7a}$$

$$\frac{\mathcal{E}_1 \ \mathsf{ectxt}}{\texttt{plus(}\mathcal{E}_1\texttt{;}e_2\texttt{)} \ \mathsf{ectxt}} \tag{8.7b}$$

$$\frac{e_1 \ \mathsf{val} \quad \mathcal{E}_2 \ \mathsf{ectxt}}{\texttt{plus(}e_1\texttt{;}\mathcal{E}_2\texttt{)} \ \mathsf{ectxt}} \tag{8.7c}$$

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 (8.7c) states that in an expression $\texttt{plus}(e_1;e_2)$, if the first principal argument, e_1, is a value, then the next instruction step, if any, lies at or within the second principal 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 judgement $e' = \mathcal{E}\{e\}$ 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:

$$\frac{}{e = \circ\{e\}} \tag{8.8a}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\texttt{plus}(e_1;e_2) = \texttt{plus}(\mathcal{E}_1;e_2)\{e\}} \tag{8.8b}$$

$$\frac{e_1\ \textsf{val} \quad e_2 = \mathcal{E}_2\{e\}}{\texttt{plus}(e_1;e_2) = \texttt{plus}(e_1;\mathcal{E}_2)\{e\}} \tag{8.8c}$$

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 $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is defined by a single rule:

$$\frac{e = \mathcal{E}\{e_0\} \quad e_0 \rightsquigarrow e_0' \quad e' = \mathcal{E}\{e_0'\}}{e \mapsto e'} \tag{8.9}$$

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 $e \mapsto_{\mathsf{s}} e'$ for the transition relation defined by the structural dynamics (Rules (8.4)), and $e \mapsto_{\mathsf{c}} e'$ for the transition relation defined by the contextual dynamics (Rules (8.9)).

Theorem 8.3. *$e \mapsto_{\mathsf{s}} e'$ if, and only if, $e \mapsto_{\mathsf{c}} e'$.*

Proof. From left to right, proceed by rule induction on Rules (8.4). 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 $e_0 \rightsquigarrow e_0'$. For example, for Rule (8.4a), take $\mathcal{E} = \circ$, and observe that $e \rightsquigarrow e'$. For Rule (8.4b), 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 $e_0 \rightsquigarrow e_0'$. Take $\mathcal{E} = \texttt{plus}(\mathcal{E}_1;e_2)$, and observe that $e = \texttt{plus}(\mathcal{E}_1;e_2)\{e_0\}$ and $e' = \texttt{plus}(\mathcal{E}_1;e_2)\{e_0'\}$ with $e_0 \rightsquigarrow e_0'$.

From right to left, observe that if $e \mapsto_{\mathsf{c}} e'$, then there exists an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \leadsto e_0'$. We prove by induction on Rules (8.8) that $e \mapsto_{\mathsf{s}} e'$. For example, for Rule (8.8a), e_0 is e, e_0' is e', and $e \leadsto e'$. Hence $e \mapsto_{\mathsf{s}} e'$. For Rule (8.8b), 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 $e_1 \mapsto_{\mathsf{s}} e_1'$. Therefore e is $\mathtt{plus}(e_1;e_2)$, e' is $\mathtt{plus}(e_1';e_2)$, and therefore by Rule (8.4b), $e \mapsto_{\mathsf{s}} e'$.

□

Since the two transition judgements 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 (8.9) in the simpler form

$$\frac{e_0 \leadsto e_0'}{\mathcal{E}\{e_0\} \mapsto \mathcal{E}\{e_0'\}} \, . \tag{8.10}$$

This formulation is simpler insofar as it leaves implicit the definition of the decomposition of the left- and right-hand sides. The deeper advantage, which we will exploit in Chapter 13, is that the transition judgement in contextual dynamics applies only to closed expressions of a *fixed* type, whereas structural dynamics transitions are necessarily defined over expressions of *every* type.

8.4 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 $x^2 + 2\,x + 1$ 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 $x^2 + 2\,x + 1$ and calculate that $2^2 + 2\,2 + 1 = 9$, 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 $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$, which we write as $\mathcal{X} \mid \Gamma \vdash e \equiv e' : \tau$, where Γ 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 judgement $\Gamma \vdash e \equiv e' : \tau$, 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 Γ.

Definitional equivalence of expressons in $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is inductively defined by the following rules:

$$\frac{}{\Gamma \vdash e \equiv e : \tau} \tag{8.11a}$$

$$\frac{\Gamma \vdash e' \equiv e : \tau}{\Gamma \vdash e \equiv e' : \tau} \tag{8.11b}$$

$$\frac{\Gamma \vdash e \equiv e' : \tau \quad \Gamma \vdash e' \equiv e'' : \tau}{\Gamma \vdash e \equiv e'' : \tau} \tag{8.11c}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \texttt{num} \quad \Gamma \vdash e_2 \equiv e_2' : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1;e_2) \equiv \texttt{plus}(e_1';e_2') : \texttt{num}} \tag{8.11d}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \texttt{str} \quad \Gamma \vdash e_2 \equiv e_2' : \texttt{str}}{\Gamma \vdash \texttt{cat}(e_1;e_2) \equiv \texttt{cat}(e_1';e_2') : \texttt{str}} \tag{8.11e}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \texttt{let}(e_1;x.e_2) \equiv \texttt{let}(e_1';x.e_2') : \tau_2} \tag{8.11f}$$

$$\frac{n_1 + n_2 = n \ \mathsf{nat}}{\Gamma \vdash \texttt{plus}(\texttt{num}[n_1];\texttt{num}[n_2]) \equiv \texttt{num}[n] : \texttt{num}} \tag{8.11g}$$

$$\frac{s_1 \,\hat{}\, s_2 = s \ \mathsf{str}}{\Gamma \vdash \texttt{cat}(\texttt{str}[s_1];\texttt{str}[s_2]) \equiv \texttt{str}[s] : \texttt{str}} \tag{8.11h}$$

$$\frac{}{\Gamma \vdash \texttt{let}(e_1;x.e_2) \equiv [e_1/x]e_2 : \tau} \tag{8.11i}$$

Rules (8.11a) through (8.11c) state that definitional equivalence is an *equivalence relation*. Rules (8.11d) through (8.11f) state that it is a *congruence relation*, which means that it is compatible with all expression-forming constructs in the language. Rules (8.11g) through (8.11i) specify the meanings of the primitive constructs of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$. For the sake of concision, Rules (8.11) may be characterized as defining the *strongest congruence* closed under Rules (8.11g), (8.11h), and (8.11i).

Rules (8.11) 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

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

by applying Rules (8.11). 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 equivalence is rather weak in that many equivalences that one might intuitively think are true are not derivable from Rules (8.11). A prototypical example is the putative equivalence

$$x : \mathtt{num}, y : \mathtt{num} \vdash x_1 + x_2 \equiv x_2 + x_1 : \mathtt{num}, \tag{8.12}$$

which, intuitively, expresses the commutativity of addition. Although we shall not prove this here, this equivalence is *not* derivable from Rules (8.11). And yet we *may* derive all of its closed instances,

$$n_1 + n_2 \equiv n_2 + n_1 : \mathtt{num}, \tag{8.13}$$

where n_1 nat and n_2 nat are particular numbers.

The "gap" between a general law, such as Equation (8.12), and all of its instances, given by Equation (8.13), 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*, or *observational*, *equivalence*, since it expresses relationships that hold by virtue of the dynamics of the expressions involved.[2] Semantic equivalence is a *synthetic judgement*, one that requires proof. It is to be distinguished from definitional equivalence, which expresses an *analytic judgement*, one that is self-evident based solely on the dynamics of the operations involved. As such definitional equivalence may be thought of as *symbolic evaluation*, which permits simplification according to the evaluation rules of a language, but which does not permit reasoning by induction.

Definitional equivalence is adequate for evaluation in that it permits the calculation of the value of any closed expression.

Theorem 8.4. *$e \equiv e' : \tau$ iff there exists e_0 val such that $e \mapsto^* e_0$ and $e' \mapsto^* e_0$.*

Proof. The proof from right to left is direct, since every transition step is a valid equation. The converse follows from the following, more general, proposition. If $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e \equiv e' : \tau$, then whenever $e_1 : \tau_1, \ldots, e_n : \tau_n$, if

$$[e_1, \ldots, e_n / x_1, \ldots, x_n]e \equiv [e_1, \ldots, e_n / x_1, \ldots, x_n]e' : \tau,$$

then there exists e_0 val such that

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

[2] This concept of equivalence is developed rigorously in Chapter 51.

and

$$[e_1, \ldots, e_n / x_1, \ldots, x_n]e' \mapsto^* e_0.$$

This is proved by rule induction on Rules (8.11). □

The formulation of definitional equivalence for the by-value dynamics of binding requires a bit of additional machinery. The key idea is motivated by the modifications required to Rule (8.11i) to express the requirement that e_1 be a value. As a first cut one might consider simply adding an additional premise to the rule:

$$\frac{e_1 \ \mathsf{val}}{\Gamma \vdash \texttt{let}(e_1; x.e_2) \equiv [e_1/x]e_2 : \tau} \tag{8.14}$$

This is almost correct, except that the judgement e val is defined only for *closed* expressions, whereas e_1 might well involve free variables in Γ. What is required is to extend the judgement e val to the hypothetical judgement

$$x_1 \ \mathsf{val}, \ldots, x_n \ \mathsf{val} \vdash e \ \mathsf{val}$$

in which the hypotheses express the assumption that variables are only ever bound to values, and hence can be regarded as values. To maintain this invariant, we must maintain a set, Ξ, of such hypotheses as part of definitional equivalence, writing $\Xi\,\Gamma \vdash e \equiv e' : \tau$, and modifying Rule (8.11f) as follows:

$$\frac{\Xi\,\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Xi, x \ \mathsf{val}\,\Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Xi\,\Gamma \vdash \texttt{let}(e_1; x.e_2) \equiv \texttt{let}(e_1'; x.e_2') : \tau_2} \tag{8.15}$$

The other rules are correspondingly modified to simply carry along Ξ is an additional set of hypotheses of the inference.

8.5 Exercises

1. For the structural dynamics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, prove that if $e \mapsto e_1$ and $e \mapsto e_2$, then $e_1 =_\alpha e_2$.

2. Formulate a variation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ with both a by-name and a by-value `let` construct.

Chapter 9

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 $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ 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 $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ may be stated as follows:

Theorem 9.1 (Type Safety)**.** *1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

2. If $e : \tau$, then either e val, or there exists e' such that $e \mapsto e'$.

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 $e \mapsto e'$. 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.

9.1 Preservation

The preservation theorem for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ defined in Chapters 7 and 8 is proved by rule induction on the transition system (rules (8.4)).

Theorem 9.2 (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. We will consider two cases, leaving the rest to the reader. Consider rule (8.4b),

$$\frac{e_1 \mapsto e_1'}{\texttt{plus}(e_1;e_2) \mapsto \texttt{plus}(e_1';e_2)} .$$

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

Now consider rule (8.4g),

$$\frac{e_1\ \mathsf{val}}{\texttt{let}(e_1;x.e_2) \mapsto [e_1/x]e_2} .$$

Assume that $\texttt{let}(e_1;x.e_2) : \tau_2$. By the inversion lemma 7.2 on page 64, $e_1 : \tau_1$ for some τ_1 such that $x : \tau_1 \vdash e_2 : \tau_2$. By the substitution lemma 7.4 on page 64 $[e_1/x]e_2 : \tau_2$, as desired. □

The proof of preservation is naturally structured as an induction on the transition judgement, since the argument hinges on examining all possible transitions from a given expression. In some cases one 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.

9.2 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 9.3 (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 (7.1) and (8.3). □

Progress is proved by rule induction on rules (7.1) defining the statics of the language.

Theorem 9.4 (Progress)**.** *If* $e : \tau$, *then either* e val, *or there exists* e' *such that* $e \mapsto e'$.

Proof. The proof proceeds by induction on the typing derivation. We will consider only one case, for rule (7.1d),

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

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 $e_1 \mapsto e_1'$. In the latter case it follows that $\mathtt{plus}(e_1;e_2) \mapsto \mathtt{plus}(e_1';e_2)$, as required. In the former we also have by induction that either e_2 val, or there exists e_2' such that $e_2 \mapsto e_2'$. In the latter case we have that $\mathtt{plus}(e_1;e_2) \mapsto \mathtt{plus}(e_1;e_2')$, as required. In the former, we have, by the Canonical Forms Lemma 9.3 on the preceding page, $e_1 = \mathtt{num}[n_1]$ and $e_2 = \mathtt{num}[n_2]$, and hence

$$\mathtt{plus}(\mathtt{num}[n_1];\mathtt{num}[n_2]) \mapsto \mathtt{num}[n_1 + n_2].$$

□

Since 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.

9.3 Run-Time Errors

Suppose that we wish to extend $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ 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 div(num[3];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 one 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 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 judgement:

$$\frac{e_1\ \mathsf{val}}{\texttt{div}(e_1;\texttt{num[0]})\ \mathsf{err}} \tag{9.1a}$$

$$\frac{e_1\ \mathsf{err}}{\mathtt{plus}(e_1;e_2)\ \mathsf{err}} \tag{9.1b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{err}}{\mathtt{plus}(e_1;e_2)\ \mathsf{err}} \tag{9.1c}$$

Rule (9.1a) 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 judgement is available, we may also consider an expression, `error`, which forcibly induces an error, with the following static and dynamic semantics:

$$\frac{}{\Gamma \vdash \mathtt{error} : \tau} \tag{9.2a}$$

$$\frac{}{\mathtt{error}\ \mathsf{err}} \tag{9.2b}$$

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

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. □

9.4 Exercises

1. Complete the proof of preservation.

2. Complete the proof of progress.

Chapter 10

Evaluation Dynamics

In Chapter 8 we defined the evaluation of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ expression 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.

10.1 Evaluation Dynamics

Another method for defining the dynamics of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$, called *evaluation dynamics*, consists of an inductive definition of the evaluation judgement, $e \Downarrow v$, stating that the closed expression, e, evaluates to the value, v.

$$\frac{}{\mathtt{num}[n] \Downarrow \mathtt{num}[n]} \tag{10.1a}$$

$$\frac{}{\mathtt{str}[s] \Downarrow \mathtt{str}[s]} \tag{10.1b}$$

$$\frac{e_1 \Downarrow \mathtt{num}[n_1] \quad e_2 \Downarrow \mathtt{num}[n_2] \quad n_1 + n_2 = n \ \mathsf{nat}}{\mathtt{plus}(e_1;e_2) \Downarrow \mathtt{num}[n]} \tag{10.1c}$$

$$\frac{e_1 \Downarrow \mathtt{str}[s_1] \quad e_2 \Downarrow \mathtt{str}[s_2] \quad s_1\,\hat{}\,s_2 = s \ \mathsf{str}}{\mathtt{cat}(e_1;e_2) \Downarrow \mathtt{str}[s]} \tag{10.1d}$$

$$\frac{e \Downarrow \mathtt{str[}s\mathtt{]} \quad |s| = n \ \mathsf{str}}{\mathtt{len(}e\mathtt{)} \Downarrow \mathtt{num[}n\mathtt{]}} \tag{10.1e}$$

$$\frac{[e_1/x]e_2 \Downarrow v_2}{\mathtt{let(}e_1\mathtt{;}x\mathtt{.}e_2\mathtt{)} \Downarrow v_2} \tag{10.1f}$$

The value of a `let` expression is determined by substitution of the binding into the body. The rules are therefore not syntax-directed, since the premise of Rule (10.1f) is not a sub-expression of the expression in the conclusion of that rule.

The evaluation judgement 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 (10.1):

1. Show that $\mathcal{P}(\mathtt{num[}n\mathtt{]} \Downarrow \mathtt{num[}n\mathtt{]})$.

2. Show that $\mathcal{P}(\mathtt{str[}s\mathtt{]} \Downarrow \mathtt{str[}s\mathtt{]})$.

3. Show that $\mathcal{P}(\mathtt{plus(}e_1\mathtt{;}e_2\mathtt{)} \Downarrow \mathtt{num[}n\mathtt{]})$, if $\mathcal{P}(e_1 \Downarrow \mathtt{num[}n_1\mathtt{]})$, $\mathcal{P}(e_2 \Downarrow \mathtt{num[}n_2\mathtt{]})$, and $n_1 + n_2 = n \ \mathsf{nat}$.

4. Show that $\mathcal{P}(\mathtt{cat(}e_1\mathtt{;}e_2\mathtt{)} \Downarrow \mathtt{str[}s\mathtt{]})$, if $\mathcal{P}(e_1 \Downarrow \mathtt{str[}s_1\mathtt{]})$, $\mathcal{P}(e_2 \Downarrow \mathtt{str[}s_2\mathtt{]})$, and $s_1 \mathbin{\hat{}} s_2 = s \ \mathsf{str}$.

5. Show that $\mathcal{P}(\mathtt{let(}e_1\mathtt{;}x\mathtt{.}e_2\mathtt{)} \Downarrow v_2)$, if $\mathcal{P}([e_1/x]e_2 \Downarrow v_2)$.

This induction principle is *not* the same as structural induction on e `exp`, because the evaluation rules are not syntax-directed!

Lemma 10.1. *If $e \Downarrow v$, then v val.*

Proof. By induction on Rules (10.1). All cases except Rule (10.1f) are immediate. For the latter case, the result follows directly by an appeal to the inductive hypothesis for the second premise of the evaluation rule. □

10.2 Relating Structural and Evaluation Dynamics

We have given two different forms of dynamics for $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$. 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, focussing 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 judgement in the evaluation dynamics. (We will consider only numeric expressions, but analogous results hold also for string-valued expressions.)

Theorem 10.2. *For all closed expressions e and values v, $e \mapsto^* v$ iff $e \Downarrow v$.*

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

Lemma 10.3. *If $e \Downarrow v$, then $e \mapsto^* v$.*

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

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

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

For the converse, recall from Chapter 8 the definitions of multi-step evaluation and complete evaluation. Since $v \Downarrow v$ whenever v val, it suffices to show that evaluation is closed under reverse execution.

Lemma 10.4. *If $e \mapsto e'$ and $e' \Downarrow v$, then $e \Downarrow v$.*

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

10.3 Type Safety, Revisited

The type safety theorem for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ (Theorem 9.1 on page 77) states that a language is safe iff it satisfies both preservation and progress. This formulation depends critically on the use of a transition system to specify the dynamics. But what if we had instead specified the dynamics as an evaluation relation, instead of using a transition system? Can we state and prove safety in such a setting?

The answer, unfortunately, is that we cannot. While 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? One might be tempted to phrase progress as saying that if $e : \tau$, then $e \Downarrow v$ for some v. While this property is true for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, it demands much more than just progress — it requires that every expression evaluate to a value! If $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ were extended to admit operations that may result in an error (as discussed in Section 9.3 on page 80), 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 run-time type errors, and to show that any expression with a type fault must be ill-typed. Re-stated in the contrapositive, this means that a well-typed program cannot incur a type error. A difficulty with this point of view is that one 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 judgement $e\Uparrow$ 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:

$$\frac{}{\texttt{plus}(\texttt{str}[s];e_2)\Uparrow} \tag{10.2a}$$

$$\frac{e_1\ \mathsf{val}}{\texttt{plus}(e_1;\texttt{str}[s])\Uparrow} \tag{10.2b}$$

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

Theorem 10.5. *If $e\Uparrow$, then there is no τ such that $e : \tau$.*

Proof. By rule induction on Rules (10.2). For example, for Rule (10.2a), we observe that $\texttt{str}[s] : \texttt{str}$, and hence $\texttt{plus}(\texttt{str}[s];e_2)$ is ill-typed. □

Corollary 10.6. *If $e : \tau$, then $\neg(e\Uparrow)$.*

Apart from the inconvenience of having to define the judgement $e\Uparrow$ 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 judgement $e\Uparrow$, the proof of Theorem 10.5 on the facing page 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, an elegant win-win situation for both the dynamics and the implementation.

10.4 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. Since 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*.

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

$$\frac{}{\texttt{num}[n] \Downarrow^0 \texttt{num}[n]} \tag{10.3a}$$

$$\frac{e_1 \Downarrow^{k_1} \texttt{num}[n_1] \quad e_2 \Downarrow^{k_2} \texttt{num}[n_2]}{\texttt{plus}(e_1;e_2) \Downarrow^{k_1+k_2+1} \texttt{num}[n_1+n_2]} \tag{10.3b}$$

$$\frac{}{\texttt{str}[s] \Downarrow^0 \texttt{str}[s]} \tag{10.3c}$$

$$\frac{e_1 \Downarrow^{k_1} s_1 \quad e_2 \Downarrow^{k_2} s_2}{\texttt{cat}(e_1;e_2) \Downarrow^{k_1+k_2+1} \texttt{str}[s_1\,\hat{}\,s_2]} \tag{10.3d}$$

$$\frac{[e_1/x]e_2 \Downarrow^{k_2} v_2}{\texttt{let}(e_1; x.e_2) \Downarrow^{k_2+1} v_2} \tag{10.3e}$$

Theorem 10.7. *For any closed expression e and closed value v of the same type, $e \Downarrow^k v$ iff $e \mapsto^k v$.*

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. □

10.5 Exercises

1. Prove that if $e \Downarrow v$, then v val.

2. Prove that if $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 = v_2$.

3. Complete the proof of equivalence of evaluation and structural dynamics.

4. Prove preservation for the instrumented evaluation dynamics, and conclude that well-typed programs cannot go wrong.

5. Is it possible to use environments in a structural dynamics? What difficulties do you encounter?

Part IV

Function Types

Chapter 11

Function Definitions and Values

In the language $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ 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 $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ 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`, since 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. Since 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.

11.1 First-Order Functions

The language $\mathcal{L}\{\texttt{num}\,\texttt{str}\,\texttt{fun}\}$ is the extension of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ with function definitions and function applications as described by the following grammar:

$$\begin{array}{llllll}
\mathsf{Expr} & e & ::= & \texttt{call[}f\texttt{](}e\texttt{)} & f\texttt{(}e\texttt{)} & \text{call} \\
 & & & \texttt{fun[}\tau_1\texttt{;}\tau_2\texttt{](}x_1\texttt{.}e_2\texttt{;}f\texttt{.}e\texttt{)} & \texttt{fun}\,f\texttt{(}x_1\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{=}\,e_2\,\texttt{in}\,e & \text{definition}
\end{array}$$

The expression $\texttt{fun[}\tau_1\texttt{;}\tau_2\texttt{](}x_1\texttt{.}e_2\texttt{;}f\texttt{.}e\texttt{)}$ binds the function name f within e to the pattern $x_1\texttt{.}e_2$, which has parameter x_1 and definition e_2. The domain and range of the function are, respectively, the types τ_1 and τ_2. The expression $\texttt{call[}f\texttt{](}e\texttt{)}$ instantiates the binding of f with the argument e.

The statics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\,\texttt{fun}\}$ defines two forms of judgement:

1. Expression typing, $e : \tau$, stating that e has type τ;

2. Function typing, $f\texttt{(}\tau_1\texttt{)} : \tau_2$, stating that f is a function with argument type τ_1 and result type τ_2.

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

The statics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\,\texttt{fun}\}$ is defined by the following rules:

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2 \quad \Gamma, f\texttt{(}\tau_1\texttt{)} : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{fun[}\tau_1\texttt{;}\tau_2\texttt{](}x_1\texttt{.}e_2\texttt{;}f\texttt{.}e\texttt{)} : \tau} \tag{11.1a}$$

$$\frac{\Gamma \vdash f\texttt{(}\tau_1\texttt{)} : \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{call[}f\texttt{](}e\texttt{)} : \tau_2} \tag{11.1b}$$

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 a call of the form $\texttt{call[}f\texttt{](}e\texttt{)}$. Function substitution for such expressions is defined by the following rule:

$$\frac{}{[\![x\,.\,e/f]\!]\texttt{call[}f\texttt{](}e'\texttt{)} = \texttt{let(}e'\texttt{;}x\,.\,e\texttt{)}} \tag{11.2}$$

At call sites to f with argument e', function substitution yields a `let` expression that binds x to e' within e.

Lemma 11.1. *If* $\Gamma, f\,(\tau_1) : \tau_2 \vdash e : \tau$ *and* $\Gamma, x_1 : \tau_2 \vdash e_2 : \tau_2$, *then* $\Gamma \vdash [\![x_1\,.\,e_2/f]\!]e : \tau$.

Proof. By induction on the structure of e'. □

The dynamics of $\mathcal{L}\{\texttt{num str fun}\}$ is defined using function substitution:

$$\frac{}{\texttt{fun[}\tau_1\texttt{;}\tau_2\texttt{](}x_1\,.\,e_2\texttt{;}f\,.\,e\texttt{)} \mapsto [\![x_1\,.\,e_2/f]\!]e} \tag{11.3}$$

Since function substitution replaces all calls to f by appropriate `let` expressions, there is no need to give a rule for function calls.

The safety of $\mathcal{L}\{\texttt{num str fun}\}$ may be obtained as an immediate corollary of the safety theorem for higher-order functions, which we discuss next.

11.2 Higher-Order Functions

The syntactic and semantic similarity between variable definitions and function definitions in $\mathcal{L}\{\texttt{num str fun}\}$ is striking. This suggests that it may be possible to consolidate the two concepts into a single definition mechanism. 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 consolidate function definitions with expression definitions it is sufficient to *reify* the abstractor into a form of expression, called a *λ-abstraction*, written $\texttt{lam[}\tau_1\texttt{](}x\,.\,e\texttt{)}$. Correspondingly, we must generalize application to have the form $\texttt{ap(}e_1\texttt{;}e_2\texttt{)}$, where e_1 is any expression, and not just a function name. These are, respectively, the introduction and elimination forms for the*function type*, $\texttt{arr(}\tau_1\texttt{;}\tau_2\texttt{)}$, whose elements are functions with domain τ_1 and range τ_2.

The language $\mathcal{L}\{\texttt{num}\,\texttt{str} \rightarrow\}$ is the enrichment of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ with function types, as specified by the following grammar:

$$
\begin{array}{llllll}
\textsf{Type} & \tau & ::= & \texttt{arr}(\tau_1;\tau_2) & \tau_1 \rightarrow \tau_2 & \text{function} \\
\textsf{Expr} & e & ::= & \texttt{lam}[\tau](x.e) & \lambda\,(x{:}\tau.\,e) & \text{abstraction} \\
 & & & \texttt{ap}(e_1;e_2) & e_1\,(e_2) & \text{application}
\end{array}
$$

Functions are now "first class" in the sense that a function is an expression of function type.

The statics of $\mathcal{L}\{\texttt{num}\,\texttt{str} \rightarrow\}$ is given by extending Rules (7.1) with the following rules:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam}[\tau_1](x.e) : \texttt{arr}(\tau_1;\tau_2)} \tag{11.4a}$$

$$\frac{\Gamma \vdash e_1 : \texttt{arr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{ap}(e_1;e_2) : \tau} \tag{11.4b}$$

Lemma 11.2 (Inversion)**.** *Suppose that $\Gamma \vdash e : \tau$.*

1. *If $e = \texttt{lam}[\tau_1](x.e)$, then $\tau = \texttt{arr}(\tau_1;\tau_2)$ and $\Gamma, x : \tau_1 \vdash e : \tau_2$.*

2. *If $e = \texttt{ap}(e_1;e_2)$, then there exists τ_2 such that $\Gamma \vdash e_1 : \texttt{arr}(\tau_2;\tau)$ 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 11.3 (Substitution)**.** *If $\Gamma, x : \tau \vdash e' : \tau'$, and $\Gamma \vdash e : \tau$, then $\Gamma \vdash [e/x]e' : \tau'$.*

Proof. By rule induction on the derivation of the first judgement. □

The dynamics of $\mathcal{L}\{\texttt{num}\,\texttt{str} \rightarrow\}$ extends that of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ with the following additional rules:

$$\frac{}{\texttt{lam}[\tau](x.e)\ \textsf{val}} \tag{11.5a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap}(e_1;e_2) \mapsto \texttt{ap}(e_1';e_2)} \tag{11.5b}$$

$$\frac{}{\texttt{ap}(\texttt{lam}[\tau_2](x.e_1);e_2) \mapsto [e_2/x]e_1} \tag{11.5c}$$

These rules specify a call-by-name discipline for function application. It is a good exercise to formulate a call-by-value discipline as well.

Theorem 11.4 (Preservation). *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. The proof is by induction on rules (11.5), which define the dynamics of the language.

Consider rule (11.5c),

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

Suppose that $\texttt{ap(lam[}\tau_2\texttt{](}x\texttt{.}e_1\texttt{);}e_2\texttt{)} : \tau_1$. By Lemma 11.2 on the preceding page $e_2 : \tau_2$ and $x : \tau_2 \vdash e_1 : \tau_1$, so by Lemma 11.3 on the facing page $[e_2/x]e_1 : \tau_1$.

The other rules governing application are handled similarly. □

Lemma 11.5 (Canonical Forms). *If e val and $e : \texttt{arr(}\tau_1\texttt{;}\tau_2\texttt{)}$, then $e = \texttt{lam[}\tau_1\texttt{](}x\texttt{.}e_2\texttt{)}$ for some x and 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 11.6 (Progress). *If $e : \tau$, then either e is a value, or there exists e' such that $e \mapsto e'$.*

Proof. The proof is by induction on rules (11.4). Note that since we consider only closed terms, there are no hypotheses on typing derivations.

Consider rule (11.4b). By induction either e_1 val or $e_1 \mapsto e_1'$. In the latter case we have $\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{ap(}e_1'\texttt{;}e_2\texttt{)}$. In the former case, we have by Lemma 11.5 that $e_1 = \texttt{lam[}\tau_2\texttt{](}x\texttt{.}e\texttt{)}$ for some x and e. But then $\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto [e_2/x]e$. □

11.3 Evaluation Dynamics and Definitional Equivalence

An inductive definition of the evaluation judgement $e \Downarrow v$ for $\mathcal{L}\{\texttt{num}\ \texttt{str} \rightarrow\}$ is given by the following rules:

$$\frac{}{\texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)} \Downarrow \texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)}} \qquad (11.6a)$$

$$\frac{e_1 \Downarrow \texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)} \quad [e_2/x]e \Downarrow v}{\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \Downarrow v} \qquad (11.6b)$$

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

Theorem 11.7. *$e \Downarrow v$ iff $e \mapsto^* v$ and v val.*

Proof. In the forward direction we proceed by rule induction on Rules (11.6). The proof makes use of a *pasting lemma* stating that, for example, if $e_1 \mapsto^* e_1'$, then $\texttt{ap}(e_1;e_2) \mapsto^* \texttt{ap}(e_1';e_2)$, and similarly for the other constructs of the language.

In the reverse direction we proceed by rule induction on Rules (8.1). The proof relies on a *converse evaluation lemma*, which states that if $e \mapsto e'$ and $e' \Downarrow v$, then $e \Downarrow v$. This is proved by rule induction on Rules (11.5). □

Definitional equivalence for the call-by-name dynamics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\rightarrow\}$ is defined by a straightforward extension to Rules (8.11).

$$\frac{}{\Gamma \vdash \texttt{ap}(\texttt{lam}[\tau](x.e_2);e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{11.7a}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \texttt{ap}(e_1;e_2) \equiv \texttt{ap}(e_1';e_2') : \tau} \tag{11.7b}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \texttt{lam}[\tau_1](x.e_2) \equiv \texttt{lam}[\tau_1](x.e_2') : \tau_1 \rightarrow \tau_2} \tag{11.7c}$$

Definitional equivalence for call-by-value requires a small bit of additional machinery. The main idea is to restrict Rule (11.7a) 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 Rule (11.7a) 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 equivalence judgement has the form

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

where Ξ is the finite set of hypotheses $x_1\ \mathsf{val}, \ldots, x_k\ \mathsf{val}$ governing the variables in scope at that point. We write $\Xi \vdash e\ \mathsf{val}$ to indicate that e is a value under these hypotheses, so that, for example, $\Xi, x\ \mathsf{val} \vdash x\ \mathsf{val}$.

The rule of definitional equivalence for call-by-value are similar to those for call-by-name, modified to take account of the scopes of value variables. Two illustrative rules are as follows:

$$\frac{\Xi, x\ \mathsf{val}\,\Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Xi\,\Gamma \vdash \texttt{lam}[\tau_1](x.e_2) \equiv \texttt{lam}[\tau_1](x.e_2') : \tau_1 \rightarrow \tau_2} \tag{11.8a}$$

$$\frac{\Xi \vdash e_1\ \mathsf{val}}{\Xi\,\Gamma \vdash \texttt{ap}(\texttt{lam}[\tau](x.e_2);e_1) \equiv [e_1/x]e_2 : \tau}\,. \tag{11.8b}$$

11.4 Dynamic Scope

The dynamics of function application given by Rules (11.5) is defined only for expressions without free variables. When a function is called, the argument is substituted for the function parameter, ensuring that the result remains closed. Moreover, since 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 3 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 call is defined similarly to dynamic 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 $\lambda\,(x:\texttt{str}.\,y + |x|)$ in which the variable y occurs free, and let e' be the expression $\lambda\,(y:\texttt{str}.\,f(y))$ with free variable f. If we *substitute* e for f in e' we obtain an expression of the form

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

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

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

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

The implications of this seemingly small change to the dynamics of $\mathcal{L}\{\rightarrow\}$ are far-reaching. The most obvious implication is that the language is not type safe. In the above example we have that $y : \texttt{nat} \vdash e : \texttt{str} \rightarrow \texttt{nat}$, and that $f : \texttt{str} \rightarrow \texttt{nat} \vdash e' : \texttt{str} \rightarrow \texttt{nat}$. It follows that $y : \texttt{nat} \vdash [e/f]e' : \texttt{str} \rightarrow \texttt{nat}$, 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 $\texttt{str}$, whereas the free occurrence in e must have type $\texttt{nat}$ in order for e to be well-formed.

One way around this difficulty is to ignore types altogether, and rely on run-time checks to ensure that bad things do not happen, despite the evident failure of safety. (See Chapter 21 for a full exploration of this approach.) But even if ignore the safety issues, 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', one 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 one disregards 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 37.)

11.5 Exercises

Chapter 12

Gödel's System T

The language $\mathcal{L}\{\texttt{nat}\rightarrow\}$, better known as *Gödel's System **T***, is the combination of function types with the type of natural numbers. In contrast to $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, which equips the naturals with some arbitrarily chosen arithmetic primitives, the language $\mathcal{L}\{\texttt{nat}\rightarrow\}$ 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 $\mathcal{L}\{\texttt{nat}\rightarrow\}$ "comes equipped" with a proof of its termination. While 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 $\mathcal{L}\{\texttt{nat}\rightarrow\}$. To do so would require a master termination proof for every possible program in the language, something that we shall prove does not exist.

12.1 Statics

The syntax of $\mathcal{L}\{\texttt{nat}\rightarrow\}$ is given by the following grammar:

Type	τ	$::=$	$\texttt{nat}$	$\texttt{nat}$	naturals
			$\texttt{arr}(\tau_1;\tau_2)$	$\tau_1 \rightarrow \tau_2$	function
Expr	e	$::=$	x	x	variable
			$\texttt{z}$	$\texttt{z}$	zero
			$\texttt{s}(e)$	$\texttt{s}(e)$	successor
			$\texttt{natrec}(e;e_0;x.y.e_1)$	$\texttt{natrec}\,e\,\{\texttt{z} \Rightarrow e_0 \mid \texttt{s}(x)\ \texttt{with}\ y \Rightarrow e_1\}$	recursion
			$\texttt{lam}[\tau](x.e)$	$\lambda\,(x{:}\tau.e)$	abstraction
			$\texttt{ap}(e_1;e_2)$	$e_1(e_2)$	application

We write $\overline{n}$ for the expression $\texttt{s}(\ldots\texttt{s}(\texttt{z}))$, in which the successor is applied $n \geq 0$ times to zero. The expression $\texttt{natrec}(e;e_0;x.y.e_1)$ is called *primitive recursion*. 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 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 *iteration*, written $\texttt{natiter}(e;e_0;y.e_1)$, is considered as an alternative to primitive recursion. It has essentially the same meaning as primitive recursion, 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 iteration is a special case of primitive recursion, since we can always ignore the predecessor binding. Conversely, primitive recursion is definable from iteration, provided that we have product types (Chapter 14) at our disposal. To define primitive recursion from iteration we simultaneously compute the predecessor while iterating the specified computation.

The statics of $\mathcal{L}\{\texttt{nat}\rightarrow\}$ is given by the following typing rules:

$$\frac{}{\Gamma, x : \texttt{nat} \vdash x : \texttt{nat}} \tag{12.1a}$$

$$\frac{}{\Gamma \vdash \texttt{z} : \texttt{nat}} \tag{12.1b}$$

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{s}(e) : \texttt{nat}} \tag{12.1c}$$

$$\frac{\Gamma \vdash e : \texttt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \texttt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \texttt{natrec}(e;e_0;x.y.e_1) : \tau} \tag{12.1d}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \texttt{lam}[\sigma](x.e) : \texttt{arr}(\sigma;\tau)} \tag{12.1e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{arr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{ap}(e_1;e_2) : \tau} \tag{12.1f}$$

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

Lemma 12.1. *If* $\Gamma \vdash e : \tau$ *and* $\Gamma, x : \tau \vdash e' : \tau'$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

12.2 Dynamics

The dynamics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ adopts a call-by-name interpretation of function application, and requires that the successor operation evaluate its argument (so that values of type nat are numerals).

The closed values of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ are determined by the following rules:

$$\frac{}{\texttt{z}\ \mathsf{val}} \tag{12.2a}$$

$$\frac{e\ \mathsf{val}}{\texttt{s}(e)\ \mathsf{val}} \tag{12.2b}$$

$$\frac{}{\texttt{lam}[\tau](x.e)\ \mathsf{val}} \tag{12.2c}$$

The dynamics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ is given by the following rules:

$$\frac{e \mapsto e'}{\texttt{s}(e) \mapsto \texttt{s}(e')} \tag{12.3a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap}(e_1;e_2) \mapsto \texttt{ap}(e_1';e_2)} \tag{12.3b}$$

$$\frac{}{\texttt{ap}(\texttt{lam}[\tau](x.e);e_2) \mapsto [e_2/x]e} \tag{12.3c}$$

$$\frac{e \mapsto e'}{\texttt{natrec}(e;e_0;x.y.e_1) \mapsto \texttt{natrec}(e';e_0;x.y.e_1)} \tag{12.3d}$$

$$\frac{}{\texttt{natrec}(\texttt{z};e_0;x.y.e_1) \mapsto e_0} \tag{12.3e}$$

$$\frac{\texttt{s}(e)\ \mathsf{val}}{\texttt{natrec}(\texttt{s}(e);e_0;x.y.e_1) \mapsto [e,\texttt{natrec}(e;e_0;x.y.e_1)/x,y]e_1} \tag{12.3f}$$

Rules (12.3e) and (12.3f) specify the behavior of the recursor on z and s(e). In the former case the recursor evaluates 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 12.2 (Canonical Forms). *If $e : \tau$ and e val, then*

1. *If $\tau = \texttt{nat}$, then $e = \texttt{s(s(}\ldots\texttt{z))}$ 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 12.3 (Safety). 1. *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

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

12.3 Definability

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

$$e_f(\overline{n}) \equiv \overline{f(n)} : \texttt{nat}. \tag{12.4}$$

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

Definitional equivalence for $\mathcal{L}\{\texttt{nat}\to\}$, written $\Gamma \vdash e \equiv e' : \tau$, is the strongest congruence containing these axioms:

$$\frac{}{\Gamma \vdash \texttt{ap(lam[}\tau\texttt{](}x\texttt{.}e_2\texttt{);}e_1\texttt{)} \equiv [e_1/x]e_2 : \tau} \tag{12.5a}$$

$$\frac{}{\Gamma \vdash \texttt{natrec(z;}e_0\texttt{;}x\texttt{.}y\texttt{.}e_1\texttt{)} \equiv e_0 : \tau} \tag{12.5b}$$

$$\frac{}{\Gamma \vdash \texttt{natrec(s(}e\texttt{);}e_0\texttt{;}x\texttt{.}y\texttt{.}e_1\texttt{)} \equiv [e, \texttt{natrec(}e\texttt{;}e_0\texttt{;}x\texttt{.}y\texttt{.}e_1\texttt{)}/x, y]e_1 : \tau} \tag{12.5c}$$

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

$$\lambda\ (x\texttt{:nat.natrec}\ x\ \{\texttt{z} \Rightarrow \texttt{z} \mid \texttt{s(}u\texttt{)}\ \texttt{with}\ v \Rightarrow \texttt{s(s(}v\texttt{))}\}\texttt{)}.$$

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

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

For the induction, assume that

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

Then calculate using the rules of definitional equivalence:

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

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

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

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 argument (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 $\texttt{nat} \to \texttt{nat}$ that is defined using primitive recursion, 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 is to showing that it is definable in $\mathcal{L}\{\texttt{nat}\rightarrow\}$ is to observe that $A(m+1,n)$ iterates the function $A(m,-)$ for n times, starting with $A(m,1)$. As an auxiliary, let us define the higher-order function

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

to be the λ-abstraction

$$\lambda\,(f\texttt{:nat} \to \texttt{nat}.\,\lambda\,(n\texttt{:nat.natrec}\,n\,\{\texttt{z} \Rightarrow \texttt{id} \mid \texttt{s(\_)}\,\texttt{with}\,g \Rightarrow f \circ g\})),$$

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

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

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

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

to be the expression

$$\lambda\,(m{:}\texttt{nat}.\,\texttt{natrec}\,m\,\{\texttt{z} \Rightarrow \texttt{succ} \mid \texttt{s}(\_)\,\texttt{with}\,f \Rightarrow \lambda\,(n{:}\texttt{nat}.\,\texttt{it}(f)(n)(f(\overline{1})))\}).$$

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

$$\begin{aligned} e_a(\overline{0})(\overline{n}) &\equiv \texttt{s}(\overline{n}) && (12.6)\\ e_a(\overline{m+1})(\overline{0}) &\equiv e_a(\overline{m})(\overline{1}) && (12.7)\\ e_a(\overline{m+1})(\overline{n+1}) &\equiv e_a(\overline{m})(e_a(\texttt{s}(\overline{m}))(\overline{n})). && (12.8)\end{aligned}$$

That is, the Ackermann function is definable in $\mathcal{L}\{\texttt{nat} \to\}$.

12.4 Non-Definability

It is impossible to define an infinite loop in $\mathcal{L}\{\texttt{nat} \to\}$.

Theorem 12.4. *If $e : \tau$, then there exists v val such that $e \equiv v : \tau$.*

Proof. See Corollary 51.9 on page 474. □

Consequently, values of function type in $\mathcal{L}\{\texttt{nat} \to\}$ behave like mathematical functions: if $f : \sigma \to \tau$ and $e : \sigma$, then $f(e)$ evaluates to a value of type τ. Moreover, if $e : \texttt{nat}$, then there exists a natural number n such that $e \equiv \overline{n} : \texttt{nat}$.

Using this, we can show, using a technique called *diagonalization*, that there are functions on the natural numbers that are not definable in the $\mathcal{L}\{\texttt{nat} \to\}$. We make use of a technique, called *Gödel-numbering*, that assigns a unique natural number to each closed expression of $\mathcal{L}\{\texttt{nat} \to\}$. This allows us to manipulate expressions as data values in $\mathcal{L}\{\texttt{nat} \to\}$, and hence permits $\mathcal{L}\{\texttt{nat} \to\}$ 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

[1] The same technique lies at the heart of the proof of Gödel's celebrated incompleteness theorem. The non-definability of certain functions on the natural numbers within $\mathcal{L}\{\texttt{nat} \to\}$ may be seen as a form of incompleteness similar to that considered by Gödel.

that α-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 kth 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 obviously 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 12.4 on the preceding page, ensure that f_{univ} is a well-defined function. It is called the *universal function* for $\mathcal{L}\{\mathtt{nat}\to\}$ 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*, $d : \mathbb{N} \to \mathbb{N}$, by the equation $d(m) = f_{univ}(m)(m)$. This function is chosen so that $d(\ulcorner e \urcorner) = n$ iff $e(\overline{\ulcorner e \urcorner}) \equiv \overline{n} : \mathtt{nat}$. (The motivation for this definition will be apparent in a moment.)

The function d is not definable in $\mathcal{L}\{\mathtt{nat}\to\}$. Suppose that d were defined by the expression e_d, so that we have

$$e_d(\overline{\ulcorner e \urcorner}) \equiv e(\overline{\ulcorner e \urcorner}) : \mathtt{nat}.$$

Let e_D be the expression

$$\lambda\,(x{:}\mathtt{nat}.\,\mathtt{s}(e_d(x)))$$

of type $\mathtt{nat} \to \mathtt{nat}$. We then have

$$\begin{aligned} e_D(\overline{\ulcorner e_D \urcorner}) &\equiv \mathtt{s}(e_d(\overline{\ulcorner e_D \urcorner})) \\ &\equiv \mathtt{s}(e_D(\overline{\ulcorner e_D \urcorner})). \end{aligned}$$

[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.

But the termination theorem implies that there exists n such that $e_D(\overline{\ulcorner e_D \urcorner}) \equiv \overline{n}$, and hence we have $\overline{n} \equiv \texttt{s}(\overline{n})$, which is impossible.

The function f_{univ} is computable (that is, one can write an interpreter for $\mathcal{L}\{\texttt{nat} \rightarrow\}$), but it is not programmable in $\mathcal{L}\{\texttt{nat} \rightarrow\}$ itself. In general a language $\mathcal{L}$ is *universal* if we can write an interpreter for $\mathcal{L}$ in the language $\mathcal{L}$ itself. The foregoing argument shows that $\mathcal{L}\{\texttt{nat} \rightarrow\}$ is *not universal*. Consequently, there are computable numeric functions, such as the diagonal function, that cannot be programmed in $\mathcal{L}\{\texttt{nat} \rightarrow\}$. Consequently, the universal function for $\mathcal{L}\{\texttt{nat} \rightarrow\}$ cannot be programmed in the language. In other words, one cannot write an interpreter for $\mathcal{L}\{\texttt{nat} \rightarrow\}$ in the language itself!

12.5 Exercises

1. Explore variant dynamics for $\mathcal{L}\{\texttt{nat} \rightarrow\}$, both separately and in combination, in which the successor does not evaluate its argument, and in which functions are called by value.

Chapter 13

Plotkin's PCF

The language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, also known as *Plotkin's **PCF***, integrates functions and natural numbers using *general recursion*, a means of defining self-referential expressions. In contrast to $\mathcal{L}\{\texttt{nat} \rightarrow\}$ expressions in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ may not terminate when evaluated; consequently, functions are partial (may be undefined for some arguments), rather than total (which explains the "partial arrow" notation for function types). Compared to $\mathcal{L}\{\texttt{nat} \rightarrow\}$, the language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ moves the termination proof from the expression itself to the mind of the programmer. The type system no longer ensures termination, which permits a wider range of functions to be defined in the system, but at the cost of admitting infinite loops when the termination proof is either incorrect or absent.

The crucial concept embodied in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ is the *fixed point* characterization of recursive definitions. In ordinary mathematical practice one may define a function f by *recursion equations* such as these:

$$\begin{aligned} f(0) &= 1 \\ f(n+1) &= (n+1) \times f(n) \end{aligned}$$

These may be viewed as simultaneous equations in the variable, f, ranging over functions on the natural numbers. The function we seek is a *solution* to these equations—a function $f : \mathbb{N} \to \mathbb{N}$ such that the above conditions are satisfied. We must, of course, show that these equations have a unique solution, which is easily shown by mathematical induction on the argument to f.

The solution to such a system of equations may be characterized as the fixed point of an associated functional (operator mapping functions to functions). To see this, let us re-write these equations in another form:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1 \end{cases}$$

Re-writing yet again, we seek f such that

$$f : n \mapsto \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1 \end{cases}$$

Now define the *functional* F by the equation $F(f) = f'$, where

$$f' : n \mapsto \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1 \end{cases}$$

Note well that the condition on f' is expressed in terms of the argument, f, to the functional F, and not in terms of f' itself! The function f we seek is then a *fixed point* of F, which is a function $f : \mathbb{N} \to \mathbb{N}$ such that $f = F(f)$. In other words f is defined to the *fix*(F), where *fix* is an operator on functionals yielding a fixed point of F.

Why does an operator such as F have a fixed point? Informally, a fixed point may be obtained as the limit of series of approximations to the desired solution obtained by iterating the functional F. This is where partial functions come into the picture. Let us say that a partial function, ϕ on the natural numbers, is an *approximation* to a total function, f, if $\phi(m) = n$ implies that $f(m) = n$. Let $\perp$: $\mathbb{N} \rightharpoonup \mathbb{N}$ 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, ϕ, of f, we may "improve" it by considering $\phi' = F(\phi)$. Intuitively, ϕ' is defined on 0 and on $m + 1$ for every $m \geq 0$ on which ϕ is defined. Continuing in this manner, $\phi'' = F(\phi') = F(F(\phi))$ is an improvement on ϕ', and hence a further improvement on ϕ. 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.

This fixed point characterization of recursion equations is taken as a primitive concept in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$—we may obtain the least fixed point of *any* functional definable in the language. Using this we may solve any set of recursion equations we like, with the proviso that there is no guarantee that the solution is a *total* function. Rather, it is guaranteed to be a *partial* function that may be undefined on some, all, or no inputs. This is the price we pay for expressive power—we may solve all systems of equations, but the solution may not be as well-behaved as we might like it to be. It is our task as programmers to ensure that the functions defined by recursion are total—all of our loops terminate.

13.1 Statics

The abstract binding syntax of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ is given by the following grammar:

Type	τ	$::=$	$\texttt{nat}$	$\texttt{nat}$	naturals
			$\texttt{parr}(\tau_1;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$	partial function
Expr	e	$::=$	x	x	variable
			$\texttt{z}$	$\texttt{z}$	zero
			$\texttt{s}(e)$	$\texttt{s}(e)$	successor
			$\texttt{ifz}(e;e_0;x.e_1)$	$\texttt{ifz}\,e\,\{\texttt{z} \Rightarrow e_0 \mid \texttt{s}(x) \Rightarrow e_1\}$	zero test
			$\texttt{lam}[\tau](x.e)$	$\lambda\,(x{:}\tau.\,e)$	abstraction
			$\texttt{ap}(e_1;e_2)$	$e_1(e_2)$	application
			$\texttt{fix}[\tau](x.e)$	$\texttt{fix}\,x{:}\tau\,\texttt{is}\,e$	recursion

The expression $\texttt{fix}[\tau](x.e)$ is called *general recursion*; it is discussed in more detail below. The expression $\texttt{ifz}(e;e_0;x.e_1)$ branches according to whether e evaluates to $\texttt{z}$ or not, binding the predecessor to x in the case that it is not.

The statics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ is inductively defined by the following rules:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{13.1a}$$

$$\frac{}{\Gamma \vdash \texttt{z} : \texttt{nat}} \tag{13.1b}$$

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{s}(e) : \texttt{nat}} \tag{13.1c}$$

$$\frac{\Gamma \vdash e : \texttt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \texttt{nat} \vdash e_1 : \tau}{\Gamma \vdash \texttt{ifz}(e;e_0;x.e_1) : \tau} \tag{13.1d}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam}[\tau_1](x.e) : \texttt{parr}(\tau_1;\tau_2)} \tag{13.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{parr}(\tau_2;\tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{13.1f}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{fix}[\tau](x.e) : \tau} \tag{13.1g}$$

Rule (13.1g) reflects the self-referential nature of general recursion. To show that $\mathtt{fix}[\tau](x.e)$ has type τ, 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 τ under this very assumption.

The structural rules, including in particular substitution, are admissible for the static semantics.

Lemma 13.1. *If $\Gamma, x : \tau \vdash e' : \tau'$, $\Gamma \vdash e : \tau$, then $\Gamma \vdash [e/x]e' : \tau'$.*

13.2 Dynamics

The dynamic semantics of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is defined by the judgements e val, specifying the closed values, and $e \mapsto e'$, specifying the steps of evaluation. We will consider a call-by-name dynamics for function application, and require that the successor evaluate its argument.

The judgement e val is defined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{13.2a}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{s}(e)\ \mathsf{val}} \tag{13.2b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{13.2c}$$

The bracketed premise on Rule (13.2b) is to be included for the *eager* interpretation of the sucessor operation, and omitted for the *lazy* interpretation. (See Section 13.4 on page 114 for more on this choice, which is further elaborated in Chapter 41).

The transition judgement $e \mapsto e'$ is defined by the following rules:

$$\left\{\frac{e \mapsto e'}{\mathtt{s}(e) \mapsto \mathtt{s}(e')}\right\} \tag{13.3a}$$

$$\frac{e \mapsto e'}{\mathtt{ifz}(e;e_0;x.e_1) \mapsto \mathtt{ifz}(e';e_0;x.e_1)} \tag{13.3b}$$

$$\frac{}{\mathtt{ifz}(\mathtt{z};e_0;x.e_1) \mapsto e_0} \tag{13.3c}$$

$$\frac{\texttt{s}(e)\ \mathsf{val}}{\texttt{ifz}(\texttt{s}(e);e_0;x.e_1) \mapsto [e/x]e_1} \tag{13.3d}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap}(e_1;e_2) \mapsto \texttt{ap}(e_1';e_2)} \tag{13.3e}$$

$$\frac{}{\texttt{ap}(\texttt{lam}[\tau](x.e);e_2) \mapsto [e_2/x]e} \tag{13.3f}$$

$$\frac{}{\texttt{fix}[\tau](x.e) \mapsto [\texttt{fix}[\tau](x.e)/x]e} \tag{13.3g}$$

The bracketed Rule (13.3a) is to be included for an eager interpretation of the successor, and omitted otherwise. Rule (13.3g) implements self-reference by substituting the recursive expression itself for the variable x in its body. This is called *unwinding* the recursion.

Theorem 13.2 (Safety). *1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

2. If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.

Proof. The proof of preservation is by induction on the derivation of the transition judgement. Consider Rule (13.3g). Suppose that $\texttt{fix}[\tau](x.e) : \tau$. By inversion and substitution we have $[\texttt{fix}[\tau](x.e)/x]e : \tau$, from which the result follows directly by transitivity of the hypothetical judgement. The proof of progress proceeds by induction on the derivation of the typing judgement. For example, for Rule (13.1g) the result follows immediately since we may make progress by unwinding the recursion. □

Definitional equivalence for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, written $\Gamma \vdash e_1 \equiv e_2 : \tau$, is defined to be the strongest congruence containing the following axioms:

$$\frac{}{\Gamma \vdash \texttt{ifz}(\texttt{z};e_0;x.e_1) \equiv e_0 : \tau} \tag{13.4a}$$

$$\frac{}{\Gamma \vdash \texttt{ifz}(\texttt{s}(e);e_0;x.e_1) \equiv [e/x]e_1 : \tau} \tag{13.4b}$$

$$\frac{}{\Gamma \vdash \texttt{fix}[\tau](x.e) \equiv [\texttt{fix}[\tau](x.e)/x]e : \tau} \tag{13.4c}$$

$$\frac{}{\Gamma \vdash \texttt{ap}(\texttt{lam}[\tau](x.e_2);e_1) \equiv [e_1/x]e_2 : \tau} \tag{13.4d}$$

These rules are sufficient to calculate the value of any closed expression of type $\texttt{nat}$: if $e : \texttt{nat}$, then $e \equiv \overline{n} : \texttt{nat}$ iff $e \mapsto^* \overline{n}$.

13.3 Definability

General recursion is a very flexible programming technique that permits a wide variety of functions to be defined within $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$. 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.

General recursive functions are definable from general recursion and non-recursive functions. Let us write $\texttt{fun}\,x\,\texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e$ for a recursive function within whose body, $e : \tau_2$, are bound two variables, $y : \tau_1$ standing for the argument and $x : \tau_1 \to \tau_2$ standing for the function itself. The dynamic semantics of this construct is given by the axiom

$$\frac{}{\texttt{fun}\,x\,\texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e\texttt{(}e_1\texttt{)} \mapsto [\texttt{fun}\,x\,\texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e, e_1/x, y]e}\,.$$

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 $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ using a combination of recursion and functions, writing

$$\texttt{fix}\,x\texttt{:}\tau_1 \rightharpoonup \tau_2\,\texttt{is}\,\lambda\,\texttt{(}y\texttt{:}\tau_1\texttt{.}\,e\texttt{)}$$

for $\texttt{fun}\,x\,\texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e$. 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 $\mathcal{L}\{\texttt{nat} \to\}$ is defined in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ using recursive functions by taking the expression

$$\texttt{natrec}\,e\,\{\texttt{z} \Rightarrow e_0 \mid \texttt{s(}x\texttt{)}\,\texttt{with}\,y \Rightarrow e_1\}$$

to stand for the application, $e'\texttt{(}e\texttt{)}$, where e' is the general recursive function

$$\texttt{fun}\,f\,\texttt{(}u\texttt{:nat)}\texttt{:}\tau\,\texttt{is}\,\texttt{ifz}\,u\,\{\texttt{z} \Rightarrow e_0 \mid \texttt{s(}x\texttt{)} \Rightarrow [f\texttt{(}x\texttt{)}/y]e_1\}.$$

The static and dynamic semantics of primitive recursion are derivable in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ using this expansion.

In general, functions definable in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ are partial in that they may be undefined for some arguments. A partial (mathematical) function, $\phi : \mathbb{N} \rightharpoonup \mathbb{N}$, is *definable* in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ iff there is an expression $e_\phi : \texttt{nat} \rightharpoonup \texttt{nat}$ such that $\phi(m) = n$ iff $e_\phi\texttt{(}\overline{m}\texttt{)} \equiv \overline{n} : \texttt{nat}$. So, for example, if ϕ is the totally undefined function, then e_ϕ is any function that loops without returning whenever it is called.

It is informative to classify those partial functions ϕ that are definable in $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. 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 13.3. *A partial function ϕ on the natural numbers is definable in $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$ iff it is partial recursive.*

Proof sketch. Minimization is readily definable in $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$, 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 $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$ as a partial recursive function, using Gödel-numbering to represent expressions as numbers. Consequently, $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$ 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 $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$ is as powerful as any other programming language with respect to the set of definable functions on the natural numbers.

The universal function, ϕ_{univ}, for $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$ is the partial function on the natural numbers defined by

$$\phi_{univ}(\ulcorner e \urcorner)(m) = n \text{ iff } e(\overline{m}) \equiv \overline{n} : \mathtt{nat}.$$

In contrast to $\mathcal{L}\{\mathtt{nat}\rightarrow\}$, the universal function ϕ_{univ} for $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$ is partial (may be undefined for some inputs). It is, in essence, an interpreter that, given the code $\ulcorner e \urcorner$ 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 $\overline{n}$. Since this process may not terminate, the universal function is not defined for all inputs.

By Church's Law the universal function is definable in $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. In contrast, we proved in Chapter 12 that the analogous function is *not* definable in $\mathcal{L}\{\mathtt{nat}\rightarrow\}$ using the technique of diagonalization. It is instructive to examine why that argument does not apply in the present setting. As in Section 12.4 on page 104, we may derive the equivalence

$$e_D(\overline{\ulcorner e_D \urcorner}) \equiv \mathtt{s}(e_D(\overline{\ulcorner e_D \urcorner}))$$

[1] See Chapter 20 for further discussion of Church's Law.

for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. The difference, however, is that this equation is not inconsistent! Rather than being contradictory, it is merely a proof that the expression $e_D(\ulcorner e_D \urcorner)$ does not terminate when evaluated, for if it did, the result would be a number equal to its own successor, which is impossible.

13.4 Co-Natural Numbers

The dynamics of the successor operation on natural numbers may be taken to be either eager or lazy, according to whether the predecessor of a successor is required to be a value. The eager interpretation represents the standard natural numbers in the sense that if $e : \mathtt{nat}$ and $e\ \mathsf{val}$, then e evaluates to a numeral. The lazy interpretation, however, admits non-standard "natural numbers," such as

$$\omega = \mathtt{fix}\, x{:}\mathtt{nat}\, \mathtt{is}\, \mathtt{s}(x).$$

The "number" ω evaluates to $\mathtt{s}(\omega)$. This "number" may be thought of as an infinite stack of successors, since whenever we peel off the outermost successor we obtain the same "number" back again. The "number" ω is therefore larger than any other natural number in the sense that one may reach zero by repeatedly taking the predecessor of a natural number, but any number of predecessors on ω leads back to ω itself.

As the scare quotes indicate, it is stretching the terminology to refer to ω as a natural number. Instead one should distinguish a new type, called conat, of *lazy natural numbers*, of which ω is an element. The prefix "co-" indicates that the co-natural numbers are "dual" to the natural numbers in the following sense. The natural numbers are inductively defined as the *least* type such that if $e \equiv \mathtt{z} : \mathtt{nat}$ or $e \equiv \mathtt{s}(e') : \mathtt{nat}$ for some $e' : \mathtt{nat}$, then $e : \mathtt{nat}$. Dually, the co-natural numbers may be regarded as the *largest* type such that if $e : \mathtt{conat}$, then either $e \equiv \mathtt{z} : \mathtt{conat}$, or $e \equiv \mathtt{s}(e') : \mathtt{nat}$ for some $e' : \mathtt{conat}$. The difference is that $\omega : \mathtt{conat}$, because ω is definitionally equivalent to its own successor, whereas it is not the case that $\omega : \mathtt{nat}$, according to these definitions.

The duality between the natural numbers and the co-natural numbers is developed further in Chapter 18, wherein we consider the concepts of inductive and co-inductive types. Eagerness and laziness in general is discussed further in Chapter 41.

13.5 Exercises

Part V

Finite Data Types

Chapter 14

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*, $\prod_{i \in I} \tau_i$, indexed by a finite set of *indices*, I. The elements of the finite product type are *I-indexed tuples* whose ith component is an element of the type τ_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 *labelled tuples*, or *records*, indexed by finite sets of symbols. Similarly to binary products, finite products admit both an eager and a lazy interpretation.

14.1 Nullary and Binary Products

The abstract syntax of products is given by the following grammar:

Type	τ	$::=$	$\mathtt{unit}$	$\mathtt{unit}$	nullary product
			$\mathtt{prod}(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$	binary product
Expr	e	$::=$	$\mathtt{triv}$	$\langle\rangle$	null tuple
			$\mathtt{pair}(e_1;e_2)$	$\langle e_1, e_2\rangle$	ordered pair
			$\mathtt{proj[l]}(e)$	$e \cdot \mathtt{l}$	left projection
			$\mathtt{proj[r]}(e)$	$e \cdot \mathtt{r}$	right projection

There is no elimination form for the unit type, there being nothing to extract from the null tuple.

The statics of product types is given by the following rules.

$$\frac{}{\Gamma \vdash \mathtt{triv} : \mathtt{unit}} \tag{14.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{pair}(e_1;e_2) : \mathtt{prod}(\tau_1;\tau_2)} \tag{14.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{prod}(\tau_1;\tau_2)}{\Gamma \vdash \mathtt{proj[l]}(e) : \tau_1} \tag{14.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{prod}(\tau_1;\tau_2)}{\Gamma \vdash \mathtt{proj[r]}(e) : \tau_2} \tag{14.1d}$$

The dynamics of product types is specified by the following rules:

$$\frac{}{\mathtt{triv}\ \mathsf{val}} \tag{14.2a}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\mathtt{pair}(e_1;e_2)\ \mathsf{val}} \tag{14.2b}$$

$$\left\{\frac{e_1 \mapsto e_1'}{\mathtt{pair}(e_1;e_2) \mapsto \mathtt{pair}(e_1';e_2)}\right\} \tag{14.2c}$$

$$\left\{\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\mathtt{pair}(e_1;e_2) \mapsto \mathtt{pair}(e_1;e_2')}\right\} \tag{14.2d}$$

$$\frac{e \mapsto e'}{\mathtt{proj[l]}(e) \mapsto \mathtt{proj[l]}(e')} \tag{14.2e}$$

$$\frac{e \mapsto e'}{\mathtt{proj[r]}(e) \mapsto \mathtt{proj[r]}(e')} \tag{14.2f}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\texttt{proj[l](pair(}e_1\texttt{;}e_2\texttt{))} \mapsto e_1} \tag{14.2g}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\texttt{proj[r](pair(}e_1\texttt{;}e_2\texttt{))} \mapsto e_2} \tag{14.2h}$$

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 14.1 (Safety). *1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

2. If $e : \tau$ then either e val or there exists e' such that $e \mapsto e'$.

Proof. Preservation is proved by induction on transition defined by Rules (14.2). Progress is proved by induction on typing defined by Rules (14.1). □

14.2 Finite Products

The syntax of finite product types is given by the following grammar:

$$\begin{array}{llllll}
\mathsf{Type} & \tau & ::= & \texttt{prod[}I\texttt{](}i \mapsto \tau_i\texttt{)} & \prod_{i \in I} \tau_i & \text{product} \\
\mathsf{Expr} & e & ::= & \texttt{tuple[}I\texttt{](}i \mapsto e_i\texttt{)} & \langle e_i \rangle_{i \in I} & \text{tuple} \\
 & & & \texttt{proj[}I\texttt{][}i\texttt{](}e\texttt{)} & e \cdot i & \text{projection}
\end{array}$$

For I a finite index set of size $n \geq 0$, the syntactic form $\texttt{prod[}I\texttt{](}i \mapsto \tau_i\texttt{)}$ specifies an n-argument operator of arity $(0, 0, \ldots, 0)$ whose ith argument is the type τ_i. When it is useful to emphasize the tree structure, such an abt is written in the form $\prod \langle i_0 : \tau_0, \ldots, i_{n-1} : \tau_{n-1} \rangle$. Similarly, the syntactic form $\texttt{tuple[}I\texttt{](}i \mapsto e_i\texttt{)}$ specifies an abt constructed from an n-argument operator whose i operand is e_i. This may alternatively be written in the form $\langle i_0 : e_0, \ldots, i_{n-1} : e_{n-1} \rangle$.

The statics of finite products is given by the following rules:

$$\frac{(\forall i \in I)\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{tuple[}I\texttt{](}i \mapsto e_i\texttt{)} : \texttt{prod[}I\texttt{](}i \mapsto \tau_i\texttt{)}} \tag{14.3a}$$

$$\frac{\Gamma \vdash e : \texttt{prod[}I\texttt{](}i \mapsto e_i\texttt{)} \quad j \in I}{\Gamma \vdash \texttt{proj[}I\texttt{][}j\texttt{](}e\texttt{)} : \tau_j} \tag{14.3b}$$

In Rule (14.3b) the index $j \in I$ is a *particular* element of the index set I, whereas in Rule (14.3a), the index i ranges over the index set I.

The dynamics of finite products is given by the following rules:

$$\frac{\{(\forall i \in I)\ e_i\ \mathsf{val}\}}{\texttt{tuple}[I](i \mapsto e_i)\ \mathsf{val}} \tag{14.4a}$$

$$\left\{\frac{e_j \mapsto e_j' \quad (\forall i \neq j)\ e_i' = e_i}{\texttt{tuple}[I](i \mapsto e_i) \mapsto \texttt{tuple}[I](i \mapsto e_i')}\right\} \tag{14.4b}$$

$$\frac{e \mapsto e'}{\texttt{proj}[I][j](e) \mapsto \texttt{proj}[I][j](e')} \tag{14.4c}$$

$$\frac{\texttt{tuple}[I](i \mapsto e_i)\ \mathsf{val}}{\texttt{proj}[I][j](\texttt{tuple}[I](i \mapsto e_i)) \mapsto e_j} \tag{14.4d}$$

Rule (14.4b) specifies that the components of a tuple are to be evaluated in *some* sequential order, without specifying the order in which they components are considered. It is straightforward, if a bit technically complicated, to impose a linear ordering on index sets that determines the evaluation order of the components of a tuple.

Theorem 14.2 (Safety). *If $e : \tau$, then either e val or there exists e' such that $e' : \tau$ and $e \mapsto e'$.*

Proof. The safety theorem may be decomposed into progress and preservation lemmas, which are proved as in Section 14.1 on page 118. □

We may define nullary and binary products as particular instances of finite products by choosing an appropriate index set. The type `unit` may be defined as the product $\prod_{\_\in\emptyset} \emptyset$ of the empty family over the empty index set, taking the expression $\langle\rangle$ to be the empty tuple, $\langle\emptyset\rangle_{\_\in\emptyset}$. Binary products $\tau_1 \times \tau_2$ may be defined as the product $\prod_{i\in\{1,2\}} \tau_i$ of the two-element family of types consisting of τ_1 and τ_2. The pair $\langle e_1, e_2\rangle$ may then be defined as the tuple $\langle e_i\rangle_{i\in\{1,2\}}$, and the projections $e \cdot \texttt{l}$ and $e \cdot \texttt{r}$ are correspondingly defined, respectively, to be $e \cdot 1$ and $e \cdot 2$.

Finite products may also be used to define *labelled tuples*, or *records*, whose components are accessed by symbolic names. If $L = \{ l_1, \ldots, l_n \}$ is a finite set of symbols, called *field names*, or *field labels*, then the product type $\prod \langle l_0 : \tau_0, \ldots, l_{n-1} : \tau_{n-1}\rangle$ has as values tuples of the form $\langle l_0 : e_0, \ldots, l_{n-1} : e_{n-1}\rangle$ in which $e_i : \tau_i$ for each $0 \leq i < n$. If e is such a tuple, then $e \cdot l$ projects the component of e labeled by $l \in L$.

14.3 Primitive and Mutual Recursion

In the presence of products we may simplify the primitive recursion construct defined in Chapter 12 so that only the result on the predecessor, and not the predecessor itself, is passed to the successor branch. Writing this as $\texttt{natiter}\,e\,\{\texttt{z}\Rightarrow e_0 \mid \texttt{s}(x)\Rightarrow e_1\}$, we may define primitive recursion in the sense of Chapter 12 to be the expression $e' \cdot \texttt{r}$, where e' is the expression

$$\texttt{natiter}\,e\,\{\texttt{z}\Rightarrow\langle\texttt{z},e_0\rangle \mid \texttt{s}(x)\Rightarrow\langle\texttt{s}(x\cdot\texttt{l}),[x\cdot\texttt{l},x\cdot\texttt{r}/x_0,x_1]e_1\rangle\}.$$

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 recursion*, which allows several mutually recursive computations to be defined simultaneously. 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 iff n is even, and $O(n)$ is non-zero iff n is odd. If we wish to define these functions in $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$, we immediately face the problem of how to define two functions simultaneously. There is a trick available in this special case that takes advantage of the fact that E and O have the same type: simply define $\texttt{eo}$ of type $\texttt{nat}\to\texttt{nat}\to\texttt{nat}$ so that $\texttt{eo}(\overline{0})$ represents E and $\texttt{eo}(\overline{1})$ represents O. (We leave the details as an exercise for the reader.)

A more general solution is to recognize that the definition of two mutually recursive functions may be thought of as the recursive definition of a pair of functions. In the case of the even and odd functions we will define the labelled tuple, e_{EO}, of type, τ_{EO}, given by

$$\prod\langle\texttt{even}:\texttt{nat}\to\texttt{nat},\texttt{odd}:\texttt{nat}\to\texttt{nat}\rangle.$$

From this we will obtain the required mutually recursive functions as the projections $e_{EO}\cdot\texttt{even}$ and $e_{EO}\cdot\texttt{odd}$.

To effect the mutual recursion the expression e_{EO} is defined to be

$$\mathtt{fix}\,\mathtt{this}\,\mathtt{:}\,\tau_{EO}\,\mathtt{is}\,\langle \mathtt{even}\,\mathtt{:}\,e_E, \mathtt{odd}\,\mathtt{:}\,e_O \rangle,$$

where e_E is the expression

$$\lambda\,\mathtt{(x:nat.\,ifz}\,\mathtt{x}\,\{\mathtt{z} \Rightarrow \mathtt{s(z)} \mid \mathtt{s(y)} \Rightarrow \mathtt{this} \cdot \mathtt{odd(y)}\}\mathtt{)},$$

and e_O is the expression

$$\lambda\,\mathtt{(x:nat.\,ifz}\,\mathtt{x}\,\{\mathtt{z} \Rightarrow \mathtt{z} \mid \mathtt{s(y)} \Rightarrow \mathtt{this} \cdot \mathtt{even(y)}\}\mathtt{)}.$$

The functions e_E and e_O refer to each other by projecting the appropriate component from the variable this standing for the object itself. The choice of variable name with which to effect the self-reference is, of course, immaterial, but it is common to use this or self to emphasize its role.

In the context of *object-oriented* languages, labelled tuples of mutually recursive functions defined in this manner are called *objects*, and their component functions are called *methods*. Component projection is called *message passing*, viewing the component name as a “message” sent to the object to invoke the method by that name in the object. Internally to the object the methods refer to one another by sending a “message” to this, the canonical name for the object itself.

14.4 Exercises

Chapter 15

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.

15.1 Binary and Nullary Sums

The abstract syntax of sums is given by the following grammar:

Type	τ	$::=$	$\mathtt{void}$	$\mathtt{void}$	nullary sum
			$\mathtt{sum}(\tau_1;\tau_2)$	$\tau_1 + \tau_2$	binary sum
Expr	e	$::=$	$\mathtt{abort}[\tau](e)$	$\mathtt{abort}_{\tau}\, e$	abort
			$\mathtt{in}[\mathtt{l}][\tau](e)$	$\mathtt{l} \cdot e$	left injection
			$\mathtt{in}[\mathtt{r}][\tau](e)$	$\mathtt{r} \cdot e$	right injection
			$\mathtt{case}(e;x_1.e_1;x_2.e_2)$	$\mathtt{case}\, e\, \{\mathtt{l} \cdot x_1 \Rightarrow e_1 \mid \mathtt{r} \cdot x_2 \Rightarrow e_2\}$	case analysis

The nullary sum represents a choice of zero alternatives, and hence admits no introductory form. The eliminatory form, $\mathtt{abort}[\tau](e)$, aborts the computation in the event that e evaluates to a value, which it cannot do. The elements of the binary sum type are labelled to indicate whether they are drawn from the left or the right summand, either $\texttt{in[l][}\tau\texttt{](}e\texttt{)}$ or $\texttt{in[r][}\tau\texttt{](}e\texttt{)}$. A value of the sum type is eliminated by case analysis.

The statics of sum types is given by the following rules.

$$\frac{\Gamma \vdash e : \texttt{void}}{\Gamma \vdash \texttt{abort[}\tau\texttt{](}e\texttt{)} : \tau} \tag{15.1a}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau = \texttt{sum(}\tau_1\texttt{;}\tau_2\texttt{)}}{\Gamma \vdash \texttt{in[l][}\tau\texttt{](}e\texttt{)} : \tau} \tag{15.1b}$$

$$\frac{\Gamma \vdash e : \tau_2 \quad \tau = \texttt{sum(}\tau_1\texttt{;}\tau_2\texttt{)}}{\Gamma \vdash \texttt{in[r][}\tau\texttt{](}e\texttt{)} : \tau} \tag{15.1c}$$

$$\frac{\Gamma \vdash e : \texttt{sum(}\tau_1\texttt{;}\tau_2\texttt{)} \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case(}e\texttt{;}x_1\texttt{.}e_1\texttt{;}x_2\texttt{.}e_2\texttt{)} : \tau} \tag{15.1d}$$

Both branches of the case analysis must have the same type. Since a type expresses a static "prediction" on the form of the value of an expression, and since a value 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:

$$\frac{e \mapsto e'}{\texttt{abort[}\tau\texttt{](}e\texttt{)} \mapsto \texttt{abort[}\tau\texttt{](}e'\texttt{)}} \tag{15.2a}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{in[l][}\tau\texttt{](}e\texttt{)}\ \mathsf{val}} \tag{15.2b}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{in[r][}\tau\texttt{](}e\texttt{)}\ \mathsf{val}} \tag{15.2c}$$

$$\left\{\frac{e \mapsto e'}{\texttt{in[l][}\tau\texttt{](}e\texttt{)} \mapsto \texttt{in[l][}\tau\texttt{](}e'\texttt{)}}\right\} \tag{15.2d}$$

$$\left\{\frac{e \mapsto e'}{\texttt{in[r][}\tau\texttt{](}e\texttt{)} \mapsto \texttt{in[r][}\tau\texttt{](}e'\texttt{)}}\right\} \tag{15.2e}$$

$$\frac{e \mapsto e'}{\texttt{case(}e\texttt{;}x_1\texttt{.}e_1\texttt{;}x_2\texttt{.}e_2\texttt{)} \mapsto \texttt{case(}e'\texttt{;}x_1\texttt{.}e_1\texttt{;}x_2\texttt{.}e_2\texttt{)}} \tag{15.2f}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{case(in[l][}\tau\texttt{](}e\texttt{);}x_1\texttt{.}e_1\texttt{;}x_2\texttt{.}e_2\texttt{)} \mapsto [e/x_1]e_1} \tag{15.2g}$$

$$\frac{\{e\ \mathsf{val}\}}{\texttt{case(in[r][}\tau\texttt{](}e\texttt{);}x_1\texttt{.}e_1\texttt{;}x_2\texttt{.}e_2\texttt{)} \mapsto [e/x_2]e_2} \tag{15.2h}$$

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 15.1 (Safety).

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

Proof. The proof proceeds by induction on Rules (15.2) for preservation, and by induction on Rules (15.1) for progress. □

15.2 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:

Type	τ	$::=$	$\texttt{sum}[I](i \mapsto \tau_i)$	$\sum_{i \in I} \tau_i$	sum
Expr	e	$::=$	$\texttt{in}[I][j](e)$	$j \cdot e$	injection
			$\texttt{case}[I](e; i \mapsto x_i . e_i)$	$\texttt{case}\, e\, \{ i \cdot x_i \Rightarrow e_i \}_{i \in I}$	case analysis

We write $\sum \langle i_0 : \tau_0, \ldots, i_{n-1} : \tau_{n-1} \rangle$ for $\sum_{i \in I} \tau_i$, where $I = \{ i_0, \ldots, i_{n-1} \}$.

The statics of finite sums is defined by the following rules:

$$\frac{\Gamma \vdash e : \tau_j \quad j \in I}{\Gamma \vdash \texttt{in}[I][j](e) : \texttt{sum}[I](i \mapsto \tau_i)} \tag{15.3a}$$

$$\frac{\Gamma \vdash e : \texttt{sum}[I](i \mapsto \tau_i) \quad (\forall i \in I)\ \Gamma, x_i : \tau_i \vdash e_i : \tau}{\Gamma \vdash \texttt{case}[I](e; i \mapsto x_i . e_i) : \tau} \tag{15.3b}$$

These rules generalize to the finite case the statics for nullary and binary sums given in Section 15.1 on page 123.

The dynamics of finite sums is defined by the following rules:

$$\frac{\{e\ \mathsf{val}\}}{\texttt{in}[I][j](e)\ \mathsf{val}} \tag{15.4a}$$

$$\left\{ \frac{e \mapsto e'}{\texttt{in}[I][j](e) \mapsto \texttt{in}[I][j](e')} \right\} \tag{15.4b}$$

$$\frac{e \mapsto e'}{\texttt{case}[I](e; i \mapsto x_i . e_i) \mapsto \texttt{case}[I](e'; i \mapsto x_i . e_i)} \tag{15.4c}$$

$$\frac{\texttt{in}[I][j](e)\ \mathsf{val}}{\texttt{case}[I](\texttt{in}[I][j](e); i \mapsto x_i . e_i) \mapsto [e/x_j]e_j} \tag{15.4d}$$

These again generalize the dynamics of binary sums given in Section 15.1 on page 123.

Theorem 15.2 (Safety)**.** *If $e : \tau$, then either e val or there exists $e' : \tau$ such that $e \mapsto e'$.*

Proof. The proof is similar to that for the binary case, as described in Section 15.1 on page 123. □

As with products, nullary and binary sums are special cases of the finite form. The type `void` may be defined to be the sum type $\sum_{\_\in\emptyset} \emptyset$ of the empty family of types. The expression `abort(`e`)` may corresponding be defined as the empty case analysis, `case` $e\,\{\emptyset\}$. Similarly, the binary sum type $\tau_1 + \tau_2$ may be defined as the sum $\sum_{i\in I} \tau_i$, where $I = \{\,\mathtt{l},\mathtt{r}\,\}$ is the two-element index set. The binary sum injections $\mathtt{l}\cdot e$ and $\mathtt{r}\cdot e$ are defined to be their counterparts, $\mathtt{l}\cdot e$ and $\mathtt{r}\cdot e$, respectively. Finally, the binary case analysis,

$$\mathtt{case}\,e\,\{\mathtt{l}\cdot x_{\mathtt{l}} \Rightarrow e_{\mathtt{l}} \mid \mathtt{r}\cdot x_{\mathtt{r}} \Rightarrow e_{\mathtt{r}}\},$$

is defined to be the case analysis, `case` $e\,\{i\cdot x_i \Rightarrow \tau_i\}_{i\in I}$. It is easy to check that the static and dynamics of sums given in Section 15.1 on page 123 is preserved by these definitions.

Two special cases of finite sums arise quite commonly. The *n-ary sum* corresponds to the finite sum over an index set of the form $\{\,0,\ldots,n-1\,\}$ for some $n \geq 0$. The *labelled sum* corresponds to the case of the index set being a finite set of symbols serving as symbolic indices for the injections.

15.3 Applications of Sum Types

Sum types have numerous uses, several of which we outline here. More interesting examples arise once we also have recursive types, which are introduced in Part VI.

15.3.1 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, `triv`, whereas the type `void` has no elements at all. Consequently, if e : `unit`, then if e evaluates to a value, it must be `unit` — in other words, e has *no interesting value* (but it could diverge). 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!

15.3.2 Booleans

Perhaps the simplest example of a sum type is the familiar type of Booleans, whose syntax is given by the following grammar:

Type	τ	$::=$	$\texttt{bool}$	$\texttt{bool}$	booleans
Expr	e	$::=$	$\texttt{tt}$	$\texttt{tt}$	truth
			$\texttt{ff}$	$\texttt{ff}$	falsity
			$\texttt{if}(e;e_1;e_2)$	$\texttt{if}\,e\,\texttt{then}\,e_1\,\texttt{else}\,e_2$	conditional

The expression $\texttt{if}(e;e_1;e_2)$ branches on the value of $e : \texttt{bool}$. We leave a precise formulation of the static and dynamics of this type as an exercise for the reader.

The type $\texttt{bool}$ is definable in terms of binary sums and nullary products:

$$\texttt{bool} = \texttt{sum}(\texttt{unit};\texttt{unit}) \tag{15.5a}$$

$$\texttt{tt} = \texttt{in[l][bool](triv)} \tag{15.5b}$$

$$\texttt{ff} = \texttt{in[r][bool](triv)} \tag{15.5c}$$

$$\texttt{if}(e;e_1;e_2) = \texttt{case}(e;x_1.e_1;x_2.e_2) \tag{15.5d}$$

In the last equation above the variables x_1 and x_2 are chosen arbitrarily such that $x_1 \notin e_1$ and $x_2 \notin e_2$. (We often write an underscore in place of a variable to stand for a variable that does not occur within its scope.) It is a simple matter to check that the evident static and dynamics of the type $\texttt{bool}$ is engendered by these definitions.

15.3.3 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 $\texttt{suit}$, whose elements are $\clubsuit$, $\diamondsuit$, $\heartsuit$, and $\spadesuit$, has as elimination form the case analysis

$$\texttt{case}\,e\,\{\clubsuit \Rightarrow e_0 \mid \diamondsuit \Rightarrow e_1 \mid \heartsuit \Rightarrow e_2 \mid \spadesuit \Rightarrow e_3\},$$

which distinguishes among the four suits. Such finite enumerations are easily representable as sums. For example, we may define $\texttt{suit} = \sum_{\_\in I} \texttt{unit}$, where $I = \{\, \clubsuit, \diamondsuit, \heartsuit, \spadesuit \,\}$ and the type family is constant over this set. The case analysis form for a labelled 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.

15.3.4 Options

Another use of sums is to define the *option* types, which have the following syntax:

$$
\begin{array}{llllll}
\mathsf{Type} & \tau & ::= & \mathtt{opt}(\tau) & \tau\,\mathtt{opt} & \text{option} \\
\mathsf{Expr} & e & ::= & \mathtt{null} & \mathtt{null} & \text{nothing} \\
 & & & \mathtt{just}(e) & \mathtt{just}(e) & \text{something} \\
 & & & \mathtt{ifnull}[\tau](e;e_1;x.e_2) & \mathtt{check}\,e\,\{\mathtt{null} \Rightarrow e_1 \mid \mathtt{just}(x) \Rightarrow e_2\} & \\
 & & & & & \text{null test}
\end{array}
$$

The type $\mathtt{opt}(\tau)$ represents the type of “optional” values of type τ. The introductory forms are `null`, corresponding to “no value”, and $\mathtt{just}(e)$, corresponding to a specified value of type τ. The elimination form discriminates between the two possibilities.

The option type is definable from sums and nullary products according to the following equations:

$$\mathtt{opt}(\tau) = \mathtt{sum}(\mathtt{unit};\tau) \tag{15.6a}$$

$$\mathtt{null} = \mathtt{in}[\mathtt{l}][\mathtt{opt}(\tau)](\mathtt{triv}) \tag{15.6b}$$

$$\mathtt{just}(e) = \mathtt{in}[\mathtt{r}][\mathtt{opt}(\tau)](e) \tag{15.6c}$$

$$\mathtt{ifnull}[\tau](e;e_1;x_2.e_2) = \mathtt{case}(e;\_.e_1;x_2.e_2) \tag{15.6d}$$

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 `tt` if its argument is null, and `ff` 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

$$\texttt{if}\,\texttt{null}(e)\,\texttt{then}\ldots\mathit{error}\ldots\texttt{else}\ldots\mathit{proceed}\ldots. \tag{15.7}$$

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 τ from the type $\texttt{opt}(\tau)$. Rather than think of the elements of type τ as pointers, and thereby have to worry about the null pointer, one instead distinguishes between a *genuine* value of type τ and an *optional* value of type τ. An optional value of type τ may or may not be present, but, if it is, the underlying value is truly a value of type τ (and cannot be null). The elimination form for the option type,

$$\texttt{ifnull}[\tau](e;e_{error};x\,.\,e_{ok}) \tag{15.8}$$

propagates the information that e is present into the non-null branch by binding a genuine value of type τ to the variable x. The case analysis effects a change of type from "optional value of type τ" to "genuine value of type τ", 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 (15.7); the advantage of option types is precisely that it does so.

15.4 Exercises

Chapter 16

Pattern Matching

Pattern matching is a natural and convenient generalization of the elimination forms for product and sum types. For example, rather than write

$$\texttt{let}\,x\,\texttt{be}\,e\,\texttt{in}\,x\cdot\texttt{l}+x\cdot\texttt{r}$$

to add the components of a pair, e, of natural numbers, we may instead write

$$\texttt{match}\,e\,\{\langle x_1,x_2\rangle \Rightarrow x_1+x_2\},$$

using pattern matching to name the components of the pair and refer to them directly. The first argument to the `match` expression 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 (as well as any others currently in scope). 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 $\texttt{l}\cdot x$ and $\texttt{r}\cdot x$ 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 $(\texttt{unit}+\texttt{unit})\times\texttt{nat}$, either doubles or squares its second component depending on the form of its first component:

$$\mathtt{match}\,e\,\{\langle \mathtt{l}\cdot\langle\rangle, x\rangle \Rightarrow x+x \mid \langle \mathtt{r}\cdot\langle\rangle, y\rangle \Rightarrow y*y\}. \tag{16.1}$$

It is an instructive exercise to express the same computation using only the primitives for sums and products given in Chapters 14 and 15.

In this chapter we study a simple language, $\mathcal{L}\{pat\}$, of pattern matching over eager product and sum types.

16.1 A Pattern Language

The abstract syntax of $\mathcal{L}\{pat\}$ is defined by the following grammar:

Expr	e	$::=$	$\mathtt{match}(e;rs)$	$\mathtt{match}\,e\,\{rs\}$	case analysis
Rules	rs	$::=$	$\mathtt{rules}[n](r_1;\ldots;r_n)$	$r_1 \mid \ldots \mid r_n$	$(n\ \mathsf{nat})$
Rule	r	$::=$	$\mathtt{rule}[k](p;x_1,\ldots,x_k.e)$	$p \Rightarrow e$	$(k\ \mathsf{nat})$
Pat	p	$::=$	$\mathtt{wild}$	$\_$	wild card
			x	x	variable
			$\mathtt{triv}$	$\langle\rangle$	unit
			$\mathtt{pair}(p_1;p_2)$	$\langle p_1, p_2\rangle$	pair
			$\mathtt{in[l]}(p)$	$\mathtt{l}\cdot p$	left injection
			$\mathtt{in[r]}(p)$	$\mathtt{r}\cdot p$	right injection

The operator `match` has arity $(0,0)$, specifying that it takes two operands, the expression to match and a series of rules. A sequence of rules is constructed using the operatator `rules[`n`]`, which has arity $(0,\ldots,0)$ specifying that it has $n \geq 0$ operands. Each rule is constructed by the operator `rule[`k`]` of arity $(0,k)$ which specifies that it has two operands, binding k variables in the second.

16.2 Statics

The statics of $\mathcal{L}\{pat\}$ makes use of a special form of hypothetical judgement, 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 *at most once* in p. When reading the judgement $\Lambda \Vdash p : \tau$ it is helpful to think of Λ as an *output*, and p and τ as *inputs*. Given p and τ, the rules determine the hypotheses Λ such that $\Lambda \Vdash p : \tau$.

$$\frac{}{\Lambda, x : \tau \Vdash x : \tau} \tag{16.2a}$$

$$\frac{}{\emptyset \Vdash \_ : \tau} \tag{16.2b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle : \texttt{unit}} \tag{16.2c}$$

$$\frac{\Lambda_1 \Vdash p_1 : \tau_1 \quad \Lambda_2 \Vdash p_2 : \tau_2 \quad \mathit{dom}(\Lambda_1) \cap \mathit{dom}(\Lambda_2) = \emptyset}{\Lambda_1\ \Lambda_2 \Vdash \langle p_1, p_2 \rangle : \tau_1 \times \tau_2} \tag{16.2d}$$

$$\frac{\Lambda_1 \Vdash p : \tau_1}{\Lambda_1 \Vdash \texttt{l} \cdot p : \tau_1 + \tau_2} \tag{16.2e}$$

$$\frac{\Lambda_2 \Vdash p : \tau_2}{\Lambda_2 \Vdash \texttt{r} \cdot p : \tau_1 + \tau_2} \tag{16.2f}$$

Rule (16.2a) states that a variable is a pattern of type τ. Rule (16.2d) states that a pair pattern consists of two patterns with disjoint variables.

The typing judgments for a rule,

$$p \Rightarrow e : \tau > \tau',$$

and for a sequence of rules,

$$r_1 \mid \ldots \mid r_n : \tau > \tau',$$

specify that rules transform a value of type τ into a value of type τ'. These judgements are inductively defined as follows:

$$\frac{\Lambda \Vdash p : \tau \quad \Gamma\, \Lambda \vdash e : \tau'}{\Gamma \vdash p \Rightarrow e : \tau > \tau'} \tag{16.3a}$$

$$\frac{\Gamma \vdash r_1 : \tau > \tau' \quad \ldots \quad \Gamma \vdash r_n : \tau > \tau'}{\Gamma \vdash r_1 \mid \ldots \mid r_n : \tau > \tau'} \tag{16.3b}$$

Using the typing judgements for rules, the typing rule for a `match` expression may be stated quite easily:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash rs : \tau > \tau'}{\Gamma \vdash \texttt{match}\, e\, \{rs\} : \tau'} \tag{16.4}$$

16.3 Dynamics

A *substitution*, θ, is a finite mapping from variables to values. If θ is the substitution $\langle x_1 : e_1\rangle \otimes \cdots \otimes \langle x_k : e_k\rangle$, we write $\hat{\theta}(e)$ for $[e_1, \ldots, e_k / x_1, \ldots, x_k]e$. The judgement $\theta : \Lambda$ is inductively defined by the following rules:

$$\frac{}{\sigma : \emptyset} \tag{16.5a}$$

$$\frac{\sigma : \Lambda \quad \sigma(x) = e \quad e : \tau}{\sigma : \Lambda, x : \tau} \tag{16.5b}$$

The judgement $\theta \Vdash p \lhd e$ states that the pattern, p, matches the value, e, as witnessed by the substitution, θ, defined on the variables of p. This judgement is inductively defined by the following rules:

$$\frac{}{\langle x : e\rangle \Vdash x \lhd e} \tag{16.6a}$$

$$\frac{}{\emptyset \Vdash \_ \lhd e} \tag{16.6b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle \lhd \langle\rangle} \tag{16.6c}$$

$$\frac{\theta_1 \Vdash p_1 \lhd e_1 \quad \theta_2 \Vdash p_2 \lhd e_2 \quad dom(\theta_1) \cap dom(\theta_2) = \emptyset}{\theta_1 \otimes \theta_2 \Vdash \langle p_1, p_2\rangle \lhd \langle e_1, e_2\rangle} \tag{16.6d}$$

$$\frac{\theta \Vdash p \lhd e}{\theta \Vdash \mathtt{l} \cdot p \lhd \mathtt{l} \cdot e} \tag{16.6e}$$

$$\frac{\theta \Vdash p \lhd e}{\theta \Vdash \mathtt{r} \cdot p \lhd \mathtt{r} \cdot e} \tag{16.6f}$$

These rules simply collect the bindings for the pattern variables required to form a substitution witnessing the success of the matching process.

The judgement $e \perp p$ states that e does not match the pattern p. It is inductively defined by the following rules:

$$\frac{e_1 \perp p_1}{\langle e_1, e_2\rangle \perp \langle p_1, p_2\rangle} \tag{16.7a}$$

$$\frac{e_2 \perp p_2}{\langle e_1, e_2\rangle \perp \langle p_1, p_2\rangle} \tag{16.7b}$$

$$\frac{}{\mathtt{l} \cdot e \perp \mathtt{r} \cdot p} \tag{16.7c}$$

$$\frac{e \perp p}{\mathtt{l} \cdot e \perp \mathtt{l} \cdot p} \tag{16.7d}$$

$$\frac{}{\texttt{r}\cdot e \perp \texttt{l}\cdot p} \tag{16.7e}$$

$$\frac{e \perp p}{\texttt{r}\cdot e \perp \texttt{r}\cdot p} \tag{16.7f}$$

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 16.1. *Suppose that $e : \tau$, e val, and $\Lambda \Vdash p : \tau$. Then either there exists θ such that $\theta : \Lambda$ and $\theta \Vdash p \lhd e$, or $e \perp p$.*

Proof. By rule induction on Rules (16.2), 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 judgements as follows:

$$\frac{e \mapsto e'}{\texttt{match}\, e\, \{rs\} \mapsto \texttt{match}\, e'\, \{rs\}} \tag{16.8a}$$

$$\frac{e\ \mathsf{val}}{\texttt{match}\, e\, \{\}\ \mathsf{err}} \tag{16.8b}$$

$$\frac{e\ \mathsf{val} \quad \theta \Vdash p_0 \lhd e}{\texttt{match}\, e\, \{p_0 \Rightarrow e_0; rs\} \mapsto \hat{\theta}(e_0)} \tag{16.8c}$$

$$\frac{e\ \mathsf{val} \quad e \perp p_0 \quad \texttt{match}\, e\, \{rs\} \mapsto e'}{\texttt{match}\, e\, \{p_0 \Rightarrow e_0; rs\} \mapsto e'} \tag{16.8d}$$

Rule (16.8b) specifies that evaluation results in a checked error once all rules are exhausted. Rules (16.8c) 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.

Theorem 16.2 (Preservation). *If $e \mapsto e'$ and $e : \tau$, then $e' : \tau$.*

Proof. By a straightforward induction on the derivation of $e \mapsto e'$. □

16.4 Exhaustiveness and Redundancy

While it is possible to state and prove a progress theorem for $\mathcal{L}\{\mathit{pat}\}$ as defined in Section 16.1 on page 132, 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. Since 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.

16.4.1 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 rule satisfies the match constraint of some rule in the sequence. A rule in a sequence is *redundant* if every value that satisfies its match contraint also satisfies the match constraint of some preceding rule.

The language of match constraints is defined by the following grammar:

Constr	ξ	$::=$	$\mathtt{all}[\tau]$	$\top$	truth
			$\mathtt{and}(\xi_1;\xi_2)$	$\xi_1 \wedge \xi_2$	conjunction
			$\mathtt{nothing}[\tau]$	$\perp$	falsity
			$\mathtt{or}(\xi_1;\xi_2)$	$\xi_1 \vee \xi_2$	disjunction
			$\mathtt{in[l]}(\xi_1)$	$\mathtt{l} \cdot \xi_1$	left injection
			$\mathtt{in[r]}(\xi_2)$	$\mathtt{r} \cdot \xi_2$	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 judgement $\xi : \tau$ specifying that the constraint ξ constrains values of type τ.

The *De Morgan Dual*, $\overline{\xi}$, of a match constraint, ξ, is defined by the following rules:

$$\begin{aligned}
\overline{\top} &= \bot \\
\overline{\xi_1 \wedge \xi_2} &= \overline{\xi_1} \vee \overline{\xi_2} \\
\overline{\bot} &= \top \\
\overline{\xi_1 \vee \xi_2} &= \overline{\xi_1} \wedge \overline{\xi_2} \\
\overline{\mathtt{l} \cdot \xi_1} &= \mathtt{l} \cdot \overline{\xi_1} \vee \mathtt{r} \cdot \top \\
\overline{\mathtt{r} \cdot \xi_1} &= \mathtt{r} \cdot \overline{\xi_1} \vee \mathtt{l} \cdot \top \\
\overline{\langle\rangle} &= \bot \\
\overline{\langle \xi_1, \xi_2 \rangle} &= \langle \overline{\xi_1}, \xi_2 \rangle \vee \langle \xi_1, \overline{\xi_2} \rangle \vee \langle \overline{\xi_1}, \overline{\xi_2} \rangle
\end{aligned}$$

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* judgement, $e \models \xi$, is defined for values e and constraints ξ of the same type by the following rules:

$$\frac{}{e \models \top} \tag{16.9a}$$

$$\frac{e \models \xi_1 \quad e \models \xi_2}{e \models \xi_1 \wedge \xi_2} \tag{16.9b}$$

$$\frac{e \models \xi_1}{e \models \xi_1 \vee \xi_2} \tag{16.9c}$$

$$\frac{e \models \xi_2}{e \models \xi_1 \vee \xi_2} \tag{16.9d}$$

$$\frac{e_1 \models \xi_1}{\mathtt{l} \cdot e_1 \models \mathtt{l} \cdot \xi_1} \tag{16.9e}$$

$$\frac{e_2 \models \xi_2}{\mathtt{r} \cdot e_2 \models \mathtt{r} \cdot \xi_2} \tag{16.9f}$$

$$\frac{}{\langle\rangle \models \langle\rangle} \tag{16.9g}$$

$$\frac{e_1 \models \xi_1 \quad e_2 \models \xi_2}{\langle e_1, e_2 \rangle \models \langle \xi_1, \xi_2 \rangle} \tag{16.9h}$$

The De Morgan dual construction negates a constraint.

Lemma 16.3. *If $\xi : \tau$, then, for every value $e : \tau$, $e \models \overline{\xi}$ 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 16.3 on the preceding page 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 $e : \tau$.

16.4.2 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.

The judgement $\Lambda \Vdash p : \tau\ [\xi]$ augments the judgement $\Lambda \Vdash p : \tau$ with a match constraint characterizing the set of values of type τ matched by the pattern p. It is inductively defined by the following rules:

$$\frac{}{x : \tau \Vdash x : \tau\ [\top]} \tag{16.10a}$$

$$\frac{}{\emptyset \Vdash \_ : \tau\ [\top]} \tag{16.10b}$$

$$\frac{}{\emptyset \Vdash \langle\rangle : \texttt{unit}\ [\langle\rangle]} \tag{16.10c}$$

$$\frac{\Lambda_1 \Vdash p : \tau_1\ [\xi_1]}{\Lambda_1 \Vdash \texttt{l} \cdot p : \tau_1 + \tau_2\ [\texttt{l} \cdot \xi_1]} \tag{16.10d}$$

$$\frac{\Lambda_2 \Vdash p : \tau_2\ [\xi_2]}{\Lambda_2 \Vdash \texttt{r} \cdot p : \tau_1 + \tau_2\ [\texttt{r} \cdot \xi_2]} \tag{16.10e}$$

$$\frac{\Lambda_1 \Vdash p_1 : \tau_1\ [\xi_1] \quad \Lambda_2 \Vdash p_2 : \tau_2\ [\xi_2] \quad \Lambda_1 \# \Lambda_2}{\Lambda_1\, \Lambda_2 \Vdash \langle p_1, p_2\rangle : \tau_1 \times \tau_2\ [\langle \xi_1, \xi_2\rangle]} \tag{16.10f}$$

Lemma 16.4. *Suppose that $\Lambda \Vdash p : \tau\ [\xi]$. For every $e : \tau$ such that e val, $e \models \xi$ iff $\theta \Vdash p \lhd e$ for some θ, and $e \not\models \xi$ iff $e \perp p$.*

The judgement $\Gamma \vdash r : \tau > \tau'\ [\xi]$ augments the formation judgement for a rule with a match constraint characterizing the pattern component of the rule. The judgement $\Gamma \vdash rs : \tau > \tau'\ [\xi]$ augments the formation judgement for a sequence of rules with a match constraint characterizing the values matched by some rule in the given rule sequence.

$$\frac{\Lambda \Vdash p : \tau\ [\xi] \quad \Gamma\, \Lambda \vdash e : \tau'}{\Gamma \vdash p \Rightarrow e : \tau > \tau'\ [\xi]} \tag{16.11a}$$

$$\frac{\begin{array}{c}(\forall 1 \leq i \leq n)\ \xi_i \not\models \xi_1 \vee \ldots \vee \xi_{i-1} \\ \Gamma \vdash r_1 : \tau > \tau'\ [\xi_1] \qquad \ldots \qquad \Gamma \vdash r_n : \tau > \tau'\ [\xi_n]\end{array}}{\Gamma \vdash r_1 \mid \ldots \mid r_n : \tau > \tau'\ [\xi_1 \vee \ldots \vee \xi_n]} \tag{16.11b}$$

Rule (16.11b) 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 τ satisfy the constraint associated with some rule.

The typing rule for `match` expressions demands that the rules that comprise it be exhaustive:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash rs : \tau > \tau'\ [\xi] \quad \models \xi}{\Gamma \vdash \mathtt{match}\, e\, \{rs\} : \tau'} \tag{16.12}$$

Rule (16.11b) ensures that ξ is a disjunction of the match constraints associated to the constituent rules of the match expression. The requirement that ξ be valid amounts to requiring that every value of type τ satisfies the constraint of at least one rule of the match.

Theorem 16.5. *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. The exhaustiveness check in Rule (16.12) ensures that if e val and $e : \tau$, then $e \models \xi$. The form of ξ given by Rule (16.11b) ensures that $e \models \xi_i$ for some constraint ξ_i corresponding to the ith rule. By Lemma 16.4 on the preceding page the value e must match the ith rule, which is enough to ensure progress. □

16.4.3 Checking Exhaustiveness and Redundancy

Checking exhaustiveness and redundacy reduces to showing that the constraint validity judgement $\models \xi$ is decidable. We will prove this by defining a judgement Ξ incon, where Ξ 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 Ξ. We will then show that either Ξ incon or not.

The rules defining inconsistency of a finite set, Ξ, of constraints of the same type are as follows:

$$\frac{\Xi\ \mathsf{incon}}{\Xi, \top\ \mathsf{incon}} \tag{16.13a}$$

$$\frac{\Xi, \xi_1, \xi_2\ \mathsf{incon}}{\Xi, \xi_1 \wedge \xi_2\ \mathsf{incon}} \tag{16.13b}$$

$$\frac{}{\Xi, \bot\ \mathsf{incon}} \tag{16.13c}$$

$$\frac{\Xi, \xi_1 \ \mathsf{incon} \quad \Xi, \xi_2 \ \mathsf{incon}}{\Xi, \xi_1 \vee \xi_2 \ \mathsf{incon}} \tag{16.13d}$$

$$\frac{}{\Xi, \mathtt{l} \cdot \xi_1, \mathtt{r} \cdot \xi_2 \ \mathsf{incon}} \tag{16.13e}$$

$$\frac{\Xi \ \mathsf{incon}}{\mathtt{l} \cdot \Xi \ \mathsf{incon}} \tag{16.13f}$$

$$\frac{\Xi \ \mathsf{incon}}{\mathtt{r} \cdot \Xi \ \mathsf{incon}} \tag{16.13g}$$

$$\frac{\Xi_1 \ \mathsf{incon}}{\langle \Xi_1, \Xi_2 \rangle \ \mathsf{incon}} \tag{16.13h}$$

$$\frac{\Xi_2 \ \mathsf{incon}}{\langle \Xi_1, \Xi_2 \rangle \ \mathsf{incon}} \tag{16.13i}$$

In Rule (16.13f) 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 (16.13g), (16.13h), and (16.13i).

Lemma 16.6. *It is decidable whether or not Ξ incon.*

Proof. The premises of each rule involving only constraints that are proper components of the constraints in the conclusion. Consequently, we can simplify Ξ by inverting each of the applicable rules until no rule applies, then determine whether or not the resulting set, Ξ', is contradictory in the sense that it contains $\perp$ or both $\mathtt{l} \cdot \xi$ and $\mathtt{r} \cdot \xi'$ for some ξ and ξ'. □

Lemma 16.7. *Ξ incon iff $\Xi \models \perp$.*

Proof. From left to right we proceed by induction on Rules (16.13). From right to left we may show that if Ξ incon is not derivable, then there exists a value e such that $e \models \Xi$, and hence $\Xi \not\models \perp$. □

16.5 Exercises

Chapter 17

Generic Programming

17.1 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 12 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 $\sigma \rightarrow \sigma'$ that transforms values of type σ into values of type σ'. For example, f might be the doubling function on natural numbers. We wish to extend f to a transformation from type $[\sigma/t]\tau$ to type $[\sigma'/t]\tau$ by applying f to various spots in the input where a value of type σ occurs to obtain a value of type σ', leaving the rest of the data structure alone. For example, τ might be $\texttt{bool} \times \sigma$, in which case f could be extended to a function of type $\texttt{bool} \times \sigma \rightarrow \texttt{bool} \times \sigma'$ 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 $\sigma \rightarrow \sigma'$, it is not obvious in general how to extend it to a function mapping $[\sigma/t]\tau$ to $[\sigma'/t]\tau$. The problem is that it is not clear which of many occurrences of σ in $[\sigma/t]\tau$ are to be transformed by f, even if there is only one occurrence of σ. To avoid ambiguity we need a way to mark which occurrences of σ in $[\sigma/t]\tau$ 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 $f : \sigma \rightarrow \sigma'$, we can extend f unambiguously to a function of type $[\sigma/t]\tau \to [\sigma'/t]\tau$.

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.

17.2 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 $t\ \mathsf{type} \vdash \tau\ \mathsf{type}$. 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, σ, for the variable, t, within the type τ. We sometimes write $\mathtt{Map}[t\,.\,\tau](\sigma)$ for the substitution instance $[\sigma/t]\tau$.

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 $\tau_1 + \tau_2$. It is a straightforward exercise to give inductive definitions of the judgement $t\,.\,\tau\ \mathsf{poly}$ stating that the operator $t\,.\,\tau$ is a polynomial type operator.

17.3 Generic Extension

The *generic extension* primitive has the form

$$\mathtt{map}[t\,.\,\tau](x\,.\,e';e)$$

with statics given by the following rule:

$$\frac{t\ \mathsf{type} \vdash \tau\ \mathsf{type} \quad \Gamma, x : \sigma \vdash e' : \sigma' \quad \Gamma \vdash e : [\sigma/t]\tau}{\Gamma \vdash \mathtt{map}[t\,.\,\tau](x\,.\,e';e) : [\sigma'/t]\tau} \tag{17.1}$$

The abstractor $x\,.\,e'$ specifies a transformation from type σ, the type of x, to type σ', the type of e'. The expression e of type $[\sigma/t]\tau$ determines the value to be transformed to obtain a value of type $[\sigma'/t]\tau$. The occurrences of t in τ determine the spots at which the transformation given by $x\,.\,e$ is to be performed.

The dynamics of generic extension is specified by the following rules. We consider here only polynomial type operators, leaving the extension to positive type operators to be considered later.

$$\frac{}{\texttt{map[}t\,.\,t\texttt{](}x\,.\,e'\texttt{;}e\texttt{)} \mapsto [e/x]e'} \tag{17.2a}$$

$$\frac{}{\texttt{map[}t\,.\,\texttt{unit](}x\,.\,e'\texttt{;}e\texttt{)} \mapsto \langle\rangle} \tag{17.2b}$$

$$\frac{}{\begin{array}{c}\texttt{map[}t\,.\,\tau_1 \times \tau_2\texttt{](}x\,.\,e'\texttt{;}e\texttt{)}\\ \mapsto \\ \langle\texttt{map[}t\,.\,\tau_1\texttt{](}x\,.\,e'\texttt{;}e\cdot\texttt{l),map[}t\,.\,\tau_2\texttt{](}x\,.\,e'\texttt{;}e\cdot\texttt{r)}\rangle\end{array}} \tag{17.2c}$$

$$\frac{}{\texttt{map[}t\,.\,\texttt{void](}x\,.\,e'\texttt{;}e\texttt{)} \mapsto \texttt{abort(}e\texttt{)}} \tag{17.2d}$$

$$\frac{}{\begin{array}{c}\texttt{map[}t\,.\,\tau_1 + \tau_2\texttt{](}x\,.\,e'\texttt{;}e\texttt{)}\\ \mapsto \\ \texttt{case}\,e\,\{\texttt{l}\cdot x_1 \Rightarrow \texttt{l}\cdot\texttt{map[}t\,.\,\tau_1\texttt{](}x\,.\,e'\texttt{;}x_1\texttt{)} \mid \texttt{r}\cdot x_2 \Rightarrow \texttt{r}\cdot\texttt{map[}t\,.\,\tau_2\texttt{](}x\,.\,e'\texttt{;}x_2\texttt{)}\}\end{array}} \tag{17.2e}$$

Rule (17.2a) applies the transformation $x\,.\,e'$ to e itself, since the operator $t\,.\,t$ specifies that the transformation is to be perfomed directly. Rule (17.2b) states that the empty tuple is transformed to itself. Rule (17.2c) 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 (17.2d) states that the transformation of a value of type `void` aborts, since there can be no such values. Rule (17.2e) 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 $t\,.\,\texttt{unit} + (\texttt{bool} \times t)$. Let $x\,.\,e$ be the abstractor $x\,.\,\texttt{s(}x\texttt{)}$, which increments a natural number. Using Rules (17.2) we may derive that

$$\texttt{map[}t\,.\,\tau\texttt{](}x\,.\,e\texttt{;r}\cdot\langle\texttt{tt},n\rangle\texttt{)} \mapsto^* \texttt{r}\cdot\langle\texttt{tt},n+1\rangle.$$

The natural number in the second component of the pair is incremented, since the type variable, t, occurs in that position in the type operator $t\,.\,\tau$.

Theorem 17.1 (Preservation)**.** *If* $\mathtt{map}[t\,.\,\tau](x\,.\,e';e) : \rho$ *and* $\mathtt{map}[t\,.\,\tau](x\,.\,e';e) \mapsto e''$*, then* $e'' : \rho$*.*

Proof. By inversion of Rule (17.1) we have

1. t type $\vdash \tau$ type;
2. $x : \sigma \vdash e' : \sigma'$ for some σ and σ';
3. $e : [\sigma/t]\tau$;
4. ρ is $[\sigma'/t]\tau$.

We proceed by cases on Rules (17.2). For example, consider Rule (17.2c). It follows from inversion that $\mathtt{map}[t\,.\,\tau_1](x\,.\,e';e\cdot\mathtt{l}) : [\sigma'/t]\tau_1$, and similarly that $\mathtt{map}[t\,.\,\tau_2](x\,.\,e';e\cdot\mathtt{r}) : [\sigma'/t]\tau_2$. 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 $[\sigma'/t]\tau_1 \times \tau_2$, as required. □

The *positive* type operators extend the polynomial type operators to admit restricted forms of function type. Specifically, $t\,.\,\tau_1 \rightarrow \tau_2$ is a positive type operator, provided that (1) t *does not occur* in τ_1, and (2) $t\,.\,\tau_2$ is a positive type operator. In general, any occurrences of a type variable t in the domain 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.

The generic extension according to a positive type operator is defined similarly to the case of a polynomial type operator, with the following additional rule:

$$\frac{}{\mathtt{map}[t\,.\,\tau_1 \rightarrow \tau_2](x\,.\,e';e) \mapsto \lambda\,(x_1\!:\!\tau_1.\,\mathtt{map}[t\,.\,\tau_2](x\,.\,e';e(x_1)))} \tag{17.3}$$

[1] The origin of this terminology appears to be that a function type $\tau_1 \rightarrow \tau_2$ is, by the propositions-as-types principle, analogous to the implication $\phi_1 \supset \phi_2$, which is classically equivalent to $\neg\phi_1 \vee \phi_2$, placing occurrences in the domain beneath the negation sign.

Since t is not permitted to occur within the domain type, the type of the result is $\tau_1 \rightarrow [\sigma'/t]\tau_2$, assuming that e is of type $\tau_1 \rightarrow [\sigma/t]\tau_2$. It is easy to verify preservation for the generic extension of a positive type operator.

It is interesting to consider what goes wrong if we relax the restriction on positive type operators to admit negative, as well as positive, occurrences of the parameter of a type operator. Consider the type operator $t\,.\,\tau_1 \rightarrow \tau_2$, without restriction on t, and suppose that $x : \sigma \vdash e' : \sigma'$. The generic extension $\mathtt{map}[t\,.\,\tau_1 \rightarrow \tau_2](x\,.\,e';e)$ should have type $[\sigma'/t]\tau_1 \rightarrow [\sigma'/t]\tau_2$, given that e has type $[\sigma/t]\tau_1 \rightarrow [\sigma/t]\tau_2$. The extension should yield a function of the form

$$\lambda\,(x_1 : [\sigma'/t]\tau_1 . \ldots (e(\ldots(x_1))))$$

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 $[\sigma/t]\tau_1$ into values of type $[\sigma'/t]\tau_1$, but *we need to go the other way around* in order to make x_1 suitable as an argument for e. But there is no obvious way to obtain the required transformation.

One solution to this is to assume that the fundamental transformation $x\,.\,e'$ is *invertible* so that we may apply the inverse transformation on x_1 to get an argument of type suitable for e, then apply the forward transformation on the result, just as in the positive case. Since we cannot invert an arbitrary transformation, we must instead pass both the transformation and its inverse to the generic extension operation so that it can "go backwards" as necessary to cover negative occurrences of the type parameter. So in the general case the generic extension applies only when we are given a *type isomorphism* (a pair of mutually inverse mappings between two types), and then results in another isomorphism pair. We leave the formulation of this as an exercise for the reader.

17.4 Exercises

Part VI

Infinite Data Types

Chapter 18

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 isomorphism 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 that element as a value of that type. Such an element is called an *generator*, or *anamorphism*.

18.1 Motivating Examples

The most important example of an inductive type is the type of natural numbers as formalized in Chapter 12. The type `nat` is defined to be the *least* type containing `z` and closed under `s(−)`. The minimality condition is witnessed by the existence of the recursor, $\texttt{natiter}\, e\, \{\texttt{z}\Rightarrow e_0 \mid \texttt{s}(x)\Rightarrow e_1\}$, which transforms a natural number into a value of type τ, 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. Put the other way around, the existence of this operation expresses the inductive nature of the type nat.

With a view towards deriving the type 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 recursor. This following rules specify the statics of this reformulation:

$$\frac{\Gamma \vdash e : \texttt{unit} + \texttt{nat}}{\Gamma \vdash \texttt{fold}_{\texttt{nat}}\,(e) : \texttt{nat}} \tag{18.1a}$$

$$\frac{\Gamma, x : \texttt{unit} + \tau \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{rec}_{\texttt{nat}}\,[x\,.\,e_1]\,(e_2) : \tau} \tag{18.1b}$$

The expression $\texttt{fold}_{\texttt{nat}}\,(e)$ is the unique introductory form of the type nat. Using this, the expression z is defined to be $\texttt{fold}_{\texttt{nat}}\,(\texttt{l} \cdot \langle\rangle)$, and $\texttt{s}\,(e)$ is defined to be $\texttt{fold}_{\texttt{nat}}\,(\texttt{r} \cdot e)$. The recursor, $\texttt{rec}_{\texttt{nat}}\,[x\,.\,e_1]\,(e_2)$, takes as argument the abstractor $x\,.\,e_1$ that consolidates the basis and inductive step into a single computation that is given a value of type $\texttt{unit} + \tau$ yields a value of type τ. Intuitively, if x is replaced by the value $\texttt{l} \cdot \langle\rangle$, then e_1 computes the base case of the recursion, and if x is replaced by the value $\texttt{r} \cdot e$, 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:

$$\frac{}{\texttt{fold}_{\texttt{nat}}\,(e)\ \mathsf{val}} \tag{18.2a}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{rec}_{\texttt{nat}}\,[x\,.\,e_1]\,(e_2) \mapsto \texttt{rec}_{\texttt{nat}}\,[x\,.\,e_1]\,(e_2')} \tag{18.2b}$$

$$\frac{}{\begin{array}{c}\texttt{rec}_{\texttt{nat}}\,[x\,.\,e_1]\,(\texttt{fold}_{\texttt{nat}}\,(e_2)) \\ \mapsto \\ [\texttt{map}\,[t\,.\,\texttt{unit} + t]\,(y\,.\,\texttt{rec}_{\texttt{nat}}\,[x\,.\,e_1]\,(y)\,;e_2)\,/\,x]e_1\end{array}} \tag{18.2c}$$

Rule (18.2c) makes use of generic extension (see Chapter 8) to apply the recursor to the predecessor, if any, of a natural number. The idea is that the result of extending the recursor from the type $\texttt{unit} + \texttt{nat}$ to the type $\texttt{unit} + \tau$ is substituted into the inductive step, given by the expression e_1. If we expand the definition of the generic extension in place, we obtain the following reformulation of this rule:

$$\frac{}{\begin{array}{c}\mathtt{rec}_{\mathtt{nat}}\,\mathtt{[}x\mathtt{.}e_1\mathtt{]}\,\mathtt{(}\mathtt{fold}_{\mathtt{nat}}\,\mathtt{(}e_2\mathtt{))}\\ \mapsto \\ [\mathtt{case}\,e_2\,\{\mathtt{l}\cdot\_ \Rightarrow \mathtt{l}\cdot\langle\rangle \mid \mathtt{r}\cdot y \Rightarrow \mathtt{r}\cdot\mathtt{rec}_{\mathtt{nat}}\,\mathtt{[}x\mathtt{.}e_1\mathtt{]}\,\mathtt{(}y\mathtt{)}\}/x]e_1\end{array}}$$

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. This characteristic of the introductory form for streams is *dual* to the analogous property of the eliminatory form for natural numbers whereby the result for a number is determined by its result for all preceding numbers.

A stream is characterized by its behavior under the elimination forms for the stream type: $\mathtt{hd}\,\mathtt{(}e\mathtt{)}$ returns the next, or head, element of the stream, and $\mathtt{tl}\,\mathtt{(}e\mathtt{)}$ 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 a 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 following rules:

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{hd}\,\mathtt{(}e\mathtt{)} : \mathtt{nat}} \tag{18.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{tl}\,\mathtt{(}e\mathtt{)} : \mathtt{stream}} \tag{18.3b}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_1 : \mathtt{nat} \quad \Gamma, x : \tau \vdash e_2 : \tau}{\Gamma \vdash \mathtt{strgen}\,e\,\mathtt{<hd}\,\mathtt{(}x\mathtt{)} \Rightarrow e_1\,\mathtt{\&}\,\mathtt{tl}\,\mathtt{(}x\mathtt{)} \Rightarrow e_2\mathtt{>} : \mathtt{stream}} \tag{18.3c}$$

In Rule (18.3c) the current state of the stream is given by the expression e of some type τ, 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 dynamics of streams is given by the following rules:

$$\frac{}{\mathtt{strgen}\,e\,\mathtt{<hd}\,\mathtt{(}x\mathtt{)} \Rightarrow e_1\,\mathtt{\&}\,\mathtt{tl}\,\mathtt{(}x\mathtt{)} \Rightarrow e_2\mathtt{>}\ \mathsf{val}} \tag{18.4a}$$

$$\frac{e \mapsto e'}{\mathtt{hd}\,\mathtt{(}e\mathtt{)} \mapsto \mathtt{hd}\,\mathtt{(}e'\mathtt{)}} \tag{18.4b}$$

$$\frac{}{\texttt{hd(strgen}\,e\,\texttt{<hd(}x\texttt{)} \Rightarrow e_1\,\texttt{\&}\,\texttt{tl(}x\texttt{)} \Rightarrow e_2\texttt{>)} \mapsto [e/x]e_1} \tag{18.4c}$$

$$\frac{e \mapsto e'}{\texttt{tl(}e\texttt{)} \mapsto \texttt{tl(}e'\texttt{)}} \tag{18.4d}$$

$$\frac{}{\begin{array}{c}\texttt{tl(strgen}\,e\,\texttt{<hd(}x\texttt{)} \Rightarrow e_1\,\texttt{\&}\,\texttt{tl(}x\texttt{)} \Rightarrow e_2\texttt{>)}\\ \mapsto \\ \texttt{strgen}\,[e/x]e_2\,\texttt{<hd(}x\texttt{)} \Rightarrow e_1\,\texttt{\&}\,\texttt{tl(}x\texttt{)} \Rightarrow e_2\texttt{>}\end{array}} \tag{18.4e}$$

Rules (18.4c) and (18.4e) 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:

$$\frac{\Gamma \vdash e : \texttt{stream}}{\Gamma \vdash \texttt{unfold}_{\texttt{stream}}\texttt{(}e\texttt{)} : \texttt{nat} \times \texttt{stream}} \tag{18.5a}$$

$$\frac{\Gamma, x : \tau \vdash e_1 : \texttt{nat} \times \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{gen}_{\texttt{stream}}\texttt{[}x\texttt{.}e_1\texttt{]}\texttt{(}e_2\texttt{)} : \texttt{stream}} \tag{18.5b}$$

Rule (18.5a) states that a stream may be unfolded into a pair consisting of its head, a natural number, and its tail, another stream. The head, $\texttt{hd(}e\texttt{)}$, and tail, $\texttt{tl(}e\texttt{)}$, of a stream, e, are defined to be the projections $\texttt{unfold}_{\texttt{stream}}\texttt{(}e\texttt{)} \cdot \texttt{l}$ and $\texttt{unfold}_{\texttt{stream}}\texttt{(}e\texttt{)} \cdot \texttt{r}$, respectively. Rule (18.5b) states 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:

$$\frac{}{\texttt{gen}_{\texttt{stream}}\texttt{[}x\texttt{.}e_1\texttt{]}\texttt{(}e_2\texttt{)}\ \mathsf{val}} \tag{18.6a}$$

$$\frac{e \mapsto e'}{\texttt{unfold}_{\texttt{stream}}\texttt{(}e\texttt{)} \mapsto \texttt{unfold}_{\texttt{stream}}\texttt{(}e'\texttt{)}} \tag{18.6b}$$

$$\frac{}{\begin{array}{c}\texttt{unfold}_{\texttt{stream}}\texttt{(gen}_{\texttt{stream}}\texttt{[}x\texttt{.}e_1\texttt{]}\texttt{(}e_2\texttt{))}\\ \mapsto \\ \texttt{map[}t\texttt{.nat} \times t\texttt{]}\texttt{(}y\texttt{.gen}_{\texttt{stream}}\texttt{[}x\texttt{.}e_1\texttt{]}\texttt{(}y\texttt{)};[e_2/x]e_1\texttt{)}\end{array}} \tag{18.6c}$$

Rule (18.6c) 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:

$$\frac{}{\begin{array}{c}\texttt{unfold}_{\texttt{stream}}(\texttt{gen}_{\texttt{stream}}[x.e_1](e_2))\\ \mapsto \\ \langle([e_2/x]e_1)\cdot\texttt{l},\texttt{gen}_{\texttt{stream}}[x.e_1](([e_2/x]e_1)\cdot\texttt{r})\rangle\end{array}}$$

18.2 Statics

We may now give a fully general account of inductive and coinductive types, which are defined in terms of positive type operators. We will consider the language $\mathcal{L}\{\mu_i\mu_f\}$, which extends $\mathcal{L}\{\rightarrow\times+\}$ with inductive and co-inductive types.

18.2.1 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:

Type	τ	::=	t	t	self-reference
			$\texttt{ind}(t.\tau)$	$\mu_i(t.\tau)$	inductive
			$\texttt{coi}(t.\tau)$	$\mu_f(t.\tau)$	coinductive

Type formation judgements have the form

$$t_1\ \mathsf{type},\ldots,t_n\ \mathsf{type}\vdash\tau\ \mathsf{type},$$

where $t_1,\ldots,t_n$ are type names. We let Δ range over finite sets of hypotheses of the form t type, where t name is a type name. The type formation judgement is inductively defined by the following rules:

$$\frac{}{\Delta,t\ \mathsf{type}\vdash t\ \mathsf{type}}\tag{18.7a}$$

$$\frac{}{\Delta\vdash\texttt{unit}\ \mathsf{type}}\tag{18.7b}$$

$$\frac{\Delta\vdash\tau_1\ \mathsf{type}\quad\Delta\vdash\tau_2\ \mathsf{type}}{\Delta\vdash\texttt{prod}(\tau_1;\tau_2)\ \mathsf{type}}\tag{18.7c}$$

$$\frac{}{\Delta\vdash\texttt{void}\ \mathsf{type}}\tag{18.7d}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \mathtt{sum}(\tau_1;\tau_2)\ \mathsf{type}} \tag{18.7e}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2)\ \mathsf{type}} \tag{18.7f}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \quad \Delta \vdash t\,.\,\tau\ \mathsf{pos}}{\Delta \vdash \mathtt{ind}(t\,.\,\tau)\ \mathsf{type}} \tag{18.7g}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \quad \Delta \vdash t\,.\,\tau\ \mathsf{pos}}{\Delta \vdash \mathtt{coi}(t\,.\,\tau)\ \mathsf{type}} \tag{18.8}$$

18.2.2 Expressions

The abstract syntax of expressions for inductive and coinductive types is given by the following grammar:

Expr	e	$::=$	$\mathtt{fold}[t\,.\,\tau](e)$	$\mathtt{fold}(e)$	constructor
			$\mathtt{rec}[t\,.\,\tau][x\,.\,e_1](e_2)$	$\mathtt{rec}[x\,.\,e_1](e_2)$	recursor
			$\mathtt{unfold}[t\,.\,\tau](e)$	$\mathtt{unfold}(e)$	destructor
			$\mathtt{gen}[t\,.\,\tau][x\,.\,e_1](e_2)$	$\mathtt{gen}[x\,.\,e_1](e_2)$	generator

The statics for inductive and coinductive types is given by the following typing rules:

$$\frac{\Gamma \vdash e : [\mathtt{ind}(t\,.\,\tau)/t]\tau}{\Gamma \vdash \mathtt{fold}[t\,.\,\tau](e) : \mathtt{ind}(t\,.\,\tau)} \tag{18.9a}$$

$$\frac{\Gamma, x : [\rho/t]\tau \vdash e_1 : \rho \quad \Gamma \vdash e_2 : \mathtt{ind}(t\,.\,\tau)}{\Gamma \vdash \mathtt{rec}[t\,.\,\tau][x\,.\,e_1](e_2) : \rho} \tag{18.9b}$$

$$\frac{\Gamma \vdash e : \mathtt{coi}(t\,.\,\tau)}{\Gamma \vdash \mathtt{unfold}[t\,.\,\tau](e) : [\mathtt{coi}(t\,.\,\tau)/t]\tau} \tag{18.9c}$$

$$\frac{\Gamma \vdash e_2 : \rho \quad \Gamma, x : \rho \vdash e_1 : [\rho/t]\tau}{\Gamma \vdash \mathtt{gen}[t\,.\,\tau][x\,.\,e_1](e_2) : \mathtt{coi}(t\,.\,\tau)} \tag{18.9d}$$

18.3 Dynamics

The dynamics of these constructs is given in terms of the generic extension operation described in Chapter 17. The following rules specify a lazy dynamics for $\mathcal{L}\{\mu_i\mu_f\}$:

$$\frac{}{\mathtt{fold}(e)\ \mathsf{val}} \tag{18.10a}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{rec}[x.e_1](e_2) \mapsto \texttt{rec}[x.e_1](e_2')} \tag{18.10b}$$

$$\frac{}{\begin{array}{c}\texttt{rec}[x.e_1](\texttt{fold}(e_2))\\ \mapsto \\ [\texttt{map}[t.\tau](y.\texttt{rec}[x.e_1](y);e_2)/x]e_1\end{array}} \tag{18.10c}$$

$$\frac{}{\texttt{gen}[x.e_1](e_2)\ \mathsf{val}} \tag{18.10d}$$

$$\frac{e \mapsto e'}{\texttt{unfold}(e) \mapsto \texttt{unfold}(e')} \tag{18.10e}$$

$$\frac{}{\begin{array}{c}\texttt{unfold}(\texttt{gen}[x.e_1](e_2))\\ \mapsto \\ \texttt{map}[t.\tau](y.\texttt{gen}[x.e_1](y);[e_2/x]e_1)\end{array}} \tag{18.10f}$$

Rule (18.10c) 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 (18.10f) is simply the dual of this rule for coinductive types.

Lemma 18.1. *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. By rule induction on Rules (18.10). □

Lemma 18.2. *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By rule induction on Rules (18.9). □

18.4 Exercises

Chapter 19

Recursive Types

Inductive and coinductive types, such as natural numbers and streams, may be seen as examples of *fixed points* of type operators *up to isomorphism*. An isomorphism between two types, τ_1 and τ_2, is given by two expressions

1. $x_1 : \tau_1 \vdash e_2 : \tau_2$, and
2. $x_2 : \tau_2 \vdash e_1 : \tau_1$

that are mutually inverse to each other.[1] For example, the types $\mathtt{nat}$ and $\mathtt{unit} + \mathtt{nat}$ are isomorphic, as witnessed by the following two expressions:

1. $x : \mathtt{unit} + \mathtt{nat} \vdash \mathtt{case}\, x \,\{\mathtt{l} \cdot \_ \Rightarrow \mathtt{z} \mid \mathtt{r} \cdot x_2 \Rightarrow \mathtt{s}(x_2)\} : \mathtt{nat}$, and
2. $x : \mathtt{nat} \vdash \mathtt{ifz}\, x \,\{\mathtt{z} \Rightarrow \mathtt{l} \cdot \langle\rangle \mid \mathtt{s}(x_2) \Rightarrow \mathtt{r} \cdot x_2\} : \mathtt{unit} + \mathtt{nat}$.

These are called, respectively, the $\mathtt{fold}$ and $\mathtt{unfold}$ operations of the isomorphism $\mathtt{nat} \cong \mathtt{unit} + \mathtt{nat}$. Thinking of $\mathtt{unit} + \mathtt{nat}$ as $[\mathtt{nat}/t](\mathtt{unit} + t)$, this means that $\mathtt{nat}$ is a *fixed point* of the type operator $t\,.\,\mathtt{unit} + t$.

In this chapter we study the language $\mathcal{L}\{+\times\rightharpoonup\mu\}$, which provides solutions to all type isomorphism equations. The *recursive type* $\mu t\,.\,\tau$ is defined to be a solution to the type isomorphism

$$\mu t\,.\,\tau \cong [\mu t\,.\,\tau/t]\tau.$$

This is witnessed by the operations

$$x : \mu t\,.\,\tau \vdash \mathtt{unfold}(x) : [\mu t\,.\,\tau/t]\tau$$

[1] To make this precise requires a discussion of equivalence of expressions to be taken up in Chapter 51. For now we will rely on an intuitive understanding of when two expressions are equivalent.

and

$$x : [\mu t\,.\,\tau/t]\tau \vdash \texttt{fold}(x) : \mu t\,.\,\tau,$$

which are mutually inverse to each other.

Requiring solutions to all type equations may seem suspicious, since we know by Cantor's Theorem that an isomorphisms such as $X \cong (X \to \mathbf{2})$ is impossible. This negative result tells us not that our requirement is untenable, but rather that *types are not sets*. To permit solution of arbitrary type equations, we must take into account that types describe computations, some of which may not even terminate. Consequently, the function space does not coincide with the set-theoretic function space, but rather is analogous to it (in a precise sense that we shall not go into here).

19.1 Solving Type Isomorphisms

The *recursive type* $\mu t\,.\,\tau$, where $t\,.\,\tau$ is a type operator, represents a solution for t to the isomorphism $t \cong \tau$. The solution is witnessed by two operations, $\texttt{fold}(e)$ and $\texttt{unfold}(e)$, that relate the recursive type $\mu t\,.\,\tau$ to its unfolding, $[\mu t\,.\,\tau/t]\tau$, and serve, respectively, as its introduction and elimination forms.

The language $\mathcal{L}\{{+}{\times}{\rightharpoonup}\mu\}$ extends $\mathcal{L}\{\rightharpoonup\}$ with recursive types and their associated operations.

Type	τ	$::=$	t	t	self-reference
			$\texttt{rec}(t\,.\,\tau)$	$\mu t\,.\,\tau$	recursive
Expr	e	$::=$	$\texttt{fold}[t\,.\,\tau](e)$	$\texttt{fold}(e)$	constructor
			$\texttt{unfold}(e)$	$\texttt{unfold}(e)$	destructor

The statics of $\mathcal{L}\{{+}{\times}{\rightharpoonup}\mu\}$ consists of two forms of judgement. The first, called *type formation*, is a general hypothetical judgement of the form

$$\Delta \vdash \tau \ \mathsf{type},$$

where Δ has the form $t_1 \ \mathsf{type}, \ldots, t_k \ \mathsf{type}$. Type formation is inductively defined by the following rules:

$$\frac{}{\Delta, t \ \mathsf{type} \vdash t \ \mathsf{type}} \tag{19.1a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type} \quad \Delta \vdash \tau_2 \ \mathsf{type}}{\Delta \vdash \texttt{arr}(\tau_1;\tau_2) \ \mathsf{type}} \tag{19.1b}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathtt{rec}(t\,.\,\tau)\ \mathsf{type}} \tag{19.1c}$$

The second form of judgement comprising the statics is the *typing judgement*, which is a hypothetical judgement of the form

$$\Gamma \vdash e : \tau,$$

where we assume that τ type. Typing for $\mathcal{L}\{{+}{\times}{\rightharpoonup}\mu\}$ is inductively defined by the following rules:

$$\frac{\Gamma \vdash e : [\mathtt{rec}(t\,.\,\tau)/t]\tau}{\Gamma \vdash \mathtt{fold}[t\,.\,\tau](e) : \mathtt{rec}(t\,.\,\tau)} \tag{19.2a}$$

$$\frac{\Gamma \vdash e : \mathtt{rec}(t\,.\,\tau)}{\Gamma \vdash \mathtt{unfold}(e) : [\mathtt{rec}(t\,.\,\tau)/t]\tau} \tag{19.2b}$$

The dynamics of $\mathcal{L}\{{+}{\times}{\rightharpoonup}\mu\}$ is specified by one axiom stating that the elimination form is inverse to the introduction form.

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{fold}[t\,.\,\tau](e)\ \mathsf{val}} \tag{19.3a}$$

$$\left\{\frac{e \mapsto e'}{\mathtt{fold}[t\,.\,\tau](e) \mapsto \mathtt{fold}[t\,.\,\tau](e')}\right\} \tag{19.3b}$$

$$\frac{e \mapsto e'}{\mathtt{unfold}(e) \mapsto \mathtt{unfold}(e')} \tag{19.3c}$$

$$\frac{\mathtt{fold}[t\,.\,\tau](e)\ \mathsf{val}}{\mathtt{unfold}(\mathtt{fold}[t\,.\,\tau](e)) \mapsto e} \tag{19.3d}$$

The bracketed premise and rule are to be included for an *eager* interpretation of the introduction form, and omitted for a *lazy* interpretation.

It is a straightforward exercise to prove type safety for $\mathcal{L}\{{+}{\times}{\rightharpoonup}\mu\}$.

Theorem 19.1 (Safety).

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

19.2 Recursive Data Structures

One important application of recursive types is to the representation of inductive data types such as the type of natural numbers. We may think of the type `nat` as a solution (up to isomorphism) of the type equation

$$\mathtt{nat} \cong [\mathtt{z} : \mathtt{unit}, \mathtt{s} : \mathtt{nat}]$$

According to this isomorphism every natural number is either zero or the successor of another natural number. A solution is given by the recursive type

$$\mu t\,.\,[\mathtt{z}:\mathtt{unit},\mathtt{s}:t]. \tag{19.4}$$

The introductory forms for the type `nat` are defined by the following equations:

$$\begin{aligned}\mathtt{z} &= \mathtt{fold}(\mathtt{z}\cdot\langle\rangle)\\ \mathtt{s}(e) &= \mathtt{fold}(\mathtt{s}\cdot e).\end{aligned}$$

The conditional branch may then be defined as follows:

$$\mathtt{ifz}\,e\,\{\mathtt{z}\Rightarrow e_0\mid\mathtt{s}(x)\Rightarrow e_1\} \quad=\quad \mathtt{case}\,\mathtt{unfold}(e)\,\{\mathtt{z}\cdot\_\Rightarrow e_0\mid\mathtt{s}\cdot x\Rightarrow e_1\},$$

where the “underscore” indicates a variable that does not occur free in e_0. It is easy to check that these definitions exhibit the expected behavior.

As another example, the type `list` of lists of natural numbers may be represented by the recursive type

$$\mu t\,.\,[\mathtt{n}:\mathtt{unit},\mathtt{c}:\mathtt{nat}\times t]$$

so that we have the isomorphism

$$\mathtt{list}\cong[\mathtt{n}:\mathtt{unit},\mathtt{c}:\mathtt{nat}\times\mathtt{list}].$$

The list formation operations are represented by the following equations:

$$\begin{aligned}\mathtt{nil} &= \mathtt{fold}(\mathtt{n}\cdot\langle\rangle)\\ \mathtt{cons}(e_1;e_2) &= \mathtt{fold}(\mathtt{c}\cdot\langle e_1,e_2\rangle).\end{aligned}$$

A conditional branch on the form of the list may be defined by the following equation:

$$\begin{aligned}&\mathtt{listcase}\,e\,\{\mathtt{nil}\Rightarrow e_0\mid\mathtt{cons}(x;y)\Rightarrow e_1\} =\\ &\qquad\qquad\mathtt{case}\,\mathtt{unfold}(e)\,\{\mathtt{n}\cdot\_\Rightarrow e_0\mid\mathtt{c}\cdot\langle x,y\rangle\Rightarrow e_1\},\end{aligned}$$

where we have used an underscore for a “don’t care” variable, and used pattern-matching syntax to bind the components of a pair.

As long as sums and products are evaluated eagerly, there is a natural correspondence between this representation of lists and the conventional “blackboard notation” for linked lists. We may think of `fold` as an abstract heap-allocated 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 a natural number and another list, which must be an abstract pointer of the same sort. If sums or products are evaluated lazily, then the blackboard notation breaks down because it is unable to depict the suspended computations that are present in the data structure. In general there is no substitute for the type itself. Drawings can be helpful, but the type determines the semantics.

We may also represent coinductive types, such as the type of streams of natural numbers, using recursive types. The representation is particularly natural in the case that $\texttt{fold}(-)$ is evaluated lazily, for then we may define the type `stream` to be the recursive type

$$\mu t.\texttt{nat} \times t.$$

This states that every stream may be thought of as a computation of a pair consisting of a number and another stream. If $\texttt{fold}(-)$ is evaluated eagerly, then we may instead consider the recursive type

$$\mu t.\texttt{unit} \to (\texttt{nat} \times t),$$

which expresses the same representation of streams. In either case streams cannot be easily depicted in blackboard notation, not so much because they are infinite, but because there is no accurate way to depict the delayed computation other than by an expression in the programming language. Here again we see that pictures can be helpful, but are not adequate for accurately defining a data structure.

19.3 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

$$\texttt{fix}[\tau](x.e) \mapsto [\texttt{fix}[\tau](x.e)/x]e,$$

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, which is to be thought of as a function of x that it implicitly implied to the recursive expression itself whenever it is used. In many well-known languages this implicit argument has a special name, such as `this` or `self`, that emphasizes 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 an *ad hoc* language feature. The derivation is based on isolating a type of self-referential expressions of type τ, written $\texttt{self(}\tau\texttt{)}$. The introduction form of this type is (a variant of) general recursion, written $\texttt{self[}\tau\texttt{](}x\texttt{.}e\texttt{)}$, and the elimination form is an operation to unroll the recursion by one step, written $\texttt{unroll(}e\texttt{)}$. The statics of these constructs is given by the following rules:

$$\frac{\Gamma, x : \texttt{self(}\tau\texttt{)} \vdash e : \tau}{\Gamma \vdash \texttt{self[}\tau\texttt{](}x\texttt{.}e\texttt{)} : \texttt{self(}\tau\texttt{)}} \tag{19.5a}$$

$$\frac{\Gamma \vdash e : \texttt{self(}\tau\texttt{)}}{\Gamma \vdash \texttt{unroll(}e\texttt{)} : \tau} \tag{19.5b}$$

The dynamics is given by the following rule for unrolling the self-reference:

$$\frac{}{\texttt{self[}\tau\texttt{](}x\texttt{.}e\texttt{)}\ \mathsf{val}} \tag{19.6a}$$

$$\frac{e \mapsto e'}{\texttt{unroll(}e\texttt{)} \mapsto \texttt{unroll(}e'\texttt{)}} \tag{19.6b}$$

$$\frac{}{\texttt{unroll(self[}\tau\texttt{](}x\texttt{.}e\texttt{))} \mapsto [\texttt{self[}\tau\texttt{](}x\texttt{.}e\texttt{)}/x]e} \tag{19.6c}$$

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 $\texttt{self(}\tau\texttt{)}$ is definable from recursive types. As suggested earlier, the key is to consider a self-referential expression of type τ to be a function of the expression itself. That is, we seek to define the type $\texttt{self(}\tau\texttt{)}$ so that it satisfies the isomorphism

$$\texttt{self(}\tau\texttt{)} \cong \texttt{self(}\tau\texttt{)} \rightarrow \tau.$$

This means that we seek a fixed point of the type operator $t\texttt{.}t \rightarrow \tau$, where $t \notin \tau$ is a type variable standing for the type in question. The required fixed point is just the recursive type

$$\texttt{rec(}t\texttt{.}t \rightarrow \tau\texttt{)},$$

which we take as the definition of $\texttt{self(}\tau\texttt{)}$.

The self-referential expression $\texttt{self[}\tau\texttt{](}x\texttt{.}e\texttt{)}$ is then defined to be the expression

$$\texttt{fold(}\lambda\texttt{ (}x\texttt{:self(}\tau\texttt{).}e\texttt{)).}$$

We may easily check that Rule (19.5a) is derivable according to this definition. The expression $\texttt{unroll(}e\texttt{)}$ is correspondingly defined to be the expression

$$\texttt{unfold(}e\texttt{)(}e\texttt{).}$$

It is easy to check that Rule (19.5b) is derivable from this definition. Moreover, we may check that

$$\texttt{unroll(self[}\tau\texttt{](}y\texttt{.}e\texttt{))} \mapsto^* [\texttt{self[}\tau\texttt{](}y\texttt{.}e\texttt{)/}y]e.$$

This completes the derivation of the type $\texttt{self(}\tau\texttt{)}$ of self-referential expressions of type τ.

One consequence of admitting the self-referential type $\texttt{self(}\tau\texttt{)}$ is that we may use it to define general recursion at *any* type. To be precise, we may define $\texttt{fix[}\tau\texttt{](}x\texttt{.}e\texttt{)}$ to stand for the expression

$$\texttt{unroll(self[}\tau\texttt{](}y\texttt{.}[\texttt{unroll(}y\texttt{)/}x]e\texttt{))}$$

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 13. Moreover, it also validates the dynamics, as evidenced by the following derivation:

$$\begin{aligned}
\texttt{fix[}\tau\texttt{](}x\texttt{.}e\texttt{)} &= \texttt{unroll(self[}\tau\texttt{](}y\texttt{.}[\texttt{unroll(}y\texttt{)/}x]e\texttt{))} \\
&\mapsto^* [\texttt{unroll(self[}\tau\texttt{](}y\texttt{.}[\texttt{unroll(}y\texttt{)/}x]e\texttt{))/}x]e \\
&= [\texttt{fix[}\tau\texttt{](}x\texttt{.}e\texttt{)/}x]e.
\end{aligned}$$

It follows that recursive types may be used to define a non-terminating expression of every type, namely $\texttt{fix[}\tau\texttt{](}x\texttt{.}x\texttt{)}$. Unlike many other type constructs we have considered, recursive types change the meaning of *every* type, not just those that involve recursion. Recursive types are therefore said to be a *non-conservative extension* of languages such as $\mathcal{L}\{\texttt{nat} \rightarrow\}$, which otherwise admits no non-terminating computations.

19.4 Exercises

Part VII

Dynamic Types

Chapter 20

The Untyped λ-Calculus

Types are the central organizing principle in the study of programming languages. Yet many languages of practical interest are said to be *untyped*. Have we missed something important? The answer is *no*! The supposed opposition between typed and untyped languages turns out to be illusory. In fact, untyped languages are special cases of typed languages with a single, pre-determined recursive type. Far from being *untyped*, such languages are instead *uni-typed*.[1]

In this chapter we study the premier example of a uni-typed programming language, the *(untyped) λ-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 λ-calculus has but one "feature", the higher-order function, with which to compute. 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 well-worn phrase, in the λ-calculus it's functions all the way down!

20.1 The λ-Calculus

The abstract syntax of $\mathcal{L}\{\lambda\}$ is given by the following grammar:

$$\begin{array}{llllll} \mathsf{Expr} & u & ::= & x & x & \text{variable} \\ & & & \lambda(x.u) & \lambda\,x.\,u & \lambda\text{-abstraction} \\ & & & \mathtt{ap}(u_1;u_2) & u_1(u_2) & \text{application} \end{array}$$

The statics of $\mathcal{L}\{\lambda\}$ is defined by general hypothetical judgements of the form $x_1\ \mathsf{ok}, \ldots, x_n\ \mathsf{ok} \vdash u\ \mathsf{ok}$, stating that u is a well-formed expression

[1] An apt description of Dana Scott's.

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:

$$\frac{}{\Gamma, x \ \mathsf{ok} \vdash x \ \mathsf{ok}} \tag{20.1a}$$

$$\frac{\Gamma \vdash u_1 \ \mathsf{ok} \quad \Gamma \vdash u_2 \ \mathsf{ok}}{\Gamma \vdash \mathtt{ap}(u_1; u_2) \ \mathsf{ok}} \tag{20.1b}$$

$$\frac{\Gamma, x \ \mathsf{ok} \vdash u \ \mathsf{ok}}{\Gamma \vdash \lambda(x.u) \ \mathsf{ok}} \tag{20.1c}$$

The dynamics is given by the following rules:

$$\frac{}{\lambda(x.u) \ \mathsf{val}} \tag{20.2a}$$

$$\frac{}{\mathtt{ap}(\lambda(x.u_1); u_2) \mapsto [u_2/x]u_1} \tag{20.2b}$$

$$\frac{u_1 \mapsto u_1'}{\mathtt{ap}(u_1; u_2) \mapsto \mathtt{ap}(u_1'; u_2)} \tag{20.2c}$$

In the λ-calculus literature this judgement is called *weak head reduction*. The first rule is called *β-reduction*; it defines the meaning of function application as substitution of argument for parameter.

Despite the apparent lack of types, $\mathcal{L}\{\lambda\}$ is nevertheless type safe!

Theorem 20.1. *If u ok, then either u val, or there exists u' such that $u \mapsto u'$ and u' ok.*

Proof. Exactly as in preceding chapters. We may show by induction on transition that well-formation is preserved by the dynamics. Since every closed value of $\mathcal{L}\{\lambda\}$ is a λ-abstraction, every closed expression is either a value or can make progress. □

Definitional equivalence for $\mathcal{L}\{\lambda\}$ is a judgement 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:

$$\frac{}{\Gamma, u \ \mathsf{ok} \vdash u \equiv u} \tag{20.3a}$$

$$\frac{\Gamma \vdash u \equiv u'}{\Gamma \vdash u' \equiv u} \tag{20.3b}$$

$$\frac{\Gamma \vdash u \equiv u' \quad \Gamma \vdash u' \equiv u''}{\Gamma \vdash u \equiv u''} \tag{20.3c}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' \quad \Gamma \vdash e_2 \equiv e_2'}{\Gamma \vdash \mathtt{ap}(e_1;e_2) \equiv \mathtt{ap}(e_1';e_2')} \tag{20.3d}$$

$$\frac{\Gamma, x \ \mathsf{ok} \vdash u \equiv u'}{\Gamma \vdash \lambda(x.u) \equiv \lambda(x.u')} \tag{20.3e}$$

$$\frac{}{\Gamma \vdash \mathtt{ap}(\lambda(x.e_2);e_1) \equiv [e_1/x]e_2} \tag{20.3f}$$

We often write just $u \equiv u'$ when the variables involved need not be emphasized or are clear from context.

20.2 Definability

Interest in the untyped λ-calculus stems from its surprising expressiveness. It is a *Turing-complete* language in the sense that it has the same capability to expression 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 λ-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 λ-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.[2]

We will sketch a proof that the untyped λ-calculus is as powerful as the language PCF described in Chapter 13. The main idea is to show that the PCF primitives for manipulating the natural numbers are definable in the untyped λ-calculus. This means, in particular, that we must show that the natural numbers are definable as λ-terms in such a way that case analysis, which 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.

[2] 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.

The first task is to represent the natural numbers as certain λ-terms, called the *Church numerals*.

$$\overline{0} = \lambda\, b.\, \lambda\, s.\, b \tag{20.4a}$$

$$\overline{n+1} = \lambda\, b.\, \lambda\, s.\, s(\overline{n}(b)(s)) \tag{20.4b}$$

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 λ-terms:

$$\mathtt{succ} = \lambda\, x.\, \lambda\, b.\, \lambda\, s.\, s(x(b)(s)) \tag{20.5}$$

$$\mathtt{plus} = \lambda\, x.\, \lambda\, y.\, y(x)(\mathtt{succ}) \tag{20.6}$$

$$\mathtt{times} = \lambda\, x.\, \lambda\, y.\, y(\overline{0})(\mathtt{plus}(x)) \tag{20.7}$$

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. We wish to find a term pred such that

$$\mathtt{pred}(\overline{0}) \equiv \overline{0} \tag{20.8}$$

$$\mathtt{pred}(\overline{n+1}) \equiv \overline{n}. \tag{20.9}$$

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 straightforward—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 nth 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.

$$\langle u_1, u_2 \rangle = \lambda\, f.\, f(u_1)(u_2) \tag{20.10}$$

$$u \cdot \mathtt{l} = u(\lambda\, x.\, \lambda\, y.\, x) \tag{20.11}$$

$$u \cdot \mathtt{r} = u(\lambda\, x.\, \lambda\, y.\, y) \tag{20.12}$$

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:

$$u'_p = \lambda\, x.\, x(\langle \overline{0}, \overline{0} \rangle)(\lambda\, y.\, \langle y \cdot \mathtt{r}, \mathtt{s}(y \cdot \mathtt{r}) \rangle) \tag{20.13}$$

$$u_p = \lambda\, x.\, u(x) \cdot \mathtt{l} \tag{20.14}$$

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 $\mathbf{Y}$ *combinator*:

$$\mathbf{Y} = \lambda\, F.\, (\lambda\, f.\, F(f(f)))(\lambda\, f.\, F(f(f))). \tag{20.15}$$

Observe that

$$\mathbf{Y}(F) \equiv F(\mathbf{Y}(F)).$$

Using the $\mathbf{Y}$ combinator, we may define general recursion by writing $\mathbf{Y}(\lambda\, x.\, u)$, where x stands for the recursive expression itself.

20.3 Scott's Theorem

Definitional equivalence for the untyped λ-calculus is undecidable: there is no algorithm to determine whether or not two untyped terms are definitionally equivalent. The proof of this result is based on two key lemmas:

1. For any untyped λ-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 12 for a discussion of Gödel-numbering.)

2. Any two non-trivial[3] properties $\mathcal{A}_0$ and $\mathcal{A}_1$ of untyped terms that respect definitional equivalence are *inseparable*. This means that there is no decidable property $\mathcal{B}$ of untyped terms such that $\mathcal{A}_0\ u$ implies that $\mathcal{B}\ u$ and $\mathcal{A}_1\ u$ implies that it is *not* the case that $\mathcal{B}\ u$. In particular, if $\mathcal{A}_0$ and $\mathcal{A}_1$ are inseparable, then neither is decidable.

For a property $\mathcal{B}$ of untyped terms to respect definitional equivalence means that if $\mathcal{B}\ u$ and $u \equiv u'$, then $\mathcal{B}\ u'$.

Lemma 20.2. *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 λ-calculus:

1. $\mathbf{ap}(\overline{\ulcorner u_1 \urcorner})(\overline{\ulcorner u_2 \urcorner}) \equiv \overline{\ulcorner u_1(u_2) \urcorner}$.

2. $\mathbf{nm}(\overline{n}) \equiv \overline{\ulcorner \overline{n} \urcorner}$.

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 $\overline{n}$. Given these, we may find the required term v by defining $v = w(\overline{\ulcorner w \urcorner})$, where $w = \lambda\, x.\, u(\mathbf{ap}(x)(\mathbf{nm}(x)))$. We have

$$\begin{aligned} v &= w(\overline{\ulcorner w \urcorner}) \\ &\equiv u(\mathbf{ap}(\overline{\ulcorner w \urcorner})(\mathbf{nm}(\overline{\ulcorner w \urcorner}))) \\ &\equiv u(\overline{\ulcorner w(\overline{\ulcorner w \urcorner}) \urcorner}) \\ &\equiv u(\overline{\ulcorner v \urcorner}). \end{aligned}$$

The definition is very similar to that of $\mathbf{Y}(u)$, 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 20.3. *Suppose that $\mathcal{A}_0$ and $\mathcal{A}_1$ are two non-vacuous properties of untyped terms that respect definitional equivalence. 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] 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.

3. *If $\mathcal{A}_1\ u$, then $w(\overline{\ulcorner u \urcorner}) \equiv \overline{1}$.*

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 $\mathcal{A}_0\ u_0$ and $\mathcal{A}_1\ u_1$. By Lemma 20.2 on the facing page 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 $\mathcal{A}_1\ t$, since $\mathcal{A}_1$ respects definitional equivalence 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 20.4. *There is no algorithm to decide whether or not $u \equiv u'$.*

Proof. For fixed u consider the property $\mathcal{E}_u\ u'$ defined by $u' \equiv u$. This is non-vacuous and respects definitional equivalence, and hence is undecidable. □

20.4 Untyped Means Uni-Typed

The untyped λ-calculus may be faithfully embedded in the typed language $\mathcal{L}\{+\times\rightharpoonup\mu\}$, enriched with recursive types. This means that every untyped λ-term has a representation as an expression in $\mathcal{L}\{+\times\rightharpoonup\mu\}$ in such a way that execution of the representation of a λ-term corresponds to execution of the term itself. If the execution model of the λ-calculus is call-by-name, this correspondence holds for the call-by-name variant of $\mathcal{L}\{+\times\rightharpoonup\mu\}$, and similarly for call-by-value.

It is important to understand that this form of embedding is *not* a matter of writing an interpreter for the λ-calculus in $\mathcal{L}\{+\times\rightharpoonup\mu\}$ (which we could surely do), but rather a direct representation of untyped λ-terms as certain typed expressions of $\mathcal{L}\{+\times\rightharpoonup\mu\}$. It is for this reason that we say that untyped languages are just a special case of typed languages, provided that we have recursive types at our disposal.

The key observation is that the *untyped* λ-calculus is really the *uni-typed* λ-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

$$D = \mu t\,.\,t \rightarrow t.$$

A value of type D is of the form $\mathtt{fold}(e)$ where e is a value of type $D \rightarrow D$ — a function whose domain and range are both D. Any such function can be regarded as a value of type D by "rolling", and any value of type D can be turned into a function by "unrolling". As usual, a recursive type may be seen as a solution to a type isomorphism equation, which in the present case is the equation

$$D \cong D \to D.$$

This specifies that D is a type that is isomorphic to the space of functions on D itself, something that is impossible in conventional set theory, but is feasible in the computationally-based setting of the λ-calculus.

This isomorphism leads to the following translation, of $\mathcal{L}\{\lambda\}$ into $\mathcal{L}\{+\times\rightharpoonup\mu\}$:

$$x^\dagger = x \tag{20.16a}$$

$$\lambda\, x.\, u^\dagger = \mathtt{fold}(\lambda\,(x{:}D.\, u^\dagger)) \tag{20.16b}$$

$$u_1(u_2)^\dagger = \mathtt{unfold}(u_1^\dagger)(u_2^\dagger) \tag{20.16c}$$

Observe that the embedding of a λ-abstraction is a value, and that the embedding of an application exposes the function being applied by unrolling 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 β-reduction is faithfully implemented by evaluation of the embedded terms.

Thus we see that the canonical *untyped* language, $\mathcal{L}\{\lambda\}$, 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 21 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 a traditional distinction can hardly be considered an opposition, since dynamic languages are but particular forms of static language in which (undue) emphasis is placed on a single recursive type.

20.5 Exercises

Chapter 21

Dynamic Typing

We saw in Chapter 20 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 λ-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. Obviously 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 λ-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 untyped λ-calculus is really unityped, so dynamic languages are special cases of static languages.

21.1 Dynamically Typed PCF

To illustrate dynamic typing we formulate a dynamically typed version of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, called $\mathcal{L}\{dyn\}$. The abstract syntax of $\mathcal{L}\{dyn\}$ is given by the following grammar:

Expr	d	$::=$	x	x	variable
			$\texttt{num}(\overline{n})$	$\overline{n}$	numeral
			$\texttt{zero}$	$\texttt{zero}$	zero
			$\texttt{succ}(d)$	$\texttt{succ}(d)$	successor
			$\texttt{ifz}(d;d_0;x.d_1)$	$\texttt{ifz}\,d\,\{\texttt{zero} \Rightarrow d_0 \mid \texttt{succ}(x) \Rightarrow d_1\}$	zero test
			$\texttt{fun}(\lambda\,x.d)$	$\lambda(x.d)$	abstraction
			$\texttt{dap}(d_1;d_2)$	$d_1(d_2)$	application
			$\texttt{fix}(x.d)$	$\texttt{fix}\,x\,\texttt{is}\,d$	recursion

There are two classes of values in $\mathcal{L}\{dyn\}$, the *numbers*, which have the form $\overline{n}$,[1] and the *functions*, which have the form $\lambda(x.d)$. The expressions $\texttt{zero}$ and $\texttt{succ}(d)$ are not in themselves values, but rather are operations that evaluate to classified values.

The concrete syntax of $\mathcal{L}\{dyn\}$ is somewhat deceptive, in keeping with common practice in dynamic languages. For example, the concrete syntax for a number is a bare numeral, $\overline{n}$, but in fact it is just a convenient notation for the classified value, $\texttt{num}(\overline{n})$, of class $\texttt{num}$. Similarly, the concrete syntax for a function is a λ-abstraction, $\lambda(x.d)$, which must be regarded as standing for the classified value $\texttt{fun}(\lambda\,x.d)$ of class $\texttt{fun}$.

The statics of $\mathcal{L}\{dyn\}$ is essentially the same as that of $\mathcal{L}\{\lambda\}$ given in Chapter 20; it merely checks that there are no free variables in the expression. The judgement

$$x_1\ \mathsf{ok}, \ldots x_n\ \mathsf{ok} \vdash d\ \mathsf{ok}$$

states that d is a well-formed expression with free variables among those in the hypothesis list.

The dynamics of $\mathcal{L}\{dyn\}$ checks for errors that would never arise in a safe statically typed language. For example, function application must ensure that its first argument is a function, signaling an error in the case that it is not, and similarly the case analysis construct must ensure that its first argument is a number, signaling an error if not. The reason for having classes labelling values is precisely to make this run-time check possible.

The value judgement, d val, states that d is a fully evaluated (closed) expression:

$$\frac{}{\texttt{num}(\overline{n})\ \mathsf{val}} \tag{21.1a}$$

$$\frac{}{\texttt{fun}(\lambda\,x.d)\ \mathsf{val}} \tag{21.1b}$$

[1] The numerals, $\overline{n}$, are n-fold compositions of the form $\texttt{s}(\texttt{s}(\ldots\texttt{s}(\texttt{z})\ldots))$.

The dynamics makes use of judgements that check the class of a value, and recover the underlying λ-abstraction in the case of a function.

$$\frac{}{\texttt{num}(\overline{n})\ \textsf{is\_num}\ \overline{n}} \tag{21.2a}$$

$$\frac{}{\texttt{fun}(\lambda\, x.\, d)\ \textsf{is\_fun}\ x\,.\,d} \tag{21.2b}$$

The second argument of each of these judgements has a special status—it is not an expression of $\mathcal{L}\{dyn\}$, but rather just a special piece of syntax used internally to the transition rules given below.

We also will need the "negations" of the class-checking judgements in order to detect run-time type errors.

$$\frac{}{\texttt{num}(\_)\ \textsf{isnt\_fun}} \tag{21.3a}$$

$$\frac{}{\texttt{fun}(\_)\ \textsf{isnt\_num}} \tag{21.3b}$$

The transition judgement, $d \mapsto d'$, and the error judgement, d $\textsf{err}$, are defined simultaneously by the following rules:[2]

$$\frac{}{\texttt{zero} \mapsto \texttt{num}(\texttt{z})} \tag{21.4a}$$

$$\frac{d \mapsto d'}{\texttt{succ}(d) \mapsto \texttt{succ}(d')} \tag{21.4b}$$

$$\frac{d\ \textsf{is\_num}\ \overline{n}}{\texttt{succ}(d) \mapsto \texttt{num}(\texttt{s}(\overline{n}))} \tag{21.4c}$$

$$\frac{d\ \textsf{isnt\_num}}{\texttt{succ}(d)\ \textsf{err}} \tag{21.4d}$$

$$\frac{d \mapsto d'}{\texttt{ifz}(d;d_0;x\,.\,d_1) \mapsto \texttt{ifz}(d';d_0;x\,.\,d_1)} \tag{21.4e}$$

$$\frac{d\ \textsf{is\_num}\ \textsf{z}}{\texttt{ifz}(d;d_0;x\,.\,d_1) \mapsto d_0} \tag{21.4f}$$

$$\frac{d\ \textsf{is\_num}\ \texttt{s}(\overline{n})}{\texttt{ifz}(d;d_0;x\,.\,d_1) \mapsto [\texttt{num}(\overline{n})/x]d_1} \tag{21.4g}$$

$$\frac{d\ \textsf{isnt\_num}}{\texttt{ifz}(d;d_0;x\,.\,d_1)\ \textsf{err}} \tag{21.4h}$$

$$\frac{d_1 \mapsto d_1'}{\texttt{dap}(d_1;d_2) \mapsto \texttt{dap}(d_1';d_2)} \tag{21.4i}$$

[2] The obvious error propagation rules discussed in Chapter 9 are omitted here for the sake of concision.

$$\frac{d_1 \ \mathsf{is\_fun} \ x\,.\,d}{\mathtt{dap}(d_1;d_2) \mapsto [d_2/x]d} \tag{21.4j}$$

$$\frac{d_1 \ \mathsf{isnt\_fun}}{\mathtt{dap}(d_1;d_2) \ \mathsf{err}} \tag{21.4k}$$

$$\frac{}{\mathtt{fix}(x\,.\,d) \mapsto [\mathtt{fix}(x\,.\,d)/x]d} \tag{21.4l}$$

Rule (21.4g) labels the predecessor with the class num to maintain the invariant that variables are bound to expressions of $\mathcal{L}\{dyn\}$.

The language $\mathcal{L}\{dyn\}$ enjoys essentially the same safety properties as $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, except that there are more opportunities for errors to arise at run-time.

Theorem 21.1 (Safety)**.** *If d ok, then either d val, or d err, or there exists d' such that $d \mapsto d'$.*

Proof. By rule induction on Rules (21.4). The rules are designed so that if d ok, then some rule, possibly an error rule, applies, ensuring progress. Since well-formedness is closed under substitution, the result of a transition is always well-formed. □

21.2 Variations and Extensions

The dynamic language $\mathcal{L}\{dyn\}$ defined in Section 21.1 on page 177 closely parallels the static language $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ defined in Chapter 13. One discrepancy, however, is in the treatment of natural numbers. Whereas in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ the zero and successor operations are introductory forms for the type nat, in $\mathcal{L}\{dyn\}$ they are elimination forms that act on separately-defined numerals. The point of this representation is to ensure that there is a well-defined class of *numbers* in the language.

It is worthwhile to explore an alternative representation that, superficially, is even closer to $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. Suppose that we eliminate the expression $\mathtt{num}(\overline{n})$ from the language, but retain zero and $\mathtt{succ}(d)$, with the idea that these are to be thought of as introductory forms for numbers in the language. We are immediately faced with the problem that such an expression is well-formed for *any* well-formed d. So, in particular, the expression $\mathtt{succ}(\lambda(x\,.\,d))$ is a value, as is succ(zero). There is no longer a well-defined class of *numbers*, but rather two separate classes of values, zero and successor, with no assurance that the successor is of a number.

The dynamics of the conditional branch changes only slightly, as described by the following rules:

$$\frac{d \mapsto d'}{\texttt{ifz}(d;d_0;x\,.\,d_1) \mapsto \texttt{ifz}(d';d_0;x\,.\,d_1)} \tag{21.5a}$$

$$\frac{d\ \mathsf{is\_zero}}{\texttt{ifz}(d;d_0;x\,.\,d_1) \mapsto d_0} \tag{21.5b}$$

$$\frac{d\ \mathsf{is\_succ}\ d'}{\texttt{ifz}(d;d_0;x\,.\,d_1) \mapsto [d'/x]d_1} \tag{21.5c}$$

$$\frac{d\ \mathsf{isnt\_zero} \quad d\ \mathsf{isnt\_succ}}{\texttt{ifz}(d;d_0;x\,.\,d_1)\ \mathsf{err}} \tag{21.5d}$$

The foregoing rules are to be augmented by the following rules that check whether a value is of class zero or successor:

$$\frac{}{\texttt{zero}\ \mathsf{is\_zero}} \tag{21.6a}$$

$$\frac{}{\texttt{succ}(d)\ \mathsf{isnt\_zero}} \tag{21.6b}$$

$$\frac{}{\texttt{succ}(d)\ \mathsf{is\_succ}\ d} \tag{21.6c}$$

$$\frac{}{\texttt{zero}\ \mathsf{isnt\_succ}} \tag{21.6d}$$

A peculiarity of this formulation of the conditional is that it can only be understood as distinguishing `zero` from `succ(_)`, rather than as distinguishing `zero` from non-`zero`. The reason is that if d is not `zero`, it might be either a successor or a function, and hence its "predecessor" is not well-defined.

Similar considerations arise when enriching $\mathcal{L}\{dyn\}$ with structured data. The classic example is to enrich the language as follows:

$$\begin{array}{llllll}
\mathsf{Expr} & d & ::= & \texttt{nil} & \texttt{nil} & \text{null} \\
 & & & \texttt{cons}(d_1;d_2) & \texttt{cons}(d_1;d_2) & \text{pair} \\
 & & & \texttt{ifnil}(d;d_0;x,y\,.\,d_1) & \multicolumn{2}{l}{\texttt{ifnil}\,d\,\{\texttt{nil} \Rightarrow d_0 \mid \texttt{cons}(x;y) \Rightarrow d_1\}} \\
 & & & & & \text{conditional}
\end{array}$$

The expression $\texttt{ifnil}(d;d_0;x,y\,.\,d_1)$ distinguishes the null structure from the pair of two structures. We leave to the reader the exercise of formulating the dynamics of this extension.

An advantage of dynamic typing is that the constructors `nil` and `cons`$(d_1; d_2)$ are sufficient to build unbounded, as well as bounded, data structures such as lists or trees. For example, the list consisting of three zero's may be represented by the value

$$\texttt{cons(zero;cons(zero;cons(zero;nil)))}.$$

But what to make of this beast?

$$\texttt{cons(zero;cons(zero;cons(zero;}\lambda\texttt{(}x\texttt{)}x\texttt{)))}.$$

It is a perfectly valid expression, but does not correspond to any natural data structure.

The disadvantage of this representation becomes apparent as soon as one wishes to define operations on lists, such as the append function:

$$\texttt{fix}\,a\,\texttt{is}\,\lambda\texttt{(}x\texttt{.}\lambda\texttt{(}y\texttt{.ifnil(}x\texttt{;}y\texttt{;}x_1\texttt{,}x_2\texttt{.cons(}x_1\texttt{;}a\texttt{(}x_2\texttt{)(}y\texttt{)))))}$$

What if x is the second list-like value given above? As it stands, the append function will signal an error upon reaching the function at the end of the list. If, however, y is this value, no error is signalled. This asymmetry may seem innocuous, but it is only one simple manifestation of a pervasive problem with dynamic languages: it is impossible to state within the language even the most rudimentary assumptions about the inputs, such as the assumption that both arguments to the append function ought to be genuine lists.

The conditional expression `ifnil`$(d; d_0; x, y.d_1)$ is rather *ad hoc* in that it makes a distinction between `nil` and all other values. Why not distinguish successors from non-succcessors, or functions from non-functions? A more systematic approach is to enrich the language with *predicates* and *destructors*. Predicates determine whether a value is of a specified class, and destructors recover the value labelled with a given class.

Expr	d	::=	`cond`$(d;d_0;d_1)$	`cond`$(d;d_0;d_1)$	conditional
			`nil?`(d)	`nil?`(d)	nil test
			`cons?`(d)	`cons?`(d)	pair test
			`car`(d)	`car`(d)	first projection
			`cdr`(d)	`cdr`(d)	second projection

The conditional `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 $\mathtt{nil?}(d)$ and $\mathtt{cons?}(d)$ 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 $\mathtt{car}(d)$ and $\mathtt{cdr}(d)$[3] decompose $\mathtt{cons}(d_1;d_2)$ into d_1 and d_2, respectively. As an example, the append function may be defined using predicates as follows:

$$\mathtt{fix}\,a\,\mathtt{is}\,\lambda(x.\lambda(y.\mathtt{cond}(x;\mathtt{cons}(\mathtt{car}(x);a(\mathtt{cdr}(x))(y));y))).$$

21.3 Critique of Dynamic Typing

The safety theorem for $\mathcal{L}\{dyn\}$ 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 $\mathcal{L}\{dyn\}$ is well-formed, and hence, by Theorem 21.1 on page 180, has a well-defined dynamics. But this can also be seen as a disadvantage, since errors that could be ruled out at compile time by type checking are not signalled until run time in $\mathcal{L}\{dyn\}$. To make this possible, the dynamics of $\mathcal{L}\{dyn\}$ must enforce conditions that need not be checked in a statically typed language.

Consider, for example, the addition function in $\mathcal{L}\{dyn\}$, whose specification is that, when passed two values of class `num`, returns their sum, which is also of class `num`:[4]

$$\mathtt{fun}(\lambda\,x.\mathtt{fix}(p.\mathtt{fun}(\lambda\,y.\mathtt{ifz}(y;x;y'.\mathtt{succ}(p(y')))))).$$

The addition function may, deceptively, be written in concrete syntax as follows:

$$\lambda(x.\mathtt{fix}\,p\,\mathtt{is}\,\lambda(y.\mathtt{ifz}\,y\,\{\mathtt{zero}\Rightarrow x\mid\mathtt{succ}(y')\Rightarrow\mathtt{succ}(p(y'))\})).$$

It is deceptive, because the concrete syntax obscures the class tags on values, and obscures the use of primitives that check 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 labelled with class `fun`. The dynamics of the fixed point construct binds p to this function. This means that the dynamic class check incurred by the application of p in

[3] This terminology for the projections is archaic, but firmly established in the literature.

[4] This specification imposes no restrictions on the behavior of addition on arguments that are not classified as numbers, but one could make the further demand that the function abort when applied to arguments that are not classified by `num`.

the recursive call is guaranteed to succeed. But $\mathcal{L}\{dyn\}$ offers no means of suppressing this 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 λ-abstraction is either x, the argument of the outer λ-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 is guaranteed 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, $\mathcal{L}\{dyn\}$ gives us no way to express this invariant; the repeated, redundant tag checks imposed by the successor operation cannot be avoided.

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 $\mathcal{L}\{dyn\}$, 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 $\mathcal{L}\{dyn\}$, because it cannot enforce the restrictions required to express the required invariants. For that we need a static type system.

21.4 Exercises

Chapter 22

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 21 may be embedded into the hybrid language by regarding a dynamically typed program as a statically typed program of type `dyn`. This shows that static and dynamic types are not opposed to one another, but may coexist harmoniously.

The notion of a hybrid language, however, is itself illusory, because the type `dyn` is really a particular recursive type. This shows that there is no need for any special mechanisms to support dynamic typing. Rather, they may be derived from the more general concept of a recursive type. Moreover, this shows that *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!

22.1 A Hybrid Language

Consider the language $\mathcal{L}\{\texttt{nat}\,\texttt{dyn}\rightharpoonup\}$, which extends $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$ (defined in Chapter 13) with the following additional constructs:

Type	τ	$::=$	`dyn`	`dyn`	dynamic
Expr	e	$::=$	$\texttt{new[}l\texttt{](}e\texttt{)}$	$l \cdot e$	construct
			$\texttt{cast[}l\texttt{](}e\texttt{)}$	$e \cdot l$	destruct
Class	l	$::=$	`num`	`num`	number
			`fun`	`fun`	function

The type dyn is the type of dynamically classified values. The new operation attaches a classifier to a value, and the cast operation checks the classifier and returns the associated value.

The statics of $\mathcal{L}\{\mathtt{nat}\,\mathtt{dyn} \rightharpoonup\}$ extends that of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ with the following additional rules:

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{new}\,[\mathtt{num}]\,(e) : \mathtt{dyn}} \tag{22.1a}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn} \rightharpoonup \mathtt{dyn}}{\Gamma \vdash \mathtt{new}\,[\mathtt{fun}]\,(e) : \mathtt{dyn}} \tag{22.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}\,[\mathtt{num}]\,(e) : \mathtt{nat}} \tag{22.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}\,[\mathtt{fun}]\,(e) : \mathtt{dyn} \rightharpoonup \mathtt{dyn}} \tag{22.1d}$$

The statics ensures that class labels are applied to objects of the appropriate type, namely num for natural numbers, and fun for functions defined over labelled values.

The dynamics of $\mathcal{L}\{\mathtt{nat}\,\mathtt{dyn} \rightharpoonup\}$ extends that of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ with the following rules:

$$\frac{e\ \mathsf{val}}{\mathtt{new}\,[l]\,(e)\ \mathsf{val}} \tag{22.2a}$$

$$\frac{e \mapsto e'}{\mathtt{new}\,[l]\,(e) \mapsto \mathtt{new}\,[l]\,(e')} \tag{22.2b}$$

$$\frac{e \mapsto e'}{\mathtt{cast}\,[l]\,(e) \mapsto \mathtt{cast}\,[l]\,(e')} \tag{22.2c}$$

$$\frac{\mathtt{new}\,[l]\,(e)\ \mathsf{val}}{\mathtt{cast}\,[l]\,(\mathtt{new}\,[l]\,(e)) \mapsto e} \tag{22.2d}$$

$$\frac{\mathtt{new}\,[l']\,(e)\ \mathsf{val} \quad l \neq l'}{\mathtt{cast}\,[l]\,(\mathtt{new}\,[l']\,(e))\ \mathsf{err}} \tag{22.2e}$$

Casting compares the class of the object to the required class, returning the underlying object if these coincide, and signalling an error otherwise.

Lemma 22.1 (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 $e' : \mathtt{dyn} \rightharpoonup \mathtt{dyn}$.*

Proof. By a straightforward rule induction on the statics of $\mathcal{L}\{\mathtt{nat}\,\mathtt{dyn} \rightharpoonup\}$. □

Theorem 22.2 (Safety)**.** *The language $\mathcal{L}\{\mathtt{nat}\,\mathtt{dyn} \rightharpoonup\}$ is safe:*

1. *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

2. *If $e : \tau$, then either e val, or e err, or $e \mapsto e'$ 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 $\mathcal{L}\{dyn\}$—a well-typed cast might fail at run-time if the class of the cast does not match the class of the value. □

22.2 Optimization of Dynamic Typing

The language $\mathcal{L}\{\texttt{nat}\,\texttt{dyn} \rightharpoonup\}$ combines static and dynamic typing by enriching $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ 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 $\mathcal{L}\{dyn\}$.

Let us examine this in the case of the addition function, which may be defined in $\mathcal{L}\{\texttt{nat}\,\texttt{dyn} \rightharpoonup\}$ as follows:

$$\texttt{fun} \cdot \lambda\,(x{:}\texttt{dyn}.\,\texttt{fix}\,p{:}\texttt{dyn}\,\texttt{is}\,\texttt{fun} \cdot \lambda\,(y{:}\texttt{dyn}.\,e_{x,p,y})),$$

where

$$x : \texttt{dyn}, p : \texttt{dyn}, y : \texttt{dyn} \vdash e_{x,p,y} : \texttt{dyn}$$

is defined to be the expression

$$\texttt{ifz}\,(y \cdot \texttt{num})\,\{\texttt{zero} \Rightarrow x \mid \texttt{succ}(y') \Rightarrow \texttt{num} \cdot (\texttt{s}((p \cdot \texttt{fun})(\texttt{num} \cdot y') \cdot \texttt{num}))\}.$$

This is a reformulation of the dynamic addition function given in Section 21.3 on page 183 in which we have made explicit the checking and imposition of classes on values. We will exploit the static type system of $\mathcal{L}\{\texttt{nat}\,\texttt{dyn} \rightharpoonup\}$ to optimize this dynamically typed implementation of addition in accordance with the specification given in Section 21.3 on page 183.

First, note that the body of the `fix` expression is an explicitly labelled function. This means that when the recursion is unwound, the variable p is bound to this value of type `dyn`. Consequently, the check that p is labelled with class `fun` is redundant, and can be eliminated. This is achieved by rewriting the function as follows:

$$\texttt{fun} \cdot \lambda\,(x{:}\texttt{dyn}.\,\texttt{fun} \cdot \texttt{fix}\,p{:}\texttt{dyn} \rightharpoonup \texttt{dyn}\,\texttt{is}\,\lambda\,(y{:}\texttt{dyn}.\,e'_{x,p,y})),$$

where $e'_{x,p,y}$ is the expression

$$\texttt{ifz}\,(y\cdot\texttt{num})\,\{\texttt{zero}\Rightarrow x\mid\texttt{succ}(y')\Rightarrow\texttt{num}\cdot(\texttt{s}(p(\texttt{num}\cdot y')\cdot\texttt{num}))\}.$$

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 $\texttt{dyn}\rightharpoonup\texttt{dyn}$, reflecting that the body is now a "bare function", rather than a labelled function value of type 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. Since 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 labelled 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 labelling 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 labelled 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 canbe enforced by replacing x in the zero branch of the conditional by $x\cdot\texttt{num}$.

Combining these optimizations we obtain the inner loop e''_x defined as follows:

$$\texttt{fix}\,p{:}\texttt{nat}\rightharpoonup\texttt{nat}\,\texttt{is}\,\lambda\,(y{:}\texttt{nat}.\,\texttt{ifz}\,y\,\{\texttt{zero}\Rightarrow x\cdot\texttt{num}\mid\texttt{succ}(y')\Rightarrow\texttt{s}(p(y'))\}).$$

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.

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 $\texttt{dyn}\rightharpoonup\texttt{dyn}$, but what we have at hand is a function of type $\texttt{nat}\rightharpoonup\texttt{nat}$. This can be converted to the required form by pre-composing with a cast to num and post-composing with a coercion to num:

$$\texttt{fun}\cdot\lambda\,(x\texttt{:dyn.}\,\texttt{fun}\cdot\lambda\,(y\texttt{:dyn.}\,\texttt{num}\cdot(e''_x(y\cdot\texttt{num})))).$$

The innermost λ-abstraction converts the function e''_x from type $\texttt{nat} \rightharpoonup \texttt{nat}$ to type $\texttt{dyn} \rightharpoonup \texttt{dyn}$ 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.

22.3 Static "Versus" Dynamic Typing

There are many attempts to distinguish dynamic from static typing, all of which are misleading or wrong. For example, it is often said that static type systems associate types with variables, but dynamic type systems associate types with values. This oft-repeated characterization appears to be justified by the absence of type annotations on λ-abstractions, and the presence of classes on values. But it is based on a confusion of classes with types—the *class* of a value (num or fun) is not its *type*. Moreover, a static type system assigns types to values just as surely as it does to variables, so the description fails on this account as well.

Another way to differentiate dynamic from static languages is to say that whereas static languages check types at compile time, dynamic languages check types at run time. But to say that static languages check types statically is to state a tautology, and to say that dynamic languages check types at run-time is to utter a falsehood. Dynamic languages perform *class checking*, not *type checking*, at run-time. For example, application checks that its first argument is labelled with fun; it does not type check the body of the function. Indeed, at no point does the dynamics compute the *type* of a value, rather it checks its class against its expectations before proceeding. Here again, a supposed contrast between static and dynamic languages evaporates under careful analysis.

Another characterization is to assert that dynamic languages admit heterogeneous collections, whereas static languages admit only homogeneous collections. For example, in a dynamic language the elements of a list may be of disparate *classes*, as illustrated by the expression

$$\texttt{cons(s(z);cons(}\lambda\texttt{(}\lambda\texttt{(}x\texttt{.}x\texttt{));nil)).}$$

But they are nevertheless all of the same *type*! Put the other way around, a static language with a dynamic type is just as capable of representing a heterogeneous collection as is a dynamic language with only one type.

What, then, are we to make of the traditional distinction between dynamic and static languages? Rather than being in opposition to each other, we see that *dynamic languages are a mode of use of static languages*. If we have a type dyn in the language, then we have all of the apparatus of dynamic languages at our disposal, so there is no loss of expressive power. But there is a very significant gain from embedding dynamic typing within a static type discipline! We can avoid much of the overhead of dynamic typing by simply limiting our use of the type dyn in our programs, as was illustrated in Section 22.2 on page 187.

22.4 Reduction to Recursive Types

The type dyn codifies the use of dynamic typing within a static language. Its introduction form labels an object of the appropriate type, and its elimination form is a (possibly undefined) casting operation. Rather than treating dyn as primitive, we may derive it as a particular use of recursive types, according to the following definitions:

$$\begin{aligned}
\texttt{dyn} &= \mu t\,.\,[\texttt{num}:\texttt{nat},\texttt{fun}:t \rightharpoonup t] && (22.3)\\
\texttt{new[num]}(e) &= \texttt{fold}(\texttt{num}\cdot e) && (22.4)\\
\texttt{new[fun]}(e) &= \texttt{fold}(\texttt{fun}\cdot e) && (22.5)\\
\texttt{cast[num]}(e) &= \texttt{case}\,\texttt{unfold}(e)\,\{\texttt{num}\cdot x \Rightarrow x \mid \texttt{fun}\cdot x \Rightarrow \texttt{error}\} && (22.6)\\
\texttt{cast[fun]}(e) &= \texttt{case}\,\texttt{unfold}(e)\,\{\texttt{num}\cdot x \Rightarrow \texttt{error} \mid \texttt{fun}\cdot x \Rightarrow x\} && (22.7)
\end{aligned}$$

One may readily check that the static and dynamics for the type dyn are derivable according to these definitions.

This encoding readily generalizes to any number of classes of values: we need only consider additional summands corresponding to each class. For example, to account for the constructors nil and cons(d_1;d_2) considered in Chapter 21, the definition of dyn is expanded to the recursive type

$$\mu t\,.\,[\texttt{num}:\texttt{nat},\texttt{fun}:t \rightharpoonup t,\texttt{nil}:\texttt{unit},\texttt{cons}:t \times t],$$

with corresponding definitions for the new and cast operations. This exemplifies the general case: dynamic typing is a mode of use of static types in which classes of values are simply names of summands in a recursive type of dynamic values.

Part VIII

Variable Types

Chapter 23

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 $\mathcal{L}\{\texttt{nat}\rightarrow\}$ there is a *distinct* identity function for each type τ, namely $\lambda\,(x{:}\tau.\,x)$, even though the behavior is the same for each choice of τ. Similarly, there is a distinct composition operator for each triple of types, namely

$$\circ_{\tau_1,\tau_2,\tau_3} = \lambda\,(f{:}\tau_2 \rightarrow \tau_3.\,\lambda\,(g{:}\tau_1 \rightarrow \tau_2.\,\lambda\,(x{:}\tau_1.\,f(g(x))))).$$

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 a language introduced by Girard under the name *System F* and by Reynolds under the name *polymorphic typed λ-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 24), 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.)

23.1 System F

*System **F***, or the *polymorphic λ-calculus*, or $\mathcal{L}\{\to\forall\}$, is a minimal functional language that illustrates the core concepts of polymorphic typing, and permits us to examine its surprising expressive power in isolation from other language features. The syntax of System **F** is given by the following grammar:

Type	τ	$::=$	t	t	variable
			$\texttt{arr}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$	function
			$\texttt{all}(t.\tau)$	$\forall(t.\tau)$	polymorphic
Expr	e	$::=$	x	x	
			$\texttt{lam}[\tau](x.e)$	$\lambda\,(x{:}\tau.e)$	abstraction
			$\texttt{ap}(e_1;e_2)$	$e_1(e_2)$	application
			$\texttt{Lam}(t.e)$	$\Lambda(t.e)$	type abstraction
			$\texttt{App}[\tau](e)$	$e[\tau]$	type application

A *type abstraction*, $\texttt{Lam}(t.e)$, defines a *generic*, or *polymorphic*, function with *type parameter t* standing for an unspecified type within e. A *type application*, or *instantiation*, $\texttt{App}[\tau](e)$, 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*, $\texttt{all}(t.\tau)$, that determines the type, τ, of the result as a function of the argument, t.

The statics of $\mathcal{L}\{\to\forall\}$ consists of two judgement forms, the *type formation* judgement,

$$\vec{t} \mid \Delta \vdash \tau \ \mathsf{type},$$

and the *typing judgement*,

$$\vec{t}\ \vec{x} \mid \Delta\ \Gamma \vdash e : \tau.$$

These are generic judgements over *type variables* $\vec{t}$ and *expression variables* $\vec{x}$. They are also hypothetical in a set Δ of *type assumptions* of the form $t\ \mathsf{type}$, where $t \in \mathcal{T}$, and *typing assumptions* of the form $x : \tau$, where $x \in \mathcal{T}$ and $\Delta \vdash \tau\ \mathsf{type}$. As usual we drop explicit mention of the parameter sets, relying on typographical conventions to determine them.

The rules defining the type formation judgement are as follows:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \tag{23.1a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \texttt{arr}(\tau_1;\tau_2)\ \mathsf{type}} \tag{23.1b}$$

$$\frac{\Delta, t \;\mathsf{type} \vdash \tau \;\mathsf{type}}{\Delta \vdash \mathtt{all}(t.\tau) \;\mathsf{type}} \tag{23.1c}$$

The rules defining the typing judgement are as follows:

$$\frac{}{\Delta\;\Gamma, x : \tau \vdash x : \tau} \tag{23.2a}$$

$$\frac{\Delta \vdash \tau_1 \;\mathsf{type} \quad \Delta\;\Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta\;\Gamma \vdash \mathtt{lam}[\tau_1](x.e) : \mathtt{arr}(\tau_1;\tau_2)} \tag{23.2b}$$

$$\frac{\Delta\;\Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Delta\;\Gamma \vdash e_2 : \tau_2}{\Delta\;\Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{23.2c}$$

$$\frac{\Delta, t \;\mathsf{type}\;\Gamma \vdash e : \tau}{\Delta\;\Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{23.2d}$$

$$\frac{\Delta\;\Gamma \vdash e : \mathtt{all}(t.\tau') \quad \Delta \vdash \tau \;\mathsf{type}}{\Delta\;\Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\tau'} \tag{23.2e}$$

Lemma 23.1 (Regularity). *If $\Delta\;\Gamma \vdash e : \tau$, and if $\Delta \vdash \tau_i$ type for each assumption $x_i : \tau_i$ in Γ, then $\Delta \vdash \tau$ type.*

Proof. By induction on Rules (23.2). □

The statics admits the structural rules for a general hypothetical judgement. In particular, we have the following critical substitution property for type formation and expression typing.

Lemma 23.2 (Substitution).

1. *If Δ, t type $\vdash \tau'$ type and $\Delta \vdash \tau$ type, then $\Delta \vdash [\tau/t]\tau'$ type.*
2. *If Δ, t type $\Gamma \vdash e' : \tau'$ and $\Delta \vdash \tau$ 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, Γ, 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 $I\,\texttt{[}\tau\texttt{]}$, where τ is some type, and have the type $\tau \to \tau$.

Similarly, the polymorphic composition function, C, is written

$$\Lambda\texttt{(}t_1\,\texttt{.}\,\Lambda\texttt{(}t_2\,\texttt{.}\,\Lambda\texttt{(}t_3\,\texttt{.}\,\lambda\,\texttt{(}f\,\texttt{:}\,t_2 \to t_3\texttt{.}\,\lambda\,\texttt{(}g\,\texttt{:}\,t_1 \to t_2\texttt{.}\,\lambda\,\texttt{(}x\,\texttt{:}\,t_1\texttt{.}\,f\texttt{(}g\texttt{(}x\texttt{)))))))}\texttt{.}$$

The function C has the polymorphic type

$$\forall\texttt{(}t_1\,\texttt{.}\,\forall\texttt{(}t_2\,\texttt{.}\,\forall\texttt{(}t_3\,\texttt{.}\,\texttt{(}t_2 \to t_3\texttt{)} \to \texttt{(}t_1 \to t_2\texttt{)} \to \texttt{(}t_1 \to t_3\texttt{))))}\texttt{.}$$

Instances of C are obtained by applying it to a triple of types, writing $C\,\texttt{[}\tau_1\texttt{]}\,\texttt{[}\tau_2\texttt{]}\,\texttt{[}\tau_3\texttt{]}$. Each such instance has the type

$$\texttt{(}\tau_2 \to \tau_3\texttt{)} \to \texttt{(}\tau_1 \to \tau_2\texttt{)} \to \texttt{(}\tau_1 \to \tau_3\texttt{)}\texttt{.}$$

Dynamics

The dynamics of $\mathcal{L}\{\to\forall\}$ is given as follows:

$$\frac{}{\texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)}\ \mathsf{val}} \tag{23.3a}$$

$$\frac{}{\texttt{Lam(}t\texttt{.}e\texttt{)}\ \mathsf{val}} \tag{23.3b}$$

$$\frac{}{\texttt{ap(lam[}\tau_1\texttt{](}x\texttt{.}e\texttt{);}e_2\texttt{)} \mapsto [e_2/x]e} \tag{23.3c}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto \texttt{ap(}e_1'\texttt{;}e_2\texttt{)}} \tag{23.3d}$$

$$\frac{}{\texttt{App[}\tau\texttt{](Lam(}t\texttt{.}e\texttt{))} \mapsto [\tau/t]e} \tag{23.3e}$$

$$\frac{e \mapsto e'}{\texttt{App[}\tau\texttt{](}e\texttt{)} \mapsto \texttt{App[}\tau\texttt{](}e'\texttt{)}} \tag{23.3f}$$

These rules endow $\mathcal{L}\{\to\forall\}$ with a call-by-name interpretation of application, but one could as well consider a call-by-value variant.

It is a simple matter to prove safety for $\mathcal{L}\{\to\forall\}$, using familiar methods.

Lemma 23.3 (Canonical Forms). *Suppose that $e : \tau$ and e val, then*

1. *If $\tau = \texttt{arr(}\tau_1\texttt{;}\tau_2\texttt{)}$, then $e = \texttt{lam[}\tau_1\texttt{](}x\texttt{.}e_2\texttt{)}$ with $x : \tau_1 \vdash e_2 : \tau_2$.*
2. *If $\tau = \texttt{all(}t\texttt{.}\tau'\texttt{)}$, then $e = \texttt{Lam(}t\texttt{.}e'\texttt{)}$ with t $\mathsf{type} \vdash e' : \tau'$.*

Proof. By rule induction on the statics. □

Theorem 23.4 (Preservation). *If $e : \sigma$ and $e \mapsto e'$, then $e' : \sigma$.*

Proof. By rule induction on the dynamics. □

Theorem 23.5 (Progress)**.** *If $e : \sigma$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By rule induction on the statics. □

23.2 Polymorphic Definability

The language $\mathcal{L}\{\to\forall\}$ 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 equivalence, which is defined to be the least congruence containing the following two axioms:

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash e : \tau_2 \quad \Delta\ \Gamma \vdash e_1 : \tau_1}{\Delta\ \Gamma \vdash \lambda\,(x{:}\tau.\,e_2)\,(e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{23.4a}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau \quad \Delta \vdash \sigma\ \mathsf{type}}{\Delta\ \Gamma \vdash \Lambda(t.e)\,[\sigma] \equiv [\sigma/t]e : [\sigma/t]\tau} \tag{23.4b}$$

In addition there are rules omitted here specifying that definitional equivalence is reflexive, symmetric, and transitive, and that it is compatible with both forms of application and abstraction.

23.2.1 Products and Sums

The nullary product, or unit, type is definable in $\mathcal{L}\{\to\forall\}$ as follows:

$$\begin{aligned} \mathtt{unit} &= \forall(r.r \to r) \\ \langle\rangle &= \Lambda(r.\lambda\,(x{:}r.\,x)) \end{aligned}$$

It is easy to check that the statics given in Chapter 14 is derivable. There being no elimination rule, there is no requirement on the dynamics.

Binary products are definable in $\mathcal{L}\{\to\forall\}$ by using encoding tricks similar to those described in Chapter 20 for the untyped λ-calculus:

$$\begin{aligned} \tau_1 \times \tau_2 &= \forall(r.(\tau_1 \to \tau_2 \to r) \to r) \\ \langle e_1, e_2\rangle &= \Lambda(r.\lambda\,(x{:}\tau_1 \to \tau_2 \to r.\,x(e_1)(e_2))) \\ e \cdot \mathtt{l} &= e[\tau_1](\lambda\,(x{:}\tau_1.\,\lambda\,(y{:}\tau_2.\,x))) \\ e \cdot \mathtt{r} &= e[\tau_2](\lambda\,(x{:}\tau_1.\,\lambda\,(y{:}\tau_2.\,y))) \end{aligned}$$

The statics given in Chapter 14 is derivable according to these definitions. Moreover, the following definitional equivalences are derivable in $\mathcal{L}\{\rightarrow\forall\}$ 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 $\mathcal{L}\{\rightarrow\forall\}$:

$$\begin{aligned} \mathtt{void} &= \forall(r\,.\,r) \\ \mathtt{abort}[\rho](e) &= e[\rho] \end{aligned}$$

There is no definitional equivalence to be checked, there being no introductory rule for the void type.

Binary sums are also definable in $\mathcal{L}\{\rightarrow\forall\}$:

$$\begin{aligned} \tau_1 + \tau_2 &= \forall(r\,.\,(\tau_1 \rightarrow r) \rightarrow (\tau_2 \rightarrow r) \rightarrow r) \\ \mathtt{l} \cdot e &= \Lambda(r\,.\,\lambda\,(x{:}\tau_1 \rightarrow r\,.\,\lambda\,(y{:}\tau_2 \rightarrow r\,.\,x(e)))) \\ \mathtt{r} \cdot e &= \Lambda(r\,.\,\lambda\,(x{:}\tau_1 \rightarrow r\,.\,\lambda\,(y{:}\tau_2 \rightarrow r\,.\,y(e)))) \\ \mathtt{case}\, e\, \{\mathtt{l} \cdot x_1 \Rightarrow e_1 \mid \mathtt{r} \cdot x_2 \Rightarrow e_2\} &= \\ e[\rho]\,(\lambda\,(x_1{:}\tau_1.\,e_1))\,(\lambda\,(x_2{:}\tau_2.\,e_2)) \end{aligned}$$

provided that the types make sense. It is easy to check that the following equivalences are derivable in $\mathcal{L}\{\rightarrow\forall\}$:

$$\mathtt{case}\,\mathtt{l} \cdot d_1\,\{\mathtt{l} \cdot x_1 \Rightarrow e_1 \mid \mathtt{r} \cdot x_2 \Rightarrow e_2\} \equiv [d_1/x_1]e_1 : \rho$$

and

$$\mathtt{case}\,\mathtt{r} \cdot d_2\,\{\mathtt{l} \cdot x_1 \Rightarrow e_1 \mid \mathtt{r} \cdot x_2 \Rightarrow e_2\} \equiv [d_2/x_2]e_2 : \rho.$$

Thus the dynamic behavior specified in Chapter 15 is correctly implemented by these definitions.

23.2.2 Natural Numbers

As we remarked above, the natural numbers (under a lazy interpretation) are also definable in $\mathcal{L}\{\rightarrow\forall\}$. The key is the representation of the iterator, whose typing rule we recall here for reference:

$$\frac{e_0 : \mathtt{nat} \quad e_1 : \tau \quad x : \tau \vdash e_2 : \tau}{\mathtt{natiter}(e_0; e_1; x\,.\,e_2) : \tau}\,.$$

Since the result type τ is arbitrary, this means that if we have an iterator, then it can be used to define a function of type

$$\texttt{nat} \to \forall(t\,.\,t \to (t \to t) \to t).$$

This function, when applied to an argument n, yields a polymorphic function that, for any result type, t, if given the initial result for $\texttt{z}$, and if given a function transforming the result for x into the result for $\texttt{s}(x)$, then it returns the result of iterating the transformer n times starting with the initial result.

Since 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 $\mathcal{L}\{\to\forall\}$ by the following equations:

$$\begin{aligned}
\texttt{nat} &= \forall(t\,.\,t \to (t \to t) \to t)\\
\texttt{z} &= \Lambda(t\,.\,\lambda\,(z\,{:}\,t.\,\lambda\,(s\,{:}\,t \to t.\,z)))\\
\texttt{s}(e) &= \Lambda(t\,.\,\lambda\,(z\,{:}\,t.\,\lambda\,(s\,{:}\,t \to t.\,s(e[t](z)(s)))))\\
\texttt{natiter}(e_0;e_1;x\,.\,e_2) &= e_0[\tau](e_1)(\lambda\,(x\,{:}\,\tau.\,e_2))
\end{aligned}$$

It is a straightforward exercise to check that the static and dynamics given in Chapter 12 is derivable in $\mathcal{L}\{\to\forall\}$ under these definitions.

This shows that $\mathcal{L}\{\to\forall\}$ is *at least as expressive* as $\mathcal{L}\{\texttt{nat}\to\}$. But is it *more* expressive? Yes! It is possible to show that the evaluation function for $\mathcal{L}\{\texttt{nat}\to\}$ is definable in $\mathcal{L}\{\to\forall\}$, even though it is not definable in $\mathcal{L}\{\texttt{nat}\to\}$ itself. However, the same diagonal argument given in Chapter 12 applies here, showing that the evaluation function for $\mathcal{L}\{\to\forall\}$ is not definable in $\mathcal{L}\{\to\forall\}$. We may enrich $\mathcal{L}\{\to\forall\}$ a bit more to define the evaluator for $\mathcal{L}\{\to\forall\}$, 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. The extension process will never close as long as all programs written in it terminate.

23.3 Parametricity Overview

A remarkable property of $\mathcal{L}\{\to\forall\}$ is that polymorphic types severely constrain the behavior of their elements. One 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, τ, and an argument of type τ, it must return a value of type τ. But since τ 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 either $\Lambda(t.\lambda\,(x{:}t.\lambda\,(y{:}t.x)))$ or $\Lambda(t.\lambda\,(x{:}t.\lambda\,(y{:}t.y)))$. For when b is applied to two arguments of some type, its only choice to return a value of that type is to return one of the two.

A full proof of these claims is somewhat involved (see Chapter 53 for details), but the core idea is relatively simple, namely to *interpret types as relations*. The *parametricity theorem* (Theorem 53.8 on page 497) states that every well-typed term respects the relational interpretation of its type. For example, the parametricity theorem implies that if $i : \forall(t.t \to t)$, then for any type τ, any predicate P on expressions of type τ, and any $e : \tau$, if $P(e)$, then $P(i[\tau](e))$. Fix τ and $e : \tau$, and define $P(x)$ to hold iff $x \cong e : \tau$.[1] By the theorem we have that for any $e' : \tau$, if $e' \cong e : \tau$, then $i[\tau](e') \cong e : \tau$, and so in particular $i[\tau](e) \cong e : \tau$. Similarly, if $c : \forall(t.t \to t \to t)$, then, fixing τ, $e_1 : \tau$, and $e_2 : \tau$, we may define $P(e)$ to hold iff either $e \cong e_1 : \tau$ or $e \cong e_2 : \tau$. It follows from the theorem that either $c[\tau](e_1)(e_2) \cong e_1 : \tau$ or $c[\tau](e_1)(e_2) \cong e_2 : \tau$.

What is remarkable is that these properties of i and c have been derived *without knowing anything about the expressions themselves*, but only their types! The theory of parametricity implies that we are able to derive theorems about the behavior of a program knowing only its type. Such theorems are sometimes called *free theorems* because they come “for free” as a consequence of typing, and require no program analysis or verification to derive (beyond the once-and-for-all proof of Theorem 53.8 on page 497). Free theorems such as those illustrated above underly the experience that in a polymorphic language, well-typed programs tend to behave as expected no further debugging or analysis required. Parametricity so constrains the behavior of a program that it is relatively easy to ensure that the code works just by checking its type. Free theorems also underly the principle of representation independence for abstract types, which is discussed further in Chapter 24.

[1] The relation $e \cong e' : \tau$ of *observational equivalence* is defined in Chapter 53. For the present it is enough to know that it is the coarsest congruence on terms of the same type that does not equate all terms.

23.4 Restricted Forms of Polymorphism

In this section we briefly examine some restricted forms of polymorphism with less than the full expressive power of $\mathcal{L}\{\to\forall\}$. These are obtained in one of two ways:

1. Restricting type quantification to unquantified types.

2. Restricting the occurrence of quantifiers within types.

23.4.1 Predicative Fragment

The remarkable expressive power of the language $\mathcal{L}\{\to\forall\}$ may be traced to the ability to instantiate a polymorphic type with another polymorphic type. For example, if we let τ be the type $\forall(t\,.\,t \to t)$, 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 τ — the very type of e!

This property of $\mathcal{L}\{\to\forall\}$ is called *impredicativity*[2]; the language $\mathcal{L}\{\to\forall\}$ 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 quasi-circularity is responsible for the surprising expressive power of $\mathcal{L}\{\to\forall\}$, and is correspondingly the prime source of complexity when reasoning about it (for example, in the proof that all expressions of $\mathcal{L}\{\to\forall\}$ terminate).

Contrast this with $\mathcal{L}\{\to\}$, 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 $[\sigma/t]\tau$ 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

[2] pronounced *im-PRED-ic-a-tiv-it-y*

type $u \to u$ to obtain the type $(u \to u) \to (u \to u)$. This type has more symbols in it than τ, but is smaller in that it has fewer quantifiers. The restriction to quantification only over unquantified types is called *predicative*[3] *polymorphism*. The predicative fragment is significantly less expressive than the full impredicative language. In particular, the natural numbers are no longer definable in it.

The formalization of $\mathcal{L}\{\to\forall_{\mathsf{p}}\}$ is left to Chapter 25, where the appropriate technical machinery is available.

23.4.2 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 $\mathcal{L}\{\to\forall\}$, because it plays an important role in the design of practical polymorphic languages.

The prenex fragment of $\mathcal{L}\{\to\forall\}$ is designated $\mathcal{L}^1\{\to\forall\}$, 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 judgements $\Delta \vdash \tau\ \mathsf{mono}$ and $\Delta \vdash \tau\ \mathsf{poly}$, where Δ is a finite set of hypotheses of the form $t\ \mathsf{mono}$, where t is a type variable not otherwise declared in Δ. The rules for deriving these judgements are as follows:

$$\frac{}{\Delta, t\ \mathsf{mono} \vdash t\ \mathsf{mono}} \tag{23.5a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{mono} \quad \Delta \vdash \tau_2\ \mathsf{mono}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2)\ \mathsf{mono}} \tag{23.5b}$$

$$\frac{\Delta \vdash \tau\ \mathsf{mono}}{\Delta \vdash \tau\ \mathsf{poly}} \tag{23.5c}$$

[3] pronounced *PRED-i-ca-tive*

$$\frac{\Delta, t\ \mathsf{mono} \vdash \tau\ \mathsf{poly}}{\Delta \vdash \mathtt{all}(t.\tau)\ \mathsf{poly}} \tag{23.5d}$$

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 $\mathcal{L}^1\{\to\forall\}$ is given by rules for deriving hypothetical judgements of the form $\Delta\ \Gamma \vdash e : \sigma$, where Δ consists of hypotheses of the form t mono, and Γ consists of hypotheses of the form $x : \sigma$, where $\Delta \vdash \sigma$ poly. The rules defining this judgement are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{23.6a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{mono} \quad \Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) : \mathtt{arr}(\tau_1;\tau_2)} \tag{23.6b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{23.6c}$$

$$\frac{\Delta, t\ \mathsf{mono}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{23.6d}$$

$$\frac{\Delta \vdash \tau\ \mathsf{mono} \quad \Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau')}{\Delta\ \Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\tau'} \tag{23.6e}$$

We tacitly exploit the inclusion of monotypes as polytypes so that all typing judgements have the form $e : \sigma$ for some expression e and polytype σ.

The restriction on the domain of a λ-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 $\mathcal{L}\{\to\forall_{\mathsf{p}}\}$ with a primitive `let` construct whose statics is as follows:

$$\frac{\Delta \vdash \tau_1\ \mathsf{poly} \quad \Delta\ \Gamma \vdash e_1 : \tau_1 \quad \Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{let}[\tau_1](e_1;x.e_2) : \tau_2}\ . \tag{23.7}$$

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 τ.

23.4.3 Rank-Restricted Fragments

The binary distinction between monomorphic and polymorphic types in $\mathcal{L}^1\{\to\forall\}$ 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 significant difference between the type $\sigma \to \forall(t\,.\,\tau)$ and the type $\forall(t\,.\,\sigma \to \tau)$ (where $t \notin \sigma$). This motivates the definition of a hierarchy of fragments of $\mathcal{L}\{\to\forall\}$ that subsumes the prenex fragment as a special case.

We will define a judgement of the form $\tau\ \mathsf{type}\ [k]$, where $k \geq 0$, to mean that τ 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 23.4.2 on page 202, 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 judgements of the form $\Delta \vdash \tau\ \mathsf{type}\ [k]$, where Δ 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 judgements are as follows:

$$\frac{}{\Delta, t\ \mathsf{type}\ [k] \vdash t\ \mathsf{type}\ [k]} \tag{23.8a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type}\ [0] \quad \Delta \vdash \tau_2\ \mathsf{type}\ [0]}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2)\ \mathsf{type}\ [0]} \tag{23.8b}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type}\ [k] \quad \Delta \vdash \tau_2\ \mathsf{type}\ [k+1]}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2)\ \mathsf{type}\ [k+1]} \tag{23.8c}$$

$$\frac{\Delta \vdash \tau\ \mathsf{type}\ [k]}{\Delta \vdash \tau\ \mathsf{type}\ [k+1]} \tag{23.8d}$$

$$\frac{\Delta, t\ \mathsf{type}\ [k] \vdash \tau\ \mathsf{type}\ [k+1]}{\Delta \vdash \mathtt{all}(t\,.\,\tau)\ \mathsf{type}\ [k+1]} \tag{23.8e}$$

With these restrictions in mind, it is a good exercise to define the statics of $\mathcal{L}^k\{\to\forall\}$, the restriction of $\mathcal{L}\{\to\forall\}$ to types of rank k (or less). It is most convenient to consider judgements of the form $e : \tau\ [k]$ specifying simultaneously that $e : \tau$ and $\tau\ \mathsf{type}\ [k]$. For example, the rank-limited rules for

λ-abstractions is phrased as follows:

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type}\,[0] \quad \Delta\ \Gamma, x : \tau_1\,[0] \vdash e_2 : \tau_2\,[0]}{\Delta\ \Gamma \vdash \mathtt{lam[}\tau_1\mathtt{]\,(}x\mathtt{.}e_2\mathtt{)} : \mathtt{arr(}\tau_1\mathtt{;}\tau_2\mathtt{)}\,[0]} \tag{23.9a}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{type}\,[k] \quad \Delta\ \Gamma, x : \tau_1\,[k] \vdash e_2 : \tau_2\,[k+1]}{\Delta\ \Gamma \vdash \mathtt{lam[}\tau_1\mathtt{]\,(}x\mathtt{.}e_2\mathtt{)} : \mathtt{arr(}\tau_1\mathtt{;}\tau_2\mathtt{)}\,[k+1]} \tag{23.9b}$$

The remaining rules follow a similar pattern.

The rank-limited languages $\mathcal{L}^k\{\to\forall\}$ clarifies the requirement for a primitive `let` construct in $\mathcal{L}^1\{\to\forall\}$. The prenex fragment of $\mathcal{L}\{\to\forall\}$ corresponds to the rank-one fragment $\mathcal{L}^1\{\to\forall\}$. The `let` construct for rank-one types is definable in $\mathcal{L}^2\{\to\forall\}$ from λ-abstraction and application. This definition only makes sense at rank two, since it abstracts over a rank-one polymorphic type.

23.5 Exercises

1. Show that primitive recursion is definable in $\mathcal{L}\{\to\forall\}$ by exploiting the definability of iteration and binary products.

2. Investigate the representation of eager products and sums in eager and lazy variants of $\mathcal{L}\{\to\forall\}$.

3. Show how to write an interpreter for $\mathcal{L}\{\mathtt{nat}\to\}$ in $\mathcal{L}\{\to\forall\}$.

Chapter 24

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 $\mathcal{L}\{\rightarrow\forall\}$ 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 23.

24.1 Existential Types

The syntax of $\mathcal{L}\{\rightarrow\forall\exists\}$ is the extension of $\mathcal{L}\{\rightarrow\forall\}$ with the following constructs:

Type	τ	::=	$\texttt{some}(t.\tau)$	$\exists(t.\tau)$	interface
Expr	e	::=	$\texttt{pack}[t.\tau][\rho](e)$	$\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau)$	implementation
			$\texttt{open}[t.\tau][\rho](e_1;t,x.e_2)$	$\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x{:}\tau\,\texttt{in}\,e_2$	client

The introductory form for the existential type $\sigma = \exists(t.\tau)$ is a *package* of the form $\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau)$, where ρ is a type and e is an expression of type $[\rho/t]\tau$. The type ρ is called the *representation type* of the package, and the expression e is called the *implementation* of the package. The eliminatory form for existentials 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 α-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 α-equivalence, and is not particularly tied to data abstraction.

24.1.1 Statics

The statics of existential types is specified by rules defining when an existential is well-formed, and by giving typing rules for the associated introductory and eliminatory forms.

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \texttt{some}(t.\tau)\ \mathsf{type}} \tag{24.1a}$$

$$\frac{\Delta \vdash \rho\ \mathsf{type} \quad \Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \quad \Delta\ \Gamma \vdash e : [\rho/t]\tau}{\Delta\ \Gamma \vdash \mathtt{pack}\,[t\,.\,\tau]\,[\rho]\,(e) : \mathtt{some}\,(t\,.\,\tau)} \tag{24.1b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{some}\,(t\,.\,\tau) \quad \Delta, t\ \mathsf{type}\ \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta\ \Gamma \vdash \mathtt{open}\,[t\,.\,\tau]\,[\tau_2]\,(e_1; t, x\,.\,e_2) : \tau_2} \tag{24.1c}$$

Rule (24.1c) is complex, so study it carefully! There are two important things to notice:

1. The type of the client, τ_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 24.1 (Regularity)**.** *Suppose that* $\Delta\ \Gamma \vdash e : \tau$. *If* $\Delta \vdash \tau_i$ *type for each* $x_i : \tau_i$ *in* Γ, *then* $\Delta \vdash \tau$ *type.*

Proof. By induction on Rules (24.1). □

24.1.2 Dynamics

The (eager or lazy) dynamics of existential types is specified as follows:

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{pack}\,[t\,.\,\tau]\,[\rho]\,(e)\ \mathsf{val}} \tag{24.2a}$$

$$\left\{\frac{e \mapsto e'}{\mathtt{pack}\,[t\,.\,\tau]\,[\rho]\,(e) \mapsto \mathtt{pack}\,[t\,.\,\tau]\,[\rho]\,(e')}\right\} \tag{24.2b}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{open}\,[t\,.\,\tau]\,[\tau_2]\,(e_1; t, x\,.\,e_2) \mapsto \mathtt{open}\,[t\,.\,\tau]\,[\tau_2]\,(e_1'; t, x\,.\,e_2)} \tag{24.2c}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{open}\,[t\,.\,\tau]\,[\tau_2]\,(\mathtt{pack}\,[t\,.\,\tau]\,[\rho]\,(e); t, x\,.\,e_2) \mapsto [\rho, e/t, x]e_2} \tag{24.2d}$$

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.

24.1.3 Safety

The safety of the extension is stated and proved as usual. The argument is a simple extension of that used for $\mathcal{L}\{\to\forall\}$ to the new constructs.

Theorem 24.2 (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. By rule induction on $e \mapsto e'$, making use of substitution for both expression- and type variables. □

Lemma 24.3 (Canonical Forms)**.** *If e : $\texttt{some(}t\texttt{.}\tau\texttt{)}$ and e val, then $e = \texttt{pack[}t\texttt{.}\tau\texttt{][}\rho\texttt{](}e'\texttt{)}$ for some type ρ 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 24.4 (Progress)**.** *If $e : \tau$ then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By rule induction on $e : \tau$, making use of the canonical forms lemma. □

24.2 Data Abstraction Via Existentials

To illustrate the use of existentials for data abstraction, we consider an abstract type of queues of natural numbers supporting three operations:

1. Formation of the empty queue.
2. Inserting an element at the tail of the queue.
3. Remove the head of the queue.

This is clearly a bare-bones interface, but is sufficient to illustrate the main ideas of data abstraction. Queue elements may be taken to be of any type, τ, of our choosing; we will not be specific about this choice, since nothing depends on it.

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, $\exists\texttt{(}t\texttt{.}\tau\texttt{)}$, which serves as the interface of the queue abstraction:

$$\exists\texttt{(}t\texttt{.}\langle\texttt{emp}:t,\texttt{ins}:\texttt{nat}\times t\to t,\texttt{rem}:t\to\texttt{nat}\times t\rangle\texttt{)}.$$

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:

$$\texttt{pack}\,\texttt{list}\,\texttt{with}\,\langle\texttt{emp}=\texttt{nil},\texttt{ins}=e_i,\texttt{rem}=e_r\rangle\,\texttt{as}\,\exists(t\,.\,\tau),$$

where

$$e_i : \texttt{nat}\times\texttt{list}\to\texttt{list} = \lambda\,(x\texttt{:nat}\times\texttt{list}.\,e_i'),$$

and

$$e_r : \texttt{list}\to\texttt{nat}\times\texttt{list} = \lambda\,(x\texttt{:list}.\,e_r').$$

Here the expression e_i' conses the first component of x, the element, onto the second component of x, the queue. Correspondingly, the expression e_r' reverses its argument, and returns the head element paired with the reversal of the tail. These operations "know" that queues are represented as values of type `list`, 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:

$$\texttt{pack}\,\texttt{list}\times\texttt{list}\,\texttt{with}\,\langle\texttt{emp}=\langle\texttt{nil},\texttt{nil}\rangle,\texttt{ins}=e_i,\texttt{rem}=e_r\rangle\,\texttt{as}\,\exists(t\,.\,\tau).$$

In this case e_i has type

$$\texttt{nat}\times(\texttt{list}\times\texttt{list})\to(\texttt{list}\times\texttt{list}),$$

and e_r has type

$$(\texttt{list}\times\texttt{list})\to\texttt{nat}\times(\texttt{list}\times\texttt{list}).$$

These operations "know" that queues are represented as values of type `list` × `list`, 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 `list` or `list` × `list` or some other type. That is, the client is *independent* of the representation of the abstract type.

24.3 Definability of Existentials

It turns out that it is not necessary to extend $\mathcal{L}\{\to\forall\}$ with existential types to model data abstraction, because they are already definable using only universal types! Before giving the details, let us consider why this should be possible. The key is to observe that the client of an abstract type is *polymorphic* in the representation type. The typing rule for

$$\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x\texttt{:}\sigma\,\texttt{in}\,e_2 : \tau,$$

where $e_1 : \exists(t.\sigma)$, specifies that $e_2 : \tau$ under the assumptions t type and $x : \sigma$. In essence, the client is a polymorphic function of type

$$\forall(t.\sigma \to \tau),$$

where t may occur in σ (the type of the operations), but not in τ (the type of the result).

This suggests the following encoding of existential types:

$$\begin{aligned}
\exists(t.\sigma) &= \forall(u.\forall(t.\sigma \to u) \to u) \\
\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\sigma) &= \Lambda(u.\lambda\,(x\texttt{:}\forall(t.\sigma \to u).\,x\texttt{[}\rho\texttt{]}\,(e))) \\
\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x\texttt{:}\sigma\,\texttt{in}\,e_2 &= e_1\texttt{[}\tau\texttt{]}\,(\Lambda(t.\lambda\,(x\texttt{:}\sigma.\,e_2)))
\end{aligned}$$

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, τ, and applies it to the polymorphic client. (The translation therefore depends on knowing the overall result type, τ, of the `open` construct.) Finally, a package consisting of a representation type ρ and an implementation e is a polymorphic function that, when given the result type, t, and the client, x, instantiates x with ρ and passes to it the implementation e.

It is then a straightforward exercise to show that this translation correctly reflects the statics and dynamics of existential types.

24.4 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. Since 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 24.3 on the preceding page. It is an immediate consequence of the definition that the client of an abstract type is polymorphic in the representation of the abstract type. A client, c, of an abstract type $\exists(t.\sigma)$ has type $\forall(t.\sigma \rightarrow \tau)$, where t does not occur free in τ (but may, of course, occur in σ). Applying the parametricity property described informally in Chapter 23 (and developed rigorously in Chapter 53), 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 operation that implement it.

To see what this means requires that we specify what is meant by a bisimulation. This is best done by example. So suppose that σ is the type

$$\langle \texttt{emp} : t, \texttt{ins} : \tau \times t \rightarrow t, \texttt{rem} : t \rightarrow \tau \times t \rangle.$$

Theorem 53.8 on page 497 ensures that if ρ and ρ' are any two closed types, R is a relation between expressions of these two types, then if any the implementations $e : [\rho/x]\sigma$ and $e' : [\rho'/x]\sigma$ 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

$$e = \langle \texttt{emp} = e_{\mathsf{m}}, \texttt{ins} = e_{\mathsf{i}}, \texttt{rem} = e_{\mathsf{r}} \rangle$$

and

$$e' = \langle \texttt{emp} = e'_{\mathsf{m}}, \texttt{ins} = e'_{\mathsf{i}}, \texttt{rem} = e'_{\mathsf{r}} \rangle.$$

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 $R(e_{\mathsf{i}}\texttt{(}d\texttt{)(}q\texttt{)}, e'_{\mathsf{i}}\texttt{(}d\texttt{)(}q'\texttt{)})$.

3. If two queues are related, their front elements are the same and their back elements are related: if $R(q, q')$, $e_{\mathsf{r}}\texttt{(}q\texttt{)} \cong \langle d, r\rangle$, $e'_{\mathsf{r}}\texttt{(}q'\texttt{)} \cong \langle d', r'\rangle$, then d is 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 ρ to be the type `list`, and define the empty queue to be the empty list, insert to add the specified element to the front of the list, and remove to remove the last element of the list. (A remove therefore takes time linear in the length of the list.) For the candidate implementation we choose ρ' to be the type `list` $\times$ `list` consisting of two lists, $\langle b, f\rangle$, where b represents the "back" of the queue, and f represents the "front" of the queue represented in reverse order of insertion. The empty queue consists of two empty lists. To insert d onto $\langle b, f\rangle$, we simply return $\langle \texttt{cons(}d\texttt{;}b\texttt{)}, f\rangle$, placing it on the "back" of the queue as expected. To remove an element from $\langle b, f\rangle$ breaks into two cases. If the front, f, of the queue is non-empty, say $\texttt{cons(}d\texttt{;}f'\texttt{)}$, then return $\langle d, \langle b, f'\rangle\rangle$ consisting of the front element and the queue with that element removed. If, on the other hand, f is empty, then we must move elements from the "back" to the "front" by reversing b and re-performing the remove operation on $\langle \texttt{nil}, \texttt{rev(}b\texttt{)}\rangle$, where `rev` is the obvious list reversal function.

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 `list` and `list` $\times$ `list` 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 $\texttt{app(}b\texttt{)(rev(}f\texttt{))}$, 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. Since 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.

24.5 Exercises

Chapter 25

Constructors and Kinds

Types such as $\tau_1 \to \tau_2$ or $\tau\,\texttt{list}$ 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 `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 operator.

These seemingly disparate cases may be treated uniformly by enriching the syntactic structure of a language with a new layer of *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, `Type`, 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. Since the dynamic phase follows the static phase (we only execute well-typed programs), we may also manipulate constructors at run-time.

Adding constructors and kinds to a language introduces more technical complications than might at first be apparent. The main difficulty is that as soon as we enrich the kind structure beyond the distinguished kind of types, it becomes essential to simplify constructors to determine whether they are equivalent. For example, if we admit product kinds, then a pair of constructors is a constructor of product kind, and projections from a constructor of product kind are also constructors. But what if we form the first projection from the pair consisiting of the constructors `nat` and `str`? This should be equivalent to `nat`, since the elimination form if post-inverse to the introduction form. Consequently, any expression (say, a variable) of the one type should also be an expression of the other. That is, typing should respect definitional equivalence of constructors.

There are two main ways to deal with this. One is to introduce a concept of definitional equivalence for constructors, and to demand that the typing judgement for expressions respect definitional equivalence of constructors of kind `Type`. This means, however, that we must show that definitional equivalence is decidable if we are to build a complete implementation of the language. The other is to prohibit formation of awkward constructors such as the projection from a pair so that there is never any issue of when two constructors are equivalent (only when they are identical). But this complicates the definition of substitution, since a projection from a constructor variable is well-formed, until you substitute a pair for the variable. Both approaches have their benefits, but the second is simplest, and is adopted here.

25.1 Statics

The syntax of kinds is given by the following grammar:

Kind	κ	::=	`Type`	`Type`	types
			`Unit`	1	nullary product
			`Prod(`κ_1`;`κ_2`)`	$\kappa_1 \times \kappa_2$	binary product
			`Arr(`κ_1`;`κ_2`)`	$\kappa_1 \to \kappa_2$	function

The kinds consist of the kind of types, `Type`, the unit kind, `Unit`, and are closed under formation of product and function kinds.

The syntax of constructors is divided into two syntactic sorts, the *neutral* and the *canonical*, according to the following grammar:

Neut	a	$::=$	u	u	variable
			`proj[l]`(a)	`pr`$_{\texttt{l}}(a)$	first projection
			`proj[r]`(a)	`pr`$_{\texttt{r}}(a)$	second projection
			`app`$(a_1;c_2)$	$a_1[c_2]$	application
Canon	c	$::=$	`atom`(a)	$\widehat{a}$	atomic
			`unit`	$\langle\rangle$	null tuple
			`pair`$(c_1;c_2)$	$\langle c_1, c_2\rangle$	pair
			`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 `pr`$_{\texttt{l}}(\langle c_1, c_2\rangle)$, which would be definitionally equivalent to c_1, is ill-formed according to Grammar (25.1). This is because the argument to a projection must be neutral, but a pair is only canonical, not neutral.

The canonical constructor `atom`(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 `Type` may be treated as canonical. This requirement is imposed to limit the forms of canonical contructors 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 is specified by the judgements

$\Delta \vdash a \Uparrow \kappa$	neutral constructor formation
$\Delta \vdash c \Downarrow \kappa$	canonical constructor formation

In each of these judgements Δ 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 judgements are to be understood as generic hypothetical judgements with parameters $u_1, \ldots, u_n$ that are determined by the forms of the hypotheses.

The rules for constructor formation are as follows:

$$\frac{}{\Delta, u \Uparrow \kappa \vdash u \Uparrow \kappa} \qquad (25.1a)$$

$$\frac{\Delta \vdash a \Uparrow \kappa_1 \times \kappa_2}{\Delta \vdash \mathtt{pr}_{\mathtt{l}}(a) \Uparrow \kappa_1} \tag{25.1b}$$

$$\frac{\Delta \vdash a \Uparrow \kappa_1 \times \kappa_2}{\Delta \vdash \mathtt{pr}_{\mathtt{r}}(a) \Uparrow \kappa_2} \tag{25.1c}$$

$$\frac{\Delta \vdash a_1 \Uparrow \kappa_2 \to \kappa \quad \Delta \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash a_1\,[c_2] \Uparrow \kappa} \tag{25.1d}$$

$$\frac{\Delta \vdash a \Uparrow \mathtt{Type}}{\Delta \vdash \widehat{a} \Downarrow \mathtt{Type}} \tag{25.1e}$$

$$\frac{}{\Delta \vdash \langle\rangle \Downarrow 1} \tag{25.1f}$$

$$\frac{\Delta \vdash c_1 \Downarrow \kappa_1 \quad \Delta \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle \Downarrow \kappa_1 \times \kappa_2} \tag{25.1g}$$

$$\frac{\Delta, u \Uparrow \kappa_1 \vdash c_2 \Downarrow \kappa_2}{\Delta \vdash \lambda\, u\,.\,c_2 \Downarrow \kappa_1 \to \kappa_2} \tag{25.1h}$$

Rule (25.1e) specifies that the only neutral constructors that are canonical are those with kind Type. This ensures that the language enjoys the following canonical forms property, which is easily proved by inspection of Rules (25.1).

Lemma 25.1. *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 $c = \lambda\, u\,.\,c_2$ with $\Delta, u \Uparrow \kappa_1 \vdash c_2 \Downarrow \kappa_2$.*

25.2 Adding Constructors and Kinds

To equip a language, $\mathcal{L}$, with constructors and kinds requires that we augment its statics with hypotheses governing constructor variables, and that we relate constructors of kind Type (types as static data) to the classifiers of dynamic expressions (types as classifiers). To achieve this the statics of $\mathcal{L}$ must be defined to have judgements of the following two forms:

$\Delta \vdash \tau\ \mathsf{type}$	type formation
$\Delta\, \Gamma \vdash e : \tau$	expression formation

where, as before, Γ is a finite set of hypotheses of the form

$$x_1 : \tau_1, \dots, x_k : \tau_k$$

for some $k \geq 0$ such that $\Delta \vdash \tau_i$ type for each $1 \leq i \leq k$.

As a general principle, every constructor of kind Type is a classifier:

$$\frac{\Delta \vdash \tau \Uparrow \mathtt{Type}}{\Delta \vdash \tau \ \mathsf{type}} \ . \tag{25.2}$$

In many cases this is the sole rule of type formation, so that every classifier is a constructor of kind Type. However, this need not be the case. In some situations we may wish to have strictly more classifiers than constructors of the distinguished kind.

To see how this might arise, let us consider two extensions of $\mathcal{L}\{\rightarrow\forall\}$ from Chapter 23. In both cases we extend the universal quantifier $\forall(t\,.\,\tau)$ to admit quantification over an arbitrary kind, written $\forall_\kappa\, u\,.\,\tau$, but the two languages differ in what constitutes a constructor of kind Type. In one case, the *impredicative*, we admit quantified types as constructors, and in the other, the *predicative*, we exclude quantified types from the domain of quantification.

The impredicative fragment includes the following two constructor constants:

$$\frac{}{\Delta \vdash \rightarrow \Uparrow \mathtt{Type} \rightarrow \mathtt{Type} \rightarrow \mathtt{Type}} \tag{25.3a}$$

$$\frac{}{\Delta \vdash \forall_\kappa \Uparrow (\kappa \rightarrow \mathtt{Type}) \rightarrow \mathtt{Type}} \tag{25.3b}$$

We regard the classifier $\tau_1 \rightarrow \tau_2$ to be the application $\rightarrow[\tau_1]\,[\tau_2]$. Similarly, we regard the classifier $\forall_\kappa\, u\,.\,\tau$ to be the application $\forall_\kappa\,[\lambda\, u\,.\,\tau]$.

The predicative fragment excludes the constant specified by Rule (25.3b) in favor of a separate rule for the formation of universally quantified types:

$$\frac{\Delta, u \Uparrow \kappa \vdash \tau \ \mathsf{type}}{\Delta \vdash \forall_\kappa\, u\,.\,\tau \ \mathsf{type}} \ . \tag{25.4}$$

The important point is that $\forall_\kappa\, u\,.\,\tau$ is a type (as classifier), but is *not* a constructor of kind type.

The signficance of this distinction becomes apparent when we consider the introduction and elimination forms for the generalized quantifier, which are the same for both fragments:

$$\frac{\Delta, u \Uparrow \kappa\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \Lambda(u::\kappa\,.\,e) : \forall_\kappa\, u\,.\,\tau} \tag{25.5a}$$

$$\frac{\Delta\;\Gamma \vdash e : \forall_{\kappa}\, u\,.\,\tau \quad \Delta \vdash c \Downarrow \kappa}{\Delta\;\Gamma \vdash e\,[c] : [c/u]\tau} \tag{25.5b}$$

(Rule (25.5b) makes use of substitution, whose definition requires some care. We will return to this point in Section 25.3.)

Rule (25.5b) makes clear that a polymorphic abstraction quantifies over the constructors of kind κ. When κ is `Type` 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 `Type`) or the predicative formulation (in which quantifiers arise only as classifiers and not as constructors).

The important principle here is that *constructors are static data*, so that a constructor abstraction $\Lambda(u::\kappa\,.\,e)$ of type $\forall_{\kappa}\, u\,.\,\tau$ is a mapping from static data c of kind κ to dynamic data $[c/u]e$ of type $[c/u]\tau$. Rule (25.1e) tells us that every constructor of kind `Type` determines a classifier, but it may or may not be the case that every classifier arises in this manner.

25.3 Substitution

Rule (25.5b) involves substitution of a canonical constructor, c, of kind κ into a family of types $u \Uparrow \kappa \vdash \tau$ type. This operation is 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 3. 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

$$u \Uparrow \texttt{Type} \to \texttt{Type} \vdash u\,[d] \Uparrow \texttt{Type},$$

where $d \Uparrow$ `Type`. (It is not important what we choose for d, so we leave it abstract.) Now if $c \Downarrow \texttt{Type} \to \texttt{Type}$, then by Lemma 25.1 on page 220 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 definitely smaller than the kind of u, since the former's kind is the domain kind of the latter's function kind. 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 well-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

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.

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa]\widehat{a'} = \widehat{a''}} \tag{25.6a}$$

$$\frac{[c/u:\kappa]a' = c'' \Downarrow \kappa''}{[c/u:\kappa]\widehat{a'} = c''} \tag{25.6b}$$

$$\frac{}{[u/\langle\rangle:\kappa]{=}\langle\rangle} \tag{25.6c}$$

$$\frac{[c/u:\kappa]c_1' = c_1'' \quad [c/u:\kappa]c_2' = c_2''}{[c/u:\kappa]\langle c_1', c_2'\rangle = \langle c_1'', c_2''\rangle} \tag{25.6d}$$

$$\frac{[c/u:\kappa]c' = c'' \quad (u \neq u') \quad (u' \notin c)}{[c/u:\kappa]\lambda\, u' \,.\, c' = \lambda\, u' \,.\, c''} \tag{25.6e}$$

The conditions on variables in Rule (25.6e) 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.

$$\frac{(u \neq u')}{[c/u:\kappa]u' = u'} \tag{25.7a}$$

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa]\mathtt{pr}_{\mathtt{l}}(a') = \mathtt{pr}_{\mathtt{l}}(a'')} \tag{25.7b}$$

$$\frac{[c/u:\kappa]a' = a''}{[c/u:\kappa]\mathtt{pr}_{\mathtt{r}}(a') = \mathtt{pr}_{\mathtt{r}}(a'')} \tag{25.7c}$$

$$\frac{[c/u:\kappa]a_1 = a_1' \quad [c/u:\kappa]c_2 = c_2'}{[c/u:\kappa]a_1[c_2] = a_1'(c_2')} \tag{25.7d}$$

Rule (25.7a) 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.

$$\frac{}{[c/u:\kappa]u = c \Downarrow \kappa} \tag{25.8a}$$

$$\frac{[c/u:\kappa]a' = \langle c_1', c_2' \rangle \Downarrow \kappa_1' \times \kappa_2'}{[c/u:\kappa]\mathtt{pr}_{\mathtt{l}}(a') = c_1' \Downarrow \kappa_1'} \tag{25.8b}$$

$$\frac{[c/u:\kappa]a' = \langle c_1', c_2' \rangle \Downarrow \kappa_1' \times \kappa_2'}{[c/u:\kappa]\mathtt{pr}_{\mathtt{r}}(a') = c_2' \Downarrow \kappa_2'} \tag{25.8c}$$

$$\frac{[c/u:\kappa]a_1' = \lambda\, u'.c' \Downarrow \kappa_2' \to \kappa' \quad [c/u:\kappa]c_2' = c_2'' \quad [c_2''/u':\kappa_2']c' = c''}{[c/u:\kappa]a_1'[c_2'] = c'' \Downarrow \kappa'} \tag{25.8d}$$

Rule (25.8a) 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 (25.8d) 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 25.2. *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 component the structure of the kind κ, and with minor component determined by Rules (25.1) governing the formation of c' and a'. For all rules except Rule (25.8d) the inductive hypothesis applies to the premise(s) of the relevant formation rules. For Rule (25.8d) we appeal to the major inductive hypothesis applied to κ_2', which is a component of the kind $\kappa_2' \rightarrow \kappa'$. □

25.4 Exercises

Chapter 26

Indexed Families of Types

26.1 Type Families

26.2 Exercises

Part IX

Subtyping

Chapter 27

Subtyping

A *subtype* relation is a pre-order (reflexive and transitive relation) on types that validates the *subsumption principle*:

> if σ is a subtype of τ, then a value of type σ may be provided whenever a value of type τ 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, while 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 this can lead to serious errors, because it fails to take account of the operations (eliminatory forms) that one can perform on values of the supertype. It is not enough to think only of the introductory forms; one must also think of the eliminatory forms. Subtyping is a matter of *behavior*, rather than *containment*.

27.1 Subsumption

A *subtyping judgement* has the form $\sigma <: \tau$, and states that σ is a subtype of τ. At a minimum we demand that the following *structural rules* of subtyping be admissible:

$$\frac{}{\tau <: \tau} \tag{27.1a}$$

$$\frac{\rho <: \sigma \quad \sigma <: \tau}{\rho <: \tau} \tag{27.1b}$$

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*:

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau} \tag{27.2}$$

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 $e : \sigma$ and $\sigma <: \tau$.

27.2 Varieties of Subtyping

In this section we will informally explore several different forms of subtyping for various extensions of $\mathcal{L}\{\rightharpoonup\}$. In Section 27.4 on page 240 we will examine some of these in more detail from the point of view of type safety.

27.2.1 Numeric Types

For languages with numeric types, our mathematical experience suggests subtyping relationships among them. 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

$$\mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$$

familiar from mathematical experience.

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 a somewhat generalized 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 out properly. For the purposes of computing, however, one cannot be quite so cavalier, because we must also take account of algorithmic efficiency and the finiteness of machine representations. Often what are called "real numbers" in a programming language are, in fact, finite precision floating point numbers, a small 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.

27.2.2 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 $\tau = \prod_{j \in J} \tau_j$ is required. The statics of finite products (Rules (14.3)) ensures that the only operation we may perform on a value of type τ, other than to bind it to a variable, is to take the jth projection from it for some $j \in J$ to obtain a value of type τ_j.

Now suppose that e is of type σ. If the projection $e \cdot j$ is to be well-formed, then σ must be a finite product type $\prod_{i \in I} \sigma_i$ such that $j \in I$. Moreover, for this to be of type τ_j, it is enough to require that $\sigma_j = \tau_j$. Since $j \in J$ is arbitrary, we arrive at the following subtyping rule for finite product types:

$$\frac{J \subseteq I}{\prod_{i \in I} \tau_i <: \prod_{j \in J} \tau_j} \qquad (27.3)$$

It is sufficient, but not necessary, to require that $\sigma_j = \tau_j$ for each $j \in J$; we will consider a more liberal form of this rule in Section 27.3 on page 236.

The argument for Rule (27.3) is based on a dynamics in which we may evaluate $e \cdot j$ regardless of the actual form of e, provided only that it has a field indexed by $j \in J$. Is this a reasonable assumption?

One common case is that I and J are initial segments of the natural numbers, say $I = [0..m-1]$ and $J = [0..n-1]$, so that the product types may be thought of as m- and n-tuples, respectively. The containment $I \subseteq J$ amounts to requiring that $m \geq n$, which is to say that a tuple type is regarded as a subtype of all of its prefixes. When specialized to this case, Rule (27.3) may be stated in the form

$$\frac{m \geq n}{\langle \tau_1, \ldots, \tau_m \rangle <: \langle \tau_1, \ldots, \tau_n \rangle} \qquad (27.4)$$

One way to justify this rule is to consider elements of the subtype to be consecutive sequences of values of type $\tau_0, \ldots, \tau_{m-1}$ from which we may calculate the jth projection for any $0 \leq j < n \leq m$, regardless of whether or not m is strictly bigger than n.

Another common case is when I and J are finite sets of symbols, so that projections are based on the field name, rather than its position. When specialized to this case, Rule (27.3) takes the following form:

$$\frac{m \geq n}{\langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} \qquad (27.5)$$

Here we are taking advantage of the implicit identification of labeled tuple types up to reordering of fields, so that the rule states that any field of the supertype must be present in the subtype with the same type.

When using symbolic labels for the components of a tuple, it is perhaps slightly less clear that Rule (27.5) is well-justified. After all, how are we to find field l_i, where $0 \leq i < n$, in a labeled tuple that may have additional fields anywhere within it? The trouble is that the label does not reveal the position of the field within the tuple, precisely because of subtyping. One way to achieve this is to associate with a labeled tuple a *dictionary* mapping labels to positions within the tuple, which the projection operation uses to find the appropriate component of the record. Since the labels are fixed statically, this may be done in constant time using a perfect hashing function mapping labels to natural numbers, so that the cost of a projection remains constant. Another method is to use *coercions* that a value of the subtype to a value of the supertype whenever subsumption is used. In the case of labeled tuples this means creating a new labeled tuple containing only the fields of the supertype, copied from those of the subtype, so that the type specifies exactly the fields present in the value. This allows for more efficient implementation (for example, by a simple offset calculation), but is not compatible with languages that permit mutation (in-place modification) of fields because it destroys sharing.

27.2.3 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 (15.3)) 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 $\sum_{i\in I}\sigma_i$ instead, no difficulty will arise so long as $I \subseteq J$ and each σ_i is equal to τ_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:

$$\frac{I \subseteq J}{\sum_{i\in I}\tau_i <: \sum_{j\in J}\tau_j}\,. \tag{27.6}$$

Note well the reversal of the containment as compared to Rule (27.3).

When I and J are initial segments of the natural numbers, we obtain the following special case of Rule (27.6):

$$\frac{m \leq n}{[l_1 : \tau_1, \ldots, l_m : \tau_m] <: [l_1 : \tau_1, \ldots, l_n : \tau_n]} \tag{27.7}$$

One may also consider a form of width subtyping for unlabeled n-ary sums, by considering any prefix of an n-ary sum to be a subtype of that sum. Here again the elimination form for the supertype, namely an n-ary case analysis, is prepared to handle any value of the subtype, which is enough to ensure type safety.

27.3 Variance

In addition to basic subtyping principles such as those considered in Section 27.2 on page 232, 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.

27.3.1 Product Types

Finite product types are *covariant* in each field. For if e is of type $\prod_{i\in I}\sigma_i$, and the projection $e \cdot j$ is expected to be of type τ_j, then it is sufficient to require that $j \in I$ and $\sigma_j <: \tau_j$. This is summarized by the following rule:

$$\frac{(\forall i \in I)\ \sigma_i <: \tau_i}{\prod_{i\in I}\sigma_i <: \prod_{i\in I}\tau_i} \tag{27.8}$$

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.

When specialized to n-tuples, Rule (27.8) reads as follows:

$$\frac{\sigma_1 <: \tau_1 \quad \dots \quad \sigma_n <: \tau_n}{\langle \sigma_1, \dots, \sigma_n \rangle <: \langle \tau_1, \dots, \tau_n \rangle}\ . \tag{27.9}$$

When specialized to symbolic labels, the covariance principle for finite products may be re-stated as follows:

$$\frac{\sigma_1 <: \tau_1 \quad \dots \quad \sigma_n <: \tau_n}{\langle l_1 : \sigma_1, \dots, l_n : \sigma_n \rangle <: \langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle}\ . \tag{27.10}$$

27.3.2 Sum Types

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:

$$\frac{(\forall i \in I)\ \sigma_i <: \tau_i}{\sum_{i\in I}\sigma_i <: \sum_{i\in I}\tau_i} \tag{27.11}$$

When specialized to symbolic labels as index sets, we obtain the following formulation of the covariance principle for sum types:

$$\frac{\sigma_1 <: \tau_1 \quad \dots \quad \sigma_n <: \tau_n}{[l_1 : \sigma_1, \dots, l_n : \sigma_n] <: [l_1 : \tau_1, \dots, l_n : \tau_n]} \,. \tag{27.12}$$

A case analysis on a value of the supertype is prepared, in the *i*th branch, to accept a value of type τ_i. By the premises of the rule, it is sufficient to provide a value of type σ_i instead.

27.3.3 Function Types

The variance of the function type constructor is a bit more subtle. Let us consider first the variance of the function type in its range. Suppose that $e : \sigma \to \tau$. This means that if $e_1 : \sigma$, then $e(e_1) : \tau$. If $\tau <: \tau'$, then $e(e_1) : \tau'$ as well. This suggests the following covariance principle for function types:

$$\frac{\tau <: \tau'}{\sigma \to \tau <: \sigma \to \tau'} \tag{27.13}$$

Every function that delivers a value of type τ must also deliver a value of type τ', provided that $\tau <: \tau'$. 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 $e : \sigma \to \tau$. This means that e may be applied to any value of type σ, and hence, by the subsumption principle, it may be applied to any value of any subtype, σ', of σ. In either case it will deliver a value of type τ. Consequently, we may just as well think of e as having type $\sigma' \to \tau$.

$$\frac{\sigma' <: \sigma}{\sigma \to \tau <: \sigma' \to \tau} \tag{27.14}$$

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:

$$\frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \to \tau <: \sigma' \to \tau'} \tag{27.15}$$

Beware of the reversal of the ordering in the domain!

27.3.4 Recursive Types

The variance principle for recursive types is rather subtle, and has been the source of errors in language design. To gain some intuition, consider the type of labeled binary trees with natural numbers at each node,

$$\mu t\,.\,\texttt{[empty}:\texttt{unit},\texttt{binode}:\langle\texttt{data}:\texttt{nat},\texttt{lft}:t,\texttt{rht}:t\rangle\texttt{]},$$

and the type of "bare" binary trees, without labels on the nodes,

$$\mu t\,.\,\texttt{[empty}:\texttt{unit},\texttt{binode}:\langle\texttt{lft}:t,\texttt{rht}:t\rangle\texttt{]}.$$

Is either a subtype of the other? Intuitively, one might expect the type of labeled binary trees to be a *subtype* of the type of bare binary trees, since 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:

$$\mu t\,.\,\texttt{[empty}:\texttt{unit},\texttt{binode}:\langle\texttt{lft}:t,\texttt{rht}:t\rangle,\texttt{trinode}:\langle\texttt{lft}:t,\texttt{mid}:t,\texttt{rht}:t\rangle\texttt{]}.$$

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, since 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:

$$\frac{t\ \mathsf{type} \vdash \sigma <: \tau}{\mu t\,.\,\sigma <: \mu t\,.\,\tau}\ \textbf{??} \tag{27.16}$$

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 $\sigma <: \tau$.

Rule (27.16) validates the intuitively plausible subtyping between labeled binary tree and bare binary trees just described. To derive this reduces to checking the subtyping relationship

$$\langle\texttt{data}:\texttt{nat},\texttt{lft}:t,\texttt{rht}:t\rangle <: \langle\texttt{lft}:t,\texttt{rht}:t\rangle,$$

generically in t, which is evidently the case.

Unfortunately, Rule (27.16) also underwrites *incorrect* subtyping relationships, as well as some correct ones. As an example of what goes wrong, consider the recursive types

$$\sigma = \mu t\,.\,\langle \mathtt{a} : t \rightarrow \mathtt{nat}, \mathtt{b} : t \rightarrow \mathtt{int} \rangle$$

and

$$\tau = \mu t\,.\,\langle \mathtt{a} : t \rightarrow \mathtt{int}, \mathtt{b} : t \rightarrow \mathtt{int} \rangle.$$

We assume for the sake of the example that $\mathtt{nat} <: \mathtt{int}$, so that by using Rule (27.16) we may derive $\sigma <: \tau$, which we will show to be incorrect. Let $e : \sigma$ be the expression

$$\mathtt{fold}(\langle \mathtt{a} = \lambda\,(x{:}\sigma.\,\mathtt{4}), \mathtt{b} = \lambda\,(x{:}\sigma.\,q((\mathtt{unfold}(x)\cdot\mathtt{a})(x)))\rangle),$$

where $q : \mathtt{nat} \rightarrow \mathtt{nat}$ is the discrete square root function. Since $\sigma <: \tau$, it follows that $e : \tau$ as well, and hence

$$\mathtt{unfold}(e) : \langle \mathtt{a} : \tau \rightarrow \mathtt{int}, \mathtt{b} : \tau \rightarrow \mathtt{int} \rangle.$$

Now let $e' : \tau$ be the expression

$$\mathtt{fold}(\langle \mathtt{a} = \lambda\,(x{:}\tau.\,\mathtt{-4}), \mathtt{b} = \lambda\,(x{:}\tau.\,\mathtt{0})\rangle).$$

(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

$$(\mathtt{unfold}(e)\cdot\mathtt{b})(e') \mapsto^* q(\mathtt{-4}),$$

which is a stuck state. We have derived a well-typed program that "gets stuck", refuting type safety!

Rule (27.16) 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 maintain a finite set, Ψ, of hypotheses of the form

$$s_1 <: t_1, \ldots, s_n <: t_n,$$

which is used to state the rule of subsumption for recursive types:

$$\frac{\Psi, s <: t \vdash \sigma <: \tau}{\Psi \vdash \mu s \,.\, \sigma <: \mu t \,.\, \tau} \,. \tag{27.17}$$

That is, to check whether $\mu s \,.\, \sigma <: \mu t \,.\, \tau$, we assume that $s <: t$, since s and t stand for the respective recursive types, and check that $\sigma <: \tau$ under this assumption.

We tacitly include the rule of reflexivity for subtyping assumptions,

$$\frac{}{\Psi, s <: t \vdash s <: t} \tag{27.18}$$

Using reflexivity in conjunction with Rule (27.17), we may verify the subtypings among the tree types sketched above. Moreover, it is instructive to check that the unsound subtyping is *not* derivable using this rule. The reason is that the assumption of the subtyping relation is at odds with the contravariance of the function type in its domain.

27.4 Safety for Subtyping

Proving safety for a language with subtyping is considerably more delicate than for languages without. The rule of subsumption means that the static type of an expression reveals only partial information about the underlying value. This changes the proof of the preservation and progress theorems, and requires some care in stating and proving the auxiliary lemmas required for the proof.

As a representative case we will sketch the proof of safety for a language with subtyping for product types. The subtyping relation is defined by Rules (27.3) and (27.8). We assume that the statics includes subsumption, Rule (27.2).

Lemma 27.1 (Structurality)**.**

1. *The tuple subtyping relation is reflexive and transitive.*

2. *The typing judgement $\Gamma \vdash e : \tau$ 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 judgements $\rho <: \sigma$ and $\sigma <: \tau$ to obtain a derivation of $\rho <: \tau$.

2. By induction on Rules (14.3), augmented by Rule (27.2).

□

Lemma 27.2 (Inversion)**.**

1. *If $e \cdot j : \tau$, then $e : \prod_{i \in I} \tau_i$, $j \in I$, and $\tau_j <: \tau$.*

2. *If $\langle e_i \rangle_{i \in I} : \tau$, then $\prod_{i \in I} \sigma_i <: \tau$ where $e_i : \sigma_i$ for each $i \in I$.*

3. *If $\sigma <: \prod_{j \in J} \tau_j$, then $\sigma = \prod_{i \in I} \sigma_i$ for some I and some types σ_i for $i \in I$.*

4. *If $\prod_{i \in I} \sigma_i <: \prod_{j \in J} \tau_j$, then $J \subseteq I$ and $\sigma_j <: \tau_j$ for each $j \in J$.*

Proof. By induction on the subtyping and typing rules, paying special attention to Rule (27.2). □

Theorem 27.3 (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Proof. By induction on Rules (14.4). For example, consider Rule (14.4d), so that $e = \langle e_i \rangle_{i \in I} \cdot k$, $e' = e_k$. By Lemma 27.2 we have that $\langle e_i \rangle_{i \in I} : \prod_{j \in J} \tau_j$, $k \in J$, and $\tau_k <: \tau$. By another application of Lemma 27.2 for each $i \in I$ there exists σ_i such that $e_i : \sigma_i$ and $\prod_{i \in I} \sigma_i <: \prod_{j \in J} \tau_j$. By Lemma 27.2 again, we have $J \subseteq I$ and $\sigma_j <: \tau_j$ for each $j \in J$. But then $e_k : \tau_k$, as desired. The remaing cases are similar. □

Lemma 27.4 (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 (14.3) augmented by Rule (27.2). □

Theorem 27.5 (Progress)**.** *If $e : \tau$, then either e val or there exists e' such that $e \mapsto e'$.*

Proof. By induction on Rules (14.3) augmented by Rule (27.2). The rule of subsumption is handled by appeal to the inductive hypothesis on the premise of the rule. Rule (14.4d) follows from Lemma 27.4. □

To account for recursive subtyping in addition to finite product subtyping, the following inversion lemma is required.

Lemma 27.6.

1. *If $\Psi, s <: t \vdash \sigma' <: \tau'$ and $\Psi \vdash \sigma <: \tau$, then $\Psi \vdash [\sigma/s]\sigma' <: [\tau/t]\tau'$.*

2. *If $\Psi \vdash \sigma <: \mu t\,.\,\tau'$, then $\sigma = \mu s\,.\,\sigma'$ and $\Psi, s <: t \vdash \sigma' <: \tau'$.*

3. *If $\Psi \vdash \mu s\,.\,\sigma <: \mu t\,.\,\tau$, then $\Psi \vdash [\mu s\,.\,\sigma/s]\sigma <: [\mu t\,.\,\tau/t]\tau$.*

4. *The subtyping relation is reflexive and transitive, and closed under weakening.*

Proof.

1. By induction on the derivation of the first premise. Wherever the assumption is used, replace it by $\sigma <: \tau$, and propagate forward.

2. By induction on the derivation of $\sigma <: \mu t\,.\,\tau$.

3. Follows immediately from the preceding two properties of subtyping.

4. Reflexivity is proved by construction. Weakening is proved by an easy induction on subtyping derivations. Transitivity is proved by induction on the sizes of the types involved. For example, suppose we have $\Psi \vdash \mu r\,.\,\rho <: \mu s\,.\,\sigma$ because $\Psi, r <: s \vdash \rho <: \sigma$, and $\Psi \vdash \mu s\,.\,\sigma <: \mu t\,.\,\tau$ because and $\Psi, s <: t \vdash \sigma <: \tau$. We may assume without loss of generality that s does not occur free in either ρ or τ. By weakening we have $\Psi, r <: s, s <: t \vdash \rho <: \sigma$ and $\Psi, r <: s, s <: t \vdash \sigma <: \tau$. Therefore by induction we have $\Psi, r <: s, s <: t \vdash \rho <: \tau$. But since $\Psi, r <: t \vdash r <: t$ and $\Psi, r <: t \vdash t <: t$, we have by the first property above that $\Psi, r <: t \vdash \rho <: \tau$, from which the result follows immediately.

□

The remainder of the proof of type safety in the presence of recursive subtyping proceeds along lines similar to that for product subtyping.

27.5 Exercises

Chapter 28

Singleton and Dependent Kinds

The expression $\texttt{let}\,e_1\texttt{:}\tau\,\texttt{be}\,x\,\texttt{in}\,e_2$ 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 $\lambda\,\texttt{(}x\texttt{:}\tau.\,e_2\texttt{)}\texttt{(}e_1\texttt{)}$, 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. The expression $\texttt{let}\,t\,\texttt{be}\,\tau\,\texttt{in}\,e$ binds t to τ within e, so that one may write expressions such as

$$\texttt{let}\,t\,\texttt{be}\,\texttt{nat}\times\texttt{nat}\,\texttt{in}\,\lambda\,\texttt{(}x\texttt{:}t.\,\texttt{s(}x\cdot\texttt{1))}.$$

For this expression to be type-correct the type variable t must be *synonymous* with the type $\texttt{nat}\times\texttt{nat}$, for otherwise the body of the λ-abstraction is not type correct.

Following the pattern of the expression-level `let`, we might guess that `lettype` is an abbreviation for the polymorphic instantiation $\Lambda\texttt{(}t\texttt{.}e\texttt{)}\texttt{[}\tau\texttt{]}$, which binds t to τ 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 `lettype`, 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 synomous with τ. Thus, in the above example, the expression $\texttt{s(}x\cdot\texttt{1)}$ fails to type check, unless the binding of t were exposed.

The proposed definition of `lettype` in terms of type abstraction and type application fails. Lacking any other idea, one might argue that type abbreviation ought to be considered as a primitive concept, rather than a derived notion. The expression $\texttt{let}\,t\,\texttt{be}\,\tau\,\texttt{in}\,e$ would be taken as a primitive form of expression whose statics is given by the following rule:

$$\frac{\Gamma \vdash [\tau/t]e : \tau'}{\Gamma \vdash \texttt{let}\, t\, \texttt{be}\, \tau\, \texttt{in}\, e : \tau'} \tag{28.1}$$

This would address the problem of supporting type abbreviations, but it does so in a rather *ad hoc* manner. One might hope for a more principled solution that arises naturally from the type structure of the language.

Our 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? And what else does this type structure suggest? By following this methodology we are led 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.

28.1 Informal 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 let t be τ in e, the principle of compositionality dictates that the only thing that e "knows" about the type variable t is its kind, namely Type, and not its binding, namely τ. 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 Eqv(τ) is the kind of types that are definitionally equivalent to τ. That is, up to definitional equality, this kind has only one inhabitant, namely τ. Consequently, if u :: Eqv(τ) is a variable of singleton kind, then within its scope, the variable u is synonymous with τ. Thus we may represent let t be τ in e by

$\Lambda(t::\texttt{Eqv}(\tau).e)[\tau]$, which correctly propagates the identity of t, namely τ, 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 `Type`, and hence is a type. Otherwise, a variable, u, 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, exception at the kind level. The fundamental axiom of subkinding is $\texttt{Eqv}(\tau)$:<: `Type`, stating that every constructor of singleton kind is a type.

Second, we must account for the occurrence of a constructor of kind `Type` within the singleton kind $\texttt{Eqv}(\tau)$. 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 $\texttt{Eqv}(\tau)$ represents a *family of kinds* indexed by constructors of kind `Type`. This, in turn, implies that we must generalize the function and product kinds to *dependent functions* and *dependent products*. The dependent function kind, $\prod u::\kappa_1.\kappa_2$ 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.[1] The dependent product kind, $\sum u::\kappa_1.\kappa_2$, 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.

Third, it is useful to consider singletons not just of kind `Type`, but also of higher kinds. To support this we introduce *higher-kind singletons*, written $\texttt{Eqv}(c::\kappa)$, where κ is a kind and c is a constructor of kind k. These are definable in terms of the primitive form of singleton kind by making use of dependent function and product kinds.

This chapter is under construction ....

[1] As we shall see in the development, the propagation of information as sketched here is managed through the use of singleton kinds.

Part X

Classes and Methods

Chapter 29

Dynamic Dispatch

It frequently arises that the values of a type are partitioned into a variety of *classes*, each classifying data with distinct internal structure. A good 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, ρ, from the origin and its angle, θ, with the polar axis. A classified value is said to be an *instance* of, or an *object* 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 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. Because it happens at run-time, this is called, rather grandly, *dynamic dispatch*. For example, the 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 $\sqrt{x^2 + y^2}$, whereas in the latter it is simply ρ itself. 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 θ divided by $\pi/2$.

Since 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_{dm}, called the *dispatch matrix*, whose rows are classes, whose columns are methods, and whose (c, d)-entry is the code for 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

$$\prod_{c \in C} \prod_{d \in D} (\sigma^c \to \rho_d),$$

where C is the set of class names, D is the set of method names, σ^c is the instance type associated with class c and ρ_d is the result type of method d. The instance type is the same for all methods acting on a given class, and that the result type is the same for all classes acted on by a given method.

There are two main ways to organize a system of classes and methods, according to whether we wish to place emphasis on the classes, thought of as a collection of methods acting on its instances, or on the methods, thought of as a collection of classes on which the methods act. These are, respectively, the *class-based* and the *method-based* organizations. Languages that place special emphasis on classes and methods, called *object-oriented languages*,[1] usually adopt one or the other of these organizations as a central design principle.

There is little point in making heavy weather of the distinction, both being applicable in different situations. What is important is that both arise from simple manipulations of the dispatch matrix based on symmetries between product and sum types. A fully expressive language supports sums and products equally well, and hence supports the class-based organization as readily as the method-based, rather than taking a doctrinal stance that cannot be maintained in the face of these symmetries.

The method-based organization starts with the *transpose* of the dispatch matrix, which has the type

$$\prod_{d \in D} \prod_{c \in C} (\sigma^c \to \rho_d).$$

By observing that each row of the transposed dispatch matrix determines a method, we obtain the *method vector*, e_{mv}, of type

$$\tau_{\mathsf{mv}} \triangleq \prod_{d \in D} (\sum_{c \in C} \sigma^c) \to \rho_d.$$

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. This organization makes it easy to add new methods for a given collection of classes by simply defining a new function of this type. It makes adding a

[1] The term "object-oriented" itself speaks to the vagueness of the concept. It is used, for the most part, to express approval.

new class relatively more difficult, however, since doing so requires that each method be updated to account for the new forms of object.

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_{cv}, of type

$$\tau_{\mathsf{cv}} \triangleq \prod_{c \in C} (\sigma^c \to (\prod_{d \in D} \rho_d)).$$

Each row of the class vector consists of a *constructor* that determines the result of each of the methods when acting on given instance data. This organization makes it easy to add a new class to the program; we need only define the method tuple on the instance data for the new class. It makes adding a new method relatively more difficult, however, because we must extend the interpretation of each class to account for it.

We will show how to give a method-based and a class-based implementation of objects by defining the following concepts:

- The type of objects arising as instances of the classes on which the methods act.
- The operation `new[`c`](`e`)` that creates an object of the class c with instance data given by the expression e.
- The operation $e \Leftarrow d$ that invokes method d on the instance given by the expression e.

Informally, under the method-based organization an object consists of the instance data tagged with its class. A new instance is created by attaching the class tag to the instance data, and a method is invoked by dispatching on the class of the instance. Conversely, under the class-based organization an object consists of a tuple of results of each of the methods acting on the instance data of the object. A new object is created by applying each of the methods to given instance data, and a method is invoked by projecting the result from the object.

29.1 The Dispatch Matrix

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

$$\sigma^{\texttt{cart}} = \langle \texttt{x}:\texttt{real}, \texttt{y}:\texttt{real} \rangle,$$

and the instance data for a polar point has the type

$$\sigma^{\texttt{pol}} = \langle \texttt{r}:\texttt{real}, \texttt{th}:\texttt{real} \rangle.$$

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 distance method is given by the tuple $e_{\texttt{dist}} = \langle \texttt{cart} = e^{\texttt{cart}}_{\texttt{dist}}, \texttt{pol} = e^{\texttt{pol}}_{\texttt{dist}} \rangle$ of type

$$\langle \texttt{cart}:\sigma^{\texttt{cart}} \rightarrow \rho_{\texttt{dist}}, \texttt{pol}:\sigma^{\texttt{pol}} \rightarrow \rho_{\texttt{dist}} \rangle,$$

where $\rho_{\texttt{dist}} = \texttt{real}$ is the result type,

$$e^{\texttt{cart}}_{\texttt{dist}} = \lambda\,(u{:}\sigma^{\texttt{cart}}.\,(u\cdot\texttt{x})^2 + (u\cdot\texttt{y})^2)$$

is the distance computation for a cartesian point, and

$$e^{\texttt{pol}}_{\texttt{dist}} = \lambda\,(v{:}\sigma^{\texttt{pol}}.\,(v\cdot\texttt{r})^2)$$

is the distance computation for a polar point. Similarly, the quadrant method is given by the tuple $e_{\texttt{quad}} = \langle \texttt{cart} = e^{\texttt{cart}}_{\texttt{quad}}, \texttt{pol} = e^{\texttt{pol}}_{\texttt{quad}} \rangle$ of type

$$\langle \texttt{cart}:\sigma^{\texttt{cart}} \rightarrow \rho_{\texttt{quad}}, \texttt{pol}:\sigma^{\texttt{pol}} \rightarrow \rho_{\texttt{quad}} \rangle,$$

where $\rho_{\texttt{quad}} = \texttt{[I,II,III,IV]}$ is the type of quadrants, and $e^{\texttt{cart}}_{\texttt{quad}}$ and $e^{\texttt{pol}}_{\texttt{quad}}$ are expressions that compute the quadrant of a point in rectangular and polar forms, respectively.

Now let $C = \{\,\texttt{cart}, \texttt{pol}\,\}$ and let $D = \{\,\texttt{dist}, \texttt{quad}\,\}$, and define the dispatch matrix, e_{dm}, to be the value of type

$$\prod_{c\in C}\prod_{d\in D}(\sigma^c \rightarrow \rho_d)$$

such that, for each class c and method d,

$$e_{\mathsf{dm}} \cdot c \cdot d \mapsto^* e^c_d.$$

That is, the entry in the dispatch matrix, e_{dm}, for class c and method d is defined to be the implementation of that method acting on an instance of that class.

29.2 Method-Based Organization

An object is a value of type $\sigma = \sum_{c \in C} \sigma^c$, the sum over the classes of the instance types. For example, the type of points in the plane is the sum type

$$[\texttt{cart} : \sigma^{\texttt{cart}}, \texttt{pol} : \sigma^{\texttt{pol}}].$$

Each point is labelled with its class, specifying its representation as having either cartesian or polar form.

An instance of a class c is just the instance data labelled with its class to form an element of the object type:

$$\texttt{new}[c](e) \triangleq c \cdot e.$$

For example, a cartesian point with coordinates x_0 and y_0 is given by the expression

$$\texttt{new[cart]}(\langle \texttt{x} = x_0, \texttt{y} = y_0 \rangle) \triangleq \texttt{cart} \cdot \langle \texttt{x} = x_0, \texttt{y} = y_0 \rangle.$$

Similarly, a polar point with distance ρ_0 and angle θ_0 is given by the expression

$$\texttt{new[pol]}(\langle \texttt{r} = r_0, \texttt{th} = \theta_0 \rangle) \triangleq \texttt{pol} \cdot \langle \texttt{r} = r_0, \texttt{th} = \theta_0 \rangle.$$

The method-based organization consolidates the implementation of each method into the *method vector*, e_{mv} of type τ_{mv}, defined by $\langle e_d \rangle_{d \in D}$, where for each $d \in D$ the expression $e_d : \sigma \rightarrow \rho_d$ is

$$\lambda\,(\mathit{this} : \sigma.\, \texttt{case}\, \mathit{this}\, \{c \cdot u \Rightarrow e_{\mathsf{dm}} \cdot c \cdot d(u)\}_{c \in C}).$$

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

$$\langle \texttt{dist} : \sigma \rightarrow \rho_{\texttt{dist}}, \texttt{quad} : \sigma \rightarrow \rho_{\texttt{quad}} \rangle.$$

The dispatch function for the dist method has the form

$$\lambda\,(\mathit{this} : \sigma.\, \texttt{case}\, \mathit{this}\, \{\texttt{cart} \cdot u \Rightarrow e_{\mathsf{dm}} \cdot \texttt{cart} \cdot \texttt{dist}(u) \mid \texttt{pol} \cdot v \Rightarrow e_{\mathsf{dm}} \cdot \texttt{pol} \cdot \texttt{dist}(v)\}),$$

and the dispatch function for the quad method has the similar form

$$\lambda\,(\mathit{this} : \sigma.\, \texttt{case}\, \mathit{this}\, \{\texttt{cart} \cdot u \Rightarrow e_{\mathsf{dm}} \cdot \texttt{cart} \cdot \texttt{quad}(u) \mid \texttt{pol} \cdot v \Rightarrow e_{\mathsf{dm}} \cdot \texttt{pol} \cdot \texttt{quad}(v)\}).$$

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,

$$\begin{aligned}(\mathtt{new}[c](e)) \Leftarrow d &\mapsto^* e_{\mathsf{mv}} \cdot d(c \cdot e)\\ &\mapsto^* e_{\mathsf{dm}} \cdot c \cdot d(e)\end{aligned}$$

That is, the message send invokes the implementation of the method d on the instance data for the given object.

29.3 Class-Based Organization

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 ρ is the product type

$$\langle \mathtt{dist} : \rho_{\mathtt{dist}}, \mathtt{quad} : \rho_{\mathtt{quad}} \rangle.$$

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_{cv}, a tuple of type τ_{cv} consisting of the *constructor* of type $\sigma^c \rightarrow \rho$ for each class $c \in C$. The class vector is defined by $e_{\mathsf{cv}} = \langle e^c \rangle_{c \in C}$, where for each $c \in C$ the expression e^c is

$$\lambda\,(u{:}\sigma^c.\,\langle e_{\mathsf{dm}} \cdot c \cdot d(u) \rangle_{d \in D}).$$

For example, the constructor for the class cart is the function $e^{\mathtt{cart}}$ given by the expression

$$\lambda\,(u{:}\sigma^{\mathtt{cart}}.\,\langle \mathtt{dist} = e_{\mathsf{dm}} \cdot \mathtt{cart} \cdot \mathtt{dist}(u), \mathtt{quad} = e_{\mathsf{dm}} \cdot \mathtt{cart} \cdot \mathtt{quad}(u) \rangle).$$

Similarly, the constructor for the class pol is the function $e^{\mathtt{pol}}$ given by the expression

$$\lambda\,(u{:}\sigma^{\mathtt{pol}}.\,\langle \mathtt{dist} = e_{\mathsf{dm}} \cdot \mathtt{pol} \cdot \mathtt{dist}(u), \mathtt{quad} = e_{\mathsf{dm}} \cdot \mathtt{pol} \cdot \mathtt{quad}(u) \rangle).$$

The class vector, e_{cv}, in this case is the tuple $\langle \mathtt{cart} = e^{\mathtt{cart}}, \mathtt{pol} = e^{\mathtt{pol}} \rangle$ of type $\langle \mathtt{cart} : \sigma^{\mathtt{cart}} \rightarrow \rho, \mathtt{pol} : \sigma^{\mathtt{pol}} \rightarrow \rho \rangle$.

An instance 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 $\mathtt{new}[\mathtt{cart}](\langle \mathtt{x} = x_0, \mathtt{y} = y_0 \rangle)$, which is defined by the expression

$$e_{\mathsf{cv}} \cdot \mathtt{cart}(\langle \mathtt{x} = x_0, \mathtt{y} = y_0 \rangle).$$

Similarly, a polar point is obtained by writing $\mathtt{new}[\mathtt{pol}](\mathtt{r} = r_0, \mathtt{th} = \theta_0)$, which is defined by the expression

$$e_{\mathsf{cv}} \cdot \mathtt{pol}(\langle \mathtt{r} = r_0, \mathtt{th} = \theta_0 \rangle).$$

It is easy to check for this organization of points that for each class c and method d, we may derive

$$\begin{aligned} (\mathtt{new}[c](e)) \Leftarrow d &\mapsto^* (e_{\mathsf{cv}} \cdot c(e)) \cdot d \\ &\mapsto^* e_{\mathsf{dm}} \cdot c \cdot d(e) \end{aligned}$$

The outcome is, of course, the same as for the method-based organization.

29.4 Self-Reference

A significant shortcoming of the foregoing account of classes and methods is that methods are not permitted to create new objects or to send messages to existing objects. The elements of the dispatch matrix are functions whose domain and range are given in advance. It is only after the dispatch matrix has been defined that we are able to choose either the method-based or class-based organization for computing with classified objects. Rectifying this will, *en passant*, also permit methods to call one another, perhaps even themselves, and allow constructors to create instances, perhaps even of their own class.

The first step to correcting this shortcoming is to change the definition and type of the dispatch matrix so that method bodies may create instances and send messages. This is not quite so straightforward as it may sound, because the meaning of instance creation and message send varies according to whether we are using a method-based or a class-based organization.

Naïvely, this would seem to imply that the dispatch matrix can no longer be organized along either the method or class axis, but must instead be defined separately according to whether we are using a method-based or class-based organization. However, the dependency can be avoided by using an abstract type to avoid representation commitments.

To allow methods to call one another and to allow constructors to generate objects of other classes, the types of the class and method vectors must be given self-referential types (see Section 19.3 on page 161). This is necessary because the definitions of message send (in the method-based setup) and instantiation (in the class-based setup) imply that the dispatchers in the method vector and the constructors in the class vector may refer to themselves indirectly via the dispatch matrix.

The type of the dispatch matrix is generalized to the polymorphic type

$$\prod_{c\in C}\prod_{d\in D}\forall(t\,.\,\tau_{\mathsf{cv}}\to\tau_{\mathsf{mv}}\to\sigma^{c}\to\rho_{d}),$$

where t is the abstract type of objects, the type of the class vector is given by the equation

$$\tau_{\mathsf{cv}}=\prod_{c\in C}(\sigma^{c}\to t),$$

and the type of the method vector is given by the equation

$$\tau_{\mathsf{mv}}=(\prod_{d\in D}t\to\rho_{d}).$$

Each class vector entry is a constructor yielding an object of type t given instance data for that class, and each method vector entry is a dispatcher that acts on an object of type t to determine the result of that method. The entry for class c and method d in the dispatch matrix has the form

$$\Lambda(t\,.\,\lambda\,(cv\!:\!\tau_{\mathsf{cv}}.\,\lambda\,(mv\!:\!\tau_{\mathsf{mv}}.\,\lambda\,(u\!:\!\sigma^{c}.\,e)))),$$

where within the body e a new object of class c' with instance data e' is obtained by writing $cv\cdot c'(e')$, and a message d' is sent to an object e' by writing $mv\cdot d'(e')$. Thus the implementation of method d on class c may create an instance of *any* class, including c itself, and may invoke *any* method, including d itself.

The change to the type of the dispatch matrix requires that we reconsider the definition of the class and method vectors. Under the method-based organization the instantiation operation is defined directly to tag the instance data with its class, just as before. The messaging operation must be generalized, however, to allow for the self-reference engendered by invoking a method that may itself invoke another method. Dually, under the class-based organizaton messaging is defined by projection, as before, but instantiation must be generalized to to account for the self-reference engendered by a constructor creating an instance.

To allow for self-reference the method vector, e_{mv} is defined to have the type $[\sigma/t]\tau_{\mathsf{mv}}\ \mathtt{self}$, in which the abstract object type is specialized to the sum over all classes of their instance types. The method vector is defined by the expression

$$\mathtt{self}\ mv\ \mathtt{is}\ \langle d = \lambda\ (\mathit{this}:\sigma.\ \mathtt{case}\ \mathit{this}\ \{c \cdot u \Rightarrow e_{\mathsf{dm}} \cdot c \cdot d\,[\sigma]\,(e'_{\mathsf{cv}})\,(e'_{\mathsf{mv}})\,(u)\}_{c \in C})\rangle_{d \in D},$$

where the class vector argument, e'_{cv}, is the tuple of tagging operations $\langle c = \lambda\ (u:\sigma^{c}.\ c \cdot u)\rangle_{c \in C}$, and the method vector argument, e'_{mv}, is the recursive unrolling of the method vector itself, $\mathtt{unroll}(mv)$. The message send operation $e \Leftarrow d$ is given by the expression $\mathtt{unroll}(e_{\mathsf{mv}}) \cdot d(e)$, whereas objection creation, $\mathtt{new}[c]\,(e)$, is defined as before to be $c \cdot e$.

Alternatively, under the class-based organization, the class vector, e_{cv}, is defined to have the type $[\rho/t]\tau_{\mathsf{cv}}\ \mathtt{self}$, which specifies that the abstract type of objects is the product over all methods of their result types. The class vector itself is given by the expression

$$\mathtt{self}\ cv\ \mathtt{is}\ \langle c = \lambda\ (u:\sigma^{c}.\ \langle d = e_{\mathsf{dm}} \cdot c \cdot d\,[\rho]\,(e''_{\mathsf{cv}})\,(e''_{\mathsf{mv}})\,(u)\rangle_{d \in D})\rangle_{c \in C}$$

where the class vector argument, e''_{cv}, is $\mathtt{unroll}(cv)$, and the method vector argument, e''_{mv}, is the tuple of projections, $\langle d = \lambda\ (\mathit{this}:\rho.\ \mathit{this} \cdot d)\rangle_{d \in D}$. Object creation, $\mathtt{new}[c]\,(e)$ is defined by the expression $\mathtt{unroll}(e_{\mathsf{cv}}) \cdot c(e)$, whereas message send, $d \Leftarrow e$, is defined, as before, by $e \cdot d$.

The symmetries between the two organizations are striking; they reflect the duality between sum and product types.

29.5 Exercises

1. Generalize the class-based table to allow each class to determine the set of methods defined on it, and similarly generalize the method-based table to allow each method to act on certain classes.

2. Extend to allow methods to return instances as results and constructors to take instances as arguments. The method-based approach has no difficulty with the former, but requires some modification to allow for the latter; dually, the class-based approach has no difficulty with the latter, but requires some modification to allow for the former.

3. Add support for an *instance test*, which allows testing whether an object is an instance of a specified class. This amounts to insisting that each object come equipped with a family of methods `instanceof`[c], where $c \in C$, which evaluates to a boolean according to whether the object is an instance of class c or not.

Chapter 30

Inheritance

Dynamic dispatch was introduced in Chapter 29 as a means of organizing the action of a collection of methods on instances of a collection of classes. The dispatch matrix assigns to each class, c, and each method, d, an implementation of type $\sigma^c \rightarrow \rho_d$ mapping the instance data appropriate for class c to a result appropriate for method d. In this chapter we consider the problem of building up the dispatch matrix by extending it with either a new class or a new method (or both) using *inheritance*. The main idea is to support code reuse by defining a new class or a new method by *inheriting* much of the implementation from one or more existing classes or methods, but allowing for modifications, called *overrides*, that alter the behavior of the new class or method. Methodologically speaking, these modifications are intended to be relatively few compared to the full extent of the implementation, but nothing precludes a wholesale redefinition of the behavior of the new class or method compared to the old one. Consequently, knowing that one class or method inherits from another tells us *nothing* about the behavior of the child compared to that of the parent(s).[1]

In this chapter we will consider the most common form of inheritance, called *subclassing*, which is defined for the class-based organization described in Chapter 29. We are given a class vector, e_{cv}, and we are to define a new class vector, e^*_{cv}, by adding a *subclass*, c^* of the *superclasses* $c_1, \ldots, c_n$, where $n \geq 0$. A dual form of inheritance, for which there is no established terminology but which one might call *submethoding*, may be defined for the method-based organization given in Chapter 29. We will not develop this idea in detail here, but rather leave it as an exercise for the reader.

[1] In view of this one may doubt the significance of programming methodologies that stress inheritance as a central organizing principle.

It is common to distinguish two forms of inheritance, *single inheritance*, which restricts the number, n, of superclasses or supermethods to 1, and *multiple inheritance*, which places no limit on n. Single inheritance is better-behaved in that each child class or method has a unique parent. Multiple inheritance introduces ambiguities when there is more than one superclass providing a method or class that is inherited by the child. Consequently, a rule is required to determine from which parent a given attribute is to be inherited.

Inheritance is often confused with subtyping. Whereas inheritance is simply a statement about how a body of code came into being, subtyping is a statement about how a body of code can be expected to behave. Many languages seek to ensure that if one class inherits from another, then the type of objects of the subclass is a subtype of the type of objects of the superclass. As we shall see, this is not automatically the case.

30.1 Subclassing

We begin with the class-based organization described in Chapter 29. Let $e_{\mathsf{cv}} : (\prod_{c\in C} \sigma^c \to \prod_{d\in D} \rho_d)$ be a class vector, and suppose that $c_1, \ldots, c_n \in C$. Defining a subclass, $c_* \notin C$, of the superclasses $c_1, \ldots, c_n$, consists of specifying the following information:

1. The instance type σ^{c_*} of the new class such that $\sigma^{c_*} <: \sigma^{c_i}$ for each $1 \leq i \leq n$.

2. The subset $D_{\mathsf{inh}} \subseteq D$ of *inherited* methods. The remaining set $D_{\mathsf{ovr}} = D \setminus D_{\mathsf{inh}}$ is defined to be the set of *overridden* methods.

3. For each $d \in D_{\mathsf{ovr}}$, an expression $e_d : \sigma^{c_*} \to \rho_d$, the action of the overridden method, d, on instances of the class c_*.

By the principles of co- and contravariance for function types (see Chapter 27) the domain type of the expression e_d may be a supertype of σ^{c_*} and its range type may be a subtype of ρ_d.

Given this data, the extended class vector, e^*_{cv}, is defined as follows:

1. For each class $c \in C$, the constructor $e^*_{\mathsf{cv}} \cdot c$ is equivalent to the constructor $e_{\mathsf{cv}} \cdot c$. That is, all existing classes are preserved intact.

2. The action of the method d on an instance of class c_* is defined as follows:

(a) If $d \in D_{\mathsf{inh}}$ is an inherited method, then, for some $1 \leq i \leq n$ and any instance data e_0 of type σ^{c_*}, the method $e^*_{\mathsf{cv}} \cdot c_*(e_0) \cdot d$ is equivalent to the method $e^*_{\mathsf{cv}} \cdot c_i(e_0) \cdot d$. (When $n \geq 2$ the choice of i must be given by some fixed rule, which we do not specify here.)

(b) If $d \in D_{\mathsf{ovr}}$ is an overridden method, then for any instance data e_0 of type σ^{c_*}, the method $e^*_{\mathsf{cv}} \cdot c_*(e_0) \cdot d$ is equivalent to $e_d(e_0)$, where e_d is given by the subclass definition.

The resulting class vector, e^*_{cv}, is of type

$$\prod_{c \in C \cup \{ c_* \}} (\sigma^c \to \prod_{d \in D} \rho_d).$$

The requirement that $\sigma^{c_*} <: \sigma^{c_i}$ ensures that the instance data for the subclass may be acted upon by a method of the superclass. The condition on inherited methods ensures that the message $\mathtt{new}[c_*](e_0) \Leftarrow d$ is equivalent to the message $\mathtt{new}[c_i](e_0) \Leftarrow d$, and the condition on overridden methods ensure that $\mathtt{new}[c_*](e_0) \Leftarrow d$ is equivalent to $e_d(e_0)$.

In this simple formulation of the class vector the object type $\rho = \prod_{d \in D} \rho_d$ is unaffected by inheritance. A more sophisticated formulation of the class vector allows each class to determine the methods supported by objects of that class, and allows the result types of these methods to vary with the class. Specifically, let us generalize the set, D, of methods into a family of sets $\{ D_c \}_{c \in C}$, with D_c being the methods supported by objects of class c. Additionally, for each $c \in C$ and each $d \in D_c$ let the result type, ρ^c_d, of method d acting on instances of class c be given. The class vector then has the type

$$\prod_{c \in C} \sigma^c \to (\prod_{d \in D_c} \rho^c_d).$$

The object type, $\rho^c = \prod_{d \in D_c} \rho^c_d$, is the type of instances of class c.[2]

Given a class vector, e_{cv}, of the above type, a subclass c_* of superclasses $c_1, \ldots, c_n$ is defined by specifying the following information:

1. The instance type, σ^{c_*}, for the new class such that $\sigma^{c_*} <: \sigma^{c_i}$ for each $1 \leq i \leq n$.

2. The set $D_{c_*} = D_{\mathsf{inh}} \uplus D_{\mathsf{ovr}} \uplus D_{\mathsf{ext}}$ of inherited, overridden, and extending methods associated with class c_* such that $D_{\mathsf{inh}} \uplus D_{\mathsf{ovr}} \subseteq \bigcup_{1 \leq i \leq n} D_{c_i}$.

[2] In many accounts the type ρ^c is *identified* with the class c, thereby confusing classes with types.

3. For each $d \in D_{\mathsf{ovr}} \uplus D_{\mathsf{ext}}$, an expression $e_d : \sigma^{c_*} \to \rho_d^{c_*}$ providing the implementation of method d for the new class c_*.

Since each class determines its own collection of methods, we now require that $D_{c_i} \cap D_{c_j} = \emptyset$ whenever $i \neq j$, so that no method is provided by more than one superclass.

This data determines a new class vector, e^*_{cv}, defined as follows:

1. For each class $c \in C$, the constructor $e^*_{\mathsf{cv}} \cdot c$ is equivalent to the constructor $e_{\mathsf{cv}} \cdot c$.
2. The action of the method d on an instance of class c_* is defined as follows:
   (a) If $d \in D_{\mathsf{inh}}$ is an inherited method, then, for a unique $1 \leq i \leq n$ and any instance data e_0 of type σ^{c_*}, the method $e^*_{\mathsf{cv}} \cdot c_*(e_0) \cdot d$ is equivalent to the method $e^*_{\mathsf{cv}} \cdot c_i(e_0) \cdot d$.
   (b) If $d \in D_{\mathsf{ovr}} \uplus D_{\mathsf{ext}}$ is an overridden or extending method, then for any instance data e_0 of type σ^{c_*}, the method $e^*_{\mathsf{cv}} \cdot c_*(e_0) \cdot d$ is equivalent to $e_d(e_0)$, where e_d is given by the subclass definition.

There is no longer any ambiguity in the choice of inherited methods, since the superclass method suites are assumed to be disjoint.

The resulting class vector has type

$$\prod_{c \in C \uplus \{c_*\}} (\sigma^c \to (\prod_{d \in D_c} \rho_d^c)).$$

Letting $\rho^c = \prod_{d \in D_c} \rho_d^c$ be the object type associated to class c, observe that the subclass object type, ρ^{c_*}, need not be a subtype of any superclass object type, ρ^{c_i}. That is, *inheritance does not imply subtyping*. For example, the subclass result type of an overriding method need not bear any relationship to any superclass result type for that method. Moreover, the subclass need not provide all of the methods provided by the superclasses. However, if we impose the additional requirements that (1) if $d \in D_{\mathsf{ovr}}$, then $\rho_d^{c_*} <: \rho_d^{c_i}$ for the unique superclass c_i providing the method d, and (2) $D_{\mathsf{inh}} \uplus D_{\mathsf{ovr}} = \bigcup_{1 \leq i \leq n} D_{c_i}$, then the type of subclass objects will be a subtype of the object types of the superclasses. Consequently, a subclass object may be used wherever a superclass object is required. Many object-oriented languages insist on this condition as a constraint on inheritance. However, this policy is not always sustainable. Certain advanced forms of inheritance[3] *preclude*

[3] Namely, those involving “self types, ” which we do not consider here.

this policy. Fundamentally, inheritance and subtyping are two different concepts. Whereas subtyping is, by the subsumption principle, a matter of *essence*, inheritance, being only an artifact of how a body of code was created is a matter of *accident*.

30.2 Exercises

1. Extend inheritance to self-referential methods.

2. Develop the idea of *submethoding* for the method-based organization. Given a method vector, e_{mv}, we are to define a new method vector, e^*_{mv}, extended with a *submethod*, d^*, of the *supermethods*, $d_1, \ldots, d_n$, where $n \geq 0$.

3. Allow overlaps among the superclasses provided that all such methods are overridden in the subclass so that there can be no ambiguity.

Part XI

Control Effects

Chapter 31

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, $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$, for the language $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$. 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 $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ with the structural dynamics of $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$.

31.1 Machine Definition

A state, s, of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ 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 $k \lhd e$, where e 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 control stack is a list of *frames*:

$$\frac{}{\epsilon\ \mathsf{stack}} \tag{31.1a}$$

$$\frac{f\ \mathsf{frame} \quad k\ \mathsf{stack}}{k;f\ \mathsf{stack}} \tag{31.1b}$$

The definition of frame depends on the language we are evaluating. The frames of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ are inductively defined by the following rules:

$$\frac{}{\mathtt{s(-)}\ \mathsf{frame}} \tag{31.2a}$$

$$\frac{}{\mathtt{ifz(-;}e_1\mathtt{;}x\mathtt{.}e_2\mathtt{)}\ \mathsf{frame}} \tag{31.2b}$$

$$\frac{}{\mathtt{ap(-;}e_2\mathtt{)}\ \mathsf{frame}} \tag{31.2c}$$

The frames correspond to rules with transition premises in the dynamics of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. 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 judgement between states of the $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ is inductively defined by a set of inference rules. We begin with the rules for natural numbers.

$$\frac{}{k \rhd \mathtt{z} \mapsto k \lhd \mathtt{z}} \tag{31.3a}$$

$$\frac{}{k \rhd \mathtt{s(}e\mathtt{)} \mapsto k;\mathtt{s(-)} \rhd e} \tag{31.3b}$$

$$\frac{}{k;\mathtt{s(-)} \lhd e \mapsto k \lhd \mathtt{s(}e\mathtt{)}} \tag{31.3c}$$

To evaluate $\mathtt{z}$ we simply return it. To evaluate $\mathtt{s(}e\mathtt{)}$, we push a frame on the stack to record the pending successor, and evaluate e; when that returns with e', we return $\mathtt{s(}e'\mathtt{)}$ to the stack.

Next, we consider the rules for case analysis.

$$\frac{}{k \rhd \mathtt{ifz(}e\mathtt{;}e_1\mathtt{;}x\mathtt{.}e_2\mathtt{)} \mapsto k;\mathtt{ifz(-;}e_1\mathtt{;}x\mathtt{.}e_2\mathtt{)} \rhd e} \tag{31.4a}$$

$$\frac{}{k;\mathtt{ifz(-;}e_1\mathtt{;}x\mathtt{.}e_2\mathtt{)} \lhd \mathtt{z} \mapsto k \rhd e_1} \tag{31.4b}$$

$$\frac{}{k;\mathtt{ifz}(-;e_1;x.e_2) \lhd \mathtt{s}(e) \mapsto k \rhd [e/x]e_2} \tag{31.4c}$$

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 and recursion.

$$\frac{}{k \rhd \mathtt{lam}[\tau](x.e) \mapsto k \lhd \mathtt{lam}[\tau](x.e)} \tag{31.5a}$$

$$\frac{}{k \rhd \mathtt{ap}(e_1;e_2) \mapsto k;\mathtt{ap}(-;e_2) \rhd e_1} \tag{31.5b}$$

$$\frac{}{k;\mathtt{ap}(-;e_2) \lhd \mathtt{lam}[\tau](x.e) \mapsto k \rhd [e_2/x]e} \tag{31.5c}$$

$$\frac{}{k \rhd \mathtt{fix}[\tau](x.e) \mapsto k \rhd [\mathtt{fix}[\tau](x.e)/x]e} \tag{31.5d}$$

These rules ensure that the function is evaluated before the argument, applying the function when both have been evaluated. Note that evaluation of general recursion requires no stack space! (But see Chapter 41 for more on evaluation of general recursion.)

The initial and final states of the $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ are defined by the following rules:

$$\frac{}{\epsilon \rhd e \ \mathsf{initial}} \tag{31.6a}$$

$$\frac{e \ \mathsf{val}}{\epsilon \lhd e \ \mathsf{final}} \tag{31.6b}$$

31.2 Safety

To define and prove safety for $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ requires that we introduce a new typing judgement, $k : \tau$, stating that the stack k expects a value of type τ. This judgement is inductively defined by the following rules:

$$\frac{}{\epsilon : \tau} \tag{31.7a}$$

$$\frac{k : \tau' \quad f : \tau \Rightarrow \tau'}{k;f : \tau} \tag{31.7b}$$

This definition makes use of an auxiliary judgement, $f : \tau \Rightarrow \tau'$, stating that a frame f transforms a value of type τ to a value of type τ'.

$$\frac{}{\mathtt{s}(-) : \mathtt{nat} \Rightarrow \mathtt{nat}} \tag{31.8a}$$

$$\frac{e_1 : \tau \quad x : \mathtt{nat} \vdash e_2 : \tau}{\mathtt{ifz}(-;e_1;x.e_2) : \mathtt{nat} \Rightarrow \tau} \tag{31.8b}$$

$$\frac{e_2 : \tau_2}{\texttt{ap(}-\texttt{;}e_2\texttt{)} : \texttt{arr(}\tau_2\texttt{;}\tau\texttt{)} \Rightarrow \tau} \tag{31.8c}$$

The two forms of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ state are well-formed provided that their stack and expression components match.

$$\frac{k : \tau \quad e : \tau}{k \triangleright e \ \mathsf{ok}} \tag{31.9a}$$

$$\frac{k : \tau \quad e : \tau \quad e \ \mathsf{val}}{k \triangleleft e \ \mathsf{ok}} \tag{31.9b}$$

We leave the proof of safety of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ as an exercise.

Theorem 31.1 (Safety). *1. If s ok and $s \mapsto s'$, then s' ok.*

2. If s ok, then either s final or there exists s' such that $s \mapsto s'$.

31.3 Correctness of the Control Machine

It is natural to ask whether $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ correctly implements $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$. If we evaluate a given expression, e, using $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$, do we get the same result as would be given by $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$, and *vice versa*?

Answering this question decomposes into two conditions relating $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ to $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$:

> **Completeness** If $e \mapsto^* e'$, where e' val, then $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e'$.
>
> **Soundness** If $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e'$, then $e \mapsto^* e'$ with e' val.

Let us consider, in turn, what is involved in the proof of each part.

For completeness it is natural 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 $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e$.

2. If $e \mapsto e'$, then, for every v val, if $\epsilon \triangleright e' \mapsto^* \epsilon \triangleleft v$, then $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft v$.

The first can be proved easily by induction on the structure of e. The second requires an inductive analysis of the derivation of $e \mapsto e'$, 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, $\texttt{ap(}e_1\texttt{;}e_2\texttt{)}$, then the first step of the machine is

$$\epsilon \triangleright \texttt{ap(}e_1\texttt{;}e_2\texttt{)} \mapsto \epsilon\texttt{;ap(}-\texttt{;}e_2\texttt{)} \triangleright e_1,$$

and so we must consider evaluation of e_1 on a non-empty stack.

A natural generalization is to prove that if $e \mapsto e'$ and $k \triangleright e' \mapsto^* k \triangleleft v$, then $k \triangleright e \mapsto^* k \triangleleft v$. Consider again the case $e = \texttt{ap}(e_1;e_2)$, $e' = \texttt{ap}(e_1';e_2)$, with $e_1 \mapsto e_1'$. We are given that $k \triangleright \texttt{ap}(e_1';e_2) \mapsto^* k \triangleleft v$, and we are to show that $k \triangleright \texttt{ap}(e_1;e_2) \mapsto^* k \triangleleft v$. It is easy to show that the first step of the former derivation is

$$k \triangleright \texttt{ap}(e_1';e_2) \mapsto k;\texttt{ap}(-;e_2) \triangleright e_1'.$$

We would like to apply induction to the derivation of $e_1 \mapsto e_1'$, but to do so we must have a v_1 such that $e_1' \mapsto^* v_1$, which is not immediately at hand.

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 10, which has the property that $e \Downarrow e'$ iff $e \mapsto^* e'$ and e' val.

Lemma 31.2. *If $e \Downarrow v$, then for every k stack, $k \triangleright e \mapsto^* k \triangleleft v$.*

The desired result follows by the analogue of Theorem 10.2 on page 85 for $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$, which states that $e \Downarrow v$ iff $e \mapsto^* v$.

For the proof of soundness, it is awkward to reason inductively about the multistep transition from $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft v$, because the intervening steps may involve alternations of evaluation and return states. Instead we regard each $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ machine state as encoding an expression, and show that $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ transitions are simulated by $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$ transitions under this encoding.

Specifically, we define a judgement, $s \looparrowright e$, 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 $s \looparrowright e$. Then we show that if $s \mapsto^* s'$, where s' final, $s \looparrowright e$, and $s' \looparrowright e'$, then e' val and $e \mapsto^* e'$. For this it is enough to show the following two facts:

1. If $s \looparrowright e$ and s final, then e val.

2. If $s \mapsto s'$, $s \looparrowright e$, $s' \looparrowright e'$, and $e' \mapsto^* v$, where v val, then $e \mapsto^* v$.

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 31.3. *If $s \mapsto s'$, $s \looparrowright e$, and $s' \looparrowright e'$, then $e \mapsto^* e'$.*

Corollary 31.4. *$e \mapsto^* \overline{n}$ iff $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft \overline{n}$.*

The remainder of this section is devoted to the proofs of the soundness and completeness lemmas.

31.3.1 Completeness

Proof of Lemma 31.2 on the preceding page. The proof is by induction on an evaluation dynamics for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$.

Consider the evaluation rule

$$\frac{e_1 \Downarrow \mathtt{lam}[\tau_2](x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1;e_2) \Downarrow v} \tag{31.10}$$

For an arbitrary control stack, k, we are to show that $k \triangleright \mathtt{ap}(e_1;e_2) \mapsto^* k \triangleleft v$. Applying both of the inductive hypotheses in succession, interleaved with steps of the abstract machine, we obtain

$$\begin{aligned} k \triangleright \mathtt{ap}(e_1;e_2) &\mapsto k;\mathtt{ap}(-;e_2) \triangleright e_1 \\ &\mapsto^* k;\mathtt{ap}(-;e_2) \triangleleft \mathtt{lam}[\tau_2](x.e) \\ &\mapsto k \triangleright [e_2/x]e \\ &\mapsto^* k \triangleleft v. \end{aligned}$$

The other cases of the proof are handled similarly. □

31.3.2 Soundness

The judgement $s \hookrightarrow e'$, where s is either $k \triangleright e$ or $k \triangleleft e$, is defined in terms of the auxiliary judgement $k \bowtie e = e'$ by the following rules:

$$\frac{k \bowtie e = e'}{k \triangleright e \hookrightarrow e'} \tag{31.11a}$$

$$\frac{k \bowtie e = e'}{k \triangleleft e \hookrightarrow e'} \tag{31.11b}$$

In words, to unravel a state we wrap the stack around the expression. The latter relation is inductively defined by the following rules:

$$\frac{}{\epsilon \bowtie e = e} \tag{31.12a}$$

$$\frac{k \bowtie \mathtt{s}(e) = e'}{k;\mathtt{s}(-) \bowtie e = e'} \tag{31.12b}$$

$$\frac{k \bowtie \mathtt{ifz}(e_1;e_2;x.e_3) = e'}{k;\mathtt{ifz}(-;e_2;x.e_3) \bowtie e_1 = e'} \tag{31.12c}$$

$$\frac{k \bowtie \mathtt{ap}(e_1;e_2) = e}{k;\mathtt{ap}(-;e_2) \bowtie e_1 = e} \tag{31.12d}$$

These judgements both define total functions.

Lemma 31.5. *The judgement $s \looparrowright e$ has mode $(\forall, \exists!)$, and the judgement $k \bowtie e = e'$ 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 31.6. *If $e \mapsto e'$, $k \bowtie e = d$, $k \bowtie e' = d'$, then $d \mapsto d'$.*

Proof. The proof is by rule induction on the transition $e \mapsto e'$. 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 $e = \texttt{ap}(e_1;e_2)$, $e' = \texttt{ap}(e_1';e_2)$, and $e_1 \mapsto e_1'$. We have $k \bowtie e = d$ and $k \bowtie e' = d'$. It follows from Rules (31.12) that $k;\texttt{ap}(-;e_2) \bowtie e_1 = d$ and $k;\texttt{ap}(-;e_2) \bowtie e_1' = d'$. So by induction $d \mapsto d'$, as desired.

For an example of a base case, suppose that $e = \texttt{ap}(\texttt{lam}[\tau_2](x.e);e_2)$ and $e' = [e_2/x]e$ with $e \mapsto e'$ directly. Assume that $k \bowtie e = d$ and $k \bowtie e' = d'$; we are to show that $d \mapsto d'$. We proceed by an inner induction on the structure of k. If $k = \epsilon$, the result follows immediately. Consider, say, the stack $k = k';\texttt{ap}(-;c_2)$. It follows from Rules (31.12) that $k' \bowtie \texttt{ap}(e;c_2) = d$ and $k' \bowtie \texttt{ap}(e';c_2) = d'$. But by the SOS rules $\texttt{ap}(e;c_2) \mapsto \texttt{ap}(e';c_2)$, so by the inner inductive hypothesis we have $d \mapsto d'$, as desired. □

We are now in a position to complete the proof of Lemma 31.3 on page 271.

Proof of Lemma 31.3 on page 271. The proof is by case analysis on the transitions of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$. In each case after unravelling the transition will correspond to zero or one transitions of $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$.

Suppose that $s = k \triangleright \texttt{s}(e)$ and $s' = k;\texttt{s}(-) \triangleright e$. Note that $k \bowtie \texttt{s}(e) = e'$ iff $k;\texttt{s}(-) \bowtie e = e'$, from which the result follows immediately.

Suppose that $s = k;\texttt{ap}(\texttt{lam}[\tau](x.e_1);-) \triangleleft e_2$ and $s' = k \triangleright [e_2/x]e_1$. Let e' be such that $k;\texttt{ap}(\texttt{lam}[\tau](x.e_1);-) \bowtie e_2 = e'$ and let e'' be such that $k \bowtie [e_2/x]e_1 = e''$. Observe that $k \bowtie \texttt{ap}(\texttt{lam}[\tau](x.e_1);e_2) = e'$. The result follows from Lemma 31.6.

□

31.4 Exercises

Chapter 32

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, one could use explicit conditionals to check for and process errors or unusual conditions, but using exceptions is often more convenient, particularly since the transfer to the handler is direct and immediate, rather than indirect via a series of explicit checks.

32.1 Failures

A *failure* is a control mechanism that permits a computation to refuse to return a value to the point of its evaluation. Failure can be detected by *catching* it, diverting evaluation to another expression, called a *handler*. Failure can be turned into success, provided that the handler does not itself fail.

The following grammar defines the syntax of failures:

Expr	e	$::=$	$\texttt{fail}$	$\texttt{fail}$	failure
			$\texttt{catch}(e_1;e_2)$	$\texttt{catch}\,e_1\,\texttt{ow}\,e_2$	handler

The expression $\texttt{fail}$ aborts the current evaluation, and the expression $\texttt{catch}(e_1;e_2)$ handles any failure in e_1 by evaluating e_2 instead.

The statics of failures is straightforward:

$$\frac{}{\Gamma \vdash \texttt{fail} : \tau} \tag{32.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{catch}(e_1;e_2) : \tau} \tag{32.1b}$$

A failure can have any type, because it never returns. The two expressions in a `catch` expression must have the same type, since either might determine the value of that expression.

The dynamics of failures may be given using *stack unwinding*. Evaluation of a `catch` installs a handler on the control stack. Evaluation of a `fail` unwinds the control stack by popping frames until it reaches the nearest enclosing handler, to which control is passed. The handler is evaluated in the context of the surrounding control stack, so that failures within it propagate further up the stack.

Stack unwinding can be defined directly using structural dynamics, but we prefer to make use of the stack machine defined in Chapter 31. In addition to states of the form $k \triangleright e$, which evaluates the expression e on the stack k, and $k \triangleleft e$, which passes the value e to the stack k, we make use of an additional form of state, $k \blacktriangleleft$, which passes a failure up the stack to the nearest enclosing handler.

The set of frames defined in Chapter 31 is extended with the additonal form $\texttt{catch}(-;e_2)$. The transition rules given in Chapter 31 are extended with the following additional rules:

$$\frac{}{k \triangleright \texttt{fail} \mapsto k \blacktriangleleft} \tag{32.2a}$$

$$\frac{}{k \triangleright \texttt{catch}(e_1;e_2) \mapsto k;\texttt{catch}(-;e_2) \triangleright e_1} \tag{32.2b}$$

$$\frac{}{k;\texttt{catch}(-;e_2) \triangleleft v \mapsto k \triangleleft v} \tag{32.2c}$$

$$\frac{}{k;\texttt{catch}(-;e_2) \blacktriangleleft \; \mapsto k \triangleright e_2} \tag{32.2d}$$

$$\frac{(f \neq \texttt{catch}(-;e_2))}{k;f \blacktriangleleft \; \mapsto k \blacktriangleleft} \tag{32.2e}$$

Evaluating `fail` propagates a failure up the stack. Evaluating $\texttt{catch}(e_1;e_2)$ 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. All other frames propagate failures.

The definition of initial state remains the same as for $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$, but we change the definition of final state to include these two forms:

$$\frac{e\ \mathsf{val}}{\epsilon \triangleleft e\ \mathsf{final}} \tag{32.3a}$$

$$\frac{}{\epsilon \blacktriangleleft\ \mathsf{final}} \tag{32.3b}$$

The first of these is as before, corresponding to a normal result with the specified value. The second is new, corresponding to an uncaught exception propagating through the entire program.

It is a straightforward exercise the extend the definition of stack typing given in Chapter 31 to account for the new forms of frame. Using this, safety can be proved by standard means. Note, however, that the meaning of the progress theorem is now significantly different: a well-typed program does not get stuck, but it may well result in an uncaught failure!

Theorem 32.1 (Safety). 1. *If s ok and $s \mapsto s'$, then s' ok.*

2. *If s ok, then either s final or there exists s' such that $s \mapsto s'$.*

32.2 Exceptions

Failures are simplistic in that they do not distinguish different causes, and hence do not permit handlers to react differently to different circumstances. An *exception* is a generalization of a failure that associates a value with the failure. This value is passed to the handler, allowing it to discriminate between various forms of failures, and to pass data appropriate to that form of failure. The type of values associated with exceptions is discussed in Section 32.3 on the following page. For now, we simply assume that there is some type, τ_{exn}, of values associated with a failure.

The syntax of exceptions is given by the following grammar:

$$\begin{array}{lllll}
\mathsf{Expr} & e & ::= & \mathtt{raise[}\tau\mathtt{](}e\mathtt{)} & \mathtt{raise(}e\mathtt{)} & \text{exception} \\
 & & & \mathtt{handle(}e_1\mathtt{;}x\mathtt{.}e_2\mathtt{)} & \mathtt{handle}\,e_1\,\mathtt{ow}\,x \Rightarrow e_2 & \text{handler}
\end{array}$$

The argument to `raise` is evaluated to determine the value passed to the handler. The expression $\mathtt{handle(}e_1\mathtt{;}x\mathtt{.}e_2\mathtt{)}$ binds a variable, x, in the handler, e_2, to which the associated value of the exception is bound, should an exception be raised during the execution of e_1.

The statics of exceptions generalizes that of failures:

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \texttt{raise[}\tau\texttt{](}e\texttt{)} : \tau} \tag{32.4a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{handle(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} : \tau} \tag{32.4b}$$

The dynamics of exceptions is a mild generalization of the dynamics of failures in which we generalize the failure state, $k \blacktriangleleft$, to the exception state, $k \blacktriangleleft e$, which passes a value of type τ_{exn} along with the failure. The syntax of stack frames is extended to include $\texttt{raise[}\tau\texttt{](}-\texttt{)}$ and $\texttt{handle(}-\texttt{;}x\texttt{.}e_2\texttt{)}$. The dynamics of exceptions is specified by the following rules:

$$\frac{}{k \triangleright \texttt{raise[}\tau\texttt{](}e\texttt{)} \mapsto k\texttt{;raise[}\tau\texttt{](}-\texttt{)} \triangleright e} \tag{32.5a}$$

$$\frac{}{k\texttt{;raise[}\tau\texttt{](}-\texttt{)} \triangleleft e \mapsto k \blacktriangleleft e} \tag{32.5b}$$

$$\frac{}{k\texttt{;raise[}\tau\texttt{](}-\texttt{)} \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{32.5c}$$

$$\frac{}{k \triangleright \texttt{handle(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} \mapsto k\texttt{;handle(}-\texttt{;}x\texttt{.}e_2\texttt{)} \triangleright e_1} \tag{32.5d}$$

$$\frac{}{k\texttt{;handle(}-\texttt{;}x\texttt{.}e_2\texttt{)} \triangleleft e \mapsto k \triangleleft e} \tag{32.5e}$$

$$\frac{}{k\texttt{;handle(}-\texttt{;}x\texttt{.}e_2\texttt{)} \blacktriangleleft e \mapsto k \triangleright [e/x]e_2} \tag{32.5f}$$

$$\frac{(f \neq \texttt{handle(}-\texttt{;}x\texttt{.}e_2\texttt{)})}{k\texttt{;}f \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{32.5g}$$

It is a straightforward exercise to extend the safety theorem given in Section 32.1 on page 275 to exceptions.

32.3 Exception Type

The statics of exceptions is parameterized by the type of exception values, τ_{exn}. This type may be chosen arbitrarily, but it must be shared by all exceptions in a program to ensure type safety. For otherwise a handler cannot tell what type of value to expect from an exception, compromising safety.

But how do we choose the type of exceptions? A very naïve choice would be to take τ_{exn} to be the type str, so that, for example, one may write

```
raise "Division by zero error."
```

to signal the obvious arithmetic fault. This is fine as far as it goes, but a handler for such an exception would have to interpret the string if it is to distinguish one exception from another!

Motivated by this, we might choose τ_{exn} to be nat, which amounts to saying that exceptional conditions are coded as natural numbers.[1] This does allow the handler to distinguish one source of failure from another, but makes no provision for associating data with the failure. Moreover, it forces the programmer to impose a single, global convention for indexing the causes of failure, compromising modular development and evolution.

The first concern—how to associate data specific to the type of failure—can be addressed by taking τ_{exn} to be a labelled sum type whose classes are the forms of failure, and whose associated types determine the form of the data attached to the exception. For example, the type τ_{exn} might have the form

$$\tau_{exn} = [\mathtt{div} : \mathtt{unit}, \mathtt{fnf} : \mathtt{string}, \ldots].$$

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.

Using a sum type for τ_{exn} makes it easy for the handler to discriminate on the source of the failure, and to recover the associated data without fear of a type safety violation. For example, we might write

```
try e₁ ow x ⇒
  match x {
    div ⟨⟩ ⇒ e_div
  | fnf s ⇒ e_fnf }
```

to handle the exceptions specified by the sum type given in the preceding paragraph.

The problem with choosing a sum type for τ_{exn} is that it imposes a *static classification* of the sources of failure in a program. There must be one, globally agreed-upon type that classifies all possible forms of failure, and specifies their associated data. Using sums in this manner impedes modular

[1] In Unix these are called errno's, for *error numbers*.

development and evolution, since all of the modules comprising a system must agree on the one, central type of exception values. A better approach is to use *dynamic classification* for exception values by choosing τ_{exn} to be an *extensible sum*, one to which new classes may be added at execution time. This allows separate program modules to introduce their own failure classification scheme without worrying about interference with one another; the initialization of the module generates new classes at run-time that are guaranteed to be distinct from all other classes previously or subsequently generated. (See Chapter 38 for more on dynamic classification.)

32.4 Encapsulation

It is sometimes useful to distinguish expressions that can fail or raise an exception from those that cannot. An expression is called *fallible*, or *exceptional*, if it can fail or raise an exception during its evaluation, and is *infallible*, or *unexceptional*, otherwise. The concept of fallibility is intentionally permissive in that an infallible expression may be considered to be (vacuously) fallible, whereas infallibility is intended to be strict in that an infallible expression cannot fail. Consequently, if e_1 and e_2 are two infallible expressions both of whose values are required in a computation, we may evaluate them in either order without affecting the outcome. If, on the other hand, one or both are fallible, then the outcome of the computation is sensitive to the evaluation order (whichever fails first determines the overall result).

To formalize this distinction we distinguish two *modes* of expression, the fallible and the infallible, linked by a *modality* classifying the fallible expressions of a type.

Type	τ	$::=$	$\texttt{fallible}(\tau)$	$\tau\ \texttt{fallible}$	fallible
Fall	f	$::=$	$\texttt{fail}$	$\texttt{fail}$	failure
			$\texttt{succ}(e)$	$\texttt{succ}\,e$	success
			$\texttt{try}(e;x.f_1;f_2)$	$\texttt{let}\,\texttt{fall}(x)\,\texttt{be}\,e\,\texttt{in}\,f_1\,\texttt{ow}\,f_2$	handler
Infall	e	$::=$	x	x	variable
			$\texttt{fall}(f)$	$\texttt{fall}\,f$	fallible
			$\texttt{try}(e;x.e_1;e_2)$	$\texttt{let}\,\texttt{fall}(x)\,\texttt{be}\,e\,\texttt{in}\,e_1\,\texttt{ow}\,e_2$	handler

The type τ `fallible` is the type of encapsulated fallible expressions of type τ. Fallible expressions include failures, successes (infallible expressions thought of as vacuously fallible), and handlers that intercept failures, but which may itself fail. Infallible expressions include variables, encapsulated fallible expressions, and handlers that intercepts failures, always yielding an infallible result.

The statics of encapsulated failures consists of two judgement forms, $\Gamma \vdash e : \tau$ for infallible expressions and $\Gamma \vdash f \sim \tau$ for fallible expressions. These judgements are defined by the following rules:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{32.6a}$$

$$\frac{\Gamma \vdash f \sim \tau}{\Gamma \vdash \texttt{fall}(f) : \texttt{fallible}(\tau)} \tag{32.6b}$$

$$\frac{\Gamma \vdash e : \texttt{fallible}(\tau) \quad \Gamma, x : \tau \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \texttt{try}(e; x . e_1; e_2) : \tau'} \tag{32.6c}$$

$$\frac{}{\Gamma \vdash \texttt{fail} \sim \tau} \tag{32.6d}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{succ}(e) \sim \tau} \tag{32.6e}$$

$$\frac{\Gamma \vdash e : \texttt{fallible}(\tau) \quad \Gamma, x : \tau \vdash f_1 \sim \tau' \quad \Gamma \vdash f_2 \sim \tau'}{\Gamma \vdash \texttt{try}(e; x . f_1; f_2) \sim \tau'} \tag{32.6f}$$

Rule (32.6c) specifies that a handler may be used to turn a fallible expression (encapsulated by e) into an infallible computation, provided that the result is infallible regardless of whether the encapsulated expression succeeds or fails.

The dynamics of encapsulated failures is readily derived, though some care must be taken with the elimination form for the modality.

$$\frac{}{\texttt{fall}(f)\ \mathsf{val}} \tag{32.7a}$$

$$\frac{}{k \triangleright \texttt{try}(e; x . e_1; e_2) \mapsto k ; \texttt{try}(-; x . e_1; e_2) \triangleright e} \tag{32.7b}$$

$$\frac{}{k ; \texttt{try}(-; x . e_1; e_2) \triangleleft \texttt{fall}(f) \mapsto k ; \texttt{try}(-; x . e_1; e_2) ; \texttt{fall}(-) \triangleright f} \tag{32.7c}$$

$$\frac{}{k \triangleright \texttt{fail} \mapsto k \blacktriangleleft} \tag{32.7d}$$

$$\frac{}{k \triangleright \texttt{succ}(e) \mapsto k;\texttt{succ}(-) \triangleright e} \tag{32.7e}$$

$$\frac{e\ \mathsf{val}}{k \triangleright \texttt{succ}(e) \mapsto k \triangleleft \texttt{succ}(e)} \tag{32.7f}$$

$$\frac{e\ \mathsf{val}}{k;\texttt{try}(-;x.e_1;e_2);\texttt{fall}(-) \triangleleft e \mapsto k \triangleright [e/x]e_1} \tag{32.7g}$$

$$\frac{}{k;\texttt{try}(-;x.e_1;e_2);\texttt{fall}(-) \blacktriangleleft\ \mapsto k \triangleright e_2} \tag{32.7h}$$

We have omitted the rules for the fallible form of handler; they are similar to Rules (32.7b) to (32.7b) and (32.7g) to (32.7h), albeit with infallible subexpresions e_1 and e_2 replaced by fallible subexpressions f_1 and f_2.

An initial state has the form $k \triangleright e$, where e is an infallible expression, and k is a stack of suitable type. Consequently, a fallible expression, f, can only be evaluated on a stack of the form

$$k;\texttt{try}(-;x.e_1;e_2);\texttt{fall}(-)$$

in which a handler for any failure that may arise from f is present. Therefore, a final state has the form $\epsilon \triangleleft e$, where e val; no uncaught failure can arise.

32.5 Exercises

Chapter 33

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*, where the qualification "first class" stresses that they are ordinary values with an indefinite lifetime 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.

33.1 Informal Overview

We will extend $\mathcal{L}\{\rightarrow\}$ with the type $\texttt{cont}(\tau)$ of continuations accepting values of type τ. 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 throw[τ](e_1;e_2), which restores the value of e_1 to the control stack that is the value of 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 nat → 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.

Here is the solution in $\mathcal{L}\{$nat ⇀$\}$, 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 version 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 τ cont and a function f of type $\tau' \rightarrow \tau$, return a continuation k' of type τ' cont with the following behavior: throwing a value v' of type τ' 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
```

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!

33.2 Semantics of Continuations

We extend the language of $\mathcal{L}\{\rightarrow\}$ expressions with these additional forms:

Type	τ	::=	cont(τ)	τ cont	continuation
Expr	e	::=	letcc[τ](x.e)	letcc x in e	mark
			throw[τ](e_1;e_2)	throw e_1 to e_2	goto
			cont(k)	cont(k)	continuation

The expression cont(k) is a reified control stack, which arises during evaluation.

The statics of this extension is defined by the following rules:

$$\frac{\Gamma, x : \texttt{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{letcc}[\tau](x.e) : \tau} \tag{33.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \texttt{cont}(\tau_1)}{\Gamma \vdash \texttt{throw}[\tau'](e_1;e_2) : \tau'} \tag{33.1b}$$

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:

$$\frac{k : \tau}{\Gamma \vdash \texttt{cont}(k) : \texttt{cont}(\tau)} \tag{33.2}$$

A continuation value `cont`(k) has type `cont`(τ) exactly if it is a stack accepting values of type τ.

To define the dynamics we extend $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ stacks with two new forms of frame:

$$\frac{e_2 \ \mathsf{exp}}{\texttt{throw}[\tau](-;e_2) \ \mathsf{frame}} \tag{33.3a}$$

$$\frac{e_1 \ \mathsf{val}}{\texttt{throw}[\tau](e_1;-) \ \mathsf{frame}} \tag{33.3b}$$

Every reified control stack is a value:

$$\frac{k \ \mathsf{stack}}{\texttt{cont}(k) \ \mathsf{val}} \tag{33.4}$$

The transition rules for the continuation constructs are as follows:

$$\frac{}{k \triangleright \texttt{letcc}[\tau](x.e) \mapsto k \triangleright [\texttt{cont}(k)/x]e} \tag{33.5a}$$

$$\frac{}{k;\texttt{throw}[\tau](v;-) \triangleleft \texttt{cont}(k') \mapsto k' \triangleleft v} \tag{33.5b}$$

$$\frac{}{k \triangleright \texttt{throw}[\tau](e_1;e_2) \mapsto k;\texttt{throw}[\tau](-;e_2) \triangleright e_1} \tag{33.5c}$$

$$\frac{e_1 \ \mathsf{val}}{k;\texttt{throw}[\tau](-;e_2) \triangleleft e_1 \mapsto k;\texttt{throw}[\tau](e_1;-) \triangleright e_2} \tag{33.5d}$$

Evaluation of a `letcc` expression duplicates the control stack; evaluation of a `throw` expression destroys the current control stack.

The safety of this extension of $\mathcal{L}\{\rightarrow\}$ may be established by a simple extension to the safety proof for $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ given in Chapter 31.

We need only add typing rules for the two new forms of frame, which are as follows:

$$\frac{e_2 : \texttt{cont}(\tau)}{\texttt{throw}[\tau](-;e_2) : \tau \Rightarrow \tau'} \tag{33.6a}$$

$$\frac{e_1 : \tau \quad e_1\ \mathsf{val}}{\texttt{throw}[\tau](e_1;-) : \texttt{cont}(\tau) \Rightarrow \tau'} \tag{33.6b}$$

The rest of the definitions remain as in Chapter 31.

Lemma 33.1 (Canonical Forms). *If e : $\texttt{cont}(\tau)$ and e val, then $e = \texttt{cont}(k)$ for some k such that $k : \tau$.*

Theorem 33.2 (Safety). *1. If s ok and $s \mapsto s'$, then s' ok.*

2. If s ok, then either s final or there exists s' such that $s \mapsto s'$.

33.3 Coroutines

A familiar pattern of control flow in a program distinguishes the *main routine* of a computation, which represents the principal control path of the program, from a *sub-routine*, which represents a subsidiary path that performs some auxiliary computation. The main routine invokes the the subroutine by passing it a data value, its *argument*, and a *control point* to return to once it has completed its work. This arrangement is asymmetric in that the main routine plays the active role, whereas the subroutine is passive. In particular the subroutine passes control directly to the return point without itself providing a return point with which it can be called back. A *coroutine* is a symmetric pattern of control flow in which each routine passes to the other the return point of the call. The asymmetric call/return pattern is symmetrized to a call/call pattern in which each routine is effectively a subroutine of the other. (This raises an interesting question of how the interaction commences, which we will discuss in more detail below.)

To see how coroutines are implemented in terms of continuations, it is best to think of the "steady state" interaction between the two routines, leaving the initialization phase to be discussed separately. A routine is represented by a continuation that, when invoked, is passed a data item, whose type is shared between the two routines, and a return continuation, which represents the partner routine. Crucially, the argument type of the other continuation is again of the very same form, consisting of a data item and another return continuation. If we think of the coroutine as a *trajectory* through a succession of such continuations, then the *state* of the continuation (which changes as the interaction progresses) satisfies the type isomorphism

$$\texttt{state} \cong (\tau \times \texttt{state})\ \texttt{cont},$$

where τ is the type of data exchanged by the routines. The solution to such an isomorphism is, of course, the recursive type

$$\texttt{state} = \mu t\,.\,(\tau \times t)\ \texttt{cont}.$$

Thus a state, s, encapsulates a pair consisting of a value of type τ together with another state.

The routines pass control from one to the other by calling the function `resume` of type

$$\tau \times \texttt{state} \rightarrow \tau \times \texttt{state}.$$

That is, given a datum, d, and a state, s, the application $\texttt{resume}(\langle d,s\rangle)$ passes d and its own return address to the routine represented by the state s. The function `resume` is defined by the following expression:

$$\lambda\ (\langle x,s\rangle : \tau \times \texttt{state}.\,\texttt{letcc}\,k\,\texttt{in}\,\texttt{throw}\,\langle x,\texttt{fold}(k)\rangle\,\texttt{to}\,\texttt{unfold}(s))$$

When applied, this function seizes the current continuation, and passes the given datum and this continuation to the partner routine, using the isomorphism between `state` and $(\tau \times \texttt{state})\ \texttt{cont}$.

The general form of a coroutine consists of a loop that, on each iteration, takes a datum, d, and a state, s, performs a transformation on d, resuming its partner routine with the result, d', of the transformation. The function `corout` builds a coroutine from a data transformation routine; it has type

$$(\tau \rightarrow \tau) \rightarrow (\tau \times \texttt{state}) \rightarrow \tau'.$$

The result type, τ', is arbitrary, because the routine never returns to the call site. A coroutine is shut down by an explicit exit operation, which will be specified shortly. The function `corout` is defined by the following expression (with types omitted for concision):

$$\lambda\,\mathit{next}.\,\texttt{fix}\,\mathit{loop}\,\texttt{is}\,\lambda\,\langle d,s\rangle.\,\mathit{loop}(\texttt{resume}(\langle \mathit{next}(d),s\rangle)).$$

Each time through the loop, the partner routine, s, is resumed with the updated datum given by applying *next* to the current datum, d.

Let ρ be the ultimate type of a computation consisting of two interacting coroutines that exchanges values of type τ during their execution. The function run, which has type

$$\tau \rightarrow ((\rho\,\texttt{cont} \rightarrow \tau \rightarrow \tau) \times (\rho\,\texttt{cont} \rightarrow \tau \rightarrow \tau)) \rightarrow \rho,$$

takes an initial value of type τ and two routines, each of type

$$\rho\,\texttt{cont} \rightarrow \tau \rightarrow \tau,$$

and builds a coroutine of type ρ from them. The first argument to each routine is the exit point, and the result is a data transformation operation. The definition of run begins as follows:

$$\lambda\,\mathit{init}.\,\lambda\,\langle r_1, r_2\rangle.\,\texttt{letcc}\,\mathit{exit}\,\texttt{in}\,\texttt{let}\,r_1'\,\texttt{be}\,r_1(\mathit{exit})\,\texttt{in}\,\texttt{let}\,r_2'\,\texttt{be}\,r_2(\mathit{exit})\,\texttt{in}\ldots$$

First, run establishes an exit point that is passed to the two routines to obtain their data transformation components. This allows either or both of the routines to terminate the computation by throwing the ultimate result value to exit. The implementation of run continues as follows:

$$\texttt{corout}(r_2')(\texttt{letcc}\,k\,\texttt{in}\,\texttt{corout}(r_1')(\langle \mathit{init}, \texttt{fold}(k)\rangle))$$

The routine r_1' is called with the initial datum, *init*, and the state $\texttt{fold}(k)$, where k is the continuation corresponding to the call to r_2'. The first resume from the coroutine built from r_1' will cause the coroutine built from r_2' to be initiated. At this point the steady state behavior is in effect, with the two routines exchanging control using resume. Either may terminate the computation by throwing a result value, v, of type ρ to the continuation *exit*.

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

[1] In practice the input and output state are implicit, but we prefer to make them explicit for the sake of clarity.

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 $\texttt{EMIT}(\langle v,\langle i,o\rangle\rangle)$, where v 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 τ of data exchanged by the routines is the labelled sum type

$$[\texttt{OK}:\tau_i\,\texttt{list}\times\tau_o\,\texttt{list},\texttt{EMIT}:\tau_i\,\texttt{opt}\times(\tau_i\,\texttt{list}\times\tau_o\,\texttt{list})].$$

This type specifies the message protocol between the producer and the consumer described in the preceding paragraph.

The producer, `producer`, is defined by the expression

$$\lambda\,\mathit{exit}.\,\lambda\,\mathit{msg}.\,\texttt{case}\,\mathit{msg}\,\{b_1\mid b_2\mid b_3\},$$

where the first branch, b_1, is

$$\texttt{OK}\cdot\langle\texttt{nil},\mathit{os}\rangle\Rightarrow\texttt{EMIT}\cdot\langle\texttt{null},\langle\texttt{nil},\mathit{os}\rangle\rangle$$

and the second branch, b_2, is

$$\texttt{OK}\cdot\langle\texttt{cons}(i;\mathit{is}),\mathit{os}\rangle\Rightarrow\texttt{EMIT}\cdot\langle\texttt{just}(i),\langle\mathit{is},\mathit{os}\rangle\rangle,$$

and the third branch, b_3, is

$$\texttt{EMIT}\cdot\_\Rightarrow\texttt{error}.$$

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, `consumer`, is defined by the expression

$$\lambda\,\mathit{exit}.\,\lambda\,\mathit{msg}.\,\texttt{case}\,\mathit{msg}\,\{b_1'\mid b_2'\mid b_3'\},$$

where the first branch, b_1', is

$$\texttt{EMIT}\cdot\langle\texttt{null},\langle\_,\mathit{os}\rangle\rangle\Rightarrow\texttt{throw}\,\mathit{os}\,\texttt{to}\,\mathit{exit},$$

the second branch, b'_2, is

$$\texttt{EMIT} \cdot \langle \texttt{just(}i\texttt{)}, \langle is, os \rangle \rangle \Rightarrow \texttt{OK} \cdot \langle is, \texttt{cons(}f\texttt{(}i\texttt{);}os\texttt{)} \rangle,$$

and the third branch, b'_3, is

$$\texttt{OK} \cdot \_ \Rightarrow \texttt{error}.$$

The consumer dispatches on the emitted datum. If it is absent, the output channel state is passed to *exit* as the ultimate value of the computation. If it is present, the function f (unspecified here) of type $\tau_i \rightarrow \tau_o$ 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 datum, `init`, 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

$$\texttt{run(init)(}\langle\texttt{producer,consumer}\rangle\texttt{)},$$

which sets up the coroutines as described earlier.

While 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*, since it is based on explicit yields, rather than implicit yields imposed by asynchronous events such as timer interrupts.

33.4 Exercises

1. Study the short-circuit multiplication example carefully to be sure you understand why it works!

2. Attempt to solve the problem of composing a continuation with a function yourself, before reading the solution.

3. Simulate the evaluation of `compose` (f, k) on the empty stack. Observe that the control stack substituted for x is

$$\epsilon;\mathtt{throw}[\tau](-;k);\mathtt{ap}(f;-)$$

This stack is returned from `compose`. Next, simulate the behavior of throwing a value v' to this continuation. Observe that the stack is reinstated and that v' is passed to it.

Part XII

Types and Propositions

Chapter 34

Constructive Logic

The correspondence between *propositions* and *types*, and the associated correspondence between *proofs* and *programs*, is the central organizing principle of programming languages. A type specifies a behavior, and a program implements it. Similarly, a proposition poses a problem, and a proof solves it. A statics relates a program to the type it implements, and a dynamics relates a program to its simplification by an execution step. Similarly, a formal logical system relates a proof to the proposition it proves, and proof reduction relates equivalent proofs. The structural rule of substitution underlies the decomposition of a program into separate modules. Similarly, the structural rule of transitivity underlies the decomposition of a theorem into lemmas.

These correspondences are neither accidental nor incidental. The *propositions as types principle*,[1] identifies propositions with types and proofs with programs. According to this principle, a proposition *is* the type of its proofs, and a proof *is* a program of that type. Consequently, every theorem has *computational content*, the its proof viewed as a program, and every program has *mathematical content*, the proof that the program represents.

Can every conceivable form of proposition also be construed as a type? Does every type correspond to a proposition? Must every proof have computational content? Is every program a proof of a theorem? To answer these questions would require a book of its own (and still not settle the matter). From a constructive perspective we may say that type theory en-

[1] The propositions-as-types principle is sometimes called the *Curry-Howard Isomorphism*. Although it is arguably snappier, this name ignores the essential contributions of Arend Heyting, Nicolaas deBruijn, and Per Martin-Löf to the development of the propositions-as-types principle.

riches logic to incorporate not only types of proofs, but also types for the objects of study. In this sense logic is a particular mode of use of type theory. If we think of type theory as a comprehensive view of mathematics, this implies that, contrary to conventional wisdom, logic is based on mathematics, rather than mathematics on logic!

In this chapter we introduce the propositions-as-types correspondence for a particularly simple system of logic, called *propositional contructive logic*. In Chapter 35 we will extend the correspondence to *propositional classical logic*. This will give rise to a computational interpretation of classical proofs that makes essential use of continuations.

34.1 Constructive Semantics

Constructive logic is concerned with two judgements, ϕ prop, stating that ϕ expresses a proposition, and ϕ true, stating that ϕ 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. *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 ϕ expresses an unsolved problem, such as the famous $\mathsf{P} \overset{?}{=} \mathsf{NP}$ 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. Since 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.

Having said that, some propositions *are* decidable, and hence may be considered to be either true or false. For example, if ϕ expresses an inequality between natural numbers, then ϕ 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 one 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's hard, but we muddle through somehow.

34.2 Constructive Logic

The judgements ϕ prop and ϕ true of constructive logic are rarely of interest by themselves, but rather in the context of a hypothetical judgement of the form

$$\phi_1 \ \mathsf{true}, \ldots, \phi_n \ \mathsf{true} \vdash \phi \ \mathsf{true}.$$

This judgement expresses that the proposition ϕ 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 judgement ϕ true.

The structural properties of the hypothetical judgement, when specialized to constructive logic, define what we mean by reasoning under hypotheses:

$$\frac{}{\Gamma, \phi \ \mathsf{true} \vdash \phi \ \mathsf{true}} \tag{34.1a}$$

$$\frac{\Gamma \vdash \phi_1 \ \mathsf{true} \quad \Gamma, \phi_1 \ \mathsf{true} \vdash \phi_2 \ \mathsf{true}}{\Gamma \vdash \phi_2 \ \mathsf{true}} \tag{34.1b}$$

$$\frac{\Gamma \vdash \phi_2 \ \mathsf{true}}{\Gamma, \phi_1 \ \mathsf{true} \vdash \phi_2 \ \mathsf{true}} \tag{34.1c}$$

$$\frac{\Gamma, \phi_1 \ \mathsf{true}, \phi_1 \ \mathsf{true} \vdash \phi_2 \ \mathsf{true}}{\Gamma, \phi_1 \ \mathsf{true} \vdash \phi_2 \ \mathsf{true}} \tag{34.1d}$$

$$\frac{\Gamma_1, \phi_2\ \mathsf{true}, \phi_1\ \mathsf{true}, \Gamma_2 \vdash \phi\ \mathsf{true}}{\Gamma_1, \phi_1\ \mathsf{true}, \phi_2\ \mathsf{true}, \Gamma_2 \vdash \phi\ \mathsf{true}} \tag{34.1e}$$

The last two rules are implicit in that we regard Γ as a *set* of hypotheses, so that two "copies" are as good as one, and the order of hypotheses does not matter.

34.2.1 Rules of Provability

The syntax of propositional logic is given by the following grammar:

Prop	ϕ	$::=$	$\mathtt{true}$	$\top$	truth
			$\mathtt{false}$	$\bot$	falsity
			$\mathtt{and}(\phi_1;\phi_2)$	$\phi_1 \wedge \phi_2$	conjunction
			$\mathtt{or}(\phi_1;\phi_2)$	$\phi_1 \vee \phi_2$	disjunction
			$\mathtt{imp}(\phi_1;\phi_2)$	$\phi_1 \supset \phi_2$	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.

$$\frac{}{\Gamma \vdash \top\ \mathsf{true}} \tag{34.2a}$$

(no elimination rule)

$$\tag{34.2b}$$

Conjunction Conjunction expresses the truth of both of its conjuncts.

$$\frac{\Gamma \vdash \phi_1\ \mathsf{true} \quad \Gamma \vdash \phi_2\ \mathsf{true}}{\Gamma \vdash \phi_1 \wedge \phi_2\ \mathsf{true}} \tag{34.3a}$$

$$\frac{\Gamma \vdash \phi_1 \wedge \phi_2\ \mathsf{true}}{\Gamma \vdash \phi_1\ \mathsf{true}} \tag{34.3b}$$

$$\frac{\Gamma \vdash \phi_1 \wedge \phi_2 \; \mathsf{true}}{\Gamma \vdash \phi_2 \; \mathsf{true}} \tag{34.3c}$$

Implication Implication states the truth of a proposition under an assumption.

$$\frac{\Gamma, \phi_1 \; \mathsf{true} \vdash \phi_2 \; \mathsf{true}}{\Gamma \vdash \phi_1 \supset \phi_2 \; \mathsf{true}} \tag{34.4a}$$

$$\frac{\Gamma \vdash \phi_1 \supset \phi_2 \; \mathsf{true} \quad \Gamma \vdash \phi_1 \; \mathsf{true}}{\Gamma \vdash \phi_2 \; \mathsf{true}} \tag{34.4b}$$

Falsehood Falsehood expresses the trivially false (refutable) proposition.

(no introduction rule)

$$\tag{34.5a}$$

$$\frac{\Gamma \vdash \perp \; \mathsf{true}}{\Gamma \vdash \phi \; \mathsf{true}} \tag{34.5b}$$

Disjunction Disjunction expresses the truth of either (or both) of two propositions.

$$\frac{\Gamma \vdash \phi_1 \; \mathsf{true}}{\Gamma \vdash \phi_1 \vee \phi_2 \; \mathsf{true}} \tag{34.6a}$$

$$\frac{\Gamma \vdash \phi_2 \; \mathsf{true}}{\Gamma \vdash \phi_1 \vee \phi_2 \; \mathsf{true}} \tag{34.6b}$$

$$\frac{\Gamma \vdash \phi_1 \vee \phi_2 \; \mathsf{true} \quad \Gamma, \phi_1 \; \mathsf{true} \vdash \phi \; \mathsf{true} \quad \Gamma, \phi_2 \; \mathsf{true} \vdash \phi \; \mathsf{true}}{\Gamma \vdash \phi \; \mathsf{true}} \tag{34.6c}$$

Negation The negation, $\neg\phi$, of a proposition, ϕ, may be defined as the implication $\phi \supset\perp$. This means that $\neg\phi \; \mathsf{true}$ if $\phi \; \mathsf{true} \vdash \perp \; \mathsf{true}$, which is to say that the truth of ϕ is *refutable* in that we may derive a proof of falsehood from any purported proof of ϕ. Because constructive truth is identified with the existence of a proof, the implied semantics of negation is rather strong. In particular, a problem, ϕ, 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.

34.2.2 Rules of Proof

The key to the propositions-as-types principle is to make explict the forms of proof. The basic judgement ϕ true, which states that ϕ has a proof, is replaced by the judgement $p : \phi$, stating that p is a proof of ϕ. (Sometimes p is called a "proof term", but we will simply call p a "proof.") The hypothetical judgement 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 Γ range over such hypothesis lists, subject to the restriction that no variable occurs more than once.

The rules of constructive propositional logic may be restated using proof terms as follows.

$$\frac{}{\Gamma \vdash \mathtt{trueI} : \top} \tag{34.7a}$$

$$\frac{\Gamma \vdash p_1 : \phi_1 \quad \Gamma \vdash p_2 : \phi_2}{\Gamma \vdash \mathtt{andI}(p_1;p_2) : \phi_1 \wedge \phi_2} \tag{34.7b}$$

$$\frac{\Gamma \vdash p_1 : \phi_1 \wedge \phi_2}{\Gamma \vdash \mathtt{andE[l]}(p_1) : \phi_1} \tag{34.7c}$$

$$\frac{\Gamma \vdash p_1 : \phi_1 \wedge \phi_2}{\Gamma \vdash \mathtt{andE[r]}(p_1) : \phi_2} \tag{34.7d}$$

$$\frac{\Gamma, x : \phi_1 \vdash p_2 : \phi_2}{\Gamma \vdash \mathtt{impI}[\phi_1](x\,.\,p_2) : \phi_1 \supset \phi_2} \tag{34.7e}$$

$$\frac{\Gamma \vdash p : \phi_1 \supset \phi_2 \quad \Gamma \vdash p_1 : \phi_1}{\Gamma \vdash \mathtt{impE}(p;p_1) : \phi_2} \tag{34.7f}$$

$$\frac{\Gamma \vdash p : \bot}{\Gamma \vdash \mathtt{falseE}[\phi](p) : \phi} \tag{34.7g}$$

$$\frac{\Gamma \vdash p_1 : \phi_1}{\Gamma \vdash \mathtt{orI[l]}[\phi_2](p_1) : \phi_1 \vee \phi_2} \tag{34.7h}$$

$$\frac{\Gamma \vdash p_2 : \phi_2}{\Gamma \vdash \mathtt{orI[r]}[\phi_1](p_2) : \phi_1 \vee \phi_2} \tag{34.7i}$$

$$\frac{\Gamma \vdash p : \phi_1 \vee \phi_2 \quad \Gamma, x_1 : \phi_1 \vdash p_1 : \phi \quad \Gamma, x_2 : \phi_2 \vdash p_2 : \phi}{\Gamma \vdash \mathtt{orE}[\phi_1;\phi_2](p;x\,.\,p_1;y\,.\,p_2) : \phi} \tag{34.7j}$$

34.3 Propositions as Types

Reviewing the rules of proof for constructive logic, we observe a striking correspondence between them and the rules for forming 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.

This correspondence extends to other forms of proposition as well, as summarized by the following chart relating a proposition, ϕ, to a type ϕ^*:

Proposition	*Type*
$\top$	`unit`
$\bot$	`void`
$\phi_1 \wedge \phi_2$	$\phi_1^* \times \phi_2^*$
$\phi_1 \supset \phi_2$	$\phi_1^* \rightarrow \phi_2^*$
$\phi_1 \vee \phi_2$	$\phi_1^* + \phi_2^*$

It is obvious that this correspondence is invertible, so that we may associate a proposition with each product, sum, or function type.

Importantly, this correspondence extends to the introductory and eliminatory forms of proofs and programs as well:

Proof	*Program*
`trueI`	$\langle\rangle$
$\mathtt{falseE[}\phi\mathtt{](}p\mathtt{)}$	$\mathtt{abort(}p^*\mathtt{)}$
$\mathtt{andI(}p_1;p_2\mathtt{)}$	$\langle p_1^*, p_2^*\rangle$
$\mathtt{andE[l](}p\mathtt{)}$	$p^* \cdot \mathtt{l}$
$\mathtt{andE[r](}p\mathtt{)}$	$p^* \cdot \mathtt{r}$
$\mathtt{impI[}\phi_1\mathtt{](}x_1\,.\,p_2\mathtt{)}$	$\lambda\,\mathtt{(}x_1{:}\phi_1^*.\,p_2^*\mathtt{)}$
$\mathtt{impE(}p;p_1\mathtt{)}$	$p^*\,\mathtt{(}p_1^*\mathtt{)}$
$\mathtt{orI[l][}\phi_2\mathtt{](}p\mathtt{)}$	$\mathtt{l} \cdot p^*$
$\mathtt{orI[r][}\phi_1\mathtt{](}p\mathtt{)}$	$\mathtt{r} \cdot p^*$
$\mathtt{orE[}\phi_1;\phi_2\mathtt{](}p;x_1\,.\,p_1;x_2\,.\,p_2\mathtt{)}$	$\mathtt{case}\;p^*\;\{\mathtt{l}\cdot x_1 \Rightarrow p_1^* \mid \mathtt{r}\cdot x_2 \Rightarrow p_2^*\}$

Here again the correspondence is easily seen to be invertible, so that we may regard a program of a product, sum, or function type as a proof of the corresponding proposition.

Theorem 34.1.

1. *If ϕ prop, then ϕ^* type.*
2. *If $\Gamma \vdash p : \phi$, then $\Gamma^* \vdash p^* : \phi^*$.*

The foregoing correspondence between the statics of propositions and proofs on one hand, and types and programs on the other extends also to the dynamics, by applying the inversion principle stating that eliminatory forms are post-inverse to introductory forms. The dynamic correspondence may be expressed by the validity of these definitional equivalences under the static correspondences given above:

$$\begin{array}{rcl}
\texttt{andE[l](andI(}p;q\texttt{))} & \equiv & p \\
\texttt{andE[r](andI(}p;q\texttt{))} & \equiv & q \\
\texttt{impE(impI[}\phi\texttt{](}x\,.\,p_2\texttt{);}p_1\texttt{)} & \equiv & [p_1/x]p_2 \\
\texttt{orE[}\phi_1;\phi_2\texttt{](orI[l][}\phi_2\texttt{](}p\texttt{);}x_1\,.\,p_2;x_2\,.\,p_2\texttt{)} & \equiv & [p/x_1]p_1 \\
\texttt{orE[}\phi_1;\phi_2\texttt{](orI[r][}\phi_1\texttt{](}p\texttt{);}x_1\,.\,p_1;x_2\,.\,p_2\texttt{)} & \equiv & [p/x_2]p_2
\end{array}$$

Observe that these equations are all valid under the static correspondence given above. For example, the first of these equations corresponds to the definitional equivalence $\langle e_1, e_2\rangle \cdot \texttt{l} \equiv e_1$, which is valid for the lazy interpretation of ordered pairs.

The significance of the dynamic correspondence is that it assigns *computational content* to proofs: a proof in constructive propositional logic may be read as a program. Put the other way around, it assigns *logical content* to programs: every expression of product, sum, or function type may be read as a proof of a proposition.

34.4 Exercises

Chapter 35

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. One 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*, the assertion that $\phi \vee \neg\phi$ true is valid for all propositions ϕ. Constructively, this principle is not universally valid, because it would mean that every proposition either has a proof or a refutation, which is manifestly not the case. Classically, however, the law of the excluded middle is valid, because every proposition is either true or false. The discrepancy between the constructive and classical interpretations can be attributed to the different meanings given to disjunction and negation by the two logics. In particular the classical truth of a disjunction cannot guarantee the constructive truth of one or the other disjunct. Something other than a constructive proof must be admitted as evidence for a disjunction if the law of the excluded middle is to hold true. And it is precisely for this reason that a classical proof expresses less than does a constructive proof of the same proposition.

Despite this weakness, classical logic admits a computational interpretation similar to, but somewhat less expressive than, that of constructive logic. The dynamics of classical proofs is derived from the complementarity of truth and falsity. A computation is initiated by juxtaposing a proof and a refutation—or, in programming terms, an expression and a *continuation*, or *control stack*. Continuations are essential to the meaning of classical proofs. In particular, the proof of the law of the excluded middle will be seen to equivocate between proving and refuting a proposition, using continuations to avoid getting caught in a contradiction.

35.1 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 ϕ true it is enough to show that ϕ false entails a contradiction, and, conversely, to show that ϕ false it is enough to show that ϕ true leads to a contradiction. While the second of these is constructively valid, the first is fundamentally classical, expressing the principle of indirect proof.

35.1.1 Provability and Refutability

There are three judgement forms in classical logic:

1. ϕ true, stating that the proposition ϕ is provable;

2. ϕ false, stating that the proposition ϕ is refutable;

3. #, stating that a contradiction has been derived.

We will consider hypothetical judgements of the form

$$\phi_1 \text{ false}, \ldots, \phi_m \text{ false } \psi_1 \text{ true}, \ldots, \psi_n \text{ true} \vdash J,$$

where J is any of the three basic judgement forms. The hypotheses are divided into two "zones" for convenience. We let Γ stand for a finite set of "true" hypotheses, and Δ stand for a finite set of "false" hypotheses.

The rules of classical logic are organized around the symmetry between truth and falsity, which is mediated by the contradiction judgement.

The hypothetical judgement is reflexive:

$$\frac{}{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \phi\ \mathsf{false}} \tag{35.1a}$$

$$\frac{}{\Delta\ \Gamma, \phi\ \mathsf{true} \vdash \phi\ \mathsf{true}} \tag{35.1b}$$

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.

$$\frac{\Delta\ \Gamma \vdash \phi\ \mathsf{false} \quad \Delta\ \Gamma \vdash \phi\ \mathsf{true}}{\Delta\ \Gamma \vdash \#} \tag{35.1c}$$

$$\frac{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \#}{\Delta\ \Gamma \vdash \phi\ \mathsf{true}} \tag{35.1d}$$

$$\frac{\Delta\ \Gamma, \phi\ \mathsf{true} \vdash \#}{\Delta\ \Gamma \vdash \phi\ \mathsf{false}} \tag{35.1e}$$

Truth is trivially true, and cannot be refuted.

$$\frac{}{\Delta\ \Gamma \vdash \top\ \mathsf{true}} \tag{35.1f}$$

A conjunction is true if both conjuncts are true, and is false if either conjunct is false.

$$\frac{\Delta\ \Gamma \vdash \phi_1\ \mathsf{true} \quad \Delta\ \Gamma \vdash \phi_2\ \mathsf{true}}{\Delta\ \Gamma \vdash \phi_1 \wedge \phi_2\ \mathsf{true}} \tag{35.1g}$$

$$\frac{\Delta\ \Gamma \vdash \phi_1\ \mathsf{false}}{\Delta\ \Gamma \vdash \phi_1 \wedge \phi_2\ \mathsf{false}} \tag{35.1h}$$

$$\frac{\Delta\ \Gamma \vdash \phi_2\ \mathsf{false}}{\Delta\ \Gamma \vdash \phi_1 \wedge \phi_2\ \mathsf{false}} \tag{35.1i}$$

Falsity is trivially false, and cannot be proved.

$$\frac{}{\Delta\ \Gamma \vdash \bot\ \mathsf{false}} \tag{35.1j}$$

A disjunction is true if either disjunct is true, and is false if both disjuncts are false.

$$\frac{\Delta\ \Gamma \vdash \phi_1 \ \mathsf{true}}{\Delta\ \Gamma \vdash \phi_1 \vee \phi_2 \ \mathsf{true}} \tag{35.1k}$$

$$\frac{\Delta\ \Gamma \vdash \phi_2 \ \mathsf{true}}{\Delta\ \Gamma \vdash \phi_1 \vee \phi_2 \ \mathsf{true}} \tag{35.1l}$$

$$\frac{\Delta\ \Gamma \vdash \phi_1 \ \mathsf{false} \quad \Delta\ \Gamma \vdash \phi_2 \ \mathsf{false}}{\Delta\ \Gamma \vdash \phi_1 \vee \phi_2 \ \mathsf{false}} \tag{35.1m}$$

Negation inverts the sense of each judgement:

$$\frac{\Delta\ \Gamma \vdash \phi \ \mathsf{false}}{\Delta\ \Gamma \vdash \neg\phi \ \mathsf{true}} \tag{35.1n}$$

$$\frac{\Delta\ \Gamma \vdash \phi \ \mathsf{true}}{\Delta\ \Gamma \vdash \neg\phi \ \mathsf{false}} \tag{35.1o}$$

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.

$$\frac{\Delta\ \Gamma, \phi_1 \ \mathsf{true} \vdash \phi_2 \ \mathsf{true}}{\Delta\ \Gamma \vdash \phi_1 \supset \phi_2 \ \mathsf{true}} \tag{35.1p}$$

$$\frac{\Delta\ \Gamma \vdash \phi_1 \ \mathsf{true} \quad \Delta\ \Gamma \vdash \phi_2 \ \mathsf{false}}{\Delta\ \Gamma \vdash \phi_1 \supset \phi_2 \ \mathsf{false}} \tag{35.1q}$$

35.1.2 Proofs and Refutations

The dynamics of classical proofs is most easily explained by introducing a notation for the derivations of each of the judgement forms of classical logic:

1. $p : \phi$, stating that p is a proof of ϕ;

2. $k \div \phi$, stating that k is a refutation of ϕ;

3. $k \# p$, stating that k and p are contradictory.

We will consider hypothetical judgements of the form

$$\underbrace{u_1 \div \phi_1, \ldots, u_m \div \phi_m}_{\Delta}\ \underbrace{x_1 : \psi_1, \ldots, x_n : \psi_n}_{\Gamma} \vdash J,$$

in which we have labelled the truth and falsity assumptions with variables.

A contradiction arises whenever a proposition is both true and false:

$$\frac{\Delta\ \Gamma \vdash k \div \phi \quad \Delta\ \Gamma \vdash p : \phi}{\Delta\ \Gamma \vdash k \mathbin{\#} p} \tag{35.2a}$$

Truth and falsity are defined symmetrically in terms of contradiction:

$$\frac{\Delta, u \div \phi\ \Gamma \vdash k \mathbin{\#} p}{\Delta\ \Gamma \vdash \mathtt{ccr}(u \div \phi . k \mathbin{\#} p) : \phi} \tag{35.2b}$$

$$\frac{\Delta\ \Gamma, x : \phi \vdash k \mathbin{\#} p}{\Delta\ \Gamma \vdash \mathtt{ccp}(x : \phi . k \mathbin{\#} p) \div \phi} \tag{35.2c}$$

Reflexivity corresponds to the use of a variable hypothesis:

$$\frac{}{\Delta, u \div \phi\ \Gamma \vdash u \div \phi} \tag{35.2d}$$

$$\frac{}{\Delta\ \Gamma, x : \phi \vdash x : \phi} \tag{35.2e}$$

The other structure properties are admissible.

Truth is trivially true, and cannot be refuted.

$$\frac{}{\Delta\ \Gamma \vdash \langle\rangle : \top} \tag{35.2f}$$

A conjunction is true if both conjuncts are true, and is false if either conjunct is false.

$$\frac{\Delta\ \Gamma \vdash p_1 : \phi_1 \quad \Delta\ \Gamma \vdash p_2 : \phi_2}{\Delta\ \Gamma \vdash \langle p_1, p_2 \rangle : \phi_1 \wedge \phi_2} \tag{35.2g}$$

$$\frac{\Delta\ \Gamma \vdash k_1 \div \phi_1}{\Delta\ \Gamma \vdash \mathtt{fst};k_1 \div \phi_1 \wedge \phi_2} \tag{35.2h}$$

$$\frac{\Delta\ \Gamma \vdash k_2 \div \phi_2}{\Delta\ \Gamma \vdash \mathtt{snd};k_2 \div \phi_1 \wedge \phi_2} \tag{35.2i}$$

Falsity is trivially false, and cannot be proved.

$$\frac{}{\Delta\ \Gamma \vdash \mathtt{abort} \div \bot} \tag{35.2j}$$

A disjunction is true if either disjunct is true, and is false if both disjuncts are false.

$$\frac{\Delta\ \Gamma \vdash p_1 : \phi_1}{\Delta\ \Gamma \vdash \mathtt{inl}(p_1) : \phi_1 \vee \phi_2} \tag{35.2k}$$

$$\frac{\Delta\;\Gamma \vdash p_2 : \phi_2}{\Delta\;\Gamma \vdash \texttt{inr}(p_2) : \phi_1 \vee \phi_2} \tag{35.2l}$$

$$\frac{\Delta\;\Gamma \vdash k_1 \div \phi_1 \quad \Delta\;\Gamma \vdash k_2 \div \phi_2}{\Delta\;\Gamma \vdash \texttt{case}(k_1;k_2) \div \phi_1 \vee \phi_2} \tag{35.2m}$$

Negation inverts the sense of each judgement:

$$\frac{\Delta\;\Gamma \vdash k \div \phi}{\Delta\;\Gamma \vdash \texttt{not}(k) : \neg\phi} \tag{35.2n}$$

$$\frac{\Delta\;\Gamma \vdash p : \phi}{\Delta\;\Gamma \vdash \texttt{not}(p) \div \neg\phi} \tag{35.2o}$$

An implication is true if its conclusion is true whenever the assumption is true, and is false if its conclusion if false yet its assumption is true.

$$\frac{\Delta\;\Gamma, x : \phi_1 \vdash p_2 : \phi_2}{\Delta\;\Gamma \vdash \lambda\,(x{:}\phi_1.\,p_2) : \phi_1 \supset \phi_2} \tag{35.2p}$$

$$\frac{\Delta\;\Gamma \vdash p_1 : \phi_1 \quad \Delta\;\Gamma \vdash k_2 \div \phi_2}{\Delta\;\Gamma \vdash \texttt{app}(p_1)\texttt{;}k_2 \div \phi_1 \supset \phi_2} \tag{35.2q}$$

35.2 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 contradicton 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

$$\lambda\,(w{:}\phi \wedge (\psi \wedge \theta).\,\texttt{ccr}(u \div \theta \wedge \phi.k \,\#\, w)),$$

where k is the refutation

$$\texttt{fst;ccp}(x : \phi.\texttt{snd;ccp}(y : \psi \wedge \theta.\texttt{snd;ccp}(z : \theta.u \,\#\, \langle z, x\rangle) \,\#\, y) \,\#\, w).$$

And yet in constructive logic this proposition has a direct proof that avoids the circumlocutions of proof by contradiction:

$$\lambda\,(w{:}\phi \wedge (\psi \wedge \theta).\,\texttt{andI(andE[r](andE[r]}(w)\texttt{);andE[l]}(w)\texttt{)}).$$

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:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \phi\ \mathsf{false}}}{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \phi \wedge \psi\ \mathsf{false}} \quad \dfrac{\Delta\ \Gamma \vdash \phi \wedge \psi\ \mathsf{true}}{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \phi \wedge \psi\ \mathsf{true}}}{\Delta, \phi\ \mathsf{false}\ \Gamma \vdash \#}}{\Delta\ \Gamma \vdash \phi\ \mathsf{true}}$$

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.

The derivations of the elimination forms of constructive logic are most easily exhibited using proof and refutation expressions, as follows:

$$\begin{aligned}
\mathtt{falseE}[\phi]\mathtt{(}p\mathtt{)} &= \mathtt{ccr(}u \div \phi\,\mathtt{.abort}\ \#\ p\mathtt{)} \\
\mathtt{andE[l](}p\mathtt{)} &= \mathtt{ccr(}u \div \phi\,\mathtt{.fst;}u\ \#\ p\mathtt{)} \\
\mathtt{andE[r](}p\mathtt{)} &= \mathtt{ccr(}u \div \psi\,\mathtt{.snd;}u\ \#\ p\mathtt{)} \\
\mathtt{impE(}p_1\mathtt{;}p_2\mathtt{)} &= \mathtt{ccr(}u \div \psi\,\mathtt{.app(}p_2\mathtt{);}u\ \#\ p_1\mathtt{)} \\
\mathtt{orE}[\phi;\psi]\mathtt{(}p_1\mathtt{;}x\,\mathtt{.}\,p_2\mathtt{;}y\,\mathtt{.}\,p\mathtt{)} &= \mathtt{ccr(}u \div \gamma\,\mathtt{.case(ccp(}x : \phi\,\mathtt{.}\,u\ \#\ p_2\mathtt{);ccp(}y : \psi\,\mathtt{.}\,u\ \#\ p\mathtt{))}\ \#\ p_1\mathtt{)}
\end{aligned}$$

It is straightforward to check that the expected elimination rules hold. For example, the rule

$$\frac{\Delta\ \Gamma \vdash p_1 : \phi \supset \psi \quad \Delta\ \Gamma \vdash p_2 : \phi}{\Delta\ \Gamma \vdash \mathtt{impE(}p_1\mathtt{;}p_2\mathtt{)} : \psi} \tag{35.3}$$

is derivable using the definition of $\mathtt{impE(}p_1\mathtt{;}p_2\mathtt{)}$ given above. By suppressing proof terms, we may derive the corresponding provability rule

$$\frac{\Delta\ \Gamma \vdash \phi \supset \psi\ \mathsf{true} \quad \Delta\ \Gamma \vdash \phi\ \mathsf{true}}{\Delta\ \Gamma \vdash \psi\ \mathsf{true}}\ . \tag{35.4}$$

35.3 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 $k \# p$ 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:

$$\texttt{fst;}k \# \langle p_1, p_2 \rangle \mapsto k \# p_1 \tag{35.5a}$$

$$\texttt{snd;}k \# \langle p_1, p_2 \rangle \mapsto k \# p_2 \tag{35.5b}$$

$$\texttt{case(}k_1\texttt{;}k_2\texttt{)} \# \texttt{inl(}p_1\texttt{)} \mapsto k_1 \# p_1 \tag{35.5c}$$

$$\texttt{case(}k_1\texttt{;}k_2\texttt{)} \# \texttt{inr(}p_2\texttt{)} \mapsto k_2 \# p_2 \tag{35.5d}$$

$$\texttt{not(}p\texttt{)} \# \texttt{not(}k\texttt{)} \mapsto k \# p \tag{35.5e}$$

$$\texttt{app(}p_1\texttt{);}k \# \lambda\,\texttt{(}x\texttt{:}\phi\texttt{.}\,p_2\texttt{)} \mapsto k \# [p_1/x]p_2 \tag{35.5f}$$

The rules of indirect proof give rise to the following transitions:

$$\texttt{ccp(}x : \phi\texttt{.}k_1 \# p_1\texttt{)} \# p_2 \mapsto [p_2/x]k_1 \# [p_2/x]p_1 \tag{35.5g}$$

$$k_1 \# \texttt{ccr(}u \div \phi\texttt{.}k_2 \# p_2\texttt{)} \mapsto [k_1/u]k_2 \# [k_1/u]p_2 \tag{35.5h}$$

The first of these defines the behavior of the refutation of ϕ that proceeds by contradicting the assumption that ϕ is true. This refutation is activated by presenting it with a proof of ϕ, which 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 ϕ that proceeds by contradicting the assumption that ϕ is false. This proof is activated by presenting it with a refutation of ϕ, which is then substituted for the assumption in the new state. Thus, “ccr” stands for “call with current refutation.”

Rules (35.5g) to (35.5h) overlap in that there are two possible transitions for a state of the form

$$\texttt{ccp(}x : \phi\texttt{.}k_1 \# p_1\texttt{)} \# \texttt{ccr(}u \div \phi\texttt{.}k_2 \# p_2\texttt{)},$$

one to the state $[p/x]k_1 \# [p/x]p_1$, where p is $\texttt{ccr(}u \div \phi\texttt{.}k_2 \# p_2\texttt{)}$, and one to the state $[k/u]k_2 \# [k/u]p_2$, where k is $\texttt{ccp(}x : \phi\texttt{.}k_1 \# p_1\texttt{)}$. 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.

Theorem 35.1 (Preservation)**.** *If $k \div \phi$, $p : \phi$, and $k \# p \mapsto k' \# p'$, then there exists ϕ' such that $k' \div \phi'$ and $p' : \phi'$.*

Proof. By rule induction on the dynamics of classical logic. □

Theorem 35.2 (Progress)**.** *If $k \div \phi$ and $p : \phi$, then either $k \# p$ final or $k \# p \mapsto k' \# p'$.*

Proof. By rule induction on the statics of classical logic. □

To initiate computation we postulate that `halt` is a refutation of any proposition. The initial and final states of computational are defined as follows:

$$\frac{}{\texttt{halt} \# p\ \mathsf{initial}} \tag{35.6a}$$

$$\frac{p\ \mathsf{canonical}}{\texttt{halt} \# p\ \mathsf{final}} \tag{35.6b}$$

The judgement p $\mathsf{canonical}$ states that p is a canonical proof, which is defined to be any proof other than an indirect proof.

35.4 Law of the Excluded Middle

The law of the excluded middle is derivable in classical logic:

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\phi \vee \neg\phi\ \mathsf{false}, \phi\ \mathsf{true} \vdash \phi\ \mathsf{true}}}{\phi \vee \neg\phi\ \mathsf{false}, \phi\ \mathsf{true} \vdash \phi \vee \neg\phi\ \mathsf{true}} \quad \dfrac{}{\phi \vee \neg\phi\ \mathsf{false}, \phi\ \mathsf{true} \vdash \phi \vee \neg\phi\ \mathsf{false}}}{\phi \vee \neg\phi\ \mathsf{false}, \phi\ \mathsf{true} \vdash \#}}{\phi \vee \neg\phi\ \mathsf{false} \vdash \phi\ \mathsf{false}}}{\phi \vee \neg\phi\ \mathsf{false} \vdash \neg\phi\ \mathsf{true}}}{\phi \vee \neg\phi\ \mathsf{false} \vdash \phi \vee \neg\phi\ \mathsf{true}} \quad \dfrac{}{\phi \vee \neg\phi\ \mathsf{false} \vdash \phi \vee \neg\phi\ \mathsf{false}}}{\dfrac{\phi \vee \neg\phi\ \mathsf{false} \vdash \#}{\phi \vee \neg\phi\ \mathsf{true}}}$$

When written out using explicit proofs and refutations, we obtain the proof term $p_0 : \phi \vee \neg\phi$:

$$\texttt{ccr(}u \div \phi \vee \neg\phi\texttt{.}u \,\#\, \texttt{inr(not(ccp(}x : \phi\texttt{.}u \,\#\, \texttt{inl(}x\texttt{))))).}$$

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 35.3 on page 310. The first step is the transition

$$\begin{gathered} k \,\#\, \texttt{ccr(}u \div \phi \vee \neg\phi\texttt{.}u \,\#\, \texttt{inr(not(ccp(}x : \phi\texttt{.}u \,\#\, \texttt{inl(}x\texttt{)))))} \\ \mapsto \\ k \,\#\, \texttt{inr(not(ccp(}x : \phi\texttt{.}k \,\#\, \texttt{inl(}x\texttt{)))),} \end{gathered}$$

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 $\texttt{case(}k_1\texttt{;}k_2\texttt{)}$, where $k_1 \div \phi$ and $k_2 \div \neg\phi$. Continuing the reduction, we obtain:

$$\begin{gathered} \texttt{case(}k_1\texttt{;}k_2\texttt{)} \,\#\, \texttt{inr(not(ccp(}x : \phi\texttt{.case(}k_1\texttt{;}k_2\texttt{)} \,\#\, \texttt{inl(}x\texttt{))))} \\ \mapsto \\ k_2 \,\#\, \texttt{not(ccp(}x : \phi\texttt{.case(}k_1\texttt{;}k_2\texttt{)} \,\#\, \texttt{inl(}x\texttt{))).} \end{gathered}$$

By virtue of its type k_2 must have the form $\texttt{not(}p_2\texttt{)}$, where $p_2 : \phi$, and hence the transition proceeds as follows:

$$\begin{gathered} \texttt{not(}p_2\texttt{)} \,\#\, \texttt{not(ccp(}x : \phi\texttt{.case(}k_1\texttt{;}k_2\texttt{)} \,\#\, \texttt{inl(}x\texttt{)))} \\ \mapsto \\ \texttt{ccp(}x : \phi\texttt{.case(}k_1\texttt{;}k_2\texttt{)} \,\#\, \texttt{inl(}x\texttt{))} \,\#\, p_2. \end{gathered}$$

Observe that p_2 is a valid proof of ϕ! Proceeding, we obtain

$$\begin{gathered} \texttt{ccp(}x : \phi\texttt{.case(}k_1\texttt{;}k_2\texttt{)} \,\#\, \texttt{inl(}x\texttt{))} \,\#\, p_2 \\ \mapsto \\ \texttt{case(}k_1\texttt{;}k_2\texttt{)} \,\#\, \texttt{inl(}p_2\texttt{)} \\ \mapsto \\ k_1 \,\#\, p_2 \end{gathered}$$

The first of these two steps is the crux of the matter: the refutation, $k = \texttt{case(}k_1\texttt{;}k_2\texttt{)}$, 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 $\texttt{inr(}p_1\texttt{)}$ 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 ϕ is false. If the context is such that it inspects this proof, it can only be by providing the proof, p_2, of ϕ that refutes the claim that ϕ is false. Should this occur, the proof of the law of the excluded middle *backtracks* the context, providing instead the proof $\mathtt{inl}(p_2)$ to k, which then passes p_2 to k_1 without further incident. The proof of the law of the excluded middle baldly asserts $\neg\phi\ \mathsf{true}$, regardless of the form of ϕ. Then, if caught in its lie by the context providing a proof of ϕ, *changes its mind* and asserts to the *original* context, k, after all! No further reversion is possible, because the context has itself provided a proof, p_2, of ϕ.

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!

35.5 Exercises

Part XIII

Symbols

Chapter 36

Symbols

A *symbol* is an atomic datum with no internal structure. Whereas variables are given meaning by substitution, symbols are given meaning by a collection of primitives associated with it. In subsequent chapters we shall consider several interpretations of symbols, giving rise to concepts such as fluid binding, dynamic classification, assignable variables, and communication channels.

A "new "symbol, a, with associated type, σ, is introduced within a scope, e, by the declaration $\texttt{new}\,a\,\texttt{:}\,\sigma\,\texttt{in}\,e$. The meaning of the type σ varies according to the interpretation of symbols under consideration. It is important to emphasize, however, that a symbol is *not* a form of expression, and hence is *not* an element of its associated type. The expression, e, in the symbol declaration $\texttt{new}\,a\,\texttt{:}\,\sigma\,\texttt{in}\,e$, is the *scope* of the symbol. As usual, bound identifiers may be renamed within their scope, and hence a may be regarded as "new" in the sense that it may be chosen to be distinct from any given finite set of symbols.

The dynamics of symbol declaration defines the *extent*, or range of significance, of the declared symbol. Under a *scoped*, or *stack-like*, dynamics, the extent of a symbol is just its scope. Once the scope of the declaration has been evaluated, the symbol may be deallocated—the statics will ensure that the result cannot depend on that symbol. Under a *scope-free*, or *heap-like*, dynamics the extent of a symbol is unlimited. A symbol may escape the scope of its declaration, which means that it cannot be deallocated once the scope has been evaluated.

Perhaps the simplest application of symbols is as an atom that may be compared for equality with a given symbol. The type $\sigma\,\texttt{sym}$ has as elements *symbolic references* of the form $\texttt{\&}\,a$. The conditional $\texttt{if}\,e\,\texttt{is}\,a\,\texttt{then}\,e_1\,\texttt{ow}\,be_2$

branches according to whether e evaluates to $\texttt{\&}\,a$ or not, in a manner to be described in more detail in Section 36.2 on page 321 below.

36.1 Symbol Declaration

The syntax for symbol declaration is given by the following grammar:

$$\begin{array}{llll} \mathsf{Expr} & e & ::= & \texttt{new[}\tau\texttt{](}a\texttt{.}e\texttt{)} \quad \texttt{new}\,a\texttt{:}\tau\,\texttt{in}\,e \quad \text{generation} \end{array}$$

Importantly, symbol declaration is not associated with any particular type, but is a primitive mechanism shared by all of the applications of symbols.

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 Σ to range over signatures, which are finite sets of pairs $a : \tau$, where a is a symbol and τ is a type. The hypothetical typing judgement $\Gamma \vdash_{\Sigma} e : \tau$ is parameterized by a signature, Σ, associating types to symbols.

The statics of symbol declaration is given by the following rule:

$$\frac{\Gamma \vdash_{\Sigma,a:\sigma} e : \tau \quad \tau \ \mathsf{mobile}}{\Gamma \vdash_{\Sigma} \texttt{new[}\sigma\texttt{](}a\texttt{.}e\texttt{)} : \tau} \tag{36.1}$$

It is implicit that a is chosen to not already be declared in Σ, ensuring that it is not otherwise in use. The premise $\tau\ \mathsf{mobile}$ of Rule (36.1) specifies that the type, τ, of the scope of the declaration must be *mobile* in the sense that it is permissible to return a value of this type from the scope of a symbol declaration.

The definition of mobility depends on the form of dynamics for symbol declaration. Under a scoped dynamics mobility is defined so that a value of a mobile type cannot involve a symbol, and hence may safely be returned from the scope of its declaration. This allows a symbol to be deallocated once it scope has been evaluated—that is, its extent coincides with its scope. Under a scope-free dynamics mobility imposes no restrictions on the type, permitting any value, including one that depends on the declared symbol, to be returned from the scope of the declaration. To support this the dynamics must give symbols indefinite extent, even though they have static scope.

36.1.1 Scoped Dynamics

The scoped dynamics of symbol declaration is given by a transition judgement of the form $e \underset{\Sigma}{\mapsto} e'$ indexed by a signature, Σ, specifying the active symbols of the transition. Either e or e' may involve the symbols declared in Σ, but no others.

$$\frac{e \underset{\Sigma,a:\sigma}{\longmapsto} e'}{\texttt{new}[\sigma]\,(a\,.\,e) \underset{\Sigma}{\mapsto} \texttt{new}[\sigma]\,(a\,.\,e')} \tag{36.2a}$$

$$\frac{e\ \mathsf{val}_{\Sigma,a:\sigma} \quad a \notin e}{\texttt{new}[\sigma]\,(a\,.\,e) \underset{\Sigma}{\mapsto} e} \tag{36.2b}$$

Rule (36.2a) specifies that evaluation takes place within the scope of the declaration of a symbol. Rule (36.2b) specifies that the declared symbol may be deallocated once its scope has been evaluated, provided that the declared symbol does not occur within that value. To ensure type safety, the definition of the judgement τ mobile must be chosen so that this condition always holds.

36.1.2 Scope-Free Dynamics

The scope-free dynamics of symbols may be specified in one of two ways. One method is to consider a transition system between states of the form $\nu\,\Sigma\,\{\,e\,\}$, where Σ is a signature and e is an expression over this signature. The judgement $\nu\,\Sigma\,\{\,e\,\} \mapsto \nu\,\Sigma'\,\{\,e'\,\}$ states that evaluation of e relative to symbols Σ results in the expression e' in the extension Σ' of Σ.

$$\frac{a \notin dom(\Sigma)}{\nu\,\Sigma\,\{\,\texttt{new}[\sigma]\,(a\,.\,e)\,\} \mapsto \nu\,\Sigma, a:\sigma\,\{\,e\,\}} \tag{36.3}$$

Rule (36.3) specifies that symbol generation enriches the signature with the newly introduced symbol by extending the signature for all future transitions.

Such a formulation of the dynamics is disadvantageous because it relies on an extra-linguistic notion of the state of the computation. One consequence is that the dynamics of all other aspects of the language must be reformulated to account for this change. For example, the dynamics of (by-name) function application cannot simply be inherited from Chapter 13, but must instead be re-formulated as follows:

$$\frac{\nu\,\Sigma\,\{\,e_1\,\} \mapsto \nu\,\Sigma'\,\{\,e_1'\,\}}{\nu\,\Sigma\,\{\,e_1(e_2)\,\} \mapsto \nu\,\Sigma'\,\{\,e_1'(e_2)\,\}} \tag{36.4a}$$

$$\frac{}{\nu\,\Sigma\,\{\,\lambda\,(x{:}\tau.\,e)\,(e_2)\,\} \mapsto \nu\,\Sigma\,\{\,[e_2/x]e\,\}} \tag{36.4b}$$

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 the language.

One way to avoid this is to impose a congruence relation on expressions, called *structural equivalence*, that manages the bureaucracy of symbol generation implicitly. First, the state

$$\nu\, a_1 : \sigma_1, \ldots, a_n : \sigma_n \,\{\, e \,\}$$

is regarded as equivalent to the cascade of declarations

$$\texttt{new}\, a_1 {:} \sigma_1 \,\texttt{in} \ldots \texttt{new}\, a_n {:} \sigma_n \,\texttt{in}\, e.$$

In particular, if e is a symbol declaration, it is tacitly absorbed into the surrounding context of the evaluation. But what if a symbol declaration is nested within another form of expression? Consider the application

$$\texttt{(new}\, a {:} \sigma \,\texttt{in}\, \lambda\,\texttt{(}x {:} \tau \texttt{.}\, e\texttt{))(}e_2\texttt{)}.$$

Progress fails because the function position is not a value, yet it cannot make a transition either. What is lacking is a means of activating the declaration by widening its scope to the surrounding context; this is called *scope extrusion*. To achieve this, the foregoing application is identified with the expression

$$\texttt{new}\, a {:} \sigma \,\texttt{in (}\lambda\,\texttt{(}x {:} \tau \texttt{.}\, e\texttt{)(}e_2\texttt{))},$$

which transitions to

$$\texttt{new}\, a {:} \sigma \,\texttt{in}\, [e_2/x]e.$$

The judgement $e_1 \equiv_\Sigma e_2$ states that the closed expressions e_1 and e_2 over the signature Σ are structurally equivalent. It is defined to be the strongest equivalence relation such that the scope of a symbol declaration in the principal argument(s) of an elimination form may be extruded to encompass that form. For example, in the case of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, the following two rules of structural congruence are required:

$$\frac{}{\texttt{(new}\, a {:} \sigma \,\texttt{in}\, e_1\texttt{)(}e_2\texttt{)} \equiv_\Sigma \texttt{new}\, a {:} \sigma \,\texttt{in (}e_1\texttt{(}e_2\texttt{))}} \tag{36.5a}$$

$$\frac{}{\texttt{ifz (new}\, a {:} \sigma \,\texttt{in}\, e\texttt{)}\, \{\texttt{z} \Rightarrow e_0 \mid \texttt{s(}x\texttt{)} \Rightarrow e_1\} \equiv_\Sigma \texttt{new}\, a {:} \sigma \,\texttt{in ifz}\, e\, \{\texttt{z} \Rightarrow e_0 \mid \texttt{s(}x\texttt{)} \Rightarrow e_1\}} \tag{36.5b}$$

Crucially, the dynamics is defined to respect structural equivalence:

$$\frac{e_1 \equiv_\Sigma e_2 \quad e_2 \underset{\Sigma}{\mapsto} e_2' \quad e_2' \equiv_\Sigma e_1'}{e_1 \underset{\Sigma}{\mapsto} e_1'} \tag{36.6}$$

A technical lemma states that structurally equivalent expressions have the same types.

Lemma 36.1. *If $e_1 \equiv_\Sigma e_2$, then $\vdash_\Sigma e_1 : \tau$ iff $\vdash_\Sigma e_2 : \tau$.*

Theorem 36.2 (Preservation)**.** *If $\vdash_\Sigma e : \tau$ and $e \underset{\Sigma}{\mapsto} e'$, then $e' : \tau$.*

Proof. The interesting case is Rule (36.6), for which we appeal to Lemma 36.1. □

Theorem 36.3 (Progress)**.** *If $\vdash_\Sigma e : \tau$, then either there exists $n \geq 0$ such that $e \equiv_\Sigma \mathtt{new}\,a_1\,\mathtt{:}\sigma_1\,\mathtt{in}\ldots\mathtt{new}\,a_n\,\mathtt{:}\sigma_n\,\mathtt{in}\,e'$ with e' $\mathsf{val}_{\Sigma,a_1:\sigma_1,\ldots,a_n:\sigma_n}$, or there exists e' such that $e \underset{\Sigma}{\mapsto} e'$.*

Proof. By induction on typing, making use of scope extrusion. For example, suppose that e is $e_1\mathtt{(}e_2\mathtt{)}$, where $\vdash_\Sigma e_1 : \tau_2 \to \tau$ and $\vdash_\Sigma e_2 : \tau_2$. By induction with have that either $e_1 \equiv_\Sigma \mathtt{new}\,a_1\mathtt{:}\sigma_1\,\mathtt{in}\ldots\mathtt{new}\,a_n\mathtt{:}\sigma_n\,\mathtt{in}\,e_1'$, where e_1' $\mathsf{val}_{\Sigma,a_1:\sigma_1,\ldots,a_n:\sigma_n}$, or $e \underset{\Sigma}{\mapsto} e'$ for some e'. In the latter case appeal to Rule (36.2a). In the former, apply Rule (36.5a) to show that $e_1\mathtt{(}e_2\mathtt{)} \equiv_\Sigma \mathtt{new}\,a_1\mathtt{:}\sigma_1\,\mathtt{in}\ldots\mathtt{new}\,a_n\mathtt{:}\sigma_n\,\mathtt{in}\,e_1'\mathtt{(}e_2\mathtt{)}$, then apply Rule (36.6) and the canonical forms lemma for function types. □

36.2 Symbolic References

As discussed in the introduction to this chapter, the type $\tau\,\mathtt{sym}$ has as values *symbolic references*, $\mathtt{\&}\,a$, to a symbol, a. Given such a value, we may branch on whether it is a reference to a specified symbol or not. The syntax of these primitives is given by the following grammar:

Type	τ	::=	$\mathtt{sym(}\tau\mathtt{)}$	$\tau\,\mathtt{sym}$	symbolic reference
Expr	e		$\mathtt{sym[}a\mathtt{]}$	$\mathtt{\&}\,a$	symbolic reference
			$\mathtt{is[}a\mathtt{][}t\mathtt{.}\tau\mathtt{](}e\mathtt{;}e_1\mathtt{;}e_2\mathtt{)}$		comparison
				$\mathtt{if}\,e\,\mathtt{is}\,a\,\mathtt{then}\,e_1\,\mathtt{ow}\,be_2$	

The expression $\mathtt{sym[}a\mathtt{]}$ is a reference to the symbol a, a value of type $\mathtt{sym(}\tau\mathtt{)}$. The expression $\mathtt{is[}a\mathtt{][}t\mathtt{.}\tau\mathtt{](}e\mathtt{;}e_1\mathtt{;}e_2\mathtt{)}$ 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.

36.2.1 Statics

The typing rules for symbolic references are as follows:

$$\frac{}{\Gamma \vdash_{\Sigma,a:\sigma} \texttt{sym[}a\texttt{]} : \texttt{sym(}\sigma\texttt{)}} \tag{36.7a}$$

$$\frac{\Gamma \vdash_{\Sigma,a:\rho} e : \texttt{sym(}\sigma\texttt{)} \quad \Gamma \vdash_{\Sigma,a:\rho} e_1 : [\rho/t]\tau \quad \Gamma \vdash_{\Sigma,a:\rho} e_2 : [\sigma/t]\tau}{\Gamma \vdash_{\Sigma,a:\rho} \texttt{is[}a\texttt{][}t\texttt{.}\tau\texttt{](}e\texttt{;}e_1\texttt{;}e_2\texttt{)} : [\sigma/t]\tau} \tag{36.7b}$$

Rule (36.7a) is the introduction rule for the type $\texttt{sym(}\sigma\texttt{)}$. It states that if a is a symbol with associated type σ, then $\texttt{sym[}a\texttt{]}$ is an expression of type $\texttt{sym(}\sigma\texttt{)}$. Rule (36.7b) is the elimination rule for the type $\texttt{sym(}\sigma\texttt{)}$. Observe that 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, of course, coincide, but if it refers to a different symbol, there is no reason to insist that it be of the same type.

With this in mind, let us examine carefully Rule (36.7b). *A priori* there is a discrepancy between the type, ρ, of a and the type, σ, of the symbol referred to by e. This discrepancy is mediated by the type operator $t\texttt{.}\tau$.[1] Regardless of the outcome of the comparison, the overall type of the expresion is $[\sigma/t]\tau$. 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 σ and ρ coincide, since 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 $[\sigma/t]\tau$; no further information about the type of the symbol is acquired in this branch.

36.2.2 Dynamics

The dynamics of symbolic references is given by the following rules:

$$\frac{}{\texttt{sym[}a\texttt{]}\ \mathsf{val}_{\Sigma,a:\sigma}} \tag{36.8a}$$

$$\frac{}{\texttt{is[}a\texttt{][}t\texttt{.}\tau\texttt{](sym[}a\texttt{];}e_1\texttt{;}e_2\texttt{)} \underset{\Sigma,a:\rho}{\longmapsto} e_1} \tag{36.8b}$$

[1] See Chapter 17 for a discussion of type operators.

$$\frac{}{\texttt{is}\,\texttt{[}a\texttt{]}\,\texttt{[}t\,\texttt{.}\,\tau\texttt{]}\,\texttt{(}\texttt{sym}\texttt{[}a'\texttt{]}\texttt{;}e_1\texttt{;}e_2\texttt{)} \xmapsto[\Sigma,a:\rho,a':\sigma]{} e_2} \tag{36.8c}$$

$$\frac{e \xmapsto[\Sigma,a:\rho]{} e'}{\texttt{is}\,\texttt{[}a\texttt{]}\,\texttt{[}t\,\texttt{.}\,\tau\texttt{]}\,\texttt{(}e\texttt{;}e_1\texttt{;}e_2\texttt{)} \xmapsto[\Sigma,a:\rho]{} \texttt{is}\,\texttt{[}a\texttt{]}\,\texttt{[}t\,\texttt{.}\,\tau\texttt{]}\,\texttt{(}e'\texttt{;}e_1\texttt{;}e_2\texttt{)}} \tag{36.8d}$$

Rules (36.8b) and (36.8c) specify that $\texttt{is}\,\texttt{[}a\texttt{]}\,\texttt{[}t\,\texttt{.}\,\tau\texttt{]}\,\texttt{(}e\texttt{;}e_1\texttt{;}e_2\texttt{)}$ branches according to whether the value of e is a reference to the symbol, a, or not.

36.2.3 Safety

Theorem 36.4 (Preservation). *If $\vdash_\Sigma e : \tau$ and $e \underset{\Sigma}{\mapsto} e'$, then $\vdash_\Sigma e' : \tau$.*

Proof. By rule induction on Rules (36.8). The most interesting case is Rule (36.8b). When the comparison is positive, the types σ and ρ must be the same, since each symbol has at most one associated type. Therefore, e_1, which has type $[\rho/t]\tau$, also has type $[\sigma/t]\tau$, as required. □

Lemma 36.5 (Canonical Forms). *If $\vdash_\Sigma e : \texttt{sym}\,\texttt{(}\sigma\texttt{)}$ and e val_Σ, then $e = \texttt{sym}\,\texttt{[}a\texttt{]}$ for some a such that $\Sigma = \Sigma', a : \sigma$.*

Proof. By rule induction on Rules (36.7), taking account of the definition of values. □

Theorem 36.6 (Progress). *Suppose that $\vdash_\Sigma e : \tau$. Then either e val_Σ, or there exists e' such that $e \underset{\Sigma}{\mapsto} e'$.*

Proof. By rule induction on Rules (36.7). For example, consider Rule (36.7b), in which we have that $\texttt{is}\,\texttt{[}a\texttt{]}\,\texttt{[}t\,\texttt{.}\,\tau\texttt{]}\,\texttt{(}e\texttt{;}e_1\texttt{;}e_2\texttt{)}$ has some type τ and that $e : \texttt{sym}\texttt{(}\sigma\texttt{)}$ for some σ. By induction either Rule (36.8d) applies, or else we have that e val_Σ, in which case we are assured by Lemma 36.5 that e is $\texttt{sym}\texttt{[}a\texttt{]}$ for some symbol b of type σ declared in Σ. But then progress is assured by Rules (36.8b) and (36.8c), since equality of symbols is decidable (either a is b or it is not). □

36.3 Exercises

1. Formulate an equality test, with type specialization, that compares two symbolic references, branching accordingly.

Chapter 37

Fluid Binding

Recall from Chapter 11 that under the dynamic scope discipline evaluation is defined for expressions with free variables whose bindings are determined by capture-incurring substitution. Evaluation aborts if the binding of a variable is required in a context in which no binding for it exists. Otherwise, it uses whatever bindings for its free variables happen to be active at the point at which it is evaluated. In essence the bindings of variables are determined as late as possible during execution—just in time for evaluation to proceed. However, we found that as a language design dynamic scoping is deficient in (at least) two respects:

- Bound variables may not always be renamed in an expression without changing its meaning.

- Since the scopes of variables are resolved dynamically, it is difficult to ensure type safety.

These difficulties can be overcome by distinguishing two different concepts, namely *static binding* of variables, which is defined by substitution, and *dynamic*, or *fluid*, *binding* of *symbols*, which is defined by storing and retrieving bindings from a table during execution.

37.1 Statics

The language $\mathcal{L}\{\mathtt{fluid}\}$ extends the language $\mathcal{L}\{\mathtt{sym}\}$ defined in Chapter 36 with the following additional constructs:

$$\begin{array}{llllll}
\mathsf{Expr} & e & ::= & \mathtt{put}[a](e_1;e_2) & \mathtt{put}\,e_1\,\mathtt{for}\,a\,\mathtt{in}\,e_2 & \text{binding} \\
 & & & \mathtt{get}[a] & \mathtt{get}\,a & \text{retrieval}
\end{array}$$

As in Chapter 36, the variable a ranges over some fixed set of *symbols*. The expression $\texttt{get[}a\texttt{]}$ evaluates to the value of the current binding of a, if it has one, and is stuck otherwise. The expression $\texttt{put[}a\texttt{](}e_1\texttt{;}e_2\texttt{)}$ 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 $\mathcal{L}\{\texttt{fluid}\}$ is defined by judgements of the form

$$\Gamma \vdash_{\Sigma} e : \tau,$$

where Σ is a finite set $a_1 : \tau_1, \ldots, a_k : \tau_k$ of declarations of the pairwise distinct symbols $a_1, \ldots, a_k$, and Γ is, as usual, a finite set $x_1 : \tau_1, \ldots, x_n : \tau_n$ of declarations of the pairwise distinct variables $x_1, \ldots, x_n$.

The statics of $\mathcal{L}\{\texttt{fluid}\}$ extends that of $\mathcal{L}\{\texttt{sym}\}$ (see Chapter 36) with the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma,a:\tau} \texttt{get[}a\texttt{]} : \tau} \tag{37.1a}$$

$$\frac{\Gamma \vdash_{\Sigma,a:\tau_1} e_1 : \tau_1 \quad \Gamma \vdash_{\Sigma,a:\tau} e_2 : \tau_2}{\Gamma \vdash_{\Sigma,a:\tau_1} \texttt{put[}a\texttt{](}e_1\texttt{;}e_2\texttt{)} : \tau_2} \tag{37.1b}$$

Rule (37.1b) specifies that the symbol a is a parameter of the expression that must be declared in Σ.

37.2 Dynamics

We assume a stack-like dynamics for symbols, as described in Chapter 36. The dynamics of $\mathcal{L}\{\texttt{fluid}\}$ maintains an association of values to symbols that changes in a stack-like manner during execution. We define a family of transition judgements of the form $e \xmapsto[\Sigma]{\mu} e'$, where Σ is as in the statics, and μ is a finite function mapping some subset of the symbols declared in Σ to values of appropriate type. If μ is defined for some symbol a, then it has the form $\mu' \otimes \langle a : e \rangle$ for some μ' and value e. If, on the other hand, μ is undefined for some symbol a, we may regard it as having the form $\mu' \otimes \langle a : \bullet \rangle$. We will write $\langle a : \_ \rangle$ to stand ambiguously for either $\langle a : \bullet \rangle$ or $\langle a : e \rangle$ for some expression e.

The dynamics of $\mathcal{L}\{\texttt{fluid}\}$ is given by the following rules:

$$\frac{e \ \mathsf{val}_{\Sigma,a:\tau}}{\texttt{get[}a\texttt{]} \xmapsto[\Sigma,a:\tau]{\mu\otimes\langle a : e\rangle} e} \tag{37.2a}$$

$$\frac{e_1 \xmapsto[\Sigma]{\mu} e_1'}{\texttt{put}\,[a]\,(e_1;e_2) \xmapsto[\Sigma]{\mu} \texttt{put}\,[a]\,(e_1';e_2)} \tag{37.2b}$$

$$\frac{e_1\ \mathsf{val}_{\Sigma,a:\tau} \quad e_2 \xmapsto[\Sigma,a:\tau]{\mu\otimes\langle a:e_1\rangle} e_2'}{\texttt{put}\,[a]\,(e_1;e_2) \xmapsto[\Sigma,a:\tau]{\mu\otimes\langle a:\_\rangle} \texttt{put}\,[a]\,(e_1;e_2')} \tag{37.2c}$$

$$\frac{e_1\ \mathsf{val}_{\Sigma,a:\tau} \quad e_2\ \mathsf{val}_{\Sigma,a:\tau}}{\texttt{put}\,[a]\,(e_1;e_2) \xmapsto[\Sigma]{\mu} e_2} \tag{37.2d}$$

Rule (37.2a) specifies that $\texttt{get}\,[a]$ evaluates to the current binding of a, if any. Rule (37.2b) specifies that the binding for the symbol a is to be evaluated before the binding is created. Rule (37.2c) 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 (37.2d) eliminates the fluid binding for a once evaluation of the extent of the binding has completed.

According to the dynamics defined by Rules (37.2), there is no transition of the form $\texttt{get}\,[a] \xmapsto[\Sigma]{\mu} e$ if $\mu(a) = \bullet$. The judgement $e\ \mathsf{unbound}_\Sigma$ states that execution of e leads to such a state. It is inductively defined by the following rules:

$$\frac{\mu(a) = \bullet}{\texttt{get}\,[a]\ \mathsf{unbound}_\mu} \tag{37.3a}$$

$$\frac{e_1\ \mathsf{unbound}_\mu}{\texttt{put}\,[a]\,(e_1;e_2)\ \mathsf{unbound}_\mu} \tag{37.3b}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad e_2\ \mathsf{unbound}_\mu}{\texttt{put}\,[a]\,(e_1;e_2)\ \mathsf{unbound}_\mu} \tag{37.3c}$$

In addition to these rules we would also have, in a richer language, rules to propagate the unbound symbol error through other language constructs, as described in Chapter 9.

37.3 Type Safety

Define the auxiliary judgement $\mu : \Sigma$ by the following rules:

$$\frac{}{\emptyset : \emptyset} \tag{37.4a}$$

$$\frac{\vdash_\Sigma e : \tau \quad \mu : \Sigma}{\mu \otimes \langle a : e \rangle : \Sigma, a : \tau} \tag{37.4b}$$

$$\frac{\mu : \Sigma}{\mu \otimes \langle a : \bullet \rangle : \Sigma, a : \tau} \tag{37.4c}$$

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 Σ. No demand is made in the case that the symbol is unbound (equivalently, bound to a "black hole").

Theorem 37.1 (Preservation). *If $e \overset{\mu}{\underset{\Sigma}{\longmapsto}} e'$, where $\mu : \Sigma$ and $\vdash_\Sigma e : \tau$, then $\vdash_\Sigma e' : \tau$.*

Proof. By rule induction on Rules (37.2). Rule (37.2a) is handled by the definition of $\mu : \Sigma$. Rule (37.2b) follows immediately by induction. Rule (37.2d) is handled by inversion of Rules (37.1). Finally, Rule (37.2c) is handled by inversion of Rules (37.1) and induction. □

Theorem 37.2 (Progress). *If $\vdash_\Sigma e : \tau$ and $\mu : \Sigma$, then either e val_Σ, or e $\mathtt{unbound}_\mu$, or there exists e' such that $e \overset{\mu}{\underset{\Sigma}{\longmapsto}} e'$.*

Proof. By induction on Rules (37.1). For Rule (37.1a), we have $\Sigma \vdash a : \tau$ from the premise of the rule, and hence, since $\mu : \Sigma$, 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 $\mathtt{unbound}_\mu$, and in the latter we have $\mathtt{get}[a] \overset{\mu}{\underset{\Sigma}{\longmapsto}} e$. For Rule (37.1b), we have by induction that either e_1 val_Σ or e_1 $\mathtt{unbound}_\mu$, or $e_1 \overset{\mu}{\underset{\Sigma}{\longmapsto}} e_1'$. In the latter two cases we may apply Rule (37.2b) or Rule (37.3b), respectively. If e_1 val_Σ, we apply induction to obtain that either e_2 val_Σ, in which case Rule (37.2d) applies; e_2 $\mathtt{unbound}_\mu$, in which case Rule (37.3b) applies; or $e_2 \overset{\mu}{\underset{\Sigma}{\longmapsto}} e_2'$, in which case Rule (37.2c) applies. □

37.4 Some Subtleties

Fluid binding in the context of a first-order language is easy to understand. If the expression $\mathtt{put}\, e_1\, \mathtt{for}\, a\, \mathtt{in}\, e_2$ has a type such as $\mathtt{nat}$, then its execution consists of the evaluation of e_2 to a number in the presence of a binding of a to the value of expression e_1. When execution is completed, the binding of a is dropped (reverted to its state in the surrounding context), and the value is returned. Since this value is a number, it cannot contain any reference to a, and so no issue of its binding arises.

But what if the type of $\texttt{put}\,e_1\,\texttt{for}\,a\,\texttt{in}\,e_2$ is a function type, so that the returned value is a λ-abstraction? In that case the body of the λ may contain references to the symbol a whose binding is dropped upon return. This raises an important question about the interaction between fluid binding and higher-order functions. For example, consider the expression

$$\texttt{put}\,\overline{17}\,\texttt{for}\,a\,\texttt{in}\,\lambda\,\texttt{(}x\texttt{:nat.}\,x+\texttt{get}\,a\texttt{)}, \tag{37.5}$$

which has type $\texttt{nat}$, given that a is a symbol of the same type. Let us assume, for the sake of discussion, that a is unbound at the point at which this expression is evaluated. Doing so binds a to the number $\overline{17}$, and returns the function $\lambda\,\texttt{(}x\texttt{:nat.}\,x+\texttt{get}\,a\texttt{)}$. This function contains the symbol a, but is returned to a context in which the symbol a is not bound. This means that, for example, application of the expression (37.5) to an argument will incur an error because the symbol a is not bound.

Contrast this with the similar expression

$$\texttt{let}\,y\,\texttt{be}\,\overline{17}\,\texttt{in}\,\lambda\,\texttt{(}x\texttt{:nat.}\,x+y\texttt{)}, \tag{37.6}$$

in which we have replaced the fluid-bound symbol, a, by a statically bound variable, y. This expression evaluates to $\lambda\,\texttt{(}x\texttt{:nat.}\,x+\overline{17}\texttt{)}$, which adds 17 to its argument when applied. There is never any possibility of an unbound symbol arising at execution time, precisely because the identification of scope and extent ensures that the association between a variable and its binding is never violated.

It is hard to say whether either of these two behaviors is “right” or “wrong.” Static binding is an important mechanism for encapsulation of behavior in a program; without static binding, one cannot ensure that the meaning of a variable is unchanged by the context in which it is used. Dynamic binding is used to avoid passing arguments to a function in order to specialize its behavior. Instead we rely on fluid binding to establish the binding of a symbol for the duration of execution of the function, avoiding the need to re-bind the fluids at each call site.

For example, let e stand for the value of expression (37.5), a λ-abstraction whose body is dependent on the binding of the symbol a. This imposes the requirement that the programmer provide a binding for a whenever e is applied to an argument. For example, the expression

$$\texttt{put}\,\overline{7}\,\texttt{for}\,a\,\texttt{in}\,\texttt{(}e\texttt{(}\overline{9}\texttt{))}$$

evaluates to $\overline{15}$, and the expression

$$\mathtt{put}\,\overline{8}\,\mathtt{for}\,a\,\mathtt{in}\,(e(\overline{9}))$$

evaluates to $\overline{17}$. Writing just $e(\overline{9})$, without a surrounding binding for a, results in a run-time error attempting to retrieve the binding of the unbound symbol a.

The alternative to fluid binding is to add an additional parameter to e for the binding of the symbol a, so that one would write

$$e'(\overline{7})(\overline{9})$$

and

$$e'(\overline{8})(\overline{9}),$$

respectively, where e' is the λ-abstraction

$$\lambda\,(a{:}\mathtt{nat}.\,\lambda\,(x{:}\mathtt{nat}.\,x+a)).$$

Using additional arguments can be slightly inconvenient, though, when several call sites have the same binding for a. Using fluid binding we may write

$$\mathtt{put}\,\overline{7}\,\mathtt{for}\,a\,\mathtt{in}\,\langle e(\overline{8}),e(\overline{9})\rangle,$$

whereas using an additional argument we must write

$$\langle e'(\overline{7})(\overline{8}),e'(\overline{7})(\overline{9})\rangle.$$

However, such redundancy can be mitigated by simply factoring out the common part, writing

$$\mathtt{let}\,f\,\mathtt{be}\,e'(\overline{7})\,\mathtt{in}\,\langle f(\overline{8}),f(\overline{9})\rangle.$$

One might argue, then, that it is all a matter of taste. However, a significant drawback of using fluid binding is that the requirement to provide a binding for a is not apparent in the type of e, whereas the type of e' reflects the demand for an additional argument. One may argue that the type system *should* record the dependency of a computation on a specified set of fluid-bound symbols. For example, the expression e might be given a type of the form $\mathtt{nat} \rightarrow_a \mathtt{nat}$, reflecting the demand that a binding for a be provided at the call site.

37.5 Fluid References

The foregoing treatment of fluid binding makes explicit the target of a `get` or `put` operation in the syntax of the language. It is sometimes useful to defer to execution time the choice of which fluid a `get` or a `put` acts on. 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.

The syntax of references as an extension to $\mathcal{L}\{\texttt{fluid}\}$ is given by the following grammar:

Type	τ	$::=$	$\texttt{fluid(}\tau\texttt{)}$	$\tau\,\texttt{fluid}$	fluid
Expr	e	$::=$	$\texttt{fl[}a\texttt{]}$	$\texttt{fl[}a\texttt{]}$	reference
			$\texttt{getfl(}e\texttt{)}$	$\texttt{getfl}\,e$	retrieval
			$\texttt{putfl(}e;e_1;e_2\texttt{)}$	$\texttt{putfl}\,e\,\texttt{is}\,e_1\,\texttt{in}\,e_2$	binding

The expression $\texttt{fl[}a\texttt{]}$ is the symbol a considered as a value of type $\texttt{fluid(}\tau\texttt{)}$. The expressions $\texttt{getfl(}e\texttt{)}$ and $\texttt{putfl(}e;e_1;e_2\texttt{)}$ are analogues of the `get` and `put` operations for fluid-bound symbols.

The statics of these constructs is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma,a:\tau} \texttt{fl[}a\texttt{]} : \texttt{fluid(}\tau\texttt{)}} \tag{37.7a}$$

$$\frac{\Gamma \vdash_{\Sigma} e : \texttt{fluid(}\tau\texttt{)}}{\Gamma \vdash_{\Sigma} \texttt{getfl(}e\texttt{)} : \tau} \tag{37.7b}$$

$$\frac{\Gamma \vdash_{\Sigma} e : \texttt{fluid(}\tau\texttt{)} \quad \Gamma \vdash_{\Sigma} e_1 : \tau \quad \Gamma \vdash_{\Sigma} e_2 : \tau_2}{\Gamma \vdash_{\Sigma} \texttt{putfl(}e;e_1;e_2\texttt{)} : \tau_2} \tag{37.7c}$$

Since we are assuming a stack-like allocation of symbols, references to fluids cannot be considered to be mobile!

The dynamics of references consists of resolving the referent and deferring to the underlying primitives acting on symbols.

$$\frac{}{\texttt{fl[}a\texttt{]}\ \mathsf{val}_{\Sigma,a:\tau}} \tag{37.8a}$$

$$\frac{e \overset{\mu}{\underset{\Sigma}{\longmapsto}} e'}{\texttt{getfl(}e\texttt{)} \overset{\mu}{\underset{\Sigma}{\longmapsto}} \texttt{getfl(}e'\texttt{)}} \tag{37.8b}$$

$$\frac{}{\texttt{getfl}(\texttt{fl}[a]) \overset{\mu}{\underset{\Sigma}{\longmapsto}} \texttt{get}[a]} \tag{37.8c}$$

$$\frac{e \overset{\mu}{\underset{\Sigma}{\longmapsto}} e'}{\texttt{putfl}(e;e_1;e_2) \overset{\mu}{\underset{\Sigma}{\longmapsto}} \texttt{putfl}(e';e_1;e_2)} \tag{37.8d}$$

$$\frac{}{\texttt{putfl}(\texttt{fl}[a];e_1;e_2) \overset{\mu}{\underset{\Sigma}{\longmapsto}} \texttt{put}[a](e_1;e_2)} \tag{37.8e}$$

37.6 Exercises

1. Formalize *deep binding* and *shallow binding* using the stack machine of Chapter 31.

Chapter 38

Dynamic Classification

In Chapters 15 and 29 we investigated the use of sums for the classification of values of disparate type. Every value of a classified type is labelled 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*. A typical situation of this kind arises when two components of a program wish to "share a secret"—that is, to compute a value that is opaque to intermediaries. This can be accomplished by creating a fresh class that is known only to the two "end points" of the communication who may create instances of this class, and pattern match against it to recover the underlying datum. In this sense dynamic classification may be regarded as a *perfect encryption* mechanism in which the class serves as an absolutely unbreakable encryption key under which data may be protected from intruders. It is absolutely unbreakable because, by α-equivalence, it is impossible to "guess" the name of a bound symbol.[1]

One may wonder why a program would ever need to keep a secret from itself. There are, in fact, many useful applications of such an idea. For example, a program may consist of many independent processes communicating over an insecure network. Perfect encryption by dynamic classification supports the creation of *private channels* between processes; see

[1] In practice this is implemented using probabilistic techniques to avoid the need for a central arbiter of unicity of symbol names. However, such methods require a source of randomness, which may be seen as just such an arbiter in disguise. There is no free lunch.

Chapter 46 for further details. Exceptions are another, less obvious, application of dynamic classification. An exception involves two parties, the raiser and the handler. Raising an exception may be viewed as sending a message to a *specific* handler (rather than to any handler that wishes to intercept it). This may be enforced by classifying the exception value with a dynamically generated class that is recognized by the intended handler, and no other.

38.1 Dynamic Classes

A dynamic class is a symbol that may be generated at run-time. A classified value consists of a symbol of type τ 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.

38.1.1 Statics

The syntax of dynamic classification is given by the following grammar:

Type	τ	$::=$	$\mathtt{clsfd}$	$\mathtt{clsfd}$	classified
Expr	e	$::=$	$\mathtt{inst}[a](e)$	$a \cdot e$	instance
			$\mathtt{ifinst}[a](e;x.e_1;e_2)$	$\mathtt{match}\,e\,\mathtt{as}\,a \cdot x \Rightarrow e_1\,\mathtt{ow} \Rightarrow e_2$	comparison

The expression $\mathtt{inst}[a](e)$ is a classified value with class a and underlying value e. The expression $\mathtt{ifinst}[a](e;x.e_1;e_2)$ 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 the following rules:

$$\frac{\Gamma \vdash_{\Sigma,a:\sigma} e : \sigma}{\Gamma \vdash_{\Sigma,a:\sigma} \mathtt{inst}[a](e) : \mathtt{clsfd}} \tag{38.1a}$$

$$\frac{\Gamma \vdash_{\Sigma,a:\sigma} e : \mathtt{clsfd} \quad \Gamma, x : \sigma \vdash_{\Sigma,a:\sigma} e_1 : \tau \quad \Gamma \vdash_{\Sigma,a:\sigma} e_2 : \tau}{\Gamma \vdash_{\Sigma,a:\sigma} \mathtt{ifinst}[a](e;x.e_1;e_2) : \tau} \tag{38.1b}$$

The type associated to the symbol in the signature determines the type of the instance data.

38.1.2 Dynamics

Dynamic classes require a heap-like dynamics for symbol generation, as described in Section 36.1.2 on page 319. This dynamics is defined by the following rules:

$$\frac{e\ \mathsf{val}_{\Sigma,a:\tau}}{\texttt{inst}\,[a]\,(e)\ \mathsf{val}_{\Sigma,a:\tau}} \tag{38.2a}$$

$$\frac{\nu\,\Sigma\,\{\,e\,\} \mapsto \nu\,\Sigma'\,\{\,e'\,\}}{\nu\,\Sigma\,\{\,\texttt{inst}\,[a]\,(e)\,\} \mapsto \nu\,\Sigma'\,\{\,\texttt{inst}\,[a]\,(e')\,\}} \tag{38.2b}$$

$$\frac{}{\nu\,\Sigma\,\{\,\texttt{ifinst}\,[a]\,(\texttt{inst}\,[a]\,(e)\,;x\,.\,e_1;e_2)\,\} \mapsto \nu\,\Sigma\,\{\,[e/x]e_1\,\}} \tag{38.2c}$$

$$\frac{(a \neq a')}{\nu\,\Sigma\,\{\,\texttt{ifinst}\,[a]\,(\texttt{inst}\,[a']\,(e')\,;x\,.\,e_1;e_2)\,\} \mapsto \nu\,\Sigma\,\{\,e_2\,\}} \tag{38.2d}$$

$$\frac{\nu\,\Sigma\,\{\,e\,\} \mapsto \nu\,\Sigma'\,\{\,e'\,\}}{\nu\,\Sigma\,\{\,\texttt{ifinst}\,[a]\,(e;x\,.\,e_1;e_2)\,\} \mapsto \nu\,\Sigma'\,\{\,\texttt{ifinst}\,[a]\,(e';x\,.\,e_1;e_2)\,\}} \tag{38.2e}$$

38.1.3 Safety

Theorem 38.1 (Safety)**.**

1. *If* $\vdash_\Sigma e : \tau$ *and* $\nu\,\Sigma\,\{\,e\,\} \mapsto \nu\,\Sigma'\,\{\,e'\,\}$, *then* $\Sigma' \supseteq \Sigma$ *and* $\vdash_{\Sigma'} e' : \tau$.
2. *If* $\vdash_\Sigma e : \tau$, *then either* $e\ \mathsf{val}_\Sigma$ *or* $\nu\,\Sigma\,\{\,e\,\} \mapsto \nu\,\Sigma'\,\{\,e'\,\}$ *for some* e' *and* Σ'.

Proof. Similar to the safety proofs given in Chapters 15, 16, and 36. □

We may also introduce a type $\texttt{class}(\tau)$ of references to dynamic classes, much as we introduced a type of references to assignables in Chapter 39.

Type	τ	$::=$	$\texttt{class}(\tau)$	$\tau\,\texttt{class}$	class reference
Expr	e	$::=$	$\texttt{cls}\,[a]$	$\&\,a$	reference
			$\texttt{mkinst}(e_1;e_2)$	$\texttt{mkinst}(e_1;e_2)$	instance
			$\texttt{ifofcls}(e_0;e_1;x\,.\,e_2;e_3)$	$\texttt{ifofcls}(e_0;e_1;x\,.\,e_2;e_3)$	dispatch

The statics of these constructs is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma,a:\tau} \texttt{cls}\,[a] : \texttt{class}(\tau)} \tag{38.3a}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \texttt{class}(\tau) \quad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma \texttt{mkinst}(e_1;e_2) : \texttt{clsfd}} \tag{38.3b}$$

$$\frac{\Gamma \vdash_\Sigma e_0 : \texttt{class}(\sigma) \quad \Gamma \vdash_\Sigma e_1 : \texttt{clsfd} \quad \Gamma, x : \sigma \vdash_\Sigma e_2 : \tau \quad \Gamma \vdash_\Sigma e_3 : \tau}{\Gamma \vdash_\Sigma \texttt{ifofcls}(e_0;e_1;x.e_2;e_3) : \tau} \tag{38.3c}$$

The corresponding dynamics is given by these rules:

$$\frac{\nu\,\Sigma\,\{\,e_1\,\} \mapsto \nu\,\Sigma'\,\{\,e_1'\,\}}{\nu\,\Sigma\,\{\,\texttt{mkinst}(e_1;e_2)\,\} \mapsto \nu\,\Sigma'\,\{\,\texttt{mkinst}(e_1';e_2)\,\}} \tag{38.4a}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad \nu\,\Sigma\,\{\,e_2\,\} \mapsto \nu\,\Sigma'\,\{\,e_2'\,\}}{\nu\,\Sigma\,\{\,\texttt{mkinst}(e_1;e_2)\,\} \mapsto \nu\,\Sigma'\,\{\,\texttt{mkinst}(e_1;e_2')\,\}} \tag{38.4b}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\texttt{mkinst}(\texttt{cls}[a];e)\,\} \mapsto \nu\,\Sigma\,\{\,\texttt{inst}[a](e)\,\}} \tag{38.4c}$$

$$\frac{\nu\,\Sigma\,\{\,e_0\,\} \mapsto \nu\,\Sigma'\,\{\,e_0'\,\}}{\nu\,\Sigma\,\{\,\texttt{ifofcls}(e_0;e_1;x.e_2;e_3)\,\} \mapsto \nu\,\Sigma'\,\{\,\texttt{ifofcls}(e_0';e_1;x.e_2;e_3)\,\}} \tag{38.4d}$$

$$\frac{}{\nu\,\Sigma\,\{\,\texttt{ifofcls}(\texttt{cls}[a];e_1;x.e_2;e_3)\,\} \mapsto \nu\,\Sigma\,\{\,\texttt{ifinst}[a](e_1;x.e_2;e_3)\,\}} \tag{38.4e}$$

Rules (38.4d) and (38.4e) 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, if 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.

38.2 Defining Dynamic Classes

The type `clsfd` may be defined in terms of symbolic references, product types, and existential types by the type expression

$$\texttt{clsfd} \triangleq \exists(t.t\,\mathsf{sym} \times t).$$

The introductory form, $\texttt{inst}[a](e)$, where a is a symbol with associated type τ, is defined by the package

$$\texttt{pack}\,\tau\,\texttt{with}\,\langle \&\,a, e\rangle\,\texttt{as}\,\exists(t.t\,\mathsf{sym} \times t).$$

The eliminatory form, $\texttt{ifinst}[a](e;x.e_1;e_2)$ is interesting, because it relies on symbol comparison in the form detailed in Chapter 36. Consider the expression $\texttt{ifinst}[a](e;x.e_1;e_2)$ of type ρ, where a is a symbol of type σ,

e is of type $\texttt{clsfd}$, e_1 is of type ρ given that x is of type σ, and e_2 is of type ρ. The class comparison is defined to be the compound expression

$$\texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,\langle x,y\rangle\,\texttt{:}\,t\,\texttt{sym}\times t\,\texttt{in}\,\texttt{(}e_{body}\texttt{(}y\texttt{))},$$

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, x, of type $t\,\texttt{sym}$, and an underlying value, y, of type t. The expression e_{body}, which is to be defined shortly, will have the type $t \rightharpoonup \rho$, so that the application to y is type correct.

The expression e_{body} compares the symbolic reference, x, to the symbol, a, of type σ, and yields a value of type $t \rightharpoonup \rho$ regardless of the outcome. It is therefore defined to be the expression

$$\texttt{is[}a\texttt{][}u\,\texttt{.}\,u \rightharpoonup \rho\texttt{](}x\texttt{;}e_1'\texttt{;}e_2'\texttt{)}$$

where, in accordance with Rule (36.7b), e_1' has type $[\sigma/u]\texttt{(}u \rightharpoonup \rho\texttt{)} = \sigma \rightharpoonup \rho$, and e_2' has type $[t/u]\texttt{(}u \rightharpoonup \rho\texttt{)} = t \rightharpoonup \rho$. The expression e_1' "knows" that the abstract type, t, is σ, 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. In the positive case we wish to propagate the classified value to e_1, which is accomplished by defining e_1' to be the expression

$$\lambda\,\texttt{(}x\texttt{:}\sigma\texttt{.}\,e_1\texttt{)} : \sigma \rightharpoonup \rho.$$

In the negative case evaluation proceeds to e_2, which is accomplished by defining e_2' to be the expression

$$\lambda\,\texttt{(\_:}t\texttt{.}\,e_2\texttt{)} : t \rightharpoonup \rho.$$

It is a good exercise to check that the statics and dynamics given in Section 38.1 on page 334 are preserved under these definitions. Note in particular that the comparison with a known symbol, a, reveals the identity of the abstract type, t, so that the underlying classified value may be passed to the branch corresponding to a. This is reflected in the type of e_1'. Should the comparison fail, no type information is gained; this is reflected in the type of e_2'. In any case the comparison results in a value of type $t \rightharpoonup \rho$, as required.

38.3 Classifying Secrets

Dynamic classification may be used to enforce *confidentiality* and *integrity* of data values in a program. A value of type $\texttt{clsfd}$ 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.[2]

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

$$(\tau \to \mathtt{clsfd}) \times (\mathtt{clsfd} \to \tau\,\mathtt{opt}),$$

called the *constructor* and *destructor* functions for that class.

```
newsym $a$:$\tau$ in
  ⟨ $\lambda$ ($x$:$\tau$.$a \cdot x$),
    $\lambda$ ($x$:clsfd.match $x$ as $a \cdot y \Rightarrow$ null ow $\Rightarrow$ just($y$)) ⟩.
```

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 τ, it is sufficient to ensure that only trusted parties have access to the constructor. To enforce the *confidentiality* of a value of type τ, 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.

38.4 Exercises

1. Show how to use dynamic classification to implement exceptions.

[2] 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.

Part XIV

Storage Effects

Chapter 39

Modernized 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 *assignable variables* (or *assignables* for short) 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. (Relaxing this restriction is the subject of Chapter 40.)

39.1 Basic Commands

The syntax of **MA** distinguishes pure *expressions* from impure *commands*. The expressions include those of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ (as described in Chapter 13), 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 *mathematical variables*, or just *variables*, and *assignable variables*, or just *assignables*. Variables are introduced by λ-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 a command is, for the time being, restricted to the natural numbers. (But see Section 39.3 on page 349 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 $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ for the sake of brevity.

Type	τ	$::=$	$\mathtt{cmd}$	$\mathtt{cmd}$	command
Expr	e	$::=$	$\mathtt{do}(m)$	$\mathtt{do}\,m$	encapsulation
Cmd	m	$::=$	$\mathtt{ret}(e)$	$\mathtt{ret}\,e$	return
			$\mathtt{bnd}(e;x.m)$	$\mathtt{bnd}\,x \leftarrow e\,;m$	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 $\mathtt{do}(m)$ 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 encapulated command, which is then executed and its returned value is substituted for x prior to executing 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.

39.1.1 Statics

The statics of **MA** consists of two forms of judgement:

1. Expression typing: $\Gamma \vdash_\Sigma e : \tau$.

2. Command formation: $\Gamma \vdash_\Sigma m \; \mathsf{ok}$.

The context, Γ, specifies the types of variables, as usual, and the signature, Σ, consists of a finite set of assignables. These judgements are inductively defined by the following rules:

$$\frac{\Gamma \vdash_\Sigma m \; \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{do}(m) : \mathtt{cmd}} \tag{39.1a}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{nat}}{\Gamma \vdash_\Sigma \mathtt{ret}(e) \; \mathsf{ok}} \tag{39.1b}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{cmd} \quad \Gamma, x : \mathtt{nat} \vdash_\Sigma m \; \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{bnd}(e; x.m) \; \mathsf{ok}} \tag{39.1c}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{nat} \quad \Gamma \vdash_{\Sigma,a} m \; \mathsf{ok}}{\Gamma \vdash_\Sigma \mathtt{dcl}(e; a.m) \; \mathsf{ok}} \tag{39.1d}$$

$$\frac{}{\Gamma \vdash_{\Sigma,a} \mathtt{get}[a] \; \mathsf{ok}} \tag{39.1e}$$

$$\frac{\Gamma \vdash_{\Sigma,a} e : \mathtt{nat}}{\Gamma \vdash_{\Sigma,a} \mathtt{set}[a](e) \; \mathsf{ok}} \tag{39.1f}$$

Rule (39.1a) is the introductory rule for the type cmd, and Rule (39.1c) is the corresponding eliminatory form. Rule (39.1d) 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 Σ. Rule (39.1e) states that the command to retrieve the contents of an assignable, a, returns a natural number. Rule (39.1f) states that we may assign a natural number to an assignable.

39.1.2 Dynamics

The dynamics of **MA** is defined in terms of a *memory*, μ, a finite function assigning a numeral to each of a finite set of assignables.

The dynamics of expressions consists of these two judgement forms:

1. $e \; \mathsf{val}_\Sigma$, stating that e is a value relative to Σ.

2. $e \underset{\Sigma}{\longmapsto} e'$, stating that the expression e steps to the expression e'.

These judgements are inductively defined by the following rules, together with the rules defining the dynamics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ (see Chapter 13). 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.

$$\frac{}{\texttt{do}(m)\ \mathsf{val}_{\Sigma}} \tag{39.2a}$$

Rule (39.2a) states that an encapsulated command is a value.

The dynamics of commands consists of these two judgement forms:

1. $m \parallel \mu\ \mathsf{final}_{\Sigma}$ stating that the state $m \parallel \mu$ is fully executed.
2. $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$ stating that the state $m \parallel \mu$ steps to the state $m' \parallel \mu'$, relative to the set of assignables, Σ.

These judgements are inductively defined by the following rules:

$$\frac{e\ \mathsf{val}_{\Sigma}}{\texttt{ret}(e) \parallel \mu\ \mathsf{final}_{\Sigma}} \tag{39.3a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\texttt{ret}(e) \parallel \mu \underset{\Sigma}{\mapsto} \texttt{ret}(e') \parallel \mu} \tag{39.3b}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\texttt{bnd}(e; x.m) \parallel \mu \underset{\Sigma}{\mapsto} \texttt{bnd}(e'; x.m) \parallel \mu} \tag{39.3c}$$

$$\frac{e\ \mathsf{val}_{\Sigma}}{\texttt{bnd}(\texttt{do}(\texttt{ret}(e)); x.m) \parallel \mu \underset{\Sigma}{\mapsto} [e/x]m \parallel \mu} \tag{39.3d}$$

$$\frac{m_1 \parallel \mu \underset{\Sigma}{\mapsto} m_1' \parallel \mu'}{\texttt{bnd}(\texttt{do}(m_1); x.m_2) \parallel \mu \underset{\Sigma}{\mapsto} \texttt{bnd}(\texttt{do}(m_1'); x.m_2) \parallel \mu'} \tag{39.3e}$$

$$\frac{}{\texttt{get}[a] \parallel \mu \otimes \langle a : e \rangle \underset{\Sigma,a}{\longmapsto} \texttt{ret}(e) \parallel \mu \otimes \langle a : e \rangle} \tag{39.3f}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\texttt{set}[a](e) \parallel \mu \underset{\Sigma}{\mapsto} \texttt{set}[a](e') \parallel \mu} \tag{39.3g}$$

$$\frac{e\ \mathsf{val}_{\Sigma}}{\texttt{set}[a](e) \parallel \mu \otimes \langle a : \_ \rangle \underset{\Sigma}{\mapsto} \texttt{ret}(e) \parallel \mu \otimes \langle a : e \rangle} \tag{39.3h}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\mathtt{dcl}(e;a.m) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{dcl}(e';a.m) \parallel \mu} \tag{39.3i}$$

$$\frac{e\ \mathsf{val}_{\Sigma} \quad m \parallel \mu \otimes \langle a : e\rangle \underset{\Sigma,a}{\longmapsto} m' \parallel \mu' \otimes \langle a : e'\rangle}{\mathtt{dcl}(e;a.m) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{dcl}(e';a.m') \parallel \mu'} \tag{39.3j}$$

$$\frac{e\ \mathsf{val}_{\Sigma} \quad e'\ \mathsf{val}_{\Sigma,a}}{\mathtt{dcl}(e;a.\mathtt{ret}(e')) \parallel \mu \underset{\Sigma}{\mapsto} \mathtt{ret}(e') \parallel \mu} \tag{39.3k}$$

Rule (39.3a) specifies that a `ret` command is final if its argument is a value. Rules (39.3c) to (39.3e) 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 (39.3i) to (39.3k) 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*.

39.1.3 Safety

The judgement $m \parallel \mu\ \mathsf{ok}_{\Sigma}$ is defined by the rule

$$\frac{\vdash_{\Sigma} m\ \mathsf{ok} \quad \mu : \Sigma}{m \parallel \mu\ \mathsf{ok}_{\Sigma}} \tag{39.4}$$

where the auxiliary judgement $\mu : \Sigma$ is defined by the rule

$$\frac{\forall a : \sigma \in \Sigma \quad \exists e \quad \mu(a) = e \text{ and } e\ \mathsf{val}_{\emptyset} \text{ and } \vdash_{\emptyset} e : \mathtt{nat}}{\mu : \Sigma} \tag{39.5}$$

That is, the memory must bind a number to each location in Σ.

Theorem 39.1 (Preservation)**.**

1. *If* $e \underset{\Sigma}{\mapsto} e'$ *and* $\vdash_{\Sigma} e : \tau$*, then* $\vdash_{\Sigma} e' : \tau$*.*

2. *If $m \parallel \mu \underset{\Sigma}{\longmapsto} m' \parallel \mu'$, with $\vdash_\Sigma m$ ok and $\mu : \Sigma$, then $\vdash_\Sigma m'$ ok and $\mu' : \Sigma$.*

Proof. Simultaneously, by induction on Rules (39.2) and (39.3).

Consider Rule (39.3j). Assume that $\vdash_\Sigma \mathtt{dcl}(e;a.m)$ ok and $\mu : \Sigma$. By inversion of typing we have $\vdash_\Sigma e : \mathtt{nat}$ and $\vdash_{\Sigma,a} m$ ok. Since e val_Σ and $\mu : \Sigma$, we have $\mu \otimes \langle a : e \rangle : \Sigma, a$. By induction we have $\vdash_{\Sigma,a} m'$ ok and $\mu' \otimes \langle a : e \rangle : \Sigma, a$, from which the result follows immediately.

Consider Rule (39.3k). Assume that $\vdash_\Sigma \mathtt{dcl}(e;a.\mathtt{ret}(e'))$ ok and $\mu : \Sigma$. By inversion we have $\vdash_\Sigma e : \mathtt{nat}$, $\vdash_{\Sigma,a} \mathtt{ret}(e')$ ok, and hence that $\vdash_{\Sigma,a} e' : \mathtt{nat}$. But since e' $\mathsf{val}_{\Sigma,a}$, we also have $\vdash_\Sigma e' : \mathtt{nat}$, as required. □

Theorem 39.2 (Progress)**.**

1. *If $\vdash_\Sigma e : \tau$, then either e val$_\Sigma$, or there exists e' such that $e \underset{\Sigma}{\longmapsto} e'$.*

2. *If $\vdash_\Sigma m$ ok and $\mu : \Sigma$, then either $m \parallel \mu$ final$_\Sigma$ or $m \parallel \mu \underset{\Sigma}{\longmapsto} m' \parallel \mu'$ for some μ' and m'.*

Proof. Simultaneously, by induction on Rules (39.1). Consider Rule (39.1d). By the first inductive hypothesis we have either $e \underset{\Sigma}{\longmapsto} e'$ or e val_Σ. In the former case Rule (39.3i) applies. In the latter, we have by the second inductive hypothesis either $m \parallel \mu \otimes \langle a : e \rangle$ $\mathsf{final}_{\Sigma,a}$ or $m \parallel \mu \otimes \langle a : e \rangle \underset{\Sigma,a}{\longmapsto} m' \parallel \mu' \otimes \langle a : e' \rangle$. In the former case we apply Rule (39.3k), and in the latter, Rule (39.3j). □

A variant of **MA** treats the operation $\mathtt{get}[a]$ as a form of *expression*, rather than as a form of *command*. This allows us to write expressions such as $a + b$ for the sum of the contents of assignables a and b, rather than have to write a command that explicitly fetches the contents of a and b, returning their sum.

To allow for this we must enrich the dynamics of expressions to allow access to the bindings of the active assignables, writing $e \overset{\mu}{\underset{\Sigma}{\longmapsto}} e'$ to state that one step of evaluation of the expression e relative to Σ and μ results in the expression e'. The definition of this judgement includes the rule

$$\frac{}{\mathtt{get}[a] \overset{\mu\otimes\langle a : e\rangle}{\underset{\Sigma,a}{\longmapsto}} e} , \tag{39.6}$$

which allows an expression to depend on the contents of an assignable.

39.2 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 $\{x \leftarrow m_1\,;m_2\}$, to stand for the command $\texttt{bnd}\,x \leftarrow \texttt{do}\;(m_1)\,;m_2$. This generalizes to an n-ary form by defining

$$\{x_1 \leftarrow m_1\,; \ldots x_{n-1} \leftarrow m_{n-1}\,; m_n\},$$

to stand for the iterated composition

$$\{x_1 \leftarrow m_1\,; \ldots \{x_{n-1} \leftarrow m_{n-1}\,; m_n\}\}.$$

We sometimes write just $\{m_1\,;m_2\}$ for the composition $\{\_ \leftarrow m_1\,;m_2\}$ 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 $\texttt{run}\,e$, executes an encapsulated command and returns its result; it for $\texttt{bnd}\,x \leftarrow e\,;\texttt{ret}\,x$.

The *conditional* command, $\texttt{if}\;(m)\;m_1\,\texttt{else}\,m_2$, executes either m_1 or m_2 according to whether the result of executing m is zero or not:

$$\{x \leftarrow m\,;\texttt{run}\;(\texttt{ifz}\,x\,\{\texttt{z} \Rightarrow \texttt{do}\,m_1 \mid \texttt{s}(\_) \Rightarrow \texttt{do}\,m_2\})\}.$$

The returned value of the conditional is the value returned by the selected command.

The *while loop* command, $\texttt{while}\;(m_1)\;m_2$, repeatedly executes the command m_2 while the command m_1 yields a non-zero number. It is defined as follows:

$$\texttt{run}\;(\texttt{fix}\,\mathit{loop}\texttt{:cmd}\,\texttt{is}\,\texttt{do}(\texttt{if}\;(m_1)\;\{\texttt{ret}\,\texttt{z}\}\;\texttt{else}\,\{m_2\,;\texttt{run}\,\mathit{loop}\})).$$

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 $\tau \rightharpoonup \texttt{cmd}$ that takes an argument of some type, τ, and yields an unexecuted command as result. 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 `call` e_1`(`e_2`)` is defined to be the command `run` `(`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 `nat` ⇀ `cmd` that returns the factorial of its argument:

```
λx:nat. do {
  dcl r := 1 in
  dcl a := x in
  { while ( a ) {
      r' ← r
    ; a' ← a
    ; r := (x-a'+1)× r'
    ; a := a'-1
    }
    ; r
 }
}
```

The loop maintains that 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.

If, as described in Section 39.2 on the previous page, we admit assignables as forms of expression, this example may be written as follows:

```
λx:nat. do {
  dcl r := 1 in
  dcl a := x in
  { while ( ret (a) ) {
      r := (x-a+1)× r
    ; a := a-1
    }
    ; ret ( r )
  }
}
```

The test governing the `while` loop is the command that returns the contents of the assignable a. However, if assignables are forms of expression, it makes sense to change the syntax of the `while` command so that the test condition is an expression, rather than a command. In this case the expression would simply be `a`, the expression that returns the contents of the assignable a, rather than the more awkward command that returns its contents.

39.3 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 other types of returned value and assignable variables is to consider the proof of Theorem 39.1 on page 345. There we relied on the fact that a value of type `nat` is a composition of successors, starting from zero, 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 we may violate safety as follows:

$$\texttt{dcl}\, a := \texttt{z}\, \texttt{in}\, \texttt{ret}\, \texttt{(}\lambda\, \texttt{(}x\texttt{:nat.do}\, \{a := x\}\texttt{))}.$$

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:

$$\texttt{dcl}\, a := \texttt{z}\, \texttt{in}\, \{b := \lambda\, \texttt{(}x\texttt{:nat.do}\, \{a := x\}\texttt{)}\, \texttt{;ret}\, \texttt{z}\}.$$

We assign to it 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, or, even worse, assign to a *different* assignable that happens to be named a!

The critical step in the proof of safety given in Section 39.1.3 on page 345 is to ensure the following *safety condition*:

$$\text{if } \vdash_{\Sigma,a} e : \tau \text{ and } e \ \mathsf{val}_{\Sigma,a} \text{ then } \vdash_{\Sigma} e : \tau. \tag{39.7}$$

When $\tau = \texttt{nat}$, this step is ensured, because e must be a numeral. If, on the other hand, τ is a procedure type, then e may contain uses of the locally declared assignable, a, and, indeed, the above counterexamples violate this safety condition.

We say that a type, τ, is *mobile*, written $\tau\ \mathsf{mobile}$, if the safety condition (39.7) is valid for *all* values of that type. The proof of safety given above shows that `nat` is mobile. The counterexamples show that procedure types are not mobile. Moreover, simple variations of these examples may be given to show that command types may not be considered mobile either. What about function types other than procedure types? One may think they are mobile, because a pure expression cannot depend on an assignable. While this is indeed the case, the safety condition (39.7) need not be satisfied for such a type. For example, consider the following value of type $\mathtt{nat} \rightharpoonup \mathtt{nat}$:

$$\lambda\,(x\mathtt{:nat.}\,(\lambda\,(\_\mathtt{:cmd.z}))\,(\mathtt{do}\,\{a\})).$$

Although the assignable a is not actually needed to compute the result, it nevertheless occurs in the value, in violation of the safety condition.

To account for this generalization, 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, Σ, of active assignables to assign a type to each active assignable so that Σ has the form of a finite set of assumptions of the form $a : \tau$, where a is an assignable. Second, we replace the judgement $\Gamma \vdash_\Sigma m\ \mathsf{ok}$ by the more general form $\Gamma \vdash_\Sigma m \sim \tau$, stating that m is a well-formed command returning a value of type τ. Third, the type `cmd` must be generalized to $\mathtt{cmd}(\tau)$, which is written in examples as $\tau\ \mathtt{cmd}$, to specify the return type of the encapsulated command.

The statics given in Section 39.1.1 on page 342 may be generalized to admit typed commands and typed assignables, as follows:

$$\frac{\Gamma \vdash_\Sigma m \sim \tau}{\Gamma \vdash_\Sigma \mathtt{do}(m) : \mathtt{cmd}(\tau)} \tag{39.8a}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau \quad \tau\ \mathsf{mobile}}{\Gamma \vdash_\Sigma \mathtt{ret}(e) \sim \tau} \tag{39.8b}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{cmd}(\tau) \quad \Gamma, x : \tau \vdash_\Sigma m \sim \tau'}{\Gamma \vdash_\Sigma \mathtt{bnd}(e; x\,.\,m) \sim \tau'} \tag{39.8c}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau \quad \tau\ \mathsf{mobile} \quad \Gamma \vdash_{\Sigma, a:\tau} m \sim \tau'}{\Gamma \vdash_\Sigma \mathtt{dcl}(e; a\,.\,m) \sim \tau'} \tag{39.8d}$$

$$\frac{}{\Gamma \vdash_{\Sigma,a:\tau} \texttt{get[}a\texttt{]} \sim \tau} \tag{39.8e}$$

$$\frac{\Gamma \vdash_{\Sigma,a:\tau} e : \tau}{\Gamma \vdash_{\Sigma,a:\tau} \texttt{set[}a\texttt{](}e\texttt{)} \sim \tau} \tag{39.8f}$$

Apart from the generalization to track returned types and content types, the most important change is to require that τ be a mobile type in Rules (39.8b) and (39.8d).

The statement of preservation and progress must be correspondingly generalized to account for types.

Theorem 39.3 (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 $\vdash_\Sigma m \sim \tau$ and $\mu : \Sigma$, then $\vdash_\Sigma m' \sim \tau$ and $\mu' : \Sigma$.*

Theorem 39.4 (Progress for Typed Commands)**.**

1. *If $\vdash_\Sigma e : \tau$, then either e val_Σ, or there exists e' such that $e \underset{\Sigma}{\mapsto} e'$.*
2. *If $\vdash_\Sigma m \sim \tau$ and $\mu : \Sigma$, then either $m \parallel \mu$ final_Σ or $m \parallel \mu \underset{\Sigma}{\mapsto} m' \parallel \mu'$ for some μ' and m'.*

The proofs of Theorems 39.3 and 39.4 follows very closely the proof of Theorems 39.1 on page 345 and 39.2 on page 346. The main difference is that we appeal to the safety condition for mobility to ensure that the returned value from a declaration does not involve the declared assignable.

39.4 Capabilities and References

The commands a and $a := e$ operate on a statically specified target assignable, a. That is, a must be in scope at the point where the command occurs. Since a is a static parameter of these commands, and not an argument determined at run-time, it would appear, at first glance, that there is no way to operate on an assignable that is not determined until run-time. For example, how can we write a procedure that, for a dynamically specified assignable, adds two to the contents of that assignable?

One way is to use a *capability* to operate on that assignment. A capability is an encapsulated command that operates on an assignable when it is activated. The *get capability*, or *getter*, for an assignable a of type τ is the command $\texttt{do}\,\{a\}$ of type $\tau\,\texttt{cmd}$ that, when executed, returns the contents of a. The *set capability*, or *setter*, for a is the procedure $\lambda\,(x\,{:}\,\tau.\,\texttt{do}\,\{a := x\})$ that, when applied, assigns its argument to a. Since capabilities are pairs of procedures, they are not mobile.

A general double-increment procedure that operates on any assignable, regardless of whether it is in scope, may be programmed as follows:

$$\lambda\,(\mathit{get}\,{:}\,\texttt{nat}\,\texttt{cmd}.\,\lambda\,(\mathit{set}\,{:}\,\texttt{nat} \rightharpoonup \texttt{nat}\,\texttt{cmd}.\,\texttt{do}\,\{x \leftarrow \mathit{get}\,;\mathit{set}(\texttt{s}(\texttt{s}(x)))\})).$$

The procedure is to be called with a getter and a setter for the same assignable. When executed, it invokes the getter to obtain the contents of that assignable, and then invokes the setter to assign its contents to be two more than the value it contained.

Although it is natural to consider the get and set capabilities for an assignable as a pair, it can be useful to separate them to provide limited access to an assignable in a particular context. If only the get capability is passed to a procedure, then its result may depend on the contents of the underlying assignable, but may not alter it. Similarly, if only the set capability is passed, then it may alter the contents of the underlying assignable, but cannot access its current contents. It is also useful to consider other forms of capability than simple getters and setters. For example, one could define an increment and a decrement capability for an assignable, and pass one or both to a procedure to limit how it may influence the value of that assignable. The possibilities are endless.

Returning to the double-increment example, the type does not constrain the caller to provide get and set capabilities that act on the same assignable. One way to ensure this is to introduce a *name*, or *reference*, to an assignable as a form of value. A reference may be thought of as a token that provides access to the get and set capabilities of an assignable. Moreover, two references may be tested for equality, so that one may determine at run-time whether they refer to the same underlying assignable.[1]

A *reference* is a value of type $\texttt{ref}(\tau)$, where τ is the type of the contents of the assignable to which it refers. A (closed) value of reference type is the name of an assignable thought of as a form of expression. Possessing a reference allows one to perform either a get or a set operation on the underlying assignable. (One may also consider read-only or write-only references, or more complex capabilities for assignables, in a similar manner.)

[1] This can also be achieved using capabilities by using the setter for one to modify the assignable and using the getter for the other to determine whether it changed.

This suggests the following syntax for references:

Type	τ	$::=$	$\texttt{ref}\,(\tau)$	$\tau\,\texttt{ref}$	assignable
Expr	e	$::=$	$\texttt{ref}\,[a]$	$\texttt{\&}\,a$	reference
Cmd	m	$::=$	$\texttt{getref}\,(e)$	$\texttt{@}\,e$	contents
			$\texttt{setref}\,(e_1;e_2)$	$e_1 := e_2$	update

The statics of references is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma,a:\tau} \texttt{ref}\,[a] : \texttt{ref}\,(\tau)} \tag{39.9a}$$

$$\frac{\Gamma \vdash_{\Sigma} e : \texttt{ref}\,(\tau)}{\Gamma \vdash_{\Sigma} \texttt{getref}\,(e) \sim \tau} \tag{39.9b}$$

$$\frac{\Gamma \vdash_{\Sigma} e_1 : \texttt{ref}\,(\tau) \quad \Gamma \vdash_{\Sigma} e_2 : \tau}{\Gamma \vdash_{\Sigma} \texttt{setref}\,(e_1;e_2) \sim \tau} \tag{39.9c}$$

Rule (39.9a) specifies that the name of any active assignable is an expression of type $\texttt{ref}\,(\tau)$.

The dynamics is defined to defer to the corresponding operation on the assignable to which a reference refers.

$$\frac{}{\texttt{ref}\,[a]\ \mathsf{val}_{\Sigma,a}} \tag{39.10a}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{getref}\,(e) \parallel \mu \underset{\Sigma}{\longmapsto} \texttt{getref}\,(e') \parallel \mu} \tag{39.10b}$$

$$\frac{}{\texttt{getref}\,(\texttt{ref}\,[a]) \parallel \mu \underset{\Sigma}{\longmapsto} \texttt{get}\,[a] \parallel \mu} \tag{39.10c}$$

$$\frac{e_1 \underset{\Sigma}{\longmapsto} e_1'}{\texttt{setref}\,(e_1;e_2) \parallel \mu \underset{\Sigma}{\longmapsto} \texttt{setref}\,(e_1';e_2) \parallel \mu} \tag{39.10d}$$

$$\frac{}{\texttt{setref}\,(\texttt{ref}\,[a];e) \parallel \mu \underset{\Sigma}{\longmapsto} \texttt{set}\,[a]\,(e) \parallel \mu} \tag{39.10e}$$

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.

Suprisingly, the addition of references to assignables does not violate the stack discipline, so long as reference types are deemed immobile. This ensures that a reference can never escape the scope of the assignable to which it refers, which is essential to maintaining safety. We leave to the reader the task of proving safety for the extension of **MA** with reference types.

As an example of programming with references, the double increment procedure given earlier can be coded using references, rather than capabilities, as follows:

$$\lambda\,(r\,\texttt{:}\,\texttt{nat}\,\texttt{ref}\,\texttt{.}\,\texttt{do}\,\{x \leftarrow \texttt{@}\,r\,;r := \texttt{s(s(}x\texttt{))}\}\texttt{)}.$$

Since the argument is a reference to an assignable, rather than a get and set capability, it is assured that the body of the procedure acts on a single assignable when performing the get and set operations on the reference.

References and capabilities allow assignables to be treated as values that can be passed as arguments to procedures. This allows us to write programs, such as the double increment procedure, that act on assignables that are not in scope within the body of the procedure. Such expressive power, however, comes at a price: we must carefully consider whether two references refer to the same assignable or not. This phenomenon is called *aliasing*; it greatly complicates reasoning about program correctness.

Consider, for example, the problem of writing 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:

$$\lambda\,(x\,\texttt{:}\,\texttt{nat}\,\texttt{ref}\,\texttt{.}\,\lambda\,(y\,\texttt{:}\,\texttt{nat}\,\texttt{ref}\,\texttt{.}\,\texttt{do}\,\{x' \leftarrow \texttt{@}\,x\,;y' \leftarrow \texttt{@}\,y\,;x := x' + y' + y'\}\texttt{))}.$$

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 twice the contents of the assignable referenced by y.

But now consider the following apparently equivalent implementation of the "same" procedure:

$$\lambda\,(x\,\texttt{:}\,\texttt{nat}\,\texttt{ref}\,\texttt{.}\,\lambda\,(y\,\texttt{:}\,\texttt{nat}\,\texttt{ref}\,\texttt{.}\,\texttt{do}\,\{x \mathrel{\texttt{+=}} y\,;x \mathrel{\texttt{+=}} y\}\texttt{))},$$

where $x \mathrel{\texttt{+=}} y$ is the command

$$\{x' \leftarrow \texttt{@}\,x\,;y' \leftarrow \texttt{@}\,y\,;x := x' + y'\}$$

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 are aliases in that they both refer to the same assignable, a, with contents n_0, the result is that a is to set $4 \times n_0$, instead of the intended $3 \times n_0$.

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 at least quadratically in n (we must consider all possible pairings) and can even be worse when more subtle relationships among three or more variables must be considered. Aliasing is a prime source of errors in imperative programs, and remains a strong argument against using imperative methods whenever possible.

39.5 Exercises

Chapter 40

Mutable Data Structures

In Chapter 39 we considered an imperative programming language that adheres to the stack discipline in that assignables are allocated and deallocated on a last-in, first-out basis. To ensure this we restricted the types of return values from a command, the types of contents of assignables, to be *mobile* types, ones whose values cannot depend on the stack of assignables. Function and command types are not mobile, nor are reference types, because these types may classify values that refer to an assignable.

A major use of references, however, is to implement *mutable* data structures whose structure may be changed at execution time. The classic example is a linked list in which the tail of any initial segment of the list may be changed to refer to another list. Crucially, any such alteration is shared among all uses of that list. (This behavior is in contrast to an *immutable* list, which can never change once created.) The usual way to implement a linked list is to specify that the tail of a list is not another list, but rather a *reference* to an assignable containing the tail of the list. The list structure is altered by setting the target assignable of the reference.

For this strategy to make sense, references must be mobile, and hence that assignables have indefinite extent—they must persist beyond the scope of their declaration. Assignables with indefinite extent are said to be *scope-free*, or simply *free*. In this chapter we consider a variation of Modernized Algol in which all assignables are free, and hence all types are mobile. The dynamics of this variation of Modernized Algol is significantly different from that given in Chapter 39 in that assignables are *heap-allocated*, rather than *stack-allocated*.

We also consider a further variation in which the distinction between commands and expressions is eliminated. This facilitates the use of *benign* *effects* to achieve purely functional behavior using references. An example is a self-adjusting data structure that, externally, is a pure dictionary structure, but which internally makes use of mutation to rebalance itself.

40.1 Free Assignables

The statics of free assignables is essentially the same as that for scoped assignables, except that all types are regarded as mobile. To account for this, the dynamics of free assignables differs fundamentally from the scoped case. The dynamics is given by a transition system between states of the form $\nu\,\Sigma\,\{\,m \parallel \mu\,\}$, in which a command, m, is executed relative to a memory, μ, that assigns values to the assignables declared in Σ. The signature, Σ, is only ever extended by transition; assignables are never deallocated.

The dynamics is inductively defined by the following rules:

$$\frac{e\ \mathsf{val}_{\Sigma}}{\nu\,\Sigma\,\{\,\mathtt{ret}(e) \parallel \mu\,\}\ \mathsf{final}} \tag{40.1a}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{ret}(e) \parallel \mu\,\} \mapsto \nu\,\Sigma\,\{\,\mathtt{ret}(e') \parallel \mu\,\}} \tag{40.1b}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{bnd}(e;x.m) \parallel \mu\,\} \mapsto \nu\,\Sigma\,\{\,\mathtt{bnd}(e';x.m) \parallel \mu\,\}} \tag{40.1c}$$

$$\frac{e\ \mathsf{val}_{\Sigma}}{\nu\,\Sigma\,\{\,\mathtt{bnd}(\mathtt{do}(\mathtt{ret}(e));x.m) \parallel \mu\,\} \mapsto \nu\,\Sigma\,\{\,[e/x]m \parallel \mu\,\}} \tag{40.1d}$$

$$\frac{\nu\,\Sigma\,\{\,m_1 \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,m_1' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{bnd}(\mathtt{do}(m_1);x.m_2) \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,\mathtt{bnd}(\mathtt{do}(m_1');x.m_2) \parallel \mu'\,\}} \tag{40.1e}$$

$$\frac{}{\nu\,\Sigma, a:\tau\,\{\,\mathtt{get}[a] \parallel \mu \otimes \langle a:e\rangle\,\} \mapsto \nu\,\Sigma, a:\tau\,\{\,\mathtt{ret}(e) \parallel \mu \otimes \langle a:e\rangle\,\}} \tag{40.1f}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{set}[a](e) \parallel \mu\,\} \mapsto \nu\,\Sigma\,\{\,\mathtt{set}[a](e') \parallel \mu\,\}} \tag{40.1g}$$

$$\frac{e\ \mathsf{val}_{\Sigma,a:\tau}}{\nu\,\Sigma, a:\tau\,\{\,\mathtt{set}[a](e) \parallel \mu \otimes \langle a:\_\rangle\,\} \mapsto \nu\,\Sigma, a:\tau\,\{\,\mathtt{ret}(e) \parallel \mu \otimes \langle a:e\rangle\,\}} \tag{40.1h}$$

$$\frac{e \underset{\Sigma}{\mapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{dcl}(e;a.m) \parallel \mu\,\} \mapsto \nu\,\Sigma\,\{\,\mathtt{dcl}(e';a.m) \parallel \mu\,\}} \tag{40.1i}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\mathtt{dcl}(e;a\,.\,m)\ \|\ \mu\,\} \mapsto \nu\,\Sigma, a:\tau\,\{\,m\ \|\ \mu \otimes \langle a : e\rangle\,\}} \tag{40.1j}$$

The most important difference is expressed by Rule (40.1j), which allows assignables to escape their scope of declaration.

40.2 Free References

References to assignables are values of type $\mathtt{ref}(\tau)$, where τ is the type of the contents of the underlying assignable. When all types are mobile, references may appear in data structures, may be stored assignables of reference type, and may be returned from commands, without restriction. For example, we may define the command $\mathtt{newref}[\tau](e)$ to stand for the command

$$\mathtt{dcl}\,a := e\,\mathtt{in}\,\mathtt{ret}\,(\&\,a), \tag{40.2}$$

which allocates and initializes an assignable, and immediately returns a reference to it. Obviously the sensibility of this definition relies on the mobility of reference types, and the scope-free allocation of assignables.

The statics and dynamics of this construct may be derived from this definition. The following typing rule is admissible:

$$\frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathtt{newref}[\tau](e) \sim \mathtt{ref}(\tau)} \tag{40.3}$$

Moreover, the dynamics is given by the following rules:

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{newref}[\tau](e)\ \|\ \mu\,\} \mapsto \nu\,\Sigma\,\{\,\mathtt{newref}[\tau](e')\ \|\ \mu\,\}} \tag{40.4a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,\mathtt{newref}[\tau](e)\ \|\ \mu\,\} \mapsto \nu\,\Sigma, a:\tau\,\{\,\mathtt{ret}(\mathtt{ref}[a])\ \|\ \mu \otimes \langle a : e\rangle\,\}} \tag{40.4b}$$

The dynamics of the getref and setref commands is essentially the same as in Chapter 39, but must be adapted to the setting of free assignables.

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\nu\,\Sigma\,\{\,\mathtt{getref}(e)\ \|\ \mu\,\} \mapsto \nu\,\Sigma\,\{\,\mathtt{getref}(e')\ \|\ \mu\,\}} \tag{40.5a}$$

$$\frac{}{\nu\,\Sigma\,\{\,\mathtt{getref}(\mathtt{ref}[a])\ \|\ \mu\,\} \mapsto \nu\,\Sigma\,\{\,\mathtt{get}[a]\ \|\ \mu\,\}} \tag{40.5b}$$

$$\frac{e_1 \underset{\Sigma}{\mapsto} e_1'}{\nu\,\Sigma\,\{\,\texttt{setref}\,(e_1;e_2)\;\|\;\mu\,\} \mapsto \nu\,\Sigma\,\{\,\texttt{setref}\,(e_1';e_2)\;\|\;\mu\,\}} \tag{40.5c}$$

$$\frac{}{\nu\,\Sigma\,\{\,\texttt{setref}\,(\texttt{ref}\,[a];e_2)\;\|\;\mu\,\} \mapsto \nu\,\Sigma\,\{\,\texttt{set}\,[a]\,(e_2)\;\|\;\mu\,\}} \tag{40.5d}$$

Observe that the evaluation of expressions cannot alter or extend the memory, only commands may do this.

40.3 Safety

The proof of safety for free assignables and references is surprisingly tricky. The main difficulty is to account for the possibility of cyclic dependencies of data structures in the memory. The contents of one assignable may contain a 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 $\texttt{nat} \rightarrow \texttt{nat cmd}$:

$$\lambda\,(x\!:\!\texttt{nat}.\,\texttt{ifz}\,x\,\{\texttt{z} \Rightarrow \texttt{do}\,\{\texttt{ret}\,(1)\}\mid \texttt{s}(x') \Rightarrow \texttt{do}\,\{f \leftarrow a\,;y \leftarrow \texttt{call}\,f\,(x')\,;\texttt{ret}\,(x*y)\}\}).$$

Let μ be a memory of the form $\mu' \otimes \langle a : e\rangle$ 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! (We will see in Section 40.4 on page 362 that such a situation can arise—the memory need not be "preloaded" for such cycles to arise.)

The possibility of cyclic dependencies means that some care in the definition of the judgement $\mu : \Sigma$ is required. The following rule defines the well-formed states:

$$\frac{\vdash_\Sigma m \sim \tau \quad \vdash_\Sigma \mu : \Sigma}{\nu\,\Sigma\,\{\,m\;\|\;\mu\,\}\;\mathsf{ok}} \tag{40.6}$$

The first premise of the rule states that the command m is well-formed relative to Σ. The second premise states that the memory, μ, conforms to Σ, *relative to the whole of* Σ so that cyclic dependencies are permitted. The judgement $\vdash_{\Sigma'} \mu : \Sigma$ is defined as follows:

$$\frac{\forall a : \sigma \in \Sigma \quad \exists e \quad \mu(a) = e \text{ and } \vdash_{\Sigma'} e : \sigma}{\vdash_{\Sigma'} \mu : \Sigma} \tag{40.7}$$

Theorem 40.1 (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* $\nu\,\Sigma\,\{\,m \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,m' \parallel \mu'\,\}$*, then* $\nu\,\Sigma'\,\{\,m' \parallel \mu'\,\}$ *ok.*

Proof. Simultaneously, by induction on transition. We prove the following stronger form of the second statement:

> If $\nu\,\Sigma\,\{\,m \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,m' \parallel \mu'\,\}$, where $\vdash_\Sigma m \sim \tau$, $\vdash_\Sigma \mu : \Sigma$, then Σ' extends Σ, and $\vdash_{\Sigma'} m' \sim \tau$, and $\vdash_{\Sigma'} \mu' : \Sigma'$.

Consider, for example, the transition

$$\nu\,\Sigma\,\{\,\mathtt{dcl}(e;a\,.\,m) \parallel \mu\,\} \mapsto \nu\,\Sigma, a:\sigma\,\{\,m \parallel \mu \otimes \langle a : e\rangle\,\}$$

where e val_Σ. By assumption and inversion of Rule (39.8d) we have σ such that $\vdash_\Sigma e : \sigma$, $\vdash_{\Sigma,a:\sigma} m \sim \tau$, and $\vdash_\Sigma \mu : \Sigma$. But since extension of Σ with a fresh assignable does not affect typing, we also have $\vdash_{\Sigma,a:\sigma} \mu : \Sigma$ and $\vdash_{\Sigma,a:\sigma} e : \sigma$, from which it follows by Rule (40.7) that $\vdash_{\Sigma,a:\sigma} \mu \otimes \langle a : e\rangle : \Sigma, a : \sigma$.

The other cases follow a similar pattern, and are left as an exercise for the reader.

□

Theorem 40.2 (Progress)**.**

1. *If* $\vdash_\Sigma e : \tau$*, then either* e val_Σ *or there exists* e' *such that* $e \underset{\Sigma}{\mapsto} e'$.

2. *If* $\nu\,\Sigma\,\{\,m \parallel \mu\,\}$ *ok then either* $\nu\,\Sigma\,\{\,m \parallel \mu\,\}$ *final or* $\nu\,\Sigma\,\{\,m \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,m' \parallel \mu'\,\}$ *for some* Σ'*,* μ'*, and* m'.

Proof. Simultaneously, by induction on typing. For the second statement we prove the stronger form

> If $\vdash_\Sigma m \sim \tau$ and $\vdash_\Sigma \mu : \Sigma$, then either $\nu\,\Sigma\,\{\,m \parallel \mu\,\}$ final, or $\nu\,\Sigma\,\{\,m \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,m' \parallel \mu'\,\}$ for some Σ', μ', and m'.

Consider, for example, the typing rule

$$\frac{\Gamma \vdash_\Sigma e : \sigma \quad \Gamma \vdash_{\Sigma,a:\sigma} m \sim \tau}{\Gamma \vdash_\Sigma \mathtt{dcl}(e;a\,.\,m) \sim \tau}$$

We have by the first inductive hypothesis that either e val_Σ or $e \underset{\Sigma}{\mapsto} e'$ for some e'. In the latter case we have by Rule (40.1i)

$$\nu\,\Sigma\,\{\,\mathtt{dcl}(e;a\,.\,m) \parallel \mu\,\} \mapsto \nu\,\Sigma\,\{\,\mathtt{dcl}(e';a\,.\,m) \parallel \mu\,\}.$$

In the former case we have by Rule (40.1j) that

$$\nu\,\Sigma\,\{\,\texttt{dcl}(e;a\,.\,m)\parallel\mu\,\}\mapsto\nu\,\Sigma,a:\sigma\,\{\,m\parallel\mu\otimes\langle a:e\rangle\,\}.$$

As another example, consider the typing rule

$$\frac{}{\Gamma\vdash_{\Sigma,a:\tau}\texttt{get}\,[a]\sim\tau}$$

By assumption $\vdash_{\Sigma,a:\tau}\mu:\Sigma,a:\tau$, and hence there exists e $\mathsf{val}_{\Sigma,a:\tau}$ such that $\mu=\mu'\otimes\langle a:e\rangle$ and $\vdash_{\Sigma,a:\tau}e:\tau$. By Rule (40.1f)

$$\nu\,\Sigma,a:\tau\,\{\,\texttt{get}\,[a]\parallel\mu'\otimes\langle a:e\rangle\,\}\mapsto\nu\,\Sigma,a:\tau\,\{\,\texttt{ret}(e)\parallel\mu'\otimes\langle a:e\rangle\,\},$$

as required. The other cases are handled similarly.

□

40.4 Integrating Commands and Expressions

The *modal* formulation of free assignables maintains the separation between expressions and commands inherited from **MA**. An important variation on this design eliminates this distinction, allowing expressions to have both value and effect. This is called the *integral* formulation of free assignables; it has advantages and disadvantages compared to the modal formulation.

The statics of the integral formulation is obtained by consolidating expressions and commands, dropping the return command since it serves no purpose. The following rules are illustrative:

$$\frac{\Gamma\vdash_{\Sigma}e:\tau}{\Gamma\vdash_{\Sigma}\texttt{do}(e):\texttt{cmd}(\tau)}\tag{40.8a}$$

$$\frac{\Gamma\vdash_{\Sigma}e_1:\texttt{cmd}(\tau_1)\quad\Gamma,x:\tau_1\vdash_{\Sigma}e_2:\tau_2}{\Gamma\vdash_{\Sigma}\texttt{bnd}(e_1;x\,.\,e_2):\tau_2}\tag{40.8b}$$

$$\frac{\Gamma\vdash_{\Sigma}e_1:\tau_1\quad\Gamma\vdash_{\Sigma,a:\tau_1}e_2:\tau_2}{\Gamma\vdash_{\Sigma}\texttt{dcl}(e_1;a\,.\,e_2):\tau_2}\tag{40.8c}$$

$$\frac{}{\Gamma\vdash_{\Sigma,a:\tau}\texttt{get}\,[a]:\tau}\tag{40.8d}$$

$$\frac{\Gamma\vdash_{\Sigma,a:\tau}e:\tau}{\Gamma\vdash_{\Sigma,a:\tau}\texttt{set}\,[a]\,(e):\tau}\tag{40.8e}$$

The dynamics of the integral formulation of mutation is defined as a transition system between states of the form $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$, where e is an expression involving the assignables declared in Σ, and μ is a memory providing values for each of these assignables. It is a straightforward exercise to reformulate Rules (40.1) to eliminate the mode distinction between commands and expressions. Rules (40.5) may similarly be adapted to the integral setting.

The modal and integral formulations of references have complementary strengths and weaknesses. The chief virtue of the modal formulation is that the use of assignment is confined to commands, leaving expressions as pure computations. One consequence is that typing judgements for expressions retain their force even in the presence of references to free assignables, so that the type $\texttt{unit} \rightharpoonup \texttt{unit}$ contains only the identity and the divergent functions, and the type $\texttt{nat} \rightharpoonup \texttt{nat}$ consists solely of partial functions on the natural numbers. By contrast the integral formulation enjoys none of these properties. Any expression may alter or allocate new assignables, and the semantics of typing assertions is therefore significantly weakened compared to the modal formulation. In particular, the type $\texttt{unit} \rightharpoonup \texttt{unit}$ contains infinitely many distinct functions, and the type $\texttt{nat} \rightharpoonup \texttt{nat}$ contains procedures that in no way represent partial functions because they retain state across calls.

While the modal separation of pure expressions from impure commands may seem like an unalloyed good, the situation is actually more complex. The central problem is that the modal formulation inhibits the use of effects to implement purely functional behavior. For example, a self-adjusting tree, such as a splay tree, uses in-place mutation to provide an efficient implementation of what is otherwise a purely functional dictionary structure mapping keys to values. This is an example of a *benign effect*, one that does not affect the behavior, but only the efficiency, of the implementation.

Many other examples arise in practice. For example, suppose that we wish to instrument an otherwise pure functional program with code to collect execution statistics for profiling. In the integral setting it is a simple matter to allocate free assignables that contain profiling information collected by assignments that update their contents at critical points in the program. In the modal setting, however, we must globally restructure the program to transform it from a pure expression to an impure command. Another example is provided by the technique of *backpatching* for implementing recursion using a free assignable, which we now describe in more detail.

In the integral formulation we may implement the factorial function using backpatching as follows:

```
dcl a := λn:nat.0 in
  { f ← λn:nat.ifz(n, 1, n'.n * a(n'))
  ; _ ← a := f
  ; f
  }
```

wherein we have used the concrete syntax for commands introduced in Chapter 39. Observe that the assignable `a` is used as an expression standing for its contents (that is, it stands for the abstract syntax `get[`*a*`]`).

This expression returns a function of type `nat` ⇀ `nat` that is obtained by (a) allocating a free assignable initialized arbitrarily (and immaterially) with a function of this type, (b) defining a λ-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 result is a value of function type that uses an assignable "under the hood" in a manner not visible to its clients.

In contrast the modal formulation forces us to make explicit the reliance on private state.

```
dcl a := λn:nat.do{ret 0} in
  { f ← ret (λ n:nat. ...)
  ; _ ← a := f
  ; ret f
  }
```

where the elided procedure body is as follows:

```
ifz(n,do{ret(1)},n'.do{f←a; x←run(f(n')); ret (n*x)}).
```

Each branch of the conditional test returns a command. In the case that the argument is zero, the command simply returns the value `1`. Otherwise, it fetches the contents of the assignable, calls it on the predecessor, and returns the result of multiplying this by the argument.

The modal implementation of factorial is a command (not an expression) of type `nat` → `(nat cmd)`, which exposes two properties of the backpatching implementation:

1. The command that builds the recursive factorial function is impure, because it allocates and assigns to the assignable used to implement backpatching.

2. The body of the factorial function is impure, because it accesses the assignable to effect the recursive call.

As a result the factorial function (so implemented) may no longer be used as a function, but must instead be called as a procedure. For example, to compute the factorial of n, we must write

$$\{\ \texttt{f} \leftarrow \mathit{fact}\,;\ \texttt{x} \leftarrow \texttt{run}\ (\texttt{f}(n));\ \texttt{return x}\ \}$$

where *fact* stands for the command implementing factorial given above. The factorial procedure is bound to a variable, which is then applied to yield an encapsulated command that, when activated, computes the desired result.

These examples illustrate that exposing the reliance on effects in the type system is both a boon and a bane. Under the integral formulation a "boring" type such as $\texttt{unit} \rightharpoonup \texttt{unit}$ can have very "interesting" behavior—for example, it may depend on or alter the contents of an assignable, or may allocate new assignables. Under the modal formulation a value of such a boring type is indeed boring: it can only be the the identity or the divergent function. An interesting function must have an interesting type such as $\texttt{unit} \rightharpoonup \texttt{unit cmd}$, which makes clear that the body of the function engenders storage effects. On the other hand, as the example of backpatching makes clear, the integral formulation allows one to think of types as descriptions of *behavior*, rather than descriptions of *implementation*. The factorial function, whether implemented using backpatching or not, is a pure function of type $\texttt{nat} \rightharpoonup \texttt{nat}$. The reliance on assignment is an implementation detail that remains hidden from the caller. The modal formulation, however, exposes the reliance on effects in both the definition and implementation of the factorial function, and hence forces it to be treated as an imperative procedure, rather than a pure function.

40.5 Exercises

Part XV

Laziness

Chapter 41

Lazy Evaluation

Lazy evaluation refers to a variety of concepts that seek to avoid evaluation of an expression unless its value is needed, and to share the results of evaluation of an expression among all uses of its, so that no expression need be evaluated more than once. Within this broad mandate, various forms of laziness are considered.

One is the *call-by-need* evaluation strategy for functions. This is a refinement of the *call-by-name* evaluation order in which arguments are passed unevaluated to functions so that it is only evaluated if needed, and, if so, the value is shared among all occurrences of the argument in the body of the function.

Another is the *lazy* evaluation strategy for data structures, including formation of pairs, injections into summands, and recursive folding. The decisions of whether to evaluate the components of a pair, or the argument to an injection or fold, are independent of one another, and of the decision whether to pass arguments to functions in unevaluated form.

Another aspect of laziness is the use of *general recursion* to define self-referential computations, including recursive functions. The role of laziness in this setting is to defer evaluation of any self-reference until it is actually required for a computation.

Traditionally, languages are classified into one of two categories. *Lazy languages* use a call-by-need interpretation of function application, impose a lazy evaluation strategy for data structures, and allow unrestricted use of general recursion. *Strict languages* take the opposite positions: call-by-value for function application, eager evaluation of data structures, and limitations on general recursion (typically, to functions). More recently, however, language designers have come to realize that it is not *whole languages* that should be lazy or strict, but rather that the type system should distinguish lazy and strict evaluation order. In its most basic form this only requires the introduction of a type whose values are suspended computations that are evaluated by-need. (A more sophisticated approach is the subject of Chapter 42.)

41.1 Need Dynamics

The distinguishing feature of call-by-need is the use of *memoization* to record the value of an expression whenever it is computed so that, should the value of that expression ever be required again, the stored value can be returned without recomputing it. This is achieved by augmenting the computation state with a *memo table* that associates an expression (not necessarily a value) to each of a finite set of symbols. The symbols serve as *names* of the expressions to which they are associated by the memo table. Whenever the value of a name is required, the associated expression is evaluated and its value is both stored in the memo table under the same name and returned as the value of that name. This ensures that any subsequent evaluation of the same name returns the new value without recomputing it.

Another perspective on call-by-need is that it uses names to mediate *sharing* among multiple occurrences of a sub-expression within a larger expression. Ordinary substitution often replicates an expression, generating one copy for each occurrence of the target of the substitution. Under call-by-need each expression is given a name which serves as a proxy for it. In particular, expression names are substituted for variables so that all occurrences have the same name and hence refer to the same copy of the expression to which it is associated. In this way we economize on both the time required to evaluate the expression, which would be needlessly repeated under call-by-name, and the space required to store it during computation, which would be replicated under call-by-name.

The need dynamics for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is based on a transition system with states of the form $\nu\,\Sigma\,\{\, e \parallel \mu \,\}$, where Σ is a finite set of hypotheses $a_1 : \tau_1, \ldots, a_n : \tau_n$ associating types to names, e is an expression that may involve the names in Σ, and μ maps each name declared in Σ to either an expression or a special symbol, $\bullet$, called the *black hole*. (The role of the black hole will be made clear below.)

The call-by-need dynamics consists of the following two forms of judgement:

1. $e\ \mathsf{val}_{\Sigma}$, stating that e is a value that may involve the names in a.

2. $\nu\,\Sigma\,\{\,e \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}$, stating that one step of evaluation of the expression e relative to memo table μ with the names declared in Σ results in the expression e' relative to the memo table μ' with names declared in Σ'.

The dynamics is defined so that the collection of active names grows monotonically, and so that the type of a name never changes. The memo table may be altered destructively during execution to reflect progress in the evaluation of the expression associated with a given name.

The judgement $e\ \mathsf{val}_\Sigma$ is defined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}_\Sigma} \tag{41.1a}$$

$$\frac{}{\mathtt{s}(a)\ \mathsf{val}_{\Sigma,a:\mathtt{nat}}} \tag{41.1b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}_\Sigma} \tag{41.1c}$$

Rules (41.1a) through (41.1c) specify that $\mathtt{z}$ is a value, any expression of the form $\mathtt{s}(a)$, where a is a name, is a value, and that any λ-abstraction, possibly containing names, is a value. It is important that names themselves are not values, rather they stand for (possibly unevaluated) expressions as specified by the memo table.

The initial and final states of evaluation are defined as follows:

$$\frac{}{\nu\,\emptyset\,\{\,e \parallel \emptyset\,\}\ \mathsf{initial}} \tag{41.2a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{final}} \tag{41.2b}$$

Rule (41.2a) specifies that an initial state consists of an expression evaluated relative to an empty memo table. Rule (41.2b) specifies that a final state has the form $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$, where e is a value relative to Σ.

The transition judgement for the call-by-need dynamics of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$ is defined by the following rules:

$$\frac{e\ \mathsf{val}_{\Sigma,a:\tau}}{\nu\,\Sigma, a:\tau\,\{\,a \parallel \mu \otimes \langle a : e\rangle\,\} \mapsto \nu\,\Sigma, a:\tau\,\{\,e \parallel \mu \otimes \langle a : e\rangle\,\}} \tag{41.3a}$$

$$\frac{\nu\,\Sigma, a:\tau\,\{\, e \parallel \mu \otimes \langle a:\bullet\rangle \,\} \mapsto \nu\,\Sigma', a:\tau\,\{\, e' \parallel \mu' \otimes \langle a:\bullet\rangle \,\}}{\nu\,\Sigma, a:\tau\,\{\, a \parallel \mu\otimes\langle a:e\rangle\,\} \mapsto \nu\,\Sigma', a:\tau\,\{\, a \parallel \mu'\otimes\langle a:e'\rangle\,\}} \tag{41.3b}$$

$$\frac{}{\nu\,\Sigma\,\{\, \texttt{s(}e\texttt{)} \parallel \mu \,\} \mapsto \nu\,\Sigma, a:\texttt{nat}\,\{\, \texttt{s(}a\texttt{)} \parallel \mu\otimes\langle a:e\rangle \,\}} \tag{41.3c}$$

$$\frac{\nu\,\Sigma\,\{\, e \parallel \mu \,\} \mapsto \nu\,\Sigma'\,\{\, e' \parallel \mu' \,\}}{\nu\,\Sigma\,\{\, \texttt{ifz(}e\texttt{;}e_0\texttt{;}x\texttt{.}e_1\texttt{)} \parallel \mu \,\} \mapsto \nu\,\Sigma'\,\{\, \texttt{ifz(}e'\texttt{;}e_0\texttt{;}x\texttt{.}e_1\texttt{)} \parallel \mu' \,\}} \tag{41.3d}$$

$$\frac{}{\nu\,\Sigma\,\{\, \texttt{ifz(z;}e_0\texttt{;}x\texttt{.}e_1\texttt{)} \parallel \mu \,\} \mapsto \nu\,\Sigma\,\{\, e_0 \parallel \mu \,\}} \tag{41.3e}$$

$$\frac{}{\left\{\begin{array}{c} \nu\,\Sigma, a:\texttt{nat}\,\{\, \texttt{ifz(s(}a\texttt{);}e_0\texttt{;}x\texttt{.}e_1\texttt{)} \parallel \mu\otimes\langle a:e\rangle \,\} \\ \mapsto \\ \nu\,\Sigma, a:\texttt{nat}\,\{\, [a/x]e_1 \parallel \mu\otimes\langle a:e\rangle \,\} \end{array}\right\}} \tag{41.3f}$$

$$\frac{\nu\,\Sigma\,\{\, e_1 \parallel \mu \,\} \mapsto \nu\,\Sigma'\,\{\, e_1' \parallel \mu' \,\}}{\nu\,\Sigma\,\{\, \texttt{ap(}e_1\texttt{;}e_2\texttt{)} \parallel \mu \,\} \mapsto \nu\,\Sigma'\,\{\, \texttt{ap(}e_1'\texttt{;}e_2\texttt{)} \parallel \mu' \,\}} \tag{41.3g}$$

$$\frac{}{\left\{\begin{array}{c} \nu\,\Sigma\,\{\, \texttt{ap(lam[}\tau\texttt{](}x\texttt{.}e\texttt{);}e_2\texttt{)} \parallel \mu \,\} \\ \mapsto \\ \nu\,\Sigma, a:\tau\,\{\, [a/x]e \parallel \mu\otimes\langle a:e_2\rangle \,\} \end{array}\right\}} \tag{41.3h}$$

$$\frac{}{\nu\,\Sigma\,\{\, \texttt{fix[}\tau\texttt{](}x\texttt{.}e\texttt{)} \parallel \mu \,\} \mapsto \nu\,\Sigma, a:\tau\,\{\, a \parallel \mu\otimes\langle a:[a/x]e\rangle \,\}} \tag{41.3i}$$

Rule (41.3a) governs a name whose associated expression is a value; the value of the name is the value associated to that name in the memo table. Rule (41.3b) specifies that if the expression associated to a name is not a value, then it is evaluated "in place" until such time as Rule (41.3a) 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 name. The black hole represents the absence of a value for that name, 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 name depends on the name itself.

Rule (41.3c) specifies that evaluation of $\texttt{s}(e)$ allocates a fresh name, a, for the expression e, and yields the value $\texttt{s}(a)$. 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 (41.3f), which governs a conditional branch on a successor, substitutes the name, 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 (41.3g) specifies that the value of the function position of an application must be determined before the application can be executed. Rule (41.3h) specifies that to evaluate an application of a λ-abstraction we allocate a fresh name for the argument, and substitute this name 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 (41.3i). Recall from Chapter 13 that the expression $\texttt{fix}[\tau](x.e)$ stands for the solution of the recursion equation $x = e$, where x may occur within e. Rule (41.3i) computes this solution by associating a fresh name, 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 $\texttt{fix}\,x{:}\tau\,\texttt{is}\,x$ 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 $\texttt{fix}\,f{:}\sigma \to \tau\,\texttt{is}\,\lambda\,(x{:}\sigma.\,e)$ does not get stuck, because the self-reference is "hidden" within the λ-abstraction, and hence need not be evaluated to determine the value of the binding.

41.2 Safety

We write $\Sigma; \Gamma \vdash e : \tau$ to mean that e has type τ under the assumptions Σ and Γ as defined by Rules (13.1). That is, we regard the names Σ as variables for the purposes of the statics.

The judgement $\nu\,\Sigma\,\{\, e \parallel \mu \,\}\ \mathsf{ok}$ is defined by the following rules:

$$\frac{\Sigma \vdash e : \tau \quad \Sigma \vdash \mu : \Sigma}{\nu\,\Sigma\,\{\, e \parallel \mu \,\}\ \mathsf{ok}} \tag{41.4a}$$

$$\frac{\forall a : \tau \in \Sigma \quad \mu(a) = e \neq \bullet \Longrightarrow \Sigma' \vdash e : \tau}{\Sigma' \vdash \mu : \Sigma} \tag{41.4b}$$

Rule (41.4b) permits self-reference through the memo table by allowing the expression associated to a name, a, to contain a, or, more generally, to contain a name whose associated expression contains a, and so on through any finite chain of such dependencies. Moreover, a name that is bound to the "black hole" is deemed to be of any type.

Theorem 41.1 (Preservation). *Suppose that* $\nu\,\Sigma\,\{\, e \parallel \mu \,\} \mapsto \nu\,\Sigma'\,\{\, e' \parallel \mu' \,\}$*. If* $\nu\,\Sigma\,\{\, e \parallel \mu \,\}$ *ok, then* $\nu\,\Sigma'\,\{\, e' \parallel \mu' \,\}$ *ok.*

Proof. We prove by induction on Rules (41.3) that if $\nu\,\Sigma\,\{\, e \parallel \mu \,\} \mapsto \nu\,\Sigma'\,\{\, e' \parallel \mu' \,\}$ and $\Sigma \vdash \mu : \Sigma$ and $\Sigma \vdash e : \tau$, then $\Sigma' \supseteq \Sigma$ and $\Sigma' \vdash \mu' : \Sigma'$ and $\Sigma' \vdash e' : \tau$.

Consider Rule (41.3b), for which we have $e = e' = a$, $\mu = \mu_0 \otimes \langle a : e_0 \rangle$, $\mu' = \mu_0' \otimes \langle a : e_0' \rangle$, and

$$\nu\,\Sigma, a : \tau\,\{\, e_0 \parallel \mu_0 \otimes \langle a : \bullet \rangle \,\} \mapsto \nu\,\Sigma', a : \tau\,\{\, e_0' \parallel \mu_0' \otimes \langle a : \bullet \rangle \,\}.$$

Assume that $\Sigma, a : \tau \vdash \mu : \Sigma, a : \tau$. It follows that $\Sigma, a : \tau \vdash e_0 : \tau$ and $\Sigma, a : \tau \vdash \mu_0 : \Sigma$, and hence that

$$\Sigma, a : \tau \vdash \mu_0 \otimes \langle a : \bullet \rangle : \Sigma, a : \tau.$$

We have by induction that $\Sigma' \supseteq \Sigma$ and $\Sigma', a : \tau \vdash e_0' : \tau'$ and

$$\Sigma', a : \tau \vdash \mu_0 \otimes \langle a : \bullet \rangle : \Sigma, a : \tau.$$

But then

$$\Sigma', a : \tau \vdash \mu' : \Sigma', a : \tau,$$

which suffices for the result.

Consider Rule (41.3g), so that e is the application $\mathtt{ap}(e_1; e_2)$ and

$$\nu\,\Sigma\,\{\, e_1 \parallel \mu \,\} \mapsto \nu\,\Sigma'\,\{\, e_1' \parallel \mu' \,\}.$$

Suppose that $\Sigma \vdash \mu : \Sigma$ and $\Sigma \vdash e : \tau$. By inversion of typing $\Sigma \vdash e_1 : \tau_2 \to \tau$ for some type τ_2 such that $\Sigma \vdash e_2 : \tau_2$. By induction $\Sigma' \supseteq \Sigma$ and $\Sigma' \vdash \mu' : \Sigma'$ and $\Sigma' \vdash e_1' : \tau_2 \to \tau$. By weakening we have $\Sigma' \vdash e_2 : \tau_2$, so that $\Sigma' \vdash \mathtt{ap}(e_1'; e_2) : \tau$, which is enough for the result.

□

The statement of the progress theorem allows for the possibility of encountering a black hole, representing a checkable form of non-termination. The judgement $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$ loops, stating that e diverges by virtue of encountering the black hole, is defined by the following rules:

$$\frac{}{\nu\,\Sigma, a:\tau\,\{\,a \parallel \mu \otimes \langle a : \bullet\rangle\,\}\ \mathsf{loops}} \tag{41.5a}$$

$$\frac{\nu\,\Sigma, a:\tau\,\{\,e \parallel \mu \otimes \langle a : \bullet\rangle\,\}\ \mathsf{loops}}{\nu\,\Sigma, a:\tau\,\{\,a \parallel \mu \otimes \langle a : e\rangle\,\}\ \mathsf{loops}} \tag{41.5b}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\}\ \mathsf{loops}}{\nu\,\Sigma\,\{\,\mathtt{ifz}(e;e_0;x.e_1) \parallel \mu\,\}\ \mathsf{loops}} \tag{41.5c}$$

$$\frac{\nu\,\Sigma\,\{\,e_1 \parallel \mu\,\}\ \mathsf{loops}}{\nu\,\Sigma\,\{\,\mathtt{ap}(e_1;e_2) \parallel \mu\,\}\ \mathsf{loops}} \tag{41.5d}$$

Theorem 41.2 (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 μ' and e' such that $\nu\,\Sigma\,\{\,e \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}$.*

Proof. We proceed by induction on the derivations of $\Sigma \vdash e : \tau$ and $\Sigma \vdash \mu : \Sigma$ implicit in the derivation of $\nu\,\Sigma\,\{\,e \parallel \mu\,\}$ ok.

Consider Rule (13.1a), where the variable, a, is declared in Σ. Thus $\Sigma = \Sigma_0, a : \tau$ and $\Sigma \vdash \mu : \Sigma$. It follows that $\mu = \mu_0 \otimes \langle a : e_0\rangle$ with $\Sigma \vdash \mu_0 : \Sigma_0$ and $\Sigma \vdash e_0 : \tau$. Note that $\Sigma \vdash \mu_0 \otimes \langle a : \bullet\rangle : \Sigma$. Applying induction to the derivation of $\Sigma \vdash e_0 : \tau$, we consider three cases:

1. $\nu\,\Sigma\,\{\,e_0 \parallel \mu \otimes \langle a : \bullet\rangle\,\}$ final. By inversion of Rule (41.2b) we have e_0 val_Σ, and hence by Rule (41.3a) we obtain $\nu\,\Sigma\,\{\,a \parallel \mu\,\} \mapsto \nu\,\Sigma\,\{\,e_0 \parallel \mu\,\}$.

2. $\nu\,\Sigma\,\{\,e_0 \parallel \mu_0 \otimes \langle a : \bullet\rangle\,\}$ loops. By applying Rule (41.5b) we obtain $\nu\,\Sigma\,\{\,a \parallel \mu\,\}$ loops.

3. $\nu\,\Sigma\,\{\,e_0 \parallel \mu_0 \otimes \langle a : \bullet\rangle\,\} \mapsto \nu\,\Sigma'\,\{\,e_0' \parallel \mu_0' \otimes \langle a : \bullet\rangle\,\}$. By applying Rule (41.3b) we obtain

$$\nu\,\Sigma\,\{\,a \parallel \mu \otimes \langle a : e_0\rangle\,\} \mapsto \nu\,\Sigma'\,\{\,a \parallel \mu' \otimes \langle a : e_0'\rangle\,\}.$$

□

41.3 Lazy Data Structures

The call-by-need dynamics extends to product, sum, and recursive types in a straightforward manner. For example, the need dynamics of lazy product types is given by the following rules:

$$\frac{}{\texttt{pair}(a_1;a_2)\ \mathsf{val}_{\Sigma,a_1:\tau_1,a_2:\tau_2}} \tag{41.6a}$$

$$\frac{}{\left\{\begin{array}{c} \nu\,\Sigma\,\{\,\texttt{pair}(e_1;e_2)\parallel\mu\,\} \\ \mapsto \\ \nu\,\Sigma,a_1:\tau_1,a_2:\tau_2\,\{\,\texttt{pair}(a_1;a_2)\parallel\mu\otimes\langle a_1:e_1\rangle\otimes\langle a_2:e_2\rangle\,\} \end{array}\right\}} \tag{41.6b}$$

$$\frac{\nu\,\Sigma\,\{\,e\parallel\mu\,\}\mapsto\nu\,\Sigma'\,\{\,e'\parallel\mu'\,\}}{\nu\,\Sigma\,\{\,\texttt{proj[l]}(e)\parallel\mu\,\}\mapsto\nu\,\Sigma'\,\{\,\texttt{proj[l]}(e')\parallel\mu'\,\}} \tag{41.6c}$$

$$\frac{\nu\,\Sigma\,\{\,e\parallel\mu\,\}\ \mathsf{loops}}{\nu\,\Sigma\,\{\,\texttt{proj[l]}(e)\parallel\mu\,\}\ \mathsf{loops}} \tag{41.6d}$$

$$\frac{}{\left\{\begin{array}{c} \nu\,\Sigma,a_1:\tau_1,a_2:\tau_2\,\{\,\texttt{proj[l]}(\texttt{pair}(a_1;a_2))\parallel\mu\,\} \\ \mapsto \\ \nu\,\Sigma,a_1:\tau_1,a_2:\tau_2\,\{\,a_1\parallel\mu\,\} \end{array}\right\}} \tag{41.6e}$$

$$\frac{\nu\,\Sigma\,\{\,e\parallel\mu\,\}\mapsto\nu\,\Sigma'\,\{\,e'\parallel\mu'\,\}}{\nu\,\Sigma\,\{\,\texttt{proj[r]}(e)\parallel\mu\,\}\mapsto\nu\,\Sigma'\,\{\,\texttt{proj[r]}(e')\parallel\mu'\,\}} \tag{41.6f}$$

$$\frac{\nu\,\Sigma\,\{\,e\parallel\mu\,\}\ \mathsf{loops}}{\nu\,\Sigma\,\{\,\texttt{proj[r]}(e)\parallel\mu\,\}\ \mathsf{loops}} \tag{41.6g}$$

$$\frac{}{\left\{\begin{array}{c} \nu\,\Sigma,a_1:\tau_1,a_2:\tau_2\,\{\,\texttt{proj[r]}(\texttt{pair}(a_1;a_2))\parallel\mu\,\} \\ \mapsto \\ \nu\,\Sigma,a_1:\tau_1,a_2:\tau_2\,\{\,a_2\parallel\mu\,\} \end{array}\right\}} \tag{41.6h}$$

A pair is considered a value only if its arguments are names (Rule (41.6a)), which are introduced when the pair is created (Rule (41.6b)). The first and second projections evaluate to one or the other name in the pair, inducing a demand for the value of that component (Rules (41.6e) and (41.6h)).

Using similar techniques we may give a need dynamics to sums and recursive types. We leave the formalization of these as an exercise for the reader.

41.4 Suspensions

Another way to introduce laziness is to consolidate the machinery of the by-need dynamics into a single type whose values are possibly unevaluated, memoized computations. The type of *suspensions* of type τ, written $\tau\,\mathtt{susp}$, has as introductory form $\mathtt{susp}\,x : \tau\,\mathtt{is}\,e$ representing the suspended, possibly self-referential, computation, e, of type τ, and 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 other lazy types according to our needs in a particular program. For example, the type of lazy pairs with components of type τ_1 and τ_2 is expressible as the type

$$\tau_1\,\mathtt{susp} \times \tau_2\,\mathtt{susp}$$

and the type of call-by-need functions with domain τ_1 and range τ_2 is expressible as the type

$$\tau_1\,\mathtt{susp} \to \tau_2.$$

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:

$$\mu t\,.\,(\mathtt{unit} + (\mathtt{nat} \times t))\,\mathtt{susp}.$$

This type should be contrasted with the type

$$\mu t\,.\,(\mathtt{unit} + (\mathtt{nat} \times t\,\mathtt{susp}))$$

whose values are the empty list and a pair consisting of a natural number and a computation of another such value.

The syntax of suspensions is given by the following grammar:

Type	τ	$::=$	$\mathtt{susp}(\tau)$	$\tau\,\mathtt{susp}$	suspension
Expr	e	$::=$	$\mathtt{susp}[\tau](x.e)$	$\mathtt{susp}\,x : \tau\,\mathtt{is}\,e$	delay
			$\mathtt{force}(e)$	$\mathtt{force}(e)$	force
			$\mathtt{susp}[a]$	$\mathtt{susp}[a]$	self-reference

Suspensions are 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 the suspension type is given sing a judgement of the form $\Sigma\;\Gamma \vdash e : \tau$, where Σ assigns types to the names of suspensions. It is defined by the following rules:

$$\frac{\Sigma\;\Gamma, x : \mathtt{susp}(\tau) \vdash e : \tau}{\Sigma\;\Gamma \vdash \mathtt{susp}[\tau](x.e) : \mathtt{susp}(\tau)} \tag{41.7a}$$

$$\frac{\Sigma\;\Gamma \vdash e : \mathtt{susp}(\tau)}{\Sigma\;\Gamma \vdash \mathtt{force}(e) : \tau} \tag{41.7b}$$

$$\frac{}{\Sigma, a : \tau\;\Gamma \vdash \mathtt{susp}[a] : \mathtt{susp}(\tau)} \tag{41.7c}$$

Rule (41.7a) checks that the expression, e, has type τ under the assumption that x, which stands for the suspension itself, has type $\mathtt{susp}(\tau)$.

The by-need dynamics of suspensions is defined by the following rules:

$$\frac{}{\mathtt{susp}[a]\ \mathsf{val}_{\Sigma,a:\tau}} \tag{41.8a}$$

$$\frac{}{\left\{\begin{array}{c} \nu\,\Sigma\,\{\,\mathtt{susp}[\tau](x.e) \parallel \mu\,\} \\ \mapsto \\ \nu\,\Sigma, a : \tau\,\{\,\mathtt{susp}[a] \parallel \mu \otimes \langle a : [a/x]e\rangle\,\} \end{array}\right\}} \tag{41.8b}$$

$$\frac{\nu\,\Sigma\,\{\,e \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,e' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{force}(e) \parallel \mu\,\} \mapsto \nu\,\Sigma'\,\{\,\mathtt{force}(e') \parallel \mu'\,\}} \tag{41.8c}$$

$$\frac{e\ \mathsf{val}_{\Sigma,a:\tau}}{\left\{\begin{array}{c} \nu\,\Sigma, a : \tau\,\{\,\mathtt{force}(\mathtt{susp}[a]) \parallel \mu \otimes \langle a : e\rangle\,\} \\ \mapsto \\ \nu\,\Sigma, a : \tau\,\{\,e \parallel \mu \otimes \langle a : e\rangle\,\} \end{array}\right\}} \tag{41.8d}$$

$$\frac{\begin{array}{c} \nu\,\Sigma, a : \tau\,\{\,e \parallel \mu \otimes \langle a : \bullet\rangle\,\} \\ \mapsto \\ \nu\,\Sigma', a : \tau\,\{\,e' \parallel \mu' \otimes \langle a : \bullet\rangle\,\} \end{array}}{\left\{\begin{array}{c} \nu\,\Sigma, a : \tau\,\{\,\mathtt{force}(\mathtt{susp}[a]) \parallel \mu \otimes \langle a : e\rangle\,\} \\ \mapsto \\ \nu\,\Sigma', a : \tau\,\{\,\mathtt{force}(\mathtt{susp}[a]) \parallel \mu' \otimes \langle a : e'\rangle\,\} \end{array}\right\}} \tag{41.8e}$$

Rule (41.8a) specifies that a reference to a suspension is a value. Rule (41.8b) specifies that evaluation of a delayed computation consists of allocating a fresh name for it in the memo table, and returning a reference to that suspension. Rules (41.8c) to (41.8e) 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 result.

41.5 Exercises

Chapter 42

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, if you are a pessimist, *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. Although the latter choice seems rather arbitrary, it is no less so than the choice between a fully lazy or a fully eager dynamics.

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 whose arguments 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, one could very sensibly decree that products, sums, and recursive types are lazy, yet impose a call-by-value discipline on functions. Or one 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 language has its adherents, each has its drawbacks, and each has its advantages.

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 is 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.

In this chapter we will introduce *polarization* to distinguish types based on whether their elements are defined by their *values* (the *positive* types) or by their *behavior* (the *negative* types). Put in other terms, positive types are "eager" (determined by their values), whereas negative types are "lazy" (determined by their behavior). Since positive types are defined by their values, they are eliminated by pattern matching against these values. Similarly, since negative types are defined by their behavior under a range of experiments, they are eliminated by performing an experiment on them.

To make these symmetries explicit we formalize polarization using a technique called *focusing*, or *focalization*.[1] A focused presentation of a programming language distinguishes three general forms of expression, *(positive and negative) values*, *(positive and negative) continuations*, and *(neutral) computations*. Besides exposing the symmetries in a polarized type system, focusing also clarifies the design of the control machine introduced in Chapter 31. In a focused framework stacks are just continuations, and states are just computations; there is no need for any *ad hoc* apparatus to explain the flow of control in a program.

42.1 Polarization

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

[1] More precisely, we employ a weak form of focusing, rather than the stricter forms considered elsewhere in the literature.

> 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, since it is defined by zero and successor, whereas function types would be negative, since they are characterized by their behavior when applied, and not by their internal structure.

The language $\mathcal{L}^{\pm}\{\mathtt{nat}\rightharpoonup\}$ is a polarized formulation of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$ in which the syntax of types is given by the following grammar:

$$\begin{array}{llllll}
\mathsf{PType} & \tau^{+} & ::= & \mathtt{dn}(\tau^{-}) & \downarrow \tau^{-} & \text{suspension} \\
 & & & \mathtt{nat} & \mathtt{nat} & \text{naturals} \\
\mathsf{NType} & \tau^{-} & ::= & \mathtt{up}(\tau^{+}) & \uparrow \tau^{+} & \text{inclusion} \\
 & & & \mathtt{parr}(\tau_1^{+};\tau_2^{-}) & \tau_1^{+} \rightharpoonup \tau_2^{-} & \text{partial function}
\end{array}$$

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, $\tau_1^{+} \rightharpoonup \uparrow \tau_2^{+}$.

42.2 Focusing

The syntax of $\mathcal{L}^{\pm}\{\mathtt{nat}\rightharpoonup\}$ 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
			$\mathtt{del}^{\mathtt{-}}(e)$	$\mathtt{del}^{\mathtt{-}}(e)$	delay
PCont	k^+	$::=$	$\mathtt{ifz}(e_0;x.e_1)$	$\mathtt{ifz}(e_0;x.e_1)$	conditional
			$\mathtt{force}^{\mathtt{-}}(k^-)$	$\mathtt{force}^{\mathtt{-}}(k^-)$	evaluate
NVal	v^-	$::=$	$\mathtt{lam}[\tau^+](x.e)$	$\lambda\,(x{:}\tau^+.e)$	abstraction
			$\mathtt{del}^{\mathtt{+}}(v^+)$	$\mathtt{del}^{\mathtt{+}}(v^+)$	inclusion
			$\mathtt{fix}(x.v^-)$	$\mathtt{fix}\,x\,\mathtt{is}\,v^-$	recursion
NCont	k^-	$::=$	$\mathtt{ap}(v^+;k^-)$	$\mathtt{ap}(v^+;k^-)$	application
			$\mathtt{force}^{\mathtt{+}}(x.e)$	$\mathtt{force}^{\mathtt{+}}(x.e)$	evaluate
Comp	e	$::=$	$\mathtt{ret}(v^-)$	$\mathtt{ret}(v^-)$	return
			$\mathtt{cut}^{\mathtt{+}}(v^+;k^+)$	$v^+ \triangleright k^+$	cut
			$\mathtt{cut}^{\mathtt{-}}(v^-;k^-)$	$v^- \triangleright k^-$	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 $\mathtt{del}^{\mathtt{-}}(e)$, and we may consider a positive value to be a negative value using $\mathtt{del}^{\mathtt{+}}(v^+)$. 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.

42.3 Statics

The statics of $\mathcal{L}^{\pm}\{\mathtt{nat}\rightharpoonup\}$ consists of a collection of rules for deriving judgements 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: $\Gamma \vdash e : \gamma^{-}$.

Throughout Γ is a finite set of hypotheses of the form

$$x_1 : \tau_1^{+}, \dots, x_n : \tau_n^{+},$$

for some $n \geq 0$, and γ^{-} 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 judgements 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:

$$\frac{}{\Gamma, x : \tau^{+} \vdash x : \tau^{+}} \tag{42.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{42.1b}$$

$$\frac{\Gamma \vdash v^{+} : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(v^{+}) : \mathtt{nat}} \tag{42.1c}$$

$$\frac{\Gamma \vdash e : \tau^{-}}{\Gamma \vdash \mathtt{del}^{-}(e) : \downarrow\tau^{-}} \tag{42.1d}$$

Rule (42.1a) specifies that variables range over positive values. Rules (42.1b) and (42.1c) specify that the values of type nat are just the numerals. Rule (42.1d) specifies that a suspended computation (necessarily of negative type) is a positive value.

The statics of positive continuations consists of the following rules:

$$\frac{\Gamma \vdash e_0 : \gamma^{-} \quad \Gamma, x : \mathtt{nat} \vdash e_1 : \gamma^{-}}{\Gamma \vdash \mathtt{ifz}(e_0; x.e_1) : \mathtt{nat} > \gamma^{-}} \tag{42.2a}$$

$$\frac{\Gamma \vdash k^{-} : \tau^{-} > \gamma^{-}}{\Gamma \vdash \mathtt{force}^{-}(k^{-}) : \downarrow\tau^{-} > \gamma^{-}} \tag{42.2b}$$

Rule (42.2a) governs the continuation that chooses between two computations according to whether a natural number is zero or non-zero. Rule (42.2b) specifies the continuation that forces a delayed computation with the specified negative continuation.

The statics of negative values is defined by these rules:

$$\frac{\Gamma, x : \tau_1^+ \vdash e : \tau_2^-}{\Gamma \vdash \lambda\,(x{:}\tau_1^+.\,e) : \tau_1^+ \rightharpoonup \tau_2^-} \tag{42.3a}$$

$$\frac{\Gamma \vdash v^+ : \tau^+}{\Gamma \vdash \mathtt{del}^{+}(v^+) : \uparrow \tau^+} \tag{42.3b}$$

$$\frac{\Gamma, x : \downarrow \tau^- \vdash v^- : \tau^-}{\Gamma \vdash \mathtt{fix}\, x\, \mathtt{is}\, v^- : \tau^-} \tag{42.3c}$$

Rule (42.3a) specifies the statics of a λ-abstraction whose argument is a positive value, and whose result is a computation of negative type. Rule (42.3b) specifies the inclusion of positive values as negative values. Rule (42.3c) specifies that negative types admit general recursion.

The statics of negative continuations is defined by these rules:

$$\frac{\Gamma \vdash v_1^+ : \tau_1^+ \quad \Gamma \vdash k_2^- : \tau_2^- > \gamma^-}{\Gamma \vdash \mathtt{ap}(v_1^+; k_2^-) : \tau_1^+ \rightharpoonup \tau_2^- > \gamma^-} \tag{42.4a}$$

$$\frac{\Gamma, x : \tau^+ \vdash e : \gamma^-}{\Gamma \vdash \mathtt{force}^{+}(x.e) : \uparrow \tau^+ > \gamma^-} \tag{42.4b}$$

Rule (42.4a) 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 (42.4b) 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:

$$\frac{\Gamma \vdash v^- : \tau^-}{\Gamma \vdash \mathtt{ret}(v^-) : \tau^-} \tag{42.5a}$$

$$\frac{\Gamma \vdash v^+ : \tau^+ \quad \Gamma \vdash k^+ : \tau^+ > \gamma^-}{\Gamma \vdash v^+ \triangleright k^+ : \gamma^-} \tag{42.5b}$$

$$\frac{\Gamma \vdash v^- : \tau^- \quad \Gamma \vdash k^- : \tau^- > \gamma^-}{\Gamma \vdash v^- \triangleright k^- : \gamma^-} \tag{42.5c}$$

Rule (42.5a) specifies the basic form of computation that simply returns the negative value v^-. Rules (42.5b) and (42.5c) specify computations that pass a value to a contination of appropriate polarity.

42.4 Dynamics

The dynamics of $\mathcal{L}^{\pm}\{\texttt{nat}\rightharpoonup\}$ is given by a transition system $e \mapsto e'$ 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:

$$\frac{}{\texttt{z} \triangleright \texttt{ifz}(e_0;x.e_1) \mapsto e_0} \tag{42.6a}$$

$$\frac{}{\texttt{s}(v^+) \triangleright \texttt{ifz}(e_0;x.e_1) \mapsto [v^+/x]e_1} \tag{42.6b}$$

$$\frac{}{\texttt{del}^{\texttt{-}}(e) \triangleright \texttt{force}^{\texttt{-}}(k^-) \mapsto e\,;k^-} \tag{42.6c}$$

$$\frac{}{\lambda\,(x{:}\tau^+.e) \triangleright \texttt{ap}(v^+;k^-) \mapsto [v^+/x]e\,;k^-} \tag{42.6d}$$

$$\frac{}{\texttt{del}^{\texttt{+}}(v^+) \triangleright \texttt{force}^{\texttt{+}}(x.e) \mapsto [v^+/x]e} \tag{42.6e}$$

$$\frac{}{\texttt{fix}\,x\,\texttt{is}\,v^- \triangleright k^- \mapsto [\texttt{del}^{\texttt{-}}(\texttt{fix}\,x\,\texttt{is}\,v^-)/x]v^- \triangleright k^-} \tag{42.6f}$$

These rules specify the interaction between values and continuations.

Rules (42.6) make use of two forms of substitution, $[v^+/x]e$ and $[v^+/x]v^-$, which are defined as in Chapter 3. 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).

$$\frac{}{\texttt{ret}(v^-)\,;k_0^- = v^- \triangleright k_0^-} \tag{42.7a}$$

$$\frac{k^-\,;k_0^- = k_1^-}{(v^- \triangleright k^-)\,;k_0^- = v^- \triangleright k_1^-} \tag{42.7b}$$

$$\frac{k^+\,;k_0^- = k_1^+}{(v^+ \triangleright k^+)\,;k_0^- = v^+ \triangleright k_1^+} \tag{42.7c}$$

$$\frac{e_0\,;k^- = e_0' \quad x \mid e_1\,;k^- = e_1'}{\texttt{ifz}(e_0;x.e_1)\,;k^- = \texttt{ifz}(e_0';x.e_1')} \tag{42.7d}$$

$$\frac{k^-\,;k_0^- = k_1^-}{\texttt{force}^{\texttt{-}}(k^-)\,;k_0^- = \texttt{force}^{\texttt{-}}(k_1^-)} \tag{42.7e}$$

$$\frac{k^-\,;k_0^- = k_1}{\texttt{ap}(v^+;k^-)\,;k_0^- = \texttt{ap}(v^+;k_1^-)} \tag{42.7f}$$

$$\frac{x \mid e \,;k_0^- = e'}{\texttt{force}^{+}(x\,.\,e)\,;k_0^- = \texttt{force}^{+}(x\,.\,e')} \qquad (42.7g)$$

Rules (42.7d) and (42.7g) make use of the generic hypothetical judgement defined in Chapter 4 to express that the composition is defined uniformly in the bound variable.

42.5 Safety

The proof of preservation for $\mathcal{L}^{\pm}\{\texttt{nat}\rightharpoonup\}$ reduces to the proof of the typing properties of substitution and composition.

Lemma 42.1 (Substitution)**.** *Suppose that $\Gamma \vdash v^+ : \sigma^+$.*

1. *If $\Gamma, x : \sigma^+ \vdash e : \gamma^-$, then $\Gamma \vdash [v^+/x]e : \gamma^-$.*
2. *If $\Gamma, x : \sigma^+ \vdash v^- : \tau^-$, then $\Gamma \vdash [v^+/x]v^- : \tau^-$.*
3. *If $\Gamma, x : \sigma^+ \vdash k^+ : \tau^+ > \gamma^-$, then $\Gamma \vdash [v^+/x]k^+ : \tau^+ > \gamma^-$.*
4. *If $\Gamma, x : \sigma^+ \vdash v_1^+ : \tau^+$, then $\Gamma \vdash [v^+/x]v_1^+ : \tau^+$.*
5. *If $\Gamma, x : \sigma^+ \vdash k^- : \tau^- > \gamma^-$, 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 42.2 (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 42.3 (Preservation)**.** *If $\Gamma \vdash e : \gamma^-$ and $e \mapsto e'$, then $\Gamma \vdash e' : \gamma^-$.*

Proof. By induction on transition, appealing to inversion for typing and Lemmas 42.1 and 42.2. □

The progress theorem reduces to the characterization of the values of each type. Focusing makes the required properties evident, since it defines directly the values of each type.

Theorem 42.4 (Progress)**.** *If $\Gamma \vdash e : \gamma^-$, then either $e = \texttt{ret}(v^-)$ for some v^-, or there exists e' such that $e \mapsto e'$.*

42.6 Definability

The syntax of $\mathcal{L}^{\pm}\{\texttt{nat}\rightharpoonup\}$ exposes the symmetries between positive and negative types, and hence between eager and lazy computation. It is not, however, especially convenient for writing programs because it requires that each computation in a program be expressed in the stilted form of a value juxtaposed with a continuation. It would be useful to have a more natural syntax that is translatable into the present language.

But the question of what is a natural syntax begs the very question that motivated the language in the first place!

Editorial Notes

This chapter under construction.

42.7 Exercises

Part XVI

Parallelism

Chapter 43

Nested Parallelism

Parallel computation seeks to reduce the running times of programs by allowing many computations to be carried out simultaneously. For example, if one wishes 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 10) 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.

43.1 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 $\texttt{letpar}\,x_1 = e_1\,\texttt{and}\,x_2 = e_2\,\texttt{in}\,e$, 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. For example, a *parallel pair* construct might be defined as the expression

$$\texttt{letpar}\,x_1 = e_1\,\texttt{and}\,x_2 = e_2\,\texttt{in}\,\langle x_1, x_2\rangle,$$

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

$$\texttt{letpar}(e_1;e_2;x_1\,.\,x_2\,.\,e),$$

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:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{letpar}(e_1;e_2;x_1\,.\,x_2\,.\,e) : \tau} \tag{43.1}$$

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. One 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 43.3 on page 400.)

We will give both a *sequential* and a *parallel* dynamics of the parallel `let` construct. The definition of the sequential dynamics as a transition judgement of the form $e \mapsto_{\mathsf{seq}} e'$ is entirely straightforward:

$$\frac{e_1 \mapsto e_1'}{\texttt{letpar}(e_1;e_2;x_1\,.\,x_2\,.\,e) \mapsto_{\mathsf{seq}} \texttt{letpar}(e_1';e_2;x_1\,.\,x_2\,.\,e)} \tag{43.2a}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{letpar}(e_1;e_2;x_1\,.\,x_2\,.\,e) \mapsto_{\mathsf{seq}} \texttt{letpar}(e_1;e_2';x_1\,.\,x_2\,.\,e)} \tag{43.2b}$$

$$\frac{e_1\ \mathsf{val}\quad e_2\ \mathsf{val}}{\mathtt{letpar}(e_1;e_2;x_1.x_2.e) \mapsto_{\mathsf{seq}} [e_1,e_2/x_1,x_2]e} \tag{43.2c}$$

The parallel dynamics is given by a transition judgement of the form $e \mapsto_{\mathsf{par}} e'$, defined as follows:

$$\frac{e_1 \mapsto_{\mathsf{par}} e_1' \quad e_2 \mapsto_{\mathsf{par}} e_2'}{\mathtt{letpar}(e_1;e_2;x_1.x_2.e) \mapsto_{\mathsf{par}} \mathtt{letpar}(e_1';e_2';x_1.x_2.e)} \tag{43.3a}$$

$$\frac{e_1 \mapsto_{\mathsf{par}} e_1' \quad e_2\ \mathsf{val}}{\mathtt{letpar}(e_1;e_2;x_1.x_2.e) \mapsto_{\mathsf{par}} \mathtt{letpar}(e_1';e_2;x_1.x_2.e)} \tag{43.3b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto_{\mathsf{par}} e_2'}{\mathtt{letpar}(e_1;e_2;x_1.x_2.e) \mapsto_{\mathsf{par}} \mathtt{letpar}(e_1;e_2';x_1.x_2.e)} \tag{43.3c}$$

$$\frac{e_1\ \mathsf{val}\quad e_2\ \mathsf{val}}{\mathtt{letpar}(e_1;e_2;x_1.x_2.e) \mapsto_{\mathsf{par}} [e_1,e_2/x_1,x_2]e} \tag{43.3d}$$

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 one need never be concerned with the *semantics* of a parallel program (its meaning is determined by the sequential dynamics), but only with its *performance*. Since the sequential dynamics is deterministic (every expression has at most one value), the implicit parallelism theorem implies that the parallel dynamics is also deterministic. This clearly distinguishes *parallelism*, which is deterministic, from *concurrency*, which is non-deterministic (see Chapters 45 and 46 for more on concurrency).

A proof of the implicit parallelism theorem may be given by giving an evaluation dynamics, $e \Downarrow v$, in the style of Chapter 10, 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 $v\ \mathsf{val}$). The crucial rule of the evaluation dynamics is the one governing the parallel `let` construct:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad [v_1,v_2/x_1,x_2]e \Downarrow v}{\mathtt{letpar}(e_1;e_2;x_1.x_2.e) \Downarrow v} \tag{43.4}$$

It is easy to show that the sequential dynamics agrees with the evaluation dynamics by a straightforward extension of the proof of Theorem 10.2 on page 85.

Lemma 43.1. *$e \mapsto^*_{seq} v$ iff $e \Downarrow v$.*

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

$$\texttt{letpar}\, x_1 = e_1\, \texttt{and}\, x_2 = e_2\, \texttt{in}\, e \mapsto^*_{\mathsf{seq}} v.$$

We leave the details of the proof as an exercise for the reader. □

By a similar argument we may show that the parallel dynamics also agrees with the evaluation dynamics, and hence with the sequential dynamics.

Lemma 43.2. *$e \mapsto^*_{par} v$ iff $e \Downarrow v$.*

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

$$\texttt{letpar}\, x_1 = e_1\, \texttt{and}\, x_2 = e_2\, \texttt{in}\, e \mapsto^*_{\mathsf{par}} v.$$

The proof of the first is by a straightforward 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 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 $\texttt{letpar}\, x_1 = e_1'\, \texttt{and}\, x_2 = v_2\, \texttt{in}\, e \mapsto^*_{\mathsf{par}} v$, and hence the result follows by an application of Rule (43.3b). The symmetric case follows similarly by an application of Rule (43.3c), and in case both e_1 and e_2 take a step, the result follows by induction and Rule (43.3a). □

Theorem 43.3 (Implicit Parallelism)**.** *The sequential and parallel dynamics coincide: for all v val, $e \mapsto^*_{seq} v$ iff $e \mapsto^*_{par} v$.*

Proof. By Lemmas 43.1 and 43.2. □

Theorem 43.3 states that parallelism is *implicit* in that the use of a parallel evaluation strategy does not affect the *semantics* of a program, but only its *efficiency*. The program means the same thing under a parallel execution strategy as it does under a sequential one. Correctness concerns are factored out, focusing attention on time (and space) complexity of a parallel execution strategy.

43.2 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:

$$\begin{array}{llll}
\textsf{Cost} & c & ::= & \mathbf{0} & \text{zero cost} \\
 & & & \mathbf{1} & \text{unit cost} \\
 & & & c_1 \otimes c_2 & \text{parallel combination} \\
 & & & c_1 \oplus c_2 & \text{sequential combination}
\end{array}$$

A cost graph is a form of *series-parallel* directed acyclic graph, with a designated *source* node and *sink* node. For **0** the graph consists of one node and no edges, with the source and sink both being the node itself. For **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*, $dp(c)$. The work is defined by the following equations:

$$wk(c) = \begin{cases} 0 & \text{if } c = \mathbf{0} \\ 1 & \text{if } c = \mathbf{1} \\ wk(c_1) + wk(c_2) & \text{if } c = c_1 \otimes c_2 \\ wk(c_1) + wk(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases} \tag{43.5}$$

The depth is defined by the following equations:

$$dp(c) = \begin{cases} 0 & \text{if } c = \mathbf{0} \\ 1 & \text{if } c = \mathbf{1} \\ \max(dp(c_1), dp(c_2)) & \text{if } c = c_1 \otimes c_2 \\ dp(c_1) + dp(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases} \tag{43.6}$$

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 10 we introduced *cost dynamics* as a means of assigning time complexity to evaluation. The proof of Theorem 10.7 on page 88 shows that $e \Downarrow^k v$ iff $e \mapsto^k v$. That is, the step complexity of an evaluation of e to a value v is just the number of transitions required to derive $e \mapsto^* v$. Here we use cost graphs as the measure of complexity, then relate these cost graphs to the structural dynamics given in Section 43.1 on page 394.

The judgement $e \Downarrow^c v$, 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:

$$\frac{e_1 \Downarrow^{c_1} v_1 \quad e_2 \Downarrow^{c_2} v_2 \quad [v_1, v_2 / x_1, x_2]e \Downarrow^{c} v}{\texttt{letpar}(e_1; e_2; x_1 . x_2 . e) \Downarrow^{(c_1 \otimes c_2) \oplus \mathbf{1} \oplus c} v} \tag{43.7}$$

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 43.4. *$e \Downarrow v$ iff $e \Downarrow^c v$ for some c.*

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 43.5. *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 43.6. *If $e \Downarrow^{c} v$, then $e \mapsto^{w}_{\mathsf{seq}} v$ and $e \mapsto^{d}_{\mathsf{par}} v$, where $w = wk(c)$ and $d = dp(c)$. Conversely, if $e \mapsto^{w}_{\mathsf{seq}} v$, then there exists c such that $e \Downarrow^{c} v$ with $wk(c) = w$, and if $e \mapsto^{d}_{\mathsf{par}} v'$, then there exists c' such that $e \Downarrow^{c'} v'$ with $dp(c') = d$. Therefore if $e \mapsto^{w}_{\mathsf{seq}} v$ and $e \mapsto^{d}_{\mathsf{par}} v'$, then v is v' and $e \Downarrow^{c} v$ for some c such that $wk(c) = w$ and $dp(c) = d$.*

Proof. The first part is proved by induction on the derivation of $e \Downarrow^{c} v$, the interesting case being Rule (43.7). 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 $w_1 = wk(c_1)$, $w_2 = wk(c_2)$, and $w = wk(c)$. By pasting together derivations we obtain a derivation

$$\begin{aligned}
\mathtt{letpar}(e_1;e_2;x_1.x_2.e) &\mapsto^{w_1}_{\mathsf{seq}} \mathtt{letpar}(v_1;e_2;x_1.x_2.e) \\
&\mapsto^{w_2}_{\mathsf{seq}} \mathtt{letpar}(v_1;v_2;x_1.x_2.e) \\
&\mapsto_{\mathsf{seq}} [v_1, v_2/x_1, x_2]e \\
&\mapsto^{w}_{\mathsf{seq}} v.
\end{aligned}$$

Noting that $wk((c_1 \otimes c_2) \oplus \mathbf{1} \oplus c) = w_1 + w_2 + 1 + w$ 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 $e \mapsto^{d}_{\mathsf{par}} v$, where $d_1 = dp(c_1)$, $d_2 = dp(c_2)$, and $d = dp(c)$. 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:

$$\begin{aligned}
\mathtt{letpar}(e_1;e_2;x_1.x_2.e) &\mapsto^{d_1}_{\mathsf{par}} \mathtt{letpar}(v_1;e_2';x_1.x_2.e) \\
&\mapsto^{d_2-d_1}_{\mathsf{par}} \mathtt{letpar}(v_1;v_2;x_1.x_2.e) \\
&\mapsto_{\mathsf{par}} [v_1, v_2/x_1, x_2]e \\
&\mapsto^{d}_{\mathsf{par}} v.
\end{aligned}$$

Calculating $dp((c_1 \otimes c_2) \oplus \mathbf{1} \oplus c) = \max(d_1, d_2) + 1 + d$ 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 $wk(c) = wk(c') + 1$, and if $e \mapsto_{\mathsf{par}} e'$ with $e' \Downarrow^{c'} v$, then $e \Downarrow^{c} v$ with $dp(c) = dp(c') + 1$.

Suppose that $e = \mathtt{letpar}(e_1;e_2;x_1\,.\,x_2\,.\,e_0)$ 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 $wk(c) = wk(c') + 1$, as required. Similarly, $e \mapsto_{\mathsf{par}} e'$ for e' as above, and hence $e \Downarrow^{c} v$ for some c such that $dp(c) = dp(c') + 1$, as required.

Suppose that $e = \mathtt{letpar}(e_1;e_2;x_1\,.\,x_2\,.\,e_0)$ and $e \mapsto_{\mathsf{seq}} e'$, where $e' = \mathtt{letpar}(e_1';e_2;x_1\,.\,x_2\,.\,e_0)$ 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 $e = \mathtt{letpar}(e_1;e_2;x_1\,.\,x_2\,.\,e_0)$ and $e \mapsto_{\mathsf{par}} e'$, where $e' = \mathtt{letpar}(e_1';e_2';x_1\,.\,x_2\,.\,e_0)$ 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. □

43.3 Multiple Fork-Join

So far we have confined attention to binary fork/join parallelism induced by the parallel `let` construct. While technically sufficient for many purposes, a more natural programming model admit 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, one 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.

Type	τ	$::=$	$\texttt{seq}(\tau)$	$\tau\,\texttt{seq}$	sequence
Expr	e	$::=$	$\texttt{seq}(e_0,\ldots,e_{n-1})$	$[e_0,\ldots,e_{n-1}]$	sequence
			$\texttt{len}(e)$	$\lvert e\rvert$	size
			$\texttt{sub}(e_1;e_2)$	$e_1[e_2]$	element
			$\texttt{tab}(x.e_1;e_2)$	$\texttt{tab}(x.e_1;e_2)$	tabulate
			$\texttt{map}(x.e_1;e_2)$	$[e_1 \mid x \in e_2]$	map
			$\texttt{cat}(e_1;e_2)$	$\texttt{cat}(e_1;e_2)$	concatenate

The expression $\texttt{seq}(e_0,\ldots,e_{n-1})$ evaluates to an n-sequence whose elements are given by the expressions $e_0,\ldots,e_{n-1}$. The operation $\texttt{len}(e)$ returns the number of elements in the sequence given by e. The operation $\texttt{sub}(e_1;e_2)$ retrieves the element of the sequence given by e_1 at the index given by e_2. The operation $\texttt{tab}(x.e_1;e_2)$ creates the sequence whose ith element is the value of e_1 with x bound to i. The operation $\texttt{map}(x.e_1;e_2)$ computes the sequence whose ith element is the result of evaluating e_1 with x bound to the ith 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:

$$\frac{\Gamma \vdash e_0 : \tau \quad \ldots \quad \Gamma \vdash e_{n-1} : \tau}{\Gamma \vdash \texttt{seq}(e_0,\ldots,e_{n-1}) : \texttt{seq}(\tau)} \tag{43.8a}$$

$$\frac{\Gamma \vdash e : \texttt{seq}(\tau)}{\Gamma \vdash \texttt{len}(e) : \texttt{nat}} \tag{43.8b}$$

$$\frac{\Gamma \vdash e_1 : \texttt{seq}(\tau) \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{sub}(e_1;e_2) : \tau} \tag{43.8c}$$

$$\frac{\Gamma, x : \texttt{nat} \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{tab}(x.e_1;e_2) : \texttt{seq}(\tau)} \tag{43.8d}$$

$$\frac{\Gamma \vdash e_2 : \texttt{seq}(\tau) \quad \Gamma, x : \tau \vdash e_1 : \tau'}{\Gamma \vdash \texttt{map}(x.e_1;e_2) : \texttt{seq}(\tau')} \tag{43.8e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{seq}(\tau) \quad \Gamma \vdash e_2 : \texttt{seq}(\tau)}{\Gamma \vdash \texttt{cat}(e_1;e_2) : \texttt{seq}(\tau)} \tag{43.8f}$$

The cost dynamics of these constructs is defined by the following rules:

$$\frac{e_0 \Downarrow^{c_0} v_0 \quad \dots \quad e_{n-1} \Downarrow^{c_{n-1}} v_{n-1}}{\mathtt{seq}(e_0,\dots,e_{n-1}) \Downarrow^{\bigotimes_{i=0}^{n-1} c_i} \mathtt{seq}(v_0,\dots,v_{n-1})} \tag{43.9a}$$

$$\frac{e \Downarrow^{c} \mathtt{seq}(v_0,\dots,v_{n-1})}{\mathtt{len}(e) \Downarrow^{c\oplus\mathbf{1}} \mathtt{num}[n]} \tag{43.9b}$$

$$\frac{e_1 \Downarrow^{c_1} \mathtt{seq}(v_0,\dots,v_{n-1}) \quad e_2 \Downarrow^{c_2} \mathtt{num}[i] \quad (0 \leq i < n)}{\mathtt{sub}(e_1;e_2) \Downarrow^{c_1\oplus c_2\oplus\mathbf{1}} v_i} \tag{43.9c}$$

$$\frac{e_2 \Downarrow^{c} \mathtt{num}[n] \quad [\mathtt{num}[0]/x]e_1 \Downarrow^{c_0} v_0 \quad \dots \quad [\mathtt{num}[n-1]/x]e_1 \Downarrow^{c_{n-1}} v_{n-1}}{\mathtt{tab}(x.e_1;e_2) \Downarrow^{c\oplus\bigotimes_{i=0}^{n-1} c_i} \mathtt{seq}(v_0,\dots,v_{n-1})} \tag{43.9d}$$

$$\frac{\begin{array}{c} e_2 \Downarrow^{c} \mathtt{seq}(v_0,\dots,v_{n-1}) \\ [v_0/x]e_1 \Downarrow^{c_0} v_0' \quad \dots \quad [v_{n-1}/x]e_1 \Downarrow^{c_{n-1}} v_{n-1}' \end{array}}{\mathtt{map}(x.e_1;e_2) \Downarrow^{c\oplus\bigotimes_{i=0}^{n-1} c_i} \mathtt{seq}(v_0',\dots,v_{n-1}')} \tag{43.9e}$$

$$\frac{e_1 \Downarrow^{c_1} \mathtt{seq}(v_0,\dots,v_{m-1}) \quad e_2 \Downarrow^{c_2} \mathtt{seq}(v_0',\dots,v_{n-1}')}{\mathtt{cat}(e_1;e_2) \Downarrow^{c_1\oplus c_2\oplus\bigotimes_{i=0}^{m+n-1}\mathbf{1}} \mathtt{seq}(v_0,\dots,v_{m-1},v_0',\dots,v_{n-1}')} \tag{43.9f}$$

The cost dynamics for sequence operations may be validated by introducing a sequential and parallel cost dynamics and extending the proof of Theorem 43.6 on page 399 to cover this extension.

43.4 Provably Efficient Implementations

Theorem 43.6 on page 399 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. Since 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 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 43.7. *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 d is proportional to w, then the

[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.

overall time is again $O(w)$, which is to say that we are unable to exploit parallelism in that case.

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 $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ as described in Chapter 13. The dynamics consists of two transition relations defined on states of the form

$$\nu\, x_1 : \tau_1, \ldots x_n : \tau_n \,\{\, \langle x_1 : e_1 \rangle \otimes \ldots \otimes \langle x_n : e_n \rangle \,\}.$$

Such a state represents the remaining work of a computation, decomposed into n tasks, with each task binding its value to a variable. Importantly, we do not distinguish states that differ in the order of the variable declarations or variable bindings.

The occurrences of variables in a state determine the *dependency ordering* among the tasks: if x_i occurs free in x_j, then e_j cannot be evaluated before evaluation of e_i is complete. Such dependencies reflect data flow dependencies among the tasks, and are therefore manifestations of the depth complexity of the program. A *closed* expression e_i in a state is said to be *ready* in that state; otherwise, e_i is said to be *blocked*, awaiting completion of evaluation of the expression on which it depends.

We will consider two forms of state transition, the *local* and the *global*. Local transitions represent the steps of computation of the individual processors, which we will model using the dynamics of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ given in Chapter 13 as a guide. Global transitions represent the scheduling and load-balancing steps that allocate tasks to processors with the intent of maximizing parallelism insofar as possible consistently with the dependency ordering among the tasks.

Local transitions apply only to ready expressions (those with no free variables). The following two rules are illustrative examples of local transitions:

$$\frac{}{\left\{\begin{array}{c}\nu\, x : \tau \,\{\, \langle x : \mathtt{letpar}(e_1;e_2;x_1.x_2.e)\rangle \,\}\\ \mapsto_{loc} \\ \nu\, x : \tau, x_1 : \tau_1, x_2 : \tau_2 \,\{\, \langle x_1 : e_1\rangle \otimes \langle x_2 : e_2\rangle \otimes \langle x : e\rangle \,\}\end{array}\right\}} \tag{43.10a}$$

$$\frac{\begin{array}{c}v_0 \;\mathsf{val} \quad \dots \quad v_{n-1} \;\mathsf{val}\\ \Gamma = y_0 : \tau, \dots, y_{n-1} : \tau \qquad \gamma = \langle y_0 : [v_0/y]e_1\rangle \otimes \dots \otimes \langle y_{n-1} : [v_{n-1}/y]e_1\rangle\end{array}}{\left\{\begin{array}{c}\nu\, x : \mathtt{seq}(\tau) \,\{\, \langle x : \mathtt{map}(y.e_1;\mathtt{seq}(v_0,\dots,v_{n-1}))\rangle \,\}\\ \mapsto_{loc} \\ \nu\, x : \mathtt{seq}(\tau)\, \Gamma \,\{\, \langle x : \mathtt{seq}(y_0,\dots,y_{n-1})\rangle \otimes \gamma \,\}\end{array}\right\}} \tag{43.10b}$$

Rule (43.10a) states that if $\mathtt{letpar}\; x_1 = e_1 \;\mathtt{and}\; x_2 = e_2 \;\mathtt{in}\; e$ is ready, then executing it consists of creating two new, independent tasks, one to evaluate e_1 and one to evaluate e_2, and to update the current task, which is in general dependent on the other two, to represent the join point of the parallel binding. Observe that both e_1 and e_2 are ready in the resulting state, whereas e is, in general, not ready.

Rule (43.10b) states that a map operation on a sequence of length n is to be executed by creating n new tasks, with the ith task devoted to evaluating the substitution $[v_i/y]e_1$ of the ith sequence element for y in the expression e_1. When all n tasks complete, their results are joined to form a new sequence consisting of the results of each task in the same order. The tabulate operation $\mathtt{tab}(x.e_1;e_2)$ is executed similarly, except that the ith task evaluates $[\mathtt{num}[i]/x]e_1$ for each $0 \leq i < n$, where n is determined by evaluating e_2.

Global transitions are parameterized by $p \geq 0$, representing the number of processors available for simultaneous execution. Each transition consists of selecting $n \leq p$ ready tasks from the state, applying a local transition to each, then reconstructing the state with the task(s) resulting from the local step.

$$\frac{\begin{array}{c} e_1\ \mathsf{val} \quad \dots \quad e_n\ \mathsf{val} \\ \Gamma = \Gamma_0, x_1 : \tau_1, \dots x_n : \tau_n \quad (\exists 1 \leq n \leq p) \end{array}}{\left\{ \begin{array}{c} \nu\,\Gamma\,\{\, \langle x_1 : e_1 \rangle \otimes \dots \otimes \langle x_n : e_n \rangle \otimes \gamma \,\} \\ \mapsto_{glo} \\ \nu\,\Gamma_0\,\{\, [e_1, \dots, e_n / x_1, \dots, x_n]\gamma \,\} \end{array} \right\}} \tag{43.11a}$$

$$\frac{\begin{array}{c} \Gamma = \Gamma_0, x_1 : \tau_1, \dots, x_n : \tau_n \quad (\exists 1 \leq n \leq p) \\ (\forall 1 \leq i \leq n)\ \nu\, x_i : \tau_i\,\{\, \langle x_i : e_i \rangle \,\} \mapsto_{loc} \nu\, x_i : \tau_i\,\Gamma_i\,\{\, \gamma_i \,\} \end{array}}{\left\{ \begin{array}{c} \nu\,\Gamma\,\{\, \langle x_1 : e_1 \rangle \otimes \dots \otimes \langle x_n : e_n \rangle \otimes \gamma \,\} \\ \mapsto_{glo} \\ \nu\,\Gamma\,\Gamma_1\, \dots\, \Gamma_n\,\{\, \gamma_1 \otimes \dots \otimes \gamma_n \otimes \gamma \,\} \end{array} \right\}} \tag{43.11b}$$

Rule (43.11a) states that if any $n \leq p$ tasks are complete, their values may be propagated to the other tasks by substitution. This has the effect of eliminating dependencies on the substituted variables, enabling some tasks for evaluation at the next global transition. Rule (43.11b) combines the results of $n \leq p$ local steps by consolidating the freshly allocated tasks into the state of the computation. We tacitly assume that the Γ_i's have pairwise disjoint domains and that no x_j is in the domain of Γ_i; this ensures that the combined context $\Gamma\,\Gamma_1\,\dots\,\Gamma_n$ does not declare any variable more than once. In implementation terms this means that the processors must synchronize memory allocated to ensure that they do not interfere with one another.

The proof of Brent's Theorem for this high-level dynamics is now obvious. 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.

43.5 Exercises

Chapter 44

Futures and Speculation

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 τ is just an expression of type τ. 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 *suspension* is a delayed computation whose result may or may not be needed for the overall computation to finish. *Speculation* is a parallel dynamics for suspensions in which suspended computations are executed in parallel with the main thread of computation without regard to whether the suspension is forced. If the value of the suspension is eventually required, then speculation pays off, but if not, the effort to evaluate it wasted. Speculation is therefore not work-efficient: if the value of the suspension is never needed, more work has been undertaken than is necessary to determine the outcome of the computation. Speculation can be useful in situations where there is an excess of computing resources available, more than can be used in a guaranteed work-efficient manner. In such situations it cannot hurt to perform extra work as long as resources are used that would otherwise be idle.

Parallel futures, in contrast to speculatively evaluated suspensions, 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. Speculative suspensions, 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!

44.1 Futures

The syntax of futures is given by the following grammar:

Type	τ	$::=$	$\mathtt{fut}(\tau)$	$\tau\,\mathtt{fut}$	future
Expr	e	$::=$	$\mathtt{fut}(e)$	$\mathtt{fut}(e)$	future
			$\mathtt{syn}(e)$	$\mathtt{syn}(e)$	synchronize

The type $\tau\,\mathtt{fut}$ is the type of futures of type τ. 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 $\mathtt{syn}(e)$, which synchronizes with the future referred to by e, returning its value.

44.1.1 Statics

The statics of futures is given by the following rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{fut}(e) : \mathtt{fut}(\tau)} \tag{44.1a}$$

$$\frac{\Gamma \vdash e : \mathtt{fut}(\tau)}{\Gamma \vdash \mathtt{syn}(e) : \tau} \tag{44.1b}$$

These rules are unsurprising, since futures add no new capabilities to the language beyond providing an opportunity for parallel evaluation.

44.1.2 Sequential Dynamics

The sequential dynamics of futures is easily defined. Futures are evaluated eagerly; synchronization returns the value of the future.

$$\frac{e\ \mathsf{val}}{\mathtt{fut}(e)\ \mathsf{val}} \tag{44.2a}$$

$$\frac{e \mapsto e'}{\mathtt{fut}(e) \mapsto \mathtt{fut}(e')} \tag{44.2b}$$

$$\frac{e \mapsto e'}{\mathtt{syn}(e) \mapsto \mathtt{syn}(e')} \tag{44.2c}$$

$$\frac{e\ \mathsf{val}}{\mathtt{syn}(\mathtt{fut}(e)) \mapsto e} \tag{44.2d}$$

44.2 Suspensions

The syntax of (non-recursive) suspensions is given by the following grammar:[1]

Type	τ	::=	$\mathtt{susp}(\tau)$	$\tau\,\mathtt{susp}$	suspension
Expr	e	::=	$\mathtt{susp}(e)$	$\mathtt{susp}(e)$	delay
			$\mathtt{force}(e)$	$\mathtt{force}(e)$	force

The type $\tau\,\mathtt{susp}$ is the type of suspended computations of type τ. The introductory form, $\mathtt{susp}(e)$, delays the computation of e until forced, and the eliminatory form, $\mathtt{force}(e)$, forces evaluation of a delayed computation.

44.2.1 Statics

The statics of suspensions is given by the following rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{susp}(e) : \mathtt{susp}(\tau)} \tag{44.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{susp}(\tau)}{\Gamma \vdash \mathtt{force}(e) : \tau} \tag{44.3b}$$

Thus, the statics for suspensions as given by Rules (44.3) is essentially equivalent to the statics for futures given by Rules (44.1).

[1] We confine ourselves to the non-recursive case to facilitate the comparison with futures.

44.2.2 Sequential Dynamics

The definition of the sequential dynamics of suspensions is similar to that of futures, except that suspended computations are values.

$$\frac{}{\texttt{susp}(e)\ \mathsf{val}} \tag{44.4a}$$

$$\frac{e \mapsto e'}{\texttt{susp}(e) \mapsto \texttt{susp}(e')} \tag{44.4b}$$

$$\frac{}{\texttt{force}(\texttt{susp}(e)) \mapsto e} \tag{44.4c}$$

Compared with futures, the sole difference is that a suspension is only evaluated when forced, whereas a future is always evaluated, regardless of whether its value is needed.

44.3 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. Suspensions are (as we saw in Chapter 41) useful for reasons other than parallelism, but they also admit a parallel, speculative interpretation. In this section we give a parallel dynamics of futures and suspensions in which the creation, execution, and synchronization of tasks is made explicit. Interestingly, the parallel dynamics of futures and suspensions is *identical*, except for the termination condition. Whereas futures require all concurrently executing evaluations to be completed before termination, speculatively evaluated suspensions may be abandoned before they are completed. For the sake of concreteness we will give the parallel dynamics of futures, remarking only where alterations must be made for speculative evaluation of suspensions.

The parallel dynamics of futures relies on a modest extension to the language given in Section 44.1 on page 408 to introduce *names* for tasks. Let Σ be a finite mapping assigning types to names. The expression $\texttt{fut}[a]$ is a value referring to the outcome of task a. The statics of this expression is given by the following rule:[2]

$$\frac{}{\Gamma \vdash_{\Sigma,a:\tau} \texttt{fut}[a] : \texttt{fut}(\tau)} \tag{44.5}$$

[2] A similar rule governs the analogous construct, $\texttt{susp}[a]$, in the case of suspensions.

Rules (44.1) carry over in the obvious way with Σ 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 μ represents the parallel futures (or suspensions) that have been activated thus far in the computation. Formally, μ is a finite mapping assigning expressions to the task names declared in Σ. A state is well-formed according to the following rule:

$$\frac{\vdash_\Sigma e : \tau \quad (\forall a \in dom(\Sigma))\ \vdash_\Sigma \mu(a) : \Sigma(a)}{\nu\,\Sigma\,\{\, e \parallel \mu \,\}\ \mathsf{ok}} \tag{44.6}$$

As discussed in Chapter 40 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 is defined by the following rules:

$$\frac{}{\texttt{fut}\,[a]\ \mathsf{val}_{\Sigma,a:\tau}} \tag{44.7a}$$

$$\frac{}{\nu\,\Sigma\,\{\, \texttt{fut}\,(e) \parallel \mu \,\} \mapsto_{loc} \nu\,\Sigma, a:\tau\,\{\, \texttt{fut}\,[a] \parallel \mu \otimes \langle a : e\rangle \,\}} \tag{44.7b}$$

$$\frac{\nu\,\Sigma\,\{\, e \parallel \mu \,\} \mapsto_{loc} \nu\,\Sigma'\,\{\, e' \parallel \mu' \,\}}{\nu\,\Sigma\,\{\, \texttt{syn}\,(e) \parallel \mu \,\} \mapsto_{loc} \nu\,\Sigma'\,\{\, \texttt{syn}\,(e') \parallel \mu' \,\}} \tag{44.7c}$$

$$\frac{e'\ \mathsf{val}_{\Sigma,a:\tau}}{\left\{\begin{array}{c} \nu\,\Sigma, a:\tau\,\{\, \texttt{syn}\,(\texttt{fut}\,[a]) \parallel \mu \otimes \langle a : e'\rangle \,\} \\ \mapsto_{loc} \\ \nu\,\Sigma, a:\tau\,\{\, e' \parallel \mu \otimes \langle a : e'\rangle \,\} \end{array}\right\}} \tag{44.7d}$$

Rule (44.7b) activates a future named a executing the expression e and returns a reference to it. Rule (44.7d) synchronizes with a future whose value has been determined. Note that a local transition always has the form

$$\nu\,\Sigma\,\{\, e \parallel \mu \,\} \mapsto_{loc} \nu\,\Sigma\,\Sigma'\,\{\, e' \parallel \mu \otimes \mu' \,\}$$

where Σ' is either empty or declares the type of a single symbol, and μ' is either empty or of the form $\langle a : e'\rangle$ 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.

$$\dfrac{\begin{array}{c}\mu = \mu_0 \otimes \langle a_1 : e_1 \rangle \otimes \cdots \otimes \langle a_n : e_n \rangle \\ \mu'' = \mu_0 \otimes \langle a_1 : e_1' \rangle \otimes \cdots \otimes \langle a_n : e_n' \rangle \\ \nu\,\Sigma\,\{\, e \parallel \mu \,\} \mapsto^{\mathbf{0,1}}_{loc} \nu\,\Sigma\,\Sigma'\,\{\, e' \parallel \mu \otimes \mu' \,\} \\ (\forall 1 \leq i \leq n) \quad \nu\,\Sigma\,\{\, e_i \parallel \mu \,\} \mapsto_{loc} \nu\,\Sigma\,\Sigma_i'\,\{\, e_i' \parallel \mu \otimes \mu_i' \,\}\end{array}}{\left\{\begin{array}{c}\nu\,\Sigma\,\{\, e \parallel \mu \,\} \\ \mapsto_{glo} \\ \nu\,\Sigma\,\Sigma'\,\Sigma_1' \;\ldots\; \Sigma_n'\,\{\, e' \parallel \mu'' \otimes \mu' \otimes \mu_1' \otimes \cdots \otimes \mu_n' \,\}\end{array}\right\}} \tag{44.8a}$$

Rule (44.8a) allows the focus expression to take either zero or one steps since it may be blocked awaiting the completion of evaluation of a parallel future (or forcing a suspension). 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 suspensions, is defined by the rule

$$\dfrac{}{\nu\,\emptyset\,\{\, e \parallel \emptyset \,\}\ \mathsf{initial}} \tag{44.9}$$

Final states differ according to whether we are considering futures or suspensions. In the case of futures a state is final iff both the focus and all parallel futures have completed evaluation:

$$\dfrac{e\ \mathsf{val}_{\Sigma} \quad \mu\ \mathsf{val}_{\Sigma}}{\nu\,\Sigma\,\{\, e \parallel \mu \,\}\ \mathsf{final}} \tag{44.10a}$$

$$\dfrac{(\forall a \in dom(\Sigma))\ \mu(a)\ \mathsf{val}_{\Sigma}}{\mu\ \mathsf{val}_{\Sigma}} \tag{44.10b}$$

In the case of suspensions a state is final iff the focus is a value:

$$\dfrac{e\ \mathsf{val}_{\Sigma}}{\nu\,\Sigma\,\{\, e \parallel \mu \,\}\ \mathsf{final}} \tag{44.11}$$

This corresponds to the speculative nature of the parallel evaluation of suspensions whose outcome may not be needed to determine the final outcome of the program.

44.4 Applications of Futures

Pipelining provides a good 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* that 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 $\mu t.\texttt{unit} + (\texttt{nat} \times t\,\texttt{fut})$, whose elements are `nil`, defined to be $\texttt{fold}(\texttt{l} \cdot \langle\rangle)$, and $\texttt{cons}(e_1, e_2)$, defined to be $\texttt{fold}(\texttt{r} \cdot \langle e_1, \texttt{fut}(e_2)\rangle)$. 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 overlap subsequent computation.

The consumer folds an operation over the work list as follows:

```
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)) (syn 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.

Another application of futures is to provide more control over parallelism in a language with suspensions. Rather than evaluate suspensions speculatively, which is not work efficient, we may instead add futures to the language in addition to suspensions. One application of futures in such a setting is called a *spark*. A spark is a computation that is executed in parallel with another purely for its effect on suspensions. The spark traverses a data structure, forcing the suspensions within so that their values are computed and stored, but otherwise yielding no useful result. The idea is that the spark forces the suspensions that will be needed by the main computation, but taking advantage of parallelism in the hope that their values will have been computed by the time the main computation requires them.

The sequential dynamics of the spark expression $\texttt{spark}(e_1;e_2)$ is simply to evaluate e_1 before evaluating e_2. This is useful in the context of a by-need dynamics for suspensions, since evaluation of e_1 will record the values of some suspensions in the memo table for subsequent use by the computation e_2. The parallel dynamics specifies, in addition, that e_1 and e_2 are to be evaluated in parallel. The behavior of sparks is captured by the definition of $\texttt{spark}(e_1;e_2)$ in terms of futures:

$$\texttt{let \_ be fut}(e_1)\ \texttt{in}\ e_2.$$

Evaluation of e_1 commences immediately, but its value, if any, is abandoned. This encoding does not allow for evaluation of e_1 to be abandoned as soon as e_2 reaches a value, but this scenario is not expected to arise for the intended mode of use of sparks. The expression e_1 should be a quick traversal that does nothing other than force the suspensions in some data structure, exiting as soon as this is complete. Presumably this computation takes less time than it takes for e_2 to perform its work before forcing the suspensions that were forced by e_2, otherwise there is little to be gained from the use of sparks in the first place!

As an example, consider the type strm of streams of numbers defined by the recursive type $\mu t.(\texttt{unit} + (\texttt{nat} \times t))\,\texttt{susp}$. 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 $\texttt{spark}(\texttt{mapforce}(s);\texttt{map}(f)(s))$ 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.

Finally, note that it is easy to encode binary nested parallelism using futures. This may be accomplished by defining $\texttt{letpar}(e_1;e_2;x_1.x_2.e)$ to stand for the expression

$$\texttt{let}\ x_1'\ \texttt{be}\ \texttt{fut}(e_1)\ \texttt{in}\ \texttt{let}\ x_2\ \texttt{be}\ e_2\ \texttt{in}\ \texttt{let}\ x_1\ \texttt{be}\ \texttt{syn}(x_1')\ \texttt{in}\ e$$

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.

44.5 Exercises

Part XVII

Concurrency

Chapter 45

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 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.

Our presentation differs from that in the literature in several respects. Most significantly, we maintain a distinction between *processes* and *events*. The basic form of process is one that awaits the arrival of one of several events. Other forms of process include parallel composition and the declaration of a communication channel. The basic forms of event are *signalling* and *querying* on a channel. Events are combined using a non-deterministic choice operator that signals the arrival any one of a specified collection of events.

45.1 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 several possible actions, known as an event.

Proc	P	$::=$	`await(`E`)`	$\$\, E$	synchronize
Evt	E	$::=$	`null`	$\mathbf{0}$	nullary choice
			`or(`E_1`;`E_2`)`	$E_1 + E_2$	binary choice
			`que[`a`](`P`)`	$?a\,;P$	query
			`sig[`a`](`P`)`	$!a\,;P$	signal

The variables a, b, and c range over *channels*, which serve as synchronization sites between 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:

$$\frac{E \equiv E'}{\$\, E \equiv \$\, E'} \tag{45.1a}$$

$$\frac{E_1 \equiv E_1' \quad E_2 \equiv E_2'}{E_1 + E_2 \equiv E_1' + E_2'} \tag{45.1b}$$

$$\frac{P \equiv P'}{?a\,;P \equiv\ ?a\,;P'} \tag{45.1c}$$

$$\frac{P \equiv P'}{!a\,;P \equiv\ !a\,;P'} \tag{45.1d}$$

$$\frac{}{E + \mathbf{0} \equiv E} \tag{45.1e}$$

$$\frac{}{E_1 + E_2 \equiv E_2 + E_1} \tag{45.1f}$$

$$\frac{}{E_1 + (E_2 + E_3) \equiv (E_1 + E_2) + E_3} \tag{45.1g}$$

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.

$$V = \$\,\texttt{(?2p;}\$\,\texttt{(!tea;}V + \texttt{?2p;}\$\,\texttt{(!cof;}V\texttt{)))}$$

As the example indicates, we tacitly permit recursive definitions of processes, with the understanding that a defined identifier may always be replaced with its definition wherever it occurs.

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.

45.2 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*:

$$\begin{array}{llllll}
\mathsf{Proc} & P & ::= & \texttt{await(}E\texttt{)} & \$\,E & \text{synchronize} \\
 & & & \texttt{stop} & \mathbf{1} & \text{inert} \\
 & & & \texttt{par(}P_1\texttt{;}P_2\texttt{)} & P_1 \parallel P_2 & \text{composition}
\end{array}$$

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. One 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:

$$\frac{}{P \parallel \mathbf{1} \equiv P} \tag{45.2a}$$

$$\frac{}{P_1 \parallel P_2 \equiv P_2 \parallel P_1} \tag{45.2b}$$

$$\frac{}{P_1 \parallel \texttt{(}P_2 \parallel P_3\texttt{)} \equiv \texttt{(}P_1 \parallel P_2\texttt{)} \parallel P_3} \tag{45.2c}$$

$$\frac{P_1 \equiv P_1' \quad P_2 \equiv P_2'}{P_1 \parallel P_2 \equiv P_1' \parallel P_2'} \tag{45.2d}$$

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, **1**.

Interaction between processes consists of synchronization of two complementary actions. The dynamics of interaction is defined by two forms of judgement. The transition judgement $P \longmapsto P'$ states that the process P evolves to the process P' as a result of a single step of computation. The family of transition judgements, $P \overset{\alpha}{\longmapsto} P'$, where α is an *action*, states that the process P may evolve to the process P' provided that the action α is permissible in the context in which the transition occurs (in a sense to be made precise momentarily). The possible actions are given by the following grammar:

$$\begin{array}{llllll} \mathsf{Act} & \alpha & ::= & \mathtt{que}[a] & ?a & \text{query} \\ & & & \mathtt{sig}[a] & !a & \text{signal} \\ & & & \mathtt{sil} & \varepsilon & \text{silent} \end{array}$$

The *query action*, $?a$, and the *signal action*, $!a$, are complementary, and the *silent action*, ε, is self-complementary. We define the *complementary* action to α to be the action $\overline{\alpha}$ given by the equations $\overline{?a} = !a$, $\overline{!a} = ?a$, and $\overline{\varepsilon} = \varepsilon$. As a notational convenience, we often regard the unlabelled transition $P \longmapsto P'$ to be the labelled transition $P \overset{\varepsilon}{\longmapsto} P'$.

$$\frac{}{\$\,(!a\,;P + E) \overset{!a}{\longmapsto} P} \tag{45.3a}$$

$$\frac{}{\$\,(?a\,;P + E) \overset{?a}{\longmapsto} P} \tag{45.3b}$$

$$\frac{P_1 \overset{\alpha}{\longmapsto} P_1'}{P_1 \parallel P_2 \overset{\alpha}{\longmapsto} P_1' \parallel P_2} \tag{45.3c}$$

$$\frac{P_1 \overset{\alpha}{\longmapsto} P_1' \quad P_2 \overset{\overline{\alpha}}{\longmapsto} P_2'}{P_1 \parallel P_2 \longmapsto P_1' \parallel P_2'} \tag{45.3d}$$

Rules (45.3a) and (45.3b) specify that any of the events on which a process is synchronizing may occur. Rule (45.3d) synchronizes two processes that take complementary actions. (When α is the silent action, Rule (45.3d) is derivable by two applications of Rule (45.3c).)

As an example, let us consider the interaction of the vending machine, V, with the user process, U, defined as follows:

$$U = \$\,\texttt{!2p}\,;\$\,\texttt{!2p}\,;\$\,\texttt{?cof}\,;\mathbf{1}.$$

Here is a trace of the interaction between V and U:

$$\begin{aligned} V \parallel U &\mapsto \$\,\texttt{!tea}\,;V + \texttt{?2p}\,;\$\,\texttt{!cof}\,;V \parallel \$\,\texttt{!2p}\,;\$\,\texttt{?cof}\,;\mathbf{1} \\ &\mapsto \$\,\texttt{!cof}\,;V \parallel \$\,\texttt{?cof}\,;\mathbf{1} \\ &\mapsto V \end{aligned}$$

These steps are justified, respectively, by the following pairs of labelled transitions:

$$\begin{aligned} U &\xrightarrow{\texttt{!2p}} U' = \$\,\texttt{!2p}\,;\$\,\texttt{?cof}\,;\mathbf{1} \\ V &\xrightarrow{\texttt{?2p}} V' = \$\,(\texttt{!tea}\,;V + \texttt{?2p}\,;\$\,\texttt{!cof}\,;V) \end{aligned}$$

$$\begin{aligned} U' &\xrightarrow{\texttt{!2p}} U'' = \$\,\texttt{?cof}\,;\mathbf{1} \\ V' &\xrightarrow{\texttt{?2p}} V'' = \$\,\texttt{!cof}\,;V \end{aligned}$$

$$\begin{aligned} U'' &\xrightarrow{\texttt{?cof}} \mathbf{1} \\ V'' &\xrightarrow{\texttt{!cof}} V \end{aligned}$$

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.

45.3 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 one may require, which may be modeled by the structural congruence

$$* P \equiv P \parallel * P. \tag{45.4}$$

Taking this as a principle of structural congruence hides the overhead of process creation, and gives no hint as to how often it can or should be applied. One could alternatively build replication into the dynamics to model the details of replication more closely:

$$* P \mapsto P \parallel * P. \tag{45.5}$$

Since 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 involving occurrences of the process variable, X, to refer to itself. This may be simulated by defining the activator process

$$A = * \mathtt{\$}\,\mathtt{(}?a\,\mathtt{;}\,P(\mathtt{\$}\,\mathtt{(}!a\,\mathtt{;}\,\mathbf{1}\mathtt{)})\mathtt{)},$$

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

$$\mathtt{\$}\,\mathtt{(}?a\,\mathtt{;}\,P(\mathtt{\$}\,\mathtt{(}!a\,\mathtt{;}\,\mathbf{1}\mathtt{)})\mathtt{)}.$$

To start process P we concurrently compose the activator, A, with an initiator process, $\mathtt{\$}\,\mathtt{(}!a\,\mathtt{;}\,\mathbf{1}\mathtt{)}$. Observe that

$$A \parallel \mathtt{\$}\,\mathtt{(}!a\,\mathtt{;}\,\mathbf{1}\mathtt{)} \mapsto A \parallel P(!a\,\mathtt{;}\,\mathbf{1}),$$

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:

$$V_0 = \mathtt{\$}\,\mathtt{(}!v\,\mathtt{;}\,\mathbf{1}\mathtt{)} \tag{45.6}$$

$$V_1 = * \mathtt{\$}\,\mathtt{(}?v\,\mathtt{;}\,V_2\mathtt{)} \tag{45.7}$$

$$V_2 = \mathtt{\$}\,\mathtt{(?2p;\$\,(!tea;}V_0 + \mathtt{?2p;\$\,(!cof;}V_0\mathtt{)))} \tag{45.8}$$

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 restriction on replication that avoids the indeterminacy inherent in accounting for it either as part of structural congruence (Rule (45.4)) or as a computation step (Rule (45.5)). Rather than take replication as a primitive notion, we may instead take *replicated synchronization* as a primitive notion governed by the following rules:

$$\frac{}{* \,\$\,(!a\,;P + E) \overset{!a}{\longmapsto} P \parallel * \,\$\,(!a\,;P + E)} \tag{45.9a}$$

$$\frac{}{* \,\$\,(?a\,;P + E) \overset{?a}{\longmapsto} P \parallel * \,\$\,(?a\,;P + E)} \tag{45.9b}$$

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.

45.4 Allocating Channels

It is often useful (particularly once we have introduce 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:

$$\mathsf{Proc} \quad P \quad ::= \quad \texttt{new}(a\,.\,P) \quad \nu\, a\,.\,P \quad \text{new channel}$$

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 $\nu\, a_1 \ldots \nu\, a_k\,.\,P$.

Structural congruence is extended with the following rules:

$$\frac{P =_\alpha P'}{P \equiv P'} \tag{45.10a}$$

$$\frac{P \equiv P'}{\nu\, a\,.\,P \equiv \nu\, a\,.\,P'} \tag{45.10b}$$

$$\frac{a \notin P_2}{(\nu\, a\,.\, P_1) \parallel P_2 \equiv \nu\, a\,.\,(P_1 \parallel P_2)} \tag{45.10c}$$

$$\frac{(a \notin P)}{\nu\, a\,.\, P \equiv P} \tag{45.10d}$$

The last rule, called *scope extrusion*, will be important in the treatment of communication in Section 45.5 on page 428. Since we identify processes up to renaming of bound names, the requirement that $a \notin P_2$ in Rule (45.10c) may always be met by choosing the name a suitably. Rule (45.10d) 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*, Σ, is, for the time being, a finite set of channels. The judgement $\vdash_\Sigma P\ \mathsf{proc}$ states that a process, P, is well-formed relative to the channels declared in the signature, Σ.

$$\frac{}{\vdash_\Sigma \mathbf{1}\ \mathsf{proc}} \tag{45.11a}$$

$$\frac{\vdash_\Sigma P_1\ \mathsf{proc} \quad \vdash_\Sigma P_2\ \mathsf{proc}}{\vdash_\Sigma P_1 \parallel P_2\ \mathsf{proc}} \tag{45.11b}$$

$$\frac{\vdash_\Sigma E\ \mathsf{event}}{\vdash_\Sigma \$\, E\ \mathsf{proc}} \tag{45.11c}$$

$$\frac{\vdash_{\Sigma,a} P\ \mathsf{proc}}{\vdash_\Sigma \nu\, a\,.\, P\ \mathsf{proc}} \tag{45.11d}$$

The foregoing rules make use of an auxiliary judgement, $\vdash_\Sigma E\ \mathsf{event}$, stating that E is a well-formed event relative to Σ.

$$\frac{}{\vdash_\Sigma \mathbf{0}\ \mathsf{event}} \tag{45.12a}$$

$$\frac{\vdash_{\Sigma,a} P\ \mathsf{proc}}{\vdash_{\Sigma,a}\ ?a\,;P\ \mathsf{event}} \tag{45.12b}$$

$$\frac{\vdash_{\Sigma,a} P\ \mathsf{proc}}{\vdash_{\Sigma,a}\ !a\,;P\ \mathsf{event}} \tag{45.12c}$$

$$\frac{\vdash_\Sigma E_1\ \mathsf{event} \quad \vdash_\Sigma E_2\ \mathsf{event}}{\vdash_\Sigma E_1 + E_2\ \mathsf{event}} \tag{45.12d}$$

We shall also have need of the judgement $\vdash_{\Sigma} \alpha$ action stating that α is a well-formed action relative to Σ:

$$\frac{}{\vdash_{\Sigma,a} ?a \ \mathsf{action}} \tag{45.13a}$$

$$\frac{}{\vdash_{\Sigma,a} !a \ \mathsf{action}} \tag{45.13b}$$

$$\frac{}{\vdash_{\Sigma} \varepsilon \ \mathsf{action}} \tag{45.13c}$$

The dynamics is correspondingly generalized to keep track of the set of active channels. The judgement $P \underset{\Sigma}{\overset{\alpha}{\longmapsto}} P'$ states that P transitions to P' with action α relative to channels Σ. 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.

$$\frac{}{\$\,(!a\,;P + E) \underset{\Sigma,a}{\overset{!a}{\longmapsto}} P} \tag{45.14a}$$

$$\frac{}{\$\,(?a\,;P + E) \underset{\Sigma,a}{\overset{?a}{\longmapsto}} P} \tag{45.14b}$$

$$\frac{P_1 \underset{\Sigma}{\overset{\alpha}{\longmapsto}} P_1'}{P_1 \parallel P_2 \underset{\Sigma}{\overset{\alpha}{\longmapsto}} P_1' \parallel P_2} \tag{45.14c}$$

$$\frac{P_1 \underset{\Sigma}{\overset{\alpha}{\longmapsto}} P_1' \quad P_2 \underset{\Sigma}{\overset{\overline{\alpha}}{\longmapsto}} P_2'}{P_1 \parallel P_2 \underset{\Sigma}{\longmapsto} P_1' \parallel P_2'} \tag{45.14d}$$

$$\frac{P \underset{\Sigma,a}{\overset{\alpha}{\longmapsto}} P' \quad \vdash_{\Sigma} \alpha \ \mathsf{action}}{\nu\, a\,.\,P \underset{\Sigma}{\overset{\alpha}{\longmapsto}} \nu\, a\,.\,P'} \tag{45.14e}$$

Rule (45.14e) states that no process may interact with $\nu\, a\,.\,P$ along the locally-allocated channel, a, since to do so would require that a already be declared in Σ, 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 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:

$$(\nu\, v\,.\,(V_0 \parallel V_1)) \parallel U \underset{\Sigma}{\longmapsto} (\nu\, v\,.\,V_2) \parallel U \equiv \nu\, v\,.\,(V_2 \parallel U).$$

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.

45.5 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. However, 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. Communication may thereby be used to alter the interconnection topology among processes as the program executes.

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:

Evt	E	$::=$	$\texttt{snd}[\tau][a](e;P)$	$!a(e);P$	send
			$\texttt{rcv}[\tau][a](x.P)$	$?a(x.P)$	receive

The event $\texttt{rcv[}\tau\texttt{][}a\texttt{](}x\,.\,P\texttt{)}$ represents the receipt of a value, x, of type τ on the channel a, passing x to the process P. The variable, x, of type τ 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 $\texttt{snd[}\tau\texttt{][}a\texttt{](}e;P\texttt{)}$ 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.

$$\mathsf{Proc}\quad P \quad ::= \quad \texttt{new[}\tau\texttt{](}a\,.\,P\texttt{)} \quad \nu\, a\!:\!\tau\,.\,P \quad \text{typed channel}$$

The process $\texttt{new[}\tau\texttt{](}a\,.\,P\texttt{)}$ introduces a new channel name, a, with associated type τ 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.

The statics of communication extends that of synchronization by associating types to channels and by considering variables that range over a type. The judgement $\Gamma \vdash_\Sigma P\ \mathsf{proc}$ states that P is a well-formed process involving the channels declared in Σ and the variables declared in Γ. It is inductively defined by the following rules, wherein we assume that the typing judgement $\Gamma \vdash_\Sigma e : \tau$ is given separately.

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{1}\ \mathsf{proc}} \tag{45.15a}$$

$$\frac{\Gamma \vdash_\Sigma P_1\ \mathsf{proc} \quad \Gamma \vdash_\Sigma P_2\ \mathsf{proc}}{\Gamma \vdash_\Sigma P_1 \parallel P_2\ \mathsf{proc}} \tag{45.15b}$$

$$\frac{\Gamma \vdash_{\Sigma,a:\tau} P\ \mathsf{proc}}{\Gamma \vdash_\Sigma \nu\, a\!:\!\tau\,.\,P\ \mathsf{proc}} \tag{45.15c}$$

$$\frac{\Gamma \vdash_\Sigma E\ \mathsf{event}}{\Gamma \vdash_\Sigma \$\, E\ \mathsf{proc}} \tag{45.15d}$$

Rules (45.15) make use of the auxiliary judgement $\Gamma \vdash_\Sigma E\ \mathsf{event}$, stating that E is a well-formed event relative to Γ and Σ, which is defined as follows:

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{0}\ \mathsf{event}} \tag{45.16a}$$

$$\frac{\Gamma \vdash_\Sigma E_1\ \mathsf{event} \quad \Gamma \vdash_\Sigma E_2\ \mathsf{event}}{\Gamma \vdash_\Sigma E_1 + E_2\ \mathsf{event}} \tag{45.16b}$$

$$\frac{\Gamma, x : \tau \vdash_{\Sigma,a:\tau} P \; \mathsf{proc}}{\Gamma \vdash_{\Sigma,a:\tau} ?a(x.P) \; \mathsf{event}} \tag{45.16c}$$

$$\frac{\Gamma \vdash_{\Sigma,a:\tau} e : \tau \quad \Gamma \vdash_{\Sigma,a:\tau} P \; \mathsf{proc}}{\Gamma \vdash_{\Sigma,a:\tau} !a(e) ; P \; \mathsf{event}} \tag{45.16d}$$

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:

$$\begin{array}{llllll} \mathsf{Act} & \alpha & ::= & \mathtt{rcv}[\tau][a](e) & ?\,a(e) & \text{receive} \\ & & & \mathtt{snd}[\tau][a](e) & !\,a(e) & \text{send} \\ & & & \mathtt{sil} & \varepsilon & \text{silent} \end{array}$$

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:

$$\frac{\vdash_{\Sigma,a:\tau} e : \tau \quad e \; \mathsf{val}_{\Sigma,a:\tau}}{\vdash_{\Sigma,a:\tau} !\,a(e) \; \mathsf{action}} \tag{45.17a}$$

$$\frac{\vdash_{\Sigma,a:\tau} e : \tau \quad e \; \mathsf{val}_{\Sigma,a:\tau}}{\vdash_{\Sigma,a:\tau} ?\,a(e) \; \mathsf{action}} \tag{45.17b}$$

$$\frac{}{\vdash_{\Sigma} \varepsilon \; \mathsf{action}} \tag{45.17c}$$

The dynamics of synchronous communication is defined by replacing Rules (45.14a) and (45.14b) with the following rules:

$$\frac{e \underset{\Sigma,a:\tau}{\longmapsto} e'}{\$\,(!a(e) ; P + E) \underset{\Sigma,a:\tau}{\longmapsto} \$\,(!a(e') ; P + E)} \tag{45.18a}$$

$$\frac{e \; \mathsf{val}_{\Sigma,a:\tau}}{\$\,(!a(e) ; P + E) \xrightarrow[\Sigma,a:\tau]{!\,a(e)} P} \tag{45.18b}$$

$$\frac{e \; \mathsf{val}_{\Sigma,a:\tau}}{\$\,(?a(x.P) + E) \xrightarrow[\Sigma,a:\tau]{?\,a(e)} [e/x]P} \tag{45.18c}$$

Rule (45.18c) is non-deterministic in that it "guesses" the value, e, to be received along channel a.

The characteristic feature of synchronous communication is that both the sender and the receiver of the message are blocked awaiting the interaction and are resumed after its completion. While it is natural to consider that the *receiver* be continued on receipt of a message, it is less obvious that the *sender* should be informed of its receipt. In effect there is an implicit acknowledgement protocol whereby the chosen receiver (among many executing concurrently) informs the sender of the receipt of its message. Put in other terms, there is an implicit "backchannel" on which the receiver signals the successful receipt of a message, and which is queried by the sender to ensure that the message has been delivered. 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:

$$\mathsf{Proc}\quad P \quad ::= \quad \texttt{asnd[}\tau\texttt{][}a\texttt{](}e\texttt{)} \quad !a\texttt{(}e\texttt{)} \quad \text{send}$$

The process $\texttt{asnd[}\tau\texttt{][}a\texttt{](}e\texttt{)}$ 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 asychronous send is given by the following rule:

$$\frac{\Gamma \vdash_{\Sigma,a:\tau} e : \tau}{\Gamma \vdash_{\Sigma,a:\tau} !a\texttt{(}e\texttt{)}\ \mathsf{proc}} \tag{45.19}$$

The dynamics is similarly straightforward:

$$\frac{e\ \mathsf{val}_{\Sigma}}{!a\texttt{(}e\texttt{)} \xmapsto[\Sigma]{!\,a(e)} \mathbf{1}} \tag{45.20}$$

The rule for interprocess communication remains unchanged, since the action associated with the asychronous send is the same as in the synchronous case. One may regard a pending asynchronous send as a "buffer" in which the message is held until a receiver is selected.

45.6 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, $\tau\,\texttt{chan}$, whose values are references to channels carrying values of type τ. 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

$$(\nu\, a{:}\tau\,.\,(P \parallel Q)) \parallel R,$$

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 τ 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 τ, then b will be a channel carrying values of type $\tau\,\texttt{chan}$, which are references to τ-carrying channels. The channel b must be known to at least one of P and Q, and also to channel R. This can be described by the following process

[1] It may be helpful to compare channel types with reference types as described in Chapters 39 and 40. Channels correspond to assignables, and channel types correspond to reference types.

expression:

$$\nu\, b\,\mathtt{:}\,\tau\,\mathtt{chan}\,\mathtt{.}\,\mathtt{(}\mathtt{(}\nu\, a\,\mathtt{:}\,\tau\,\mathtt{.}\,\mathtt{(}P \parallel Q\mathtt{)}\mathtt{)} \parallel R\mathtt{)}.$$

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 b, and commences communication with P and Q along that channel. Thus P has the form $\mathtt{\$}\,\mathtt{(}!b\mathtt{(}\&\,a\mathtt{)}\,\mathtt{;}\,P'\mathtt{)}$ and R has the form $\mathtt{\$}\,\mathtt{(}?b\mathtt{(}x\,\mathtt{.}\,R'\mathtt{)}\mathtt{)}$. The overall process has the form

$$\nu\, b\,\mathtt{:}\,\tau\,\mathtt{chan}\,\mathtt{.}\,\mathtt{(}\nu\, a\,\mathtt{:}\,\tau\,\mathtt{.}\,\mathtt{(}\mathtt{\$}\,\mathtt{(}!b\mathtt{(}a\mathtt{)}\,\mathtt{;}\,P'\mathtt{)} \parallel Q\mathtt{)} \parallel \mathtt{\$}\,\mathtt{(}?b\mathtt{(}x\,\mathtt{.}\,R'\mathtt{)}\mathtt{)}\mathtt{)}.$$

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 45.4 on page 425 to obtain the structurally equivalent process

$$\nu\, b\,\mathtt{:}\,\tau\,\mathtt{chan}\,\mathtt{.}\,\nu\, a\,\mathtt{:}\,\tau\,\mathtt{.}\,\mathtt{(}\mathtt{\$}\,\mathtt{(}!b\mathtt{(}a\mathtt{)}\,\mathtt{;}\,P'\mathtt{)} \parallel Q \parallel \mathtt{\$}\,\mathtt{(}?b\mathtt{(}x\,\mathtt{.}\,R'\mathtt{)}\mathtt{)}\mathtt{)}.$$

The scope of a has been expanded to encompass R, preparing the ground for communication between P and R, which results in the process

$$\nu\, b\,\mathtt{:}\,\tau\,\mathtt{chan}\,\mathtt{.}\,\nu\, a\,\mathtt{:}\,\tau\,\mathtt{.}\,\mathtt{(}P' \parallel Q \parallel [\&\,a/x]R'\mathtt{)}.$$

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 τ along the channel referred to by x, the process R' would have the form

$$\mathtt{\$}\,\mathtt{(}!!\mathtt{(}x;e\mathtt{)}\,\mathtt{;}\,R''\mathtt{)}.$$

Similarly, to receive along the referenced channel, the process R' would have the form

$$\mathtt{\$}\,\mathtt{(}??\mathtt{(}x;y\,\mathtt{.}\,R''\mathtt{)}\mathtt{)}.$$

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	τ	$::=$	$\mathtt{chan}(\tau)$	$\tau\ \mathtt{chan}$	channel type
Exp	e	$::=$	$\mathtt{ch}[a]$	$\&\,a$	reference
Evt	E	$::=$	$\mathtt{sndref}[\tau](e_1;e_2;P)$	$\mathtt{!!}(e_1;e_2)\,;P$	send
			$\mathtt{rcvref}[\tau](e;x\,.\,P)$	$\mathtt{??}(e;x\,.\,P)$	receive

The events $\mathtt{sndref}[\tau](e_1;e_2;P)$ and $\mathtt{rcvref}[\tau](e;x\,.\,P)$ are dynamic versions of the events $\mathtt{snd}[\tau][a](e;P)$ and $\mathtt{rcv}[\tau][a](x\,.\,P)$ 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:

$$\frac{}{\Gamma \vdash_{\Sigma,a:\tau} \&\,a : \tau\ \mathtt{chan}} \tag{45.21a}$$

$$\frac{\Gamma \vdash_{\Sigma} e_1 : \tau\ \mathtt{chan} \quad \Gamma \vdash_{\Sigma} e_2 : \tau \quad \Gamma \vdash_{\Sigma} P\ \mathsf{proc}}{\Gamma \vdash_{\Sigma} \mathtt{!!}(e_1;e_2)\,;P\ \mathsf{event}} \tag{45.21b}$$

$$\frac{\Gamma \vdash_{\Sigma} e : \tau\ \mathtt{chan} \quad \Gamma, x : \tau \vdash_{\Sigma} P\ \mathsf{proc}}{\Gamma \vdash_{\Sigma} \mathtt{??}(e;x\,.\,P)\ \mathsf{event}} \tag{45.21c}$$

The dynamics is given by the following rules, in which we have omitted the obvious rules for evaluation of the expressions occurring within events to focus attention on the crucial transitions:

$$\frac{e\ \mathsf{val}_{\Sigma}}{\$\,(\mathtt{!!}(\&\,a;e)\,;P+E) \underset{\Sigma,a:\tau}{\longmapsto} \$\,(\mathtt{!}a(e)\,;P+E)} \tag{45.22a}$$

$$\frac{}{\$\,(\mathtt{??}(\&\,a;x\,.\,P)+E) \underset{\Sigma,a:\tau}{\longmapsto} \$\,(\mathtt{?}a(x\,.\,P)+E)} \tag{45.22b}$$

These rules may be viewed as providing a dynamics for events themselves, which have thus far been essentially static data structures representing a choice of statically-given events.

45.7 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 λ calculus with a call-by-name dynamics (see Chapter 20). The main idea of the encoding is to associate each untyped λ-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. Since every term in the untyped λ-calculus is a function, a *context* is a "call site" for the function consisting of an *argument* and the *return context* 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 $u \,\texttt{@}\, z$, where u is an untyped λ-term and z is a channel reference representing the context in which u is to be evaluated. The free variables of u will be represented by channels on which we may pass a a context. Thus, the channel reference z will be a value of type π, and a free variable, x, will be a value of type $\pi\,\texttt{chan}$. The type π is chosen to satisfy the isomorphism

$$\pi \cong (\pi\,\texttt{chan} \times \pi)\,\texttt{chan}.$$

That is, a context is a channel on which is passed an argument and another context. An argument, in turn, is a channel on which is passed a context.

The encoding of untyped λ-terms as processes is given by the following equations:

$$\begin{aligned}
x \,\texttt{@}\, z &= \texttt{!!}(x;z) \\
\lambda\, x.\, u \,\texttt{@}\, z &= \texttt{\$??(unfold}(z);\langle x,z'\rangle\texttt{.}\,u \,\texttt{@}\, z'\texttt{)} \\
u_1(u_2) \,\texttt{@}\, z &= \\
&\qquad \nu\, a_1 \texttt{:} \pi\,\texttt{chan} \times \pi\texttt{.}(u_1 \,\texttt{@}\,\texttt{fold}(\&\, a_1)) \parallel \nu\, a \texttt{:} \pi\texttt{.} * \texttt{\$}\,?a(z_2\texttt{.}\,u_2 \,\texttt{@}\, z_2) \parallel !a_1(\langle \&\, a, z\rangle)
\end{aligned}$$

Here we have taken a few liberties with the syntax for the sake of readability. We use the asynchronous form of a dynamic send operation, since 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 explanation. The call site of a λ-term is determined dynamically; one cannot predict at translation time the context in which the term will be used. 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 context 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:

$$
\begin{aligned}
&(\lambda\, x.\, x)(y) \mathbin{@} z \mapsto^* \\
&\qquad\qquad \nu\, a_1 : \tau\,.\,(\$\, ?a_1(\langle x, z' \rangle\,.\,!!(x; z'))) \parallel \nu\, a : \pi\,.\, *\, \$\, ?a(z_2\,.\,!!(y; z_2)) \parallel !a_1(\langle \&\, a, z \rangle) \\
&\qquad\qquad \mapsto^* \nu\, a : \pi\,.\, *\, \$\, ?a(z_2\,.\,!!(y; z_2)) \parallel !a(z) \\
&\qquad\qquad \mapsto^* \nu\, a : \pi\,.\, *\, \$\, ?a(z_2\,.\,!!(y; z_2)) \parallel !!(y; z)
\end{aligned}
$$

Apart from the idle server process listening on channel a, this is just the translation $y \mathbin{@} z$.

45.8 Exercises

1. Rather than distinguish multiple types of channels, one may instead regard all channels as carrying a value of type π, where $\pi \cong \pi\,\texttt{chan}$. A slightly more flexible type allows channels to carry any number of channels as arguments, so that π satisfies the isomorphism $\pi \cong \pi\,\texttt{list}\,\texttt{chan}$. Show how to encode the untyped λ-calculus into a process calculus in which all channels carry values of type π as given by the second of these two equations.

Chapter 46

Concurrent Algol

In this chapter we integrate concurrency into a full-scale programming language based on the Modernized Algol to obtain Concurrent Algol, **CA**. Assignables in **CA** are replaced by more general primitives for communication among processes. Communication consists of broadcasting a *message* consisting of a *channel* attached to a *payload* of type appropriate to that channel. Such messages are simply dynamically classified values, and channels are therefore just dynamic classes (see Chapter 38 for more on dynamic classification). A broadcast message may be received by any process, but only those processes that know its channel (class) may extract the payload from the message; all others must handle it as an inscrutable value of message type.

46.1 Concurrent Algol

The syntax of **CA** is obtained by stripping out assignables from **MA**, and adding a syntactic level of *processes*:

Type	τ	$::=$	$\texttt{cmd}(\tau)$	$\tau\,\texttt{cmd}$	commands
Expr	e	$::=$	$\texttt{do}(m)$	$\texttt{do}\,m$	command
Cmd	m	$::=$	$\texttt{ret}\,e$	$\texttt{ret}\,e$	return
			$\texttt{bnd}(e;x.m)$	$\texttt{bnd}\,x \leftarrow e\,;m$	sequence
Proc	p	$::=$	$\texttt{stop}$	$\mathbf{1}$	idle
			$\texttt{proc}(m)$	$\texttt{proc}(m)$	atomic
			$\texttt{par}(p_1;p_2)$	$p_1 \parallel p_2$	parallel
			$\texttt{new}[\tau](a.p)$	$\nu\, a{:}\tau.p$	new channel

The process $\mathtt{proc}(m)$ is an atomic process executing the command, m. The other forms of process are adapted from Chapter 45. If Σ has the form $a_1 : \tau_1, \ldots, a_n : \tau_n$, then we sometimes write $\nu\,\Sigma\{p\}$ for the iterated form $\nu\,a_1{:}\tau_1 \ldots . \nu\,a_n{:}\tau_n\,.\,p$.

The statics is given by the judgements $\Gamma \vdash_\Sigma e : \tau$ and $\Gamma \vdash_\Sigma m \sim \tau$ introduced in Chapter 39, augmented by the judgement $\vdash_\Sigma p\ \mathsf{proc}$ stating that p is a well-formed process over the signature Σ. The latter judgement is defined by the following rules:

$$\frac{}{\vdash_\Sigma \mathbf{1}\ \mathsf{proc}} \tag{46.1a}$$

$$\frac{\vdash_\Sigma m \sim \tau}{\vdash_\Sigma \mathtt{proc}(m)\ \mathsf{proc}} \tag{46.1b}$$

$$\frac{\vdash_\Sigma p_1\ \mathsf{proc} \quad \vdash_\Sigma p_2\ \mathsf{proc}}{\vdash_\Sigma p_1 \parallel p_2\ \mathsf{proc}} \tag{46.1c}$$

$$\frac{\vdash_{\Sigma,a:\tau} p\ \mathsf{proc}}{\vdash_\Sigma \nu\,a{:}\tau\,.\,p\ \mathsf{proc}} \tag{46.1d}$$

Processes are tacitly identified up to structural equivalence, as described in Chapter 45.

The transition judgement $p \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p'$ states that the process p evolves in one step to the process p' with associated action α. The particular actions are specified when specific commands are introduced in Section 46.2 on page 440. As in Chapter 45 we assume that to each action is associated a complementary action, and that the silent action indexes the unlabelled transition judgement.

$$\frac{m \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} \nu\,\Sigma'\,\{\,\mathtt{proc}(m') \parallel p\,\}}{\mathtt{proc}(m) \overset{\Sigma}{\underset{\alpha}{\longmapsto}} \nu\,\Sigma'\{\mathtt{proc}(m') \parallel p\}} \tag{46.2a}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{proc}(\mathtt{ret}\,e) \underset{\Sigma}{\longmapsto} \mathbf{1}} \tag{46.2b}$$

$$\frac{p_1 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p_1'}{p_1 \parallel p_2 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p_1' \parallel p_2} \tag{46.2c}$$

$$\frac{p_1 \xmapsto[\Sigma]{\alpha} p_1' \quad p_2 \xmapsto[\Sigma]{\overline{\alpha}} p_2'}{p_1 \parallel p_2 \xmapsto[\Sigma]{} p_1' \parallel p_2'} \tag{46.2d}$$

$$\frac{p \xmapsto[\Sigma,a:\tau]{\alpha} p' \quad \vdash_\Sigma \alpha \ \mathsf{action}}{\nu\, a{:}\tau\,.\,p \xmapsto[\Sigma]{\alpha} \nu\, a{:}\tau\,.\,p'} \tag{46.2e}$$

Rule (46.2a) 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, Σ', 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 (46.2b) states that a completed command evolves to the inert (stopped) process; processes are executed solely for their effect, and not for their value.

The auxiliary judgement $m \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} \nu\,\Sigma' \{ \mathtt{proc}(m') \parallel p' \}$ defines the execution behavior of commands. It states that the command, m, transitions to the command, m', while creating new channels, Σ', and new processes, p'. The action, α, 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 judgement be construed as a triple consisting of Σ', m', and p', rather than as a process expression comprising these parts.

The generic rules defining the dynamics of **CA** are as follows:

$$\frac{e \xmapsto[\Sigma]{} e'}{\mathtt{ret}\, e \overset{\varepsilon}{\underset{\Sigma}{\hookrightarrow}} \mathtt{proc}(\mathtt{ret}\, e')} \tag{46.3a}$$

$$\frac{m_1 \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} \nu\,\Sigma' \{ \mathtt{proc}(m_1') \parallel p' \}}{\mathtt{bnd}\, x \leftarrow \mathtt{do}\, m_1 \,;\, m_2 \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} \nu\,\Sigma' \{ \mathtt{proc}(\mathtt{bnd}\, x \leftarrow \mathtt{do}\, m_1' \,;\, m_2) \parallel p' \}} \tag{46.3b}$$

$$\frac{e \ \mathsf{val}_\Sigma}{\mathtt{bnd}\, x \leftarrow \mathtt{do}\, \mathtt{ret}\, e \,;\, m_2 \overset{\varepsilon}{\underset{\Sigma}{\hookrightarrow}} \mathtt{proc}([e/x]m_2)} \tag{46.3c}$$

$$\frac{e_1 \xmapsto[\Sigma]{} e_1'}{\mathtt{bnd}\, x \leftarrow e_1 \,;\, m_2 \overset{\varepsilon}{\underset{\Sigma}{\hookrightarrow}} \mathtt{proc}(\mathtt{bnd}\, x \leftarrow e_1' \,;\, m_2)} \tag{46.3d}$$

These generic rules are supplemented by rules governing commands for communication and synchronization among processes.

46.2 Broadcast Communication

In this section we consider a very general form of process synchronization called *broadcast*. Processes emit and receive messages of type `clsfd`, the type of dynamically classified values considered in Chapter 38. 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 38.3 on page 337 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	::=	$\texttt{spawn(}e\texttt{)}$	$\texttt{spawn(}e\texttt{)}$	spawn
			$\texttt{emit(}e\texttt{)}$	$\texttt{emit(}e\texttt{)}$	emit message
			$\texttt{recv}$	$\texttt{recv}$	receive message
			$\texttt{newch[}\tau\texttt{]}$	$\texttt{newch}$	new class

The command $\texttt{spawn(}e\texttt{)}$ spawns a process that executes the encapsulated command given by e. The commands $\texttt{emit(}e\texttt{)}$ and `recv` emit and receive messages, which are just classified values whose class is the channel on which the message is sent. The command $\texttt{newch[}\tau\texttt{]}$ returns a reference to a fresh class carrying values of type τ.

The statics of broadcast communication is given by the following rules:

$$\frac{\Gamma \vdash_\Sigma e : \texttt{cmd(unit)}}{\Gamma \vdash_\Sigma \texttt{spawn(}e\texttt{)} \sim \texttt{unit}} \tag{46.4a}$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{clsfd}}{\Gamma \vdash_\Sigma \texttt{emit(}e\texttt{)} \sim \texttt{unit}} \tag{46.4b}$$

$$\frac{}{\Gamma \vdash_\Sigma \texttt{recv} \sim \texttt{clsfd}} \tag{46.4c}$$

$$\frac{}{\Gamma \vdash_\Sigma \texttt{newch[}\tau\texttt{]} \sim \texttt{class(}\tau\texttt{)}} \tag{46.4d}$$

The execution of commands for broadcast communication is defined by these rules:

$$\frac{}{\texttt{spawn(do(}m\texttt{))} \xhookrightarrow[\Sigma]{\varepsilon} \texttt{proc(ret }\langle\rangle\texttt{)} \parallel \texttt{proc(}m\texttt{)}} \tag{46.5a}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{spawn(}e\texttt{)} \xhookrightarrow[\Sigma]{\varepsilon} \texttt{proc(spawn(}e'\texttt{))}} \tag{46.5b}$$

$$\frac{e \ \mathsf{val}_{\Sigma}}{\texttt{emit(}e\texttt{)} \xhookrightarrow[\Sigma]{!e} \texttt{proc(ret }\langle\rangle\texttt{)}} \tag{46.5c}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{emit(}e\texttt{)} \xhookrightarrow[\Sigma]{\varepsilon} \texttt{proc(emit(}e'\texttt{))}} \tag{46.5d}$$

$$\frac{e \ \mathsf{val}_{\Sigma}}{\texttt{recv} \xhookrightarrow[\Sigma]{?e} \texttt{proc(ret }e\texttt{)}} \tag{46.5e}$$

$$\frac{}{\texttt{newch[}\tau\texttt{]} \xhookrightarrow[\Sigma]{\varepsilon} \nu\, a{:}\tau\texttt{.proc(ret (\&}a\texttt{))}} \tag{46.5f}$$

Rule (46.5c) specifies that $\texttt{emit(}e\texttt{)}$ has the effect of emitting the message e. Correspondingly, Rule (46.5e) specifies that `recv` may receive (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 judgement $\vdash_{\Sigma} \alpha$ action by the following rules:

$$\frac{}{\vdash_{\Sigma} \varepsilon \ \mathsf{action}} \tag{46.6a}$$

$$\frac{\vdash_{\Sigma} e : \texttt{clsfd}}{\vdash_{\Sigma} !e \ \mathsf{action}} \tag{46.6b}$$

$$\frac{\vdash_{\Sigma} e : \texttt{clsfd}}{\vdash_{\Sigma} ?e \ \mathsf{action}} \tag{46.6c}$$

Lemma 46.1. *If $m \overset{\alpha}{\underset{\Sigma}{\hookrightarrow}} \nu\,\Sigma' \,\{\, \texttt{proc}\,(m') \parallel p' \,\}$ and $\vdash_{\Sigma} m \sim \tau$, then $\vdash_{\Sigma} \alpha$ action, $\vdash_{\Sigma\,\Sigma'} m' \sim \tau$, and $\vdash_{\Sigma\,\Sigma'} p'$ proc.*

Proof. By induction on Rules (46.3). □

With this in hand the proof of preservation is straightforward.

Theorem 46.2 (Preservation)**.** *If $\vdash_{\Sigma} p$ proc and $p \underset{\Sigma}{\mapsto} p'$, then $\vdash_{\Sigma} p'$ proc.*

Proof. By induction on transition, appealing to Lemma 46.1 for the crucial steps. □

Typing does not, however, guarantee progress with respect to unlabelled transition, for the simple reason that there may be no other process with which to communicate. By extending progress to labelled transitions we may state that this is the *only* way for the execution of a process to get stuck.

Theorem 46.3 (Progress)**.** *Suppose that e val$_{\Sigma}$ for some e such that $\vdash_{\Sigma} e : \texttt{clsfd}$. If $\vdash_{\Sigma} p$ proc, then either $p \equiv \mathbf{1}$, or there exists p' and α such that $p \overset{\alpha}{\underset{\Sigma}{\mapsto}} p'$.*

Proof. By induction on Rules (46.1) and (46.4). □

The assumption that there exists a message rules out degenerate situations in which there are no channels, or all channels carry values of an empty type.

46.3 Selective Communication

Broadcast communication provides no means of restricting a receive 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, as follows:

$$\{x \leftarrow \texttt{recv}\,;\texttt{match}\,x\,\texttt{as}\,a \cdot y \Rightarrow \texttt{ret}\,y\,\texttt{ow} \Rightarrow \texttt{emit}\,(x)\}.$$

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 rebroadcast 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 received, if ever. But polling is evidently wasteful of computing resources.

An alternative is to change the language to allow for *selective communication*. Rather than receive 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, $\texttt{event}(\tau)$, of *events* consisting of a finite choice of receives all of whose associated payload has the type τ.

Typ	τ	$::=$	$\texttt{event}(\tau)$	$\tau\,\texttt{event}$	events
Exp	e	$::=$	$\texttt{rcv}[a]$	$?\,a$	selective receive
			$\texttt{never}[\tau]$	$\texttt{never}$	impossibility
			$\texttt{or}(e_1;e_2)$	$e_1\,\texttt{or}\,e_2$	choice
Cmd	m	$::=$	$\texttt{sync}(e)$	$\texttt{sync}(e)$	synchronize

Events in **CA** correspond directly to those of the asynchronous process calculus described in Chapter 45. One exception is that the receive event need not carry with it a continuation, as it does in the process calculus; this is handled by the ambient monadic structure on commands.

The statics of these constructs is given by the following rules:

$$\frac{}{\Gamma \vdash_{\Sigma,a:\tau} \texttt{rcv}[a] : \texttt{event}(\tau)} \tag{46.7a}$$

$$\frac{}{\Gamma \vdash_{\Sigma} \texttt{never}[\tau] : \texttt{event}(\tau)} \tag{46.7b}$$

$$\frac{\Gamma \vdash_{\Sigma} e_1 : \texttt{event}(\tau) \quad \Gamma \vdash_{\Sigma} e_2 : \texttt{event}(\tau)}{\Gamma \vdash_{\Sigma} \texttt{or}(e_1;e_2) : \texttt{event}(\tau)} \tag{46.7c}$$

There are, at this stage, no non-trivial event-forming constructs, but we will shortly introduce forms that require evaluation. In anticipation of this extension the dynamics of events is given by the following rules:

$$\frac{}{\texttt{rcv}[a]\ \mathsf{val}_{\Sigma,a:\tau}} \tag{46.8a}$$

$$\frac{}{\texttt{never}[\tau]\ \mathsf{val}_{\Sigma}} \tag{46.8b}$$

$$\frac{e_1\ \mathsf{val}_{\Sigma} \quad e_2\ \mathsf{val}_{\Sigma}}{\texttt{or}(e_1;e_2)\ \mathsf{val}_{\Sigma}} \tag{46.8c}$$

$$\frac{e_1 \underset{\Sigma}{\longmapsto} e_1'}{\texttt{or}(e_1;e_2) \underset{\Sigma}{\longmapsto} \texttt{or}(e_1';e_2)} \tag{46.8d}$$

$$\frac{e_1\ \mathsf{val}_\Sigma \quad e_2 \underset{\Sigma}{\longmapsto} e_2'}{\texttt{or}(e_1;e_2) \underset{\Sigma}{\longmapsto} \texttt{or}(e_1;e_2')} \tag{46.8e}$$

Event values are to be identified up to structural congruence exactly as in Chapter 45. We write $e_1 \equiv e_2$ to mean that the closed expressions of event type e_1 and e_2 are structurally congruent, and we extend this congruence to expressions, commands, and processes in the evident manner.

It remains to define the statics and dynamics of the synchronization command.

$$\frac{\Gamma \vdash_\Sigma e : \texttt{event}(\tau)}{\Gamma \vdash_\Sigma \texttt{sync}(e) \sim \tau} \tag{46.9a}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{sync}(e) \overset{\varepsilon}{\underset{\Sigma}{\hookrightarrow}} \texttt{proc}(\texttt{sync}(e'))} \tag{46.10a}$$

$$\frac{e\ \mathsf{val}_\Sigma \quad e'\ \mathsf{val}_\Sigma}{\texttt{sync}(\texttt{or}(\texttt{rcv}[a];e)) \overset{?a\cdot e'}{\underset{\Sigma}{\hookrightarrow}} \texttt{proc}(\texttt{ret}(e'))} \tag{46.10b}$$

Rule (46.10b) specifies that a read on a channel a may synchronize only with messages that are sent on channel a (that is, are classified by a). Implicit respect for structural congruence of events ensures that we need only consider a choice of events of the given form.

Evaluation of events becomes important in the the presence of channel references. Whereas the event $?\,a$ is a value that specifies the channel, a, on which to receive explicitly, the event $\texttt{rcvref}(e)$, in which e is an expression of type $\texttt{class}(\tau)$, must be evaluated to determine the referent of the expression e. The process of resolving the referent is given by the following rules:

$$\frac{\Gamma \vdash_\Sigma e : \texttt{class}(\tau)}{\Gamma \vdash_\Sigma \texttt{rcvref}(e) : \texttt{event}(\tau)} \tag{46.11a}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{rcvref}(e) \underset{\Sigma}{\longmapsto} \texttt{rcvref}(e')} \tag{46.12a}$$

$$\frac{}{\texttt{rcvref}(\texttt{cls}[a]) \underset{\Sigma,a:\tau}{\longmapsto} \texttt{rcv}[a]} \tag{46.12b}$$

As an example, to receive a message on a channel that has been passed as a message, we may use a command of the form

$$\{x \leftarrow \texttt{sync(?}\,a\texttt{)}\,;y \leftarrow \texttt{sync(??}\,x\texttt{)}\,;\ldots\}.$$

The outer event specifies that a message is to be received on channel, a, that contains a reference to some channel, b, on which a further message is to be received as specified by the inner event. The value received on channel a must have the form $\texttt{\&}\,b$, because its type is that of a channel reference. This value is substituted for x in the inner event, resulting in the event $\texttt{??}\,\texttt{\&}\,b$. Evaluation of this event yields the event value $\texttt{?}\,b$, which specifies that a message is to be received on channel b.

46.4 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 σ is associated a server that selectively receives a message on channel a with one of two forms:

1. $\texttt{get}\cdot\texttt{(\&}\,b\texttt{)}$, where b is a channel of type σ. This message requests that the contents of a be sent on channel b.

2. $\texttt{set}\cdot\texttt{(}\langle e,\texttt{\&}\,b\rangle\texttt{)}$, where e is a value of type σ, and b is a channel of type σ. 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 τ_{srvr} given by

$$\texttt{[get}:\sigma\,\texttt{class},\texttt{set}:\sigma\times\sigma\,\texttt{class]}.$$

The server selectively receives on channel a, then dispatches on the class of the message to satisfy the request.

The server associated with the assignable, a, of type σ maintains the contents of a using recursion. When called with the current contents of the assignable, the server selectively receives on channel a, dispatching on the associated request, and calling itself recursively with the (updated, if necessary) contents:

$$\lambda\,\texttt{(}u\texttt{:}\tau_{srvr}\,\texttt{class.}\,\texttt{fix}\,srvr\texttt{:}\sigma\rightarrow\texttt{void}\,\texttt{cmd}\,\texttt{is}\,\lambda\,\texttt{(}x\texttt{:}\sigma\texttt{.}\,\texttt{do}\,\{y\leftarrow\texttt{sync(??}\,u\texttt{)}\,;e_{(46.14)}\}\texttt{))}. \tag{46.13}$$

The server is a procedure that takes an argument of type σ, 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 receives a message on channel a, and dispatches on it as follows:

$$\texttt{case}\, y\, \{\texttt{get} \cdot z \Rightarrow e_{(46.15)} \mid \texttt{set} \cdot \langle x', z\rangle \Rightarrow e_{(46.16)}\}. \tag{46.14}$$

A request to get the contents of the assignable a is served as follows:

$$\{\_ \leftarrow \texttt{emit}(\texttt{mkinst}(z;x))\,;\texttt{run}\, \mathit{srvr}(x)\} \tag{46.15}$$

A request to set the contents of the assignable a is served as follows:

$$\{\_ \leftarrow \texttt{emit}(\texttt{mkinst}(z;x'))\,;\texttt{run}\, \mathit{srvr}(x')\} \tag{46.16}$$

The type $\tau\,\texttt{ref}$ is defined to be $\tau\,\texttt{class}$, the type of channels (classes) carrying a value of type τ. A new free assignable is created by the command $\texttt{ref}\, e_0$, which is defined to be

$$\{x \leftarrow \texttt{newch}\,;\_ \leftarrow \texttt{spawn}(e_{(46.13)}(x)(e_0))\,;\texttt{ret}\, x\}. \tag{46.17}$$

A channel carrying a value of type τ_{srvr} is allocated to server as the name of the assignable, and a new server is spawned that receives requests on that channel, with initial value e_0.

The commands $\texttt{@}\, e_0$ and $e_0 := e_1$ send a message to the server to get and set the contents of an assignable. The code for $\texttt{@}\, e_0$ is as follows:

$$\{x \leftarrow \texttt{newch}\,;\_ \leftarrow \texttt{emit}(\texttt{mkinst}(e_0;\texttt{get} \cdot x))\,;\texttt{sync}(??\, x)\} \tag{46.18}$$

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:

$$\{x \leftarrow \texttt{newch}\,;\_ \leftarrow \texttt{emit}(\texttt{mkinst}(e_0;\texttt{set} \cdot \langle e_1, x\rangle))\,;\texttt{sync}(??\, x)\} \tag{46.19}$$

46.5 Exercises

Chapter 47

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 an extension of Concurrent Algol, called *Distributed Algol*, or **DA**, 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.

47.1 Statics

The statics of **DA** is loosely 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 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 king, there is no further possible world in which someone else is also king (there being only one sovereign).

The applications of possible worlds are numerous. Here we shall consider an interpretation in which possible worlds are identified with sites on a network. Accessibility between worlds corresponds to 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). In the jargon of possible worlds we are considering a version of the modal logic **S5**, which is characterized by its accessibility relation being an equivalence relation. Considering one world possible relative to another corresponds to considering that the locus of computation may move from one site to another, as discussed in the introduction to this chapter.

The syntax of **DA** is a modification and an extension of that of **CA** as given by the following grammar:

Typ	τ	$::=$	$\texttt{cmd[}w\texttt{](}\tau\texttt{)}$	$\tau\,\texttt{cmd[}w\texttt{]}$	commands
			$\texttt{chan[}w\texttt{](}\tau\texttt{)}$	$\tau\,\texttt{chan[}w\texttt{]}$	channels
			$\texttt{event[}w\texttt{](}\tau\texttt{)}$	$\tau\,\texttt{event[}w\texttt{]}$	events
Cmd	m	$::=$	$\texttt{at[}w\texttt{](}m\texttt{)}$	$\texttt{at}\,w\,\{m\}$	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, $\texttt{at[}w\texttt{](}m\texttt{)}$, that changes the locus of execution from one site to another.

A signature, Σ, is a finite set of declarations of the form $a : \sigma\,@\,w$, where σ is a type and w is a site. Such a declaration specifies that a is a channel carrying a payload of type σ located at the site w. The following judgements comprise the statics of **DA**:

$$\begin{array}{ll} \Gamma \vdash_\Sigma e : \tau & \text{expression typing} \\ \Gamma \vdash_\Sigma m \sim \tau \mathbin{@} w & \text{command typing} \end{array}$$

The expression typing judgement is independent of the site; the meaning of a type is the same at all sites. The command typing judgement, by contrast, is relative to a site. It states that m is a command that returns a value of type τ, and that this command can only be executed at the site w.

The statics of **DA** is given by a collection of rules for deriving judgements of the above two forms. A representative selection of those rules follows:

$$\frac{\Gamma \vdash_\Sigma m \sim \tau \mathbin{@} w}{\Gamma \vdash_\Sigma \mathtt{do}\, m : \tau\, \mathtt{cmd}\,[w]} \tag{47.1a}$$

$$\frac{}{\Gamma \vdash_{\Sigma,a:\sigma @ w} \mathtt{\&}\, a : \sigma\, \mathtt{chan}\,[w]} \tag{47.1b}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathtt{never} : \tau\, \mathtt{event}\,[w]} \tag{47.1c}$$

$$\frac{}{\Gamma \vdash_{\Sigma,a:\sigma @ w}\, ?\, a : \sigma\, \mathtt{event}\,[w]} \tag{47.1d}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau\, \mathtt{chan}\,[w]}{\Gamma \vdash_\Sigma\, ??\, e : \tau\, \mathtt{event}\,[w]} \tag{47.1e}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau\, \mathtt{event}\,[w] \quad \Gamma \vdash_\Sigma e_2 : \tau\, \mathtt{event}\,[w]}{\Gamma \vdash_\Sigma e_1\, \mathtt{or}\, e_2 : \tau\, \mathtt{event}\,[w]} \tag{47.1f}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau\, \mathtt{event}\,[w]}{\Gamma \vdash_\Sigma \mathtt{sync}(e) \sim \tau \mathbin{@} w} \tag{47.1g}$$

$$\frac{\Gamma \vdash_\Sigma m' \sim \tau' \mathbin{@} w'}{\Gamma \vdash_\Sigma \mathtt{at}\, w'\, \{m'\} \sim \tau' \mathbin{@} w} \tag{47.1h}$$

Rule (47.1a) states that the type of an encapsulated command records the site at which the command is to be executed. Rules (47.1d) and (47.1e) specify that the type of a (static or dynamic) receive event records the site at which the channel resides. Rules (47.1c) and (47.1f) state that a choice can only be made between events at the same site; there are no cross-site choices. Rule (47.1g) 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 (47.1h) 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.

47.2 Dynamics

The dynamics is given by a labelled transition judgement between processes, much as in Chapter 46. 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 46, except for the following rules governing the atomic process:

$$\frac{m \xhookrightarrow[\Sigma]{\alpha\,@\,w} \nu\,\Sigma'\,\{\, m' \parallel p \,\}}{\mathtt{proc}[w](m) \xmapsto[\Sigma]{\alpha} \nu\,\Sigma'\,\{\, \mathtt{proc}[w](m') \parallel p \,\}} \tag{47.2a}$$

$$\frac{}{\mathtt{proc}[w](\mathtt{ret}(\langle\rangle)) \xmapsto[\Sigma]{\varepsilon} \mathtt{stop}} \tag{47.2b}$$

The command execution judgement $m \xhookrightarrow[\Sigma]{\alpha\,@\,w} \nu\,\Sigma'\,\{\, m' \parallel p \,\}$ states that the command, m, when executed at site, w, may undertake the action, α, and in the process create new channels, Σ', 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 separable parts, the newly allocated channels, the newly created processes, and a new command. (As in Chapter 46, we omit Σ' or p when either is trivial.) This may be understood as a family of judgements indexed by sites, w. At each site there is an associated labelled transition system defining concurrent interaction of processes *at that site*. Distribution (locality) is segregated from concurrency (interaction).

The command execution judgement is defined by the following rules:

$$\frac{}{\mathtt{spawn}(m) \xhookrightarrow[\Sigma]{\varepsilon\,@\,w} \mathtt{ret}(\langle\rangle) \parallel \mathtt{proc}[w](m)} \tag{47.3a}$$

$$\frac{m \xhookrightarrow[\Sigma]{\alpha\,@\,w'} \nu\,\Sigma'\,\{\, m' \parallel p' \,\}}{\mathtt{at}[w'](m) \xhookrightarrow[\Sigma]{\alpha\,@\,w} \nu\,\Sigma'\,\{\, \mathtt{at}[w'](m') \parallel p' \,\}} \tag{47.3b}$$

$$\frac{e\ \mathsf{val}_{\Sigma}}{\mathtt{at}[w'](\mathtt{ret}(e)) \xhookrightarrow[\Sigma]{\varepsilon\,@\,w} \mathtt{ret}(e)} \tag{47.3c}$$

$$\frac{e\ \mathsf{val}_{\Sigma,a:\sigma@w} \quad e'\ \mathsf{val}_{\Sigma,a:\sigma@w}}{\mathtt{sync}(\mathtt{or}(\mathtt{rcv}[a];e')) \xrightarrow[\Sigma,a:\sigma@w]{?a\cdot e\,@\,w} \mathtt{ret}(e)} \tag{47.3d}$$

Rule (47.3a) states that new processes created at a site remain at that site—the new process executes the given command at the current site. Rules (47.3b) and (47.3c) 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 (47.3d) states that a receive action may be undertaken at site w by synchronizing on a receive event specifying a channel that resides at that site. The associated data is, in that case, returned as the result of the synchronization. The receive event associated with a channel may only be activated by a synchronization command executing at that site; no cross-site interaction is permissible.

47.3 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 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 judgement $\vdash_\Sigma p\ \mathsf{proc}$, which states that the process p is well-formed. Most importantly, the following rule governs the formation of atomic processes:

$$\frac{\vdash_\Sigma m \sim \mathtt{unit}\ @\ w}{\vdash_\Sigma \mathtt{proc}[w](m)\ \mathsf{proc}} \tag{47.4}$$

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 judgement.

Lemma 47.1 (Execution)**.** *Suppose that* $m \xrightarrow[\Sigma]{\alpha\,@\,w} \nu\,\Sigma'\,\{\,m' \parallel p\,\}$. *If* $\vdash_\Sigma m \sim \tau\ @\ w$*, then* $\vdash_\Sigma \alpha$ *action and* $\vdash_\Sigma \nu\,\Sigma\{\mathtt{proc}[w](m') \parallel p\}$ *proc.*

Proof. By a straightforward induction on Rules (47.3). □

Theorem 47.2 (Preservation)**.** *If* $p \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p'$ *and* $\vdash_\Sigma p$ *proc, then* $\vdash_\Sigma p'$ *proc.*

Proof. By induction on Rules (47.1), appealing to Lemma 47.1 on the previous page 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 47.3 (Progress)**.** *If $\vdash_\Sigma p$ proc, then either $p \equiv \mathbf{1}$ or there exists α and p' such that $p \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p'$.*

47.4 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 factor out the site indices from the skeleton of the type by introducing *site-indexed families of types*. If ϕ is such a family, then $\phi\langle w\rangle$ stands for the instance obtained by specializing the types in ϕ to the site w in a sense to be detailed shortly.

To take advantage of this separation we will re-formulate the statics of **DA** using judgements of the form $\Phi \vdash_\Sigma e : \phi\,@\,w$ and $\Phi \vdash_\Sigma m \sim \phi\ @\ w$, where Φ consists of hypotheses of the form $x_i : \phi_i\,@\,w_i$. The essential difference compared to Section 47.1 on page 447 is that expressions and commands are classified by families, rather than types, and that both are indexed by the site to which they pertain. Assumptions are correspondingly generalized so that variables are classified by families and sites. The point of this reformulation is captured by the following correctness conditions on the revised statics of **DA**:

1. If $\Phi \vdash_\Sigma e : \phi\,@\,w$, then $\widehat{\Phi} \vdash_\Sigma e : \phi\langle w\rangle$.

2. If $\Phi \vdash_\Sigma m \sim \phi\ @\ w$, then $\widehat{\Phi} \vdash_\Sigma m \sim \phi\langle w\rangle\ @\ w$.

If Φ is a context of the form $x_1 : \phi_1\ @\ w_1, \ldots, x_n : \phi_n\ @\ w_n$, then $\widehat{\Phi}$ is the context $x_1 : \phi_1\langle w_1\rangle, \ldots, x_n : \phi_n\langle w_n\rangle$ in which the family associated to each assumption is specialized to the site for which the assumption is stated.

The syntax of type families is similar to that for types, with the addition of a means of fixing the interpretation of a family at a particular site.

Fam	ϕ	::=	`nat`	`nat`	numbers
			`arr`$(\phi_1;\phi_2)$	$\phi_1 \rightarrow \phi_2$	functions
			`cmd`(ϕ)	ϕ `cmd`	computations
			`chan`(ϕ)	ϕ `chan`	channels
			`event`(ϕ)	ϕ `event`	events
			`at`$[w](\phi)$	ϕ `at` w	situated

The syntax of families mimics that of types, except that the only mention of sites is in the situated type, ϕ `at` w, which fixes the interpretation of ϕ at the site w.

The instantiation of a family, ϕ, at a site, w, is written $\phi\langle w\rangle$, and is inductively defined by the following rules:

$$\frac{}{\texttt{nat}\langle w\rangle = \texttt{nat}} \tag{47.5a}$$

$$\frac{\phi_1\langle w\rangle = \tau_1 \quad \phi_2\langle w\rangle = \tau_2}{(\phi_1 \rightarrow \phi_2)\langle w\rangle = \tau_1 \rightarrow \tau_2} \tag{47.5b}$$

$$\frac{\phi\langle w\rangle = \tau}{\phi\;\texttt{cmd}\langle w\rangle = \tau\;\texttt{cmd}[w]} \tag{47.5c}$$

$$\frac{\phi\langle w\rangle = \tau}{\phi\;\texttt{chan}\langle w\rangle = \tau\;\texttt{chan}[w]} \tag{47.5d}$$

$$\frac{\phi\langle w\rangle = \tau}{\phi\;\texttt{event}\langle w\rangle = \tau\;\texttt{event}[w]} \tag{47.5e}$$

$$\frac{\phi\langle w'\rangle = \tau'}{(\phi\;\texttt{at}\;w')\langle w\rangle = \tau'} \tag{47.5f}$$

Crucially, Rule (47.5f) states that the situated family ϕ `at` w' is to be interpreted at w by the interpretation of ϕ at w'. Otherwise instantiation serves merely to decorate the constituent command, channel, and event families with the site at which they are being interpreted.

Any type, τ, of **DA** may be embedded as a constant family, `just`(τ), such that `just`$(\tau)\langle w\rangle = \tau$ for any site w. The constant family is inductively defined by the following rules:

$$\frac{}{\texttt{just(nat)} = \texttt{nat}} \tag{47.6a}$$

$$\frac{\texttt{just}(\tau_1) = \phi_1 \quad \texttt{just}(\tau_2) = \phi_2}{\texttt{just}(\tau_1 \to \tau_2) = \phi_1 \to \phi_2} \tag{47.6b}$$

$$\frac{\texttt{just}(\tau) = \phi}{\texttt{just}(\tau\,\texttt{cmd}[w]) = \phi\,\texttt{cmd}\,\texttt{at}\,w} \tag{47.6c}$$

$$\frac{\texttt{just}(\tau) = \phi}{\texttt{just}(\tau\,\texttt{chan}[w]) = \phi\,\texttt{chan}\,\texttt{at}\,w} \tag{47.6d}$$

$$\frac{\texttt{just}(\tau) = \phi}{\texttt{just}(\tau\,\texttt{event}[w]) = \phi\,\texttt{event}\,\texttt{at}\,w} \tag{47.6e}$$

It is easy to check that $\texttt{just}(\tau)$ is a constant family:

Lemma 47.4. *For any site w, $\texttt{just}(\tau)\langle w\rangle = \tau$.*

The statics of **DA** may be reformulated using judgements of the form $\Phi \vdash_\Sigma e : \phi\,@\,w$ for expressions and $\Phi \vdash_\Sigma m \sim \phi\,@\,w$ for commands. The following rules are illustrative:

$$\frac{\Phi \vdash_\Sigma e : \phi\,@\,w}{\Phi \vdash_\Sigma \texttt{ret}\,e \sim \phi\,@\,w} \tag{47.7a}$$

$$\frac{\Phi \vdash_\Sigma e_1 : \phi_1\,@\,w \quad \Phi, x : \phi_1\,@\,w \vdash_\Sigma m_2 \sim \phi_2\,@\,w}{\Phi \vdash_\Sigma \texttt{bnd}\,x \leftarrow e_1\,;m_2 \sim \phi_2\,@\,w} \tag{47.7b}$$

$$\frac{\Phi \vdash_\Sigma m \sim \phi\,@\,w}{\Phi \vdash_\Sigma \texttt{do}\,m : \phi\,\texttt{cmd}\,@\,w} \tag{47.7c}$$

$$\frac{\texttt{just}(\sigma) = \phi}{\Phi \vdash_{\Sigma,a:\sigma@w} \&\,a : \phi\,\texttt{chan}\,@\,w} \tag{47.7d}$$

$$\frac{}{\Phi \vdash_\Sigma \texttt{never} : \phi\,\texttt{event}\,@\,w} \tag{47.7e}$$

$$\frac{\texttt{just}(\sigma) = \phi}{\Phi \vdash_{\Sigma,a:\sigma@w} ?\,a : \phi\,\texttt{event}\,@\,w} \tag{47.7f}$$

$$\frac{\Phi \vdash_\Sigma e : \phi\,\texttt{chan}\,@\,w}{\Phi \vdash_\Sigma ??\,e : \phi\,\texttt{event}\,@\,w} \tag{47.7g}$$

$$\frac{\Phi \vdash_\Sigma e_1 : \phi\,\texttt{event}\,@\,w \quad \Phi \vdash_\Sigma e_2 : \phi\,\texttt{event}\,@\,w}{\Phi \vdash_\Sigma e_1\,\texttt{or}\,e_2 : \phi\,\texttt{event}\,@\,w} \tag{47.7h}$$

$$\frac{\Phi \vdash_\Sigma e : \phi\,\texttt{event}\,@\,w}{\Phi \vdash_\Sigma \texttt{sync}(e) \sim \phi\,@\,w} \tag{47.7i}$$

$$\frac{\Phi \vdash_\Sigma m' \sim \phi' \mathbin{@} w' \quad \phi' \text{ mobile}}{\Phi \vdash_\Sigma \mathtt{at}\, w' \,\{m'\} \sim \phi' \mathbin{@} w} \tag{47.7j}$$

Rule (47.7d) specifies that a reference to a channel carrying a value of type σ is classified by the *constant* family yielding the type σ at each site. Rule (47.7j) is the most interesting rule, since it include a restriction on the family ϕ'. To see how this arises, observe that Inductively we have that $\widehat{\Phi} \vdash_\Sigma m' \sim \phi'\langle w'\rangle \mathbin{@} w'$, which is enough to ensure that $\widehat{\Phi} \vdash_\Sigma \mathtt{at}\, w' \,\{m'\} \sim \phi'\langle w'\rangle \mathbin{@} w$. But we are required to show that $\widehat{\Phi} \vdash_\Sigma \mathtt{at}\, w' \,\{m'\} \sim \phi'\langle w\rangle \mathbin{@} w$! This will only be the case if $\phi'\langle w\rangle = \phi'\langle w'\rangle$, which is to say that ϕ' is a constant family—its interpretation is site-independent, or *mobile*.

The judgement ϕ mobile stating that ϕ is a mobile family is inductively defined by the following rules:

$$\frac{}{\mathtt{nat} \text{ mobile}} \tag{47.8a}$$

$$\frac{\phi_1 \text{ mobile} \quad \phi_2 \text{ mobile}}{\phi_1 \rightarrow \phi_2 \text{ mobile}} \tag{47.8b}$$

$$\frac{}{\phi\, \mathtt{at}\, w \text{ mobile}} \tag{47.8c}$$

The remaining families are not mobile, precisely because their instantiation specifies the site of their instances; these do not determine constant families.

Lemma 47.5.

1. *If ϕ mobile, then for every w and w', $\phi\langle w\rangle = \phi\langle w'\rangle$.*

2. *For any type τ, $\mathtt{just}(\tau)$ mobile.*

We may then verify that the intended interpretation is valid:

Theorem 47.6.

1. *If $\Phi \vdash_\Sigma e : \phi \mathbin{@} w$, then $\widehat{\Phi} \vdash_\Sigma e : \phi\langle w\rangle$.*

2. *If $\Phi \vdash_\Sigma m \sim \phi \mathbin{@} w$, then $\widehat{\Phi} \vdash_\Sigma m \sim \phi\langle w\rangle \mathbin{@} w$.*

Proof. By induction on Rules (47.7). □

47.5 Exercises

Part XVIII

Modularity

Chapter 48

Separate Compilation and Linking

48.1 Linking and Substitution

48.2 Exercises

Chapter 49

Basic Modules

Chapter 50

Parameterized Modules

Part XIX

Equivalence

Chapter 51

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 $\mathcal{L}\{\texttt{nat}\rightarrow\}$ of Chapter 12 in which we can express addition as the function `plus`, the expressions

$$\lambda\,(x{:}\texttt{nat}.\,\lambda\,(y{:}\texttt{nat}.\,\texttt{plus}(x)(y)))$$

and

$$\lambda\,(x{:}\texttt{nat}.\,\lambda\,(y{:}\texttt{nat}.\,\texttt{plus}(y)(x)))$$

are equal. In other words, the addition function *as programmed in* $\mathcal{L}\{\mathit{nat}\rightarrow\}$ is commutative.

This may seem to be obviously true, but *why*, precisely, is it so? More importantly, what do we even *mean* when we say that two expressions of a programming language are 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. One 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. We say that two functions are *extensionally equivalent* if they give equal results for equal arguments—in particular, they agree on all possible arguments. Since their behavior on arguments is all that matters for calculating observable results, we may expect that extensionally 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, we say that the these functions are *observationally equivalent*. The main result of this chapter is that observational and extensional equivalence coincide for a variant of $\mathcal{L}\{\texttt{nat}\rightarrow\}$ in which the successor is evaluated eagerly, so that a value of type `nat` is a numeral.

51.1 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 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 λ-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 $\mathcal{L}\{\texttt{nat}\rightarrow\}$ 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 $\mathcal{C} = \lambda\,(x{:}\texttt{nat}.\,\circ)$, and $e = x + x$, then

$$\mathcal{C}\{e\} = \lambda\,(x{:}\texttt{nat}.\,x + x).$$

The free occurrences of x in e are captured by the λ-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 $\mathcal{L}\{\texttt{nat} \rightarrow\}$ is extended to expression contexts by defining the typing judgement

$$\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$$

so that if $\Gamma \vdash e : \tau$, then $\Gamma' \vdash \mathcal{C}\{e\} : \tau'$. This judgement may be inductively defined by a collection of rules derived from the statics of $\mathcal{L}\{\texttt{nat} \rightarrow\}$ (see Rules (12.1)). Some representative rules are as follows:

$$\frac{}{\circ : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma \triangleright \tau)} \tag{51.1a}$$

$$\frac{\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \texttt{nat})}{\texttt{s}(\mathcal{C}) : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \texttt{nat})} \tag{51.1b}$$

$$\frac{\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \texttt{nat}) \quad \Gamma' \vdash e_0 : \tau' \quad \Gamma', x : \texttt{nat}, y : \tau' \vdash e_1 : \tau'}{\texttt{natrec}\,\mathcal{C}\,\{\texttt{z} \Rightarrow e_0 \mid \texttt{s}(x)\,\texttt{with}\,y \Rightarrow e_1\} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{51.1c}$$

$$\frac{\Gamma' \vdash e : \texttt{nat} \quad \mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau') \quad \Gamma', x : \texttt{nat}, y : \tau' \vdash e_1 : \tau'}{\texttt{natrec}\,e\,\{\texttt{z} \Rightarrow \mathcal{C}_0 \mid \texttt{s}(x)\,\texttt{with}\,y \Rightarrow e_1\} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{51.1d}$$

$$\frac{\Gamma' \vdash e : \mathtt{nat} \quad \Gamma' \vdash e_0 : \tau' \quad \mathcal{C}_1 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma', x : \mathtt{nat}, y : \tau' \triangleright \tau')}{\mathtt{natrec}\, e\, \{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x)\, \mathtt{with}\, y \Rightarrow \mathcal{C}_1\} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{51.1e}$$

$$\frac{\mathcal{C}_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma', x : \tau_1 \triangleright \tau_2)}{\lambda\, (x{:}\tau_1.\mathcal{C}_2) : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_1 \rightarrow \tau_2)} \tag{51.1f}$$

$$\frac{\mathcal{C}_1 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_2 \rightarrow \tau') \quad \Gamma' \vdash e_2 : \tau_2}{\mathcal{C}_1(e_2) : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{51.1g}$$

$$\frac{\Gamma' \vdash e_1 : \tau_2 \rightarrow \tau' \quad \mathcal{C}_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_2)}{e_1(\mathcal{C}_2) : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \tag{51.1h}$$

Lemma 51.1. *If $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$, then $\Gamma' \subseteq \Gamma$, and if $\Gamma \vdash e : \tau$, then $\Gamma' \vdash \mathcal{C}\{e\} : \tau'$.*

Observe that the trivial context consisting only of a "hole" acts as the identity under replacement. Moreover, contexts are closed under composition in the following sense.

Lemma 51.2. *If $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$, and $\mathcal{C}' : (\Gamma' \triangleright \tau') \rightsquigarrow (\Gamma'' \triangleright \tau'')$, then $\mathcal{C}'\{\mathcal{C}\{\circ\}\} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma'' \triangleright \tau'')$.*

Lemma 51.3. *If $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ and $x \notin dom(\Gamma)$, then $\mathcal{C} : (\Gamma, x : \sigma \triangleright \tau) \rightsquigarrow (\Gamma', x : \sigma \triangleright \tau')$.*

Proof. By induction on Rules (51.1). □

A *complete program* is a closed expression of type `nat`.

Definition 51.1. *We say that two complete programs, e and e', are* Kleene equivalent, *written $e \simeq e'$, iff there exists $n \geq 0$ such that $e \mapsto^* \overline{n}$ and $e' \mapsto^* \overline{n}$.*

Kleene equivalence is evidently reflexive and symmetric; transitivity follows from determinacy of evaluation. Closure under converse evaluation also follows directly from determinacy. It is obviously consistent in that $\overline{0} \not\simeq \overline{1}$.

Definition 51.2. *Suppose that $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$ are two expressions of the same type. We say that e and e' are* observationally equivalent, *written $e \cong e' : \tau\ [\Gamma]$, iff $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$ for every program context $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$.*

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.

A family of equivalence relations $e_1 \mathcal{E} e_2 : \tau\ [\Gamma]$ is a *congruence* iff it is preserved by all contexts. That is,

$$\text{if } e\ \mathcal{E}\ e' : \tau\ [\Gamma],\ \text{then } \mathcal{C}\{e\}\ \mathcal{E}\ \mathcal{C}\{e'\} : \tau'\ [\Gamma']$$

for every expression context $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$. Such a family of relations is *consistent* iff $e\ \mathcal{E}\ e' : \mathtt{nat}\ [\emptyset]$ implies $e \simeq e'$.

Theorem 51.4. *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 $e\ \mathcal{E}\ e' : \tau\ [\Gamma]$ for some consistent congruence $\mathcal{E}$, and if $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$, then by congruence $\mathcal{C}\{e\}\ \mathcal{E}\ \mathcal{C}\{e'\} : \mathtt{nat}\ [\emptyset]$, and hence by consistency $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$. □

A *closing substitution*, γ, 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 Γ, 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 Γ.

Lemma 51.5. *If $e \cong e' : \tau\ [\Gamma]$ and $\gamma : \Gamma$, then $\hat{\gamma}(e) \cong \hat{\gamma}(e') : \tau$. Moreover, if $\gamma \cong \gamma' : \Gamma$, then $\hat{\gamma}(e) \cong \widehat{\gamma'}(e) : \tau$ and $\hat{\gamma}(e') \cong \widehat{\gamma'}(e') : \tau$.*

Proof. Let $\mathcal{C} : (\emptyset \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$ be a program context; we are to show that $\mathcal{C}\{\hat{\gamma}(e)\} \simeq \mathcal{C}\{\hat{\gamma}(e')\}$. Since $\mathcal{C}$ 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 51.3 on the preceding page we have $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma \triangleright \mathtt{nat})$, from which it follows directly that $\mathcal{D} : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$. Since $e \cong e' : \tau\ [\Gamma]$, 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 $\hat{\gamma}(\mathcal{C}\{e\}) \simeq \hat{\gamma}(\mathcal{C}\{e'\})$. Since $\mathcal{C}$ is arbitrary, it follows that $\hat{\gamma}(e) \cong \hat{\gamma}(e') : \tau$.

Defining $\mathcal{D}'$ similarly to $\mathcal{D}$, but based on γ', rather than γ, we may also show that $\mathcal{D}'\{e\} \simeq \mathcal{D}'\{e'\}$, and hence $\widehat{\gamma}'(e) \cong \widehat{\gamma}'(e') : \tau$. Now if $\gamma \cong \gamma' : \Gamma$, then by congruence we have $\mathcal{D}\{e\} \cong \mathcal{D}'\{e\}$: `nat`, and $\mathcal{D}\{e'\} \cong \mathcal{D}'\{e'\}$: `nat`. It follows that $\mathcal{D}\{e'\} \cong \mathcal{D}'\{e'\}$: `nat`, and so, by consistency of observational equivalence, we have $\mathcal{D}\{e'\} \simeq \mathcal{D}'\{e'\}$, which is to say that $\hat{\gamma}(e) \cong \widehat{\gamma}'(e') : \tau$. □

Theorem 51.4 on the preceding page licenses the principle of *proof by coinduction*: to show that $e \cong e' : \tau\ [\Gamma]$, it is enough to exhibit a consistent congruence, $\mathcal{E}$, such that $e\ \mathcal{E}\ e' : \tau\ [\Gamma]$. 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.

51.2 Extensional 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 τ 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 extensional equivalence alluded to in the introduction.

Definition 51.3. Extensional equivalence *is a family of relations $e \sim e' : \tau$ between closed expressions of type τ. It is defined by induction on τ as follows:*

$$
\begin{array}{lll}
e \sim e' : \mathtt{nat} & \textit{iff} & e \simeq e' \\[1em]
e \sim e' : \tau_1 \to \tau_2 & \textit{iff} & \textit{if } e_1 \sim e_1' : \tau_1,\ \textit{then } e\,(e_1) \sim e'\,(e_1') : \tau_2
\end{array}
$$

The definition of extensional equivalence at type `nat` licenses the following principle of *proof by* `nat`*-induction*. To show that $\mathcal{E}\ (e, e')$ whenever $e \sim e'$: `nat`, 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 $e \mapsto^* \overline{n}$ and $e' \mapsto^* \overline{n}$ by the definition of Kleene equivalence.

Extensional equivalence is extended to open terms by substitution of related closed terms to obtain related results. If γ and γ' are two substitutions for Γ, we define $\gamma \sim \gamma' : \Gamma$ to hold iff $\gamma(x) \sim \gamma'(x) : \Gamma(x)$ for every variable, x, such that $\Gamma \vdash x : \tau$. Finally, we define $e \sim e' : \tau\ [\Gamma]$ to mean that $\hat{\gamma}(e) \sim \widehat{\gamma'}(e') : \tau$ whenever $\gamma \sim \gamma' : \Gamma$.

51.3 Extensional and Observational Equivalence Coincide

In this section we prove the coincidence of observational and extensional equivalence.

Lemma 51.6 (Converse Evaluation)**.** *Suppose that $e \sim e' : \tau$. If $d \mapsto e$, then $d \sim e' : \tau$, and if $d' \mapsto e'$, then $e \sim d' : \tau$.*

Proof. By induction on the structure of τ. If $\tau = \mathtt{nat}$, then the result follows from the closure of Kleene equivalence under converse evaluation. If $\tau = \tau_1 \rightarrow \tau_2$, then suppose that $e \sim e' : \tau$, and $d \mapsto e$. To show that $d \sim e' : \tau$, we assume $e_1 \sim e_1' : \tau_1$ and show $d(e_1) \sim e'(e_1') : \tau_2$. It follows from the assumption that $e(e_1) \sim e'(e_1') : \tau_2$. Noting that $d(e_1) \mapsto e(e_1)$, the result follows by induction. □

Lemma 51.7 (Consistency)**.** *If $e \sim e' : \mathtt{nat}$, then $e \simeq e'$.*

Proof. Immediate, from Definition 51.3 on the preceding page. □

Theorem 51.8 (Reflexivity)**.** *If $\Gamma \vdash e : \tau$, then $e \sim e : \tau\ [\Gamma]$.*

Proof. We are to show that if $\Gamma \vdash e : \tau$ and $\gamma \sim \gamma' : \Gamma$, then $\hat{\gamma}(e) \sim \widehat{\gamma'}(e) : \tau$. The proof proceeds by induction on typing derivations; we consider a few representative cases.

Consider the case of Rule (11.4a), in which $\tau = \tau_1 \rightarrow \tau_2$ and $e = \lambda\,(x{:}\tau_1.\,e_2)$. Since e is a value, we are to show that

$$\lambda\,(x{:}\tau_1.\,\hat{\gamma}(e_2)) \sim \lambda\,(x{:}\tau_1.\,\widehat{\gamma'}(e_2)) : \tau_1 \rightarrow \tau_2.$$

Assume that $e_1 \sim e_1' : \tau_1$; we are to show that $[e_1/x]\hat{\gamma}(e_2) \sim [e_1'/x]\widehat{\gamma'}(e_2) : \tau_2$. Let $\gamma_2 = \gamma[x \mapsto e_1]$ and $\gamma_2' = \gamma'[x \mapsto e_1']$, and observe that $\gamma_2 \sim \gamma_2' :$

$\Gamma, x : \tau_1$. Therefore, by induction we have $\hat{\gamma}_2(e_2) \sim \hat{\gamma}'_2(e_2) : \tau_2$, from which the result follows directly.

Now consider the case of Rule (12.1d), for which we are to show that

$$\mathtt{natrec}(\hat{\gamma}(e);\hat{\gamma}(e_0);x.y.\hat{\gamma}(e_1)) \sim \mathtt{natrec}(\widehat{\gamma'}(e);\widehat{\gamma'}(e_0);x.y.\widehat{\gamma'}(e_1)) : \tau.$$

By the induction hypothesis applied to the first premise of Rule (12.1d), we have

$$\hat{\gamma}(e) \sim \widehat{\gamma'}(e) : \mathtt{nat}.$$

We proceed by nat-induction. It suffices to show that

$$\mathtt{natrec}(\mathtt{z};\hat{\gamma}(e_0);x.y.\hat{\gamma}(e_1)) \sim \mathtt{natrec}(\mathtt{z};\widehat{\gamma'}(e_0);x.y.\widehat{\gamma'}(e_1)) : \tau, \tag{51.2}$$

and that

$$\mathtt{natrec}(\mathtt{s}(\overline{n});\hat{\gamma}(e_0);x.y.\hat{\gamma}(e_1)) \sim \mathtt{natrec}(\mathtt{s}(\overline{n});\widehat{\gamma'}(e_0);x.y.\widehat{\gamma'}(e_1)) : \tau, \tag{51.3}$$

assuming

$$\mathtt{natrec}(\overline{n};\hat{\gamma}(e_0);x.y.\hat{\gamma}(e_1)) \sim \mathtt{natrec}(\overline{n};\widehat{\gamma'}(e_0);x.y.\widehat{\gamma'}(e_1)) : \tau. \tag{51.4}$$

To show (51.2), by Lemma 51.6 on the preceding page it is enough to show that $\hat{\gamma}(e_0) \sim \widehat{\gamma'}(e_0) : \tau$. This is assured by the outer inductive hypothesis applied to the second premise of Rule (12.1d).

To show (51.3), define

$$\delta = \gamma[x \mapsto \overline{n}][y \mapsto \mathtt{natrec}(\overline{n};\hat{\gamma}(e_0);x.y.\hat{\gamma}(e_1))]$$

and

$$\delta' = \gamma'[x \mapsto \overline{n}][y \mapsto \mathtt{natrec}(\overline{n};\widehat{\gamma'}(e_0);x.y.\widehat{\gamma'}(e_1))].$$

By (51.4) we have $\delta \sim \delta' : \Gamma, x : \mathtt{nat}, y : \tau$. Consequently, by the outer inductive hypothesis applied to the third premise of Rule (12.1d), and Lemma 51.6 on the previous page, the required follows. □

Corollary 51.9 (Termination). *If $e : \tau$, then there exists e' val such that $e \mapsto^* e'$.*

Symmetry and transitivity of extensional equivalence are easily established by induction on types; extensional equivalence is therefore an equivalence relation.

Lemma 51.10 (Congruence). *If $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_0)$, and $e \sim e' : \tau\ [\Gamma]$, then $\mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_0\ [\Gamma_0]$.*

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 $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_1 \rightarrow \tau_2)$ and $\mathcal{C}_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0, x : \tau_1 \triangleright \tau_2)$. Assuming $e \sim e' : \tau\ [\Gamma]$, we are to show that

$$\mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_1 \rightarrow \tau_2\ [\Gamma_0],$$

which is to say

$$\lambda\,(x{:}\tau_1.\,\mathcal{C}_2\{e\}) \sim \lambda\,(x{:}\tau_1.\,\mathcal{C}_2\{e'\}) : \tau_1 \rightarrow \tau_2\ [\Gamma_0].$$

We know, by induction, that

$$\mathcal{C}_2\{e\} \sim \mathcal{C}_2\{e'\} : \tau_2\ [\Gamma_0, x : \tau_1].$$

Suppose that $\gamma_0 \sim \gamma_0' : \Gamma_0$, and that $e_1 \sim e_1' : \tau_1$. Let $\gamma_1 = \gamma_0[x \mapsto e_1]$, $\gamma_1' = \gamma_0'[x \mapsto e_1']$, and observe that $\gamma_1 \sim \gamma_1' : \Gamma_0, x : \tau_1$. By Definition 51.3 on page 472 it is enough to show that

$$\hat{\gamma}_1(\mathcal{C}_2\{e\}) \sim \hat{\gamma}_1'(\mathcal{C}_2\{e'\}) : \tau_2,$$

which follows immediately from the inductive hypothesis. □

Theorem 51.11. *If $e \sim e' : \tau\ [\Gamma]$, then $e \cong e' : \tau\ [\Gamma]$.*

Proof. By Lemmas 51.7 on page 473 and 51.10 on the facing page, and Theorem 51.4 on page 471. □

Corollary 51.12. *If $e : \mathtt{nat}$, then $e \cong \overline{n} : \mathtt{nat}$, for some $n \geq 0$.*

Proof. By Theorem 51.8 on page 473 we have $e \sim e : \tau$. Hence for some $n \geq 0$, we have $e \sim \overline{n} : \mathtt{nat}$, and so by Theorem 51.11, $e \cong \overline{n} : \mathtt{nat}$. □

Lemma 51.13. *For closed expressions $e : \tau$ and $e' : \tau$, if $e \cong e' : \tau$, then $e \sim e' : \tau$.*

Proof. We proceed by induction on the structure of τ. If $\tau = \mathtt{nat}$, consider the empty context to obtain $e \simeq e'$, and hence $e \sim e' : \mathtt{nat}$. If $\tau = \tau_1 \rightarrow \tau_2$, then we are to show that whenever $e_1 \sim e_1' : \tau_1$, we have $e(e_1) \sim e'(e_1') : \tau_2$. By Theorem 51.11 we have $e_1 \cong e_1' : \tau_1$, and hence by congruence of observational equivalence it follows that $e(e_1) \cong e'(e_1') : \tau_2$, from which the result follows by induction. □

Theorem 51.14. *If $e \cong e' : \tau\ [\Gamma]$, then $e \sim e' : \tau\ [\Gamma]$.*

Proof. Assume that $e \cong e' : \tau\ [\Gamma]$, and that $\gamma \sim \gamma' : \Gamma$. By Theorem 51.11 on the previous page we have $\gamma \cong \gamma' : \Gamma$, so by Lemma 51.5 on page 471 $\hat{\gamma}(e) \cong \widehat{\gamma'}(e') : \tau$. Therefore, by Lemma 51.13 on the previous page, $\hat{\gamma}(e) \sim \hat{\gamma}(e') : \tau$. □

Corollary 51.15. *$e \cong e' : \tau\ [\Gamma]$ iff $e \sim e' : \tau\ [\Gamma]$.*

Theorem 51.16. *If $\Gamma \vdash e \equiv e' : \tau$, then $e \sim e' : \tau\ [\Gamma]$, and hence $e \cong e' : \tau\ [\Gamma]$.*

Proof. By an argument similar to that used in the proof of Theorem 51.8 on page 473 and Lemma 51.10 on page 474, then appealing to Theorem 51.11 on the preceding page. □

Corollary 51.17. *If $e \equiv e' : \mathtt{nat}$, then there exists $n \geq 0$ such that $e \mapsto^* \overline{n}$ and $e' \mapsto^* \overline{n}$.*

Proof. By Theorem 51.16 we have $e \sim e' : \mathtt{nat}$ and hence $e \simeq e'$. □

51.4 Some Laws of Equivalence

In this section we summarize some useful principles of observational equivalence for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. For the most part these may be proved as laws of extensional equivalence, and then transferred to observational equivalence by appeal to Corollary 51.15. The laws are presented as inference rules with the meaning that if all of the premises are true judgements about observational equivalence, then so are the conclusions. In other words each rule is admissible as a principle of observational equivalence.

51.4.1 General Laws

Extensional equivalence is indeed an equivalence relation: it is reflexive, symmetric, and transitive.

$$\frac{}{e \cong e : \tau\ [\Gamma]} \tag{51.5a}$$

$$\frac{e' \cong e : \tau\ [\Gamma]}{e \cong e' : \tau\ [\Gamma]} \tag{51.5b}$$

$$\frac{e \cong e' : \tau\ [\Gamma] \quad e' \cong e'' : \tau\ [\Gamma]}{e \cong e'' : \tau\ [\Gamma]} \tag{51.5c}$$

Reflexivity is an instance of a more general principle, that all definitional equivalences are observational equivalences.

$$\frac{\Gamma \vdash e \equiv e' : \tau}{e \cong e' : \tau\ [\Gamma]} \tag{51.6a}$$

This is called the *principle of symbolic evaluation*.

Observational equivalence is a congruence: we may replace equals by equals anywhere in an expression.

$$\frac{e \cong e' : \tau\ [\Gamma] \quad \mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')}{\mathcal{C}\{e\} \cong \mathcal{C}\{e'\} : \tau'\ [\Gamma']} \tag{51.7a}$$

Equivalence is stable under substitution for free variables, and substituting equivalent expressions in an expression gives equivalent results.

$$\frac{\Gamma \vdash e : \tau \quad e_2 \cong e_2' : \tau'\ [\Gamma, x : \tau]}{[e/x]e_2 \cong [e/x]e_2' : \tau'\ [\Gamma]} \tag{51.8a}$$

$$\frac{e_1 \cong e_1' : \tau\ [\Gamma] \quad e_2 \cong e_2' : \tau'\ [\Gamma, x : \tau]}{[e_1/x]e_2 \cong [e_1'/x]e_2' : \tau'\ [\Gamma]} \tag{51.8b}$$

51.4.2 Extensionality Laws

Two functions are equivalent if they are equivalent on all arguments.

$$\frac{e(x) \cong e'(x) : \tau_2\ [\Gamma, x : \tau_1]}{e \cong e' : \tau_1 \rightarrow \tau_2\ [\Gamma]} \tag{51.9}$$

Consequently, every expression of function type is equivalent to a λ-abstraction:

$$\frac{}{e \cong \lambda\,(x{:}\tau_1.\,e(x)) : \tau_1 \rightarrow \tau_2\ [\Gamma]} \tag{51.10}$$

51.4.3 Induction Law

An equation involving a free variable, x, of type `nat` can be proved by induction on x.

$$\frac{[\overline{n}/x]e \cong [\overline{n}/x]e' : \tau\ [\Gamma]\ (\text{for every } n \in \mathbb{N})}{e \cong e' : \tau\ [\Gamma, x : \mathtt{nat}]} \tag{51.11a}$$

To apply the induction rule, we proceed by mathematical induction on $n \in \mathbb{N}$, which reduces to showing:

1. $[\mathtt{z}/x]e \cong [\mathtt{z}/x]e' : \tau\ [\Gamma]$, and
2. $[\mathtt{s}(\overline{n})/x]e \cong [\mathtt{s}(\overline{n})/x]e' : \tau\ [\Gamma]$, if $[\overline{n}/x]e \cong [\overline{n}/x]e' : \tau\ [\Gamma]$.

51.5 Exercises

Chapter 52

Equational Reasoning for PCF

In this Chapter we develop the theory of observational equivalence for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, with an eager interpretation of the type of natural numbers. The development proceeds long lines similar to those in Chapter 51, 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*.

52.1 Observational Equivalence

The definition of observational equivalence, along with the auxiliary notion of Kleene equivalence, are defined similarly to Chapter 51, but modified to account for the possibility of non-termination.

The collection of well-formed $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ contexts is inductively defined in a manner directly analogous to that in Chapter 51. Specifically, we define the judgement $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ by rules similar to Rules (51.1), modified for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$. (We leave the precise definition as an exercise for the reader.) When Γ and Γ' are empty, we write just $\mathcal{C} : \tau \rightsquigarrow \tau'$.

A *complete program* is a closed expression of type $\texttt{nat}$.

Definition 52.1. *We say that two complete programs, e and e', are* Kleene equivalent, *written $e \simeq e'$, iff for every $n \geq 0$, $e \mapsto^* \overline{n}$ iff $e' \mapsto^* \overline{n}$.*

Kleene equivalence 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 as in Chapter 51.

Definition 52.2. *We say that* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e : \tau$ *are* observationally, *or* contextually, equivalent *iff for every program context* $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{nat})$, $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$.

Theorem 52.1. *Observational equivalence is the coarsest consistent congruence.*

Proof. See the proof of Theorem 51.4 on page 471. □

Lemma 52.2 (Substitution and Functionality)**.** *If* $e \cong e' : \tau\ [\Gamma]$ *and* $\gamma : \Gamma$, *then* $\hat{\gamma}(e) \cong \hat{\gamma}(e') : \tau$. *Moreover, if* $\gamma \cong \gamma' : \Gamma$, *then* $\hat{\gamma}(e) \cong \hat{\gamma}'(e) : \tau$ *and* $\hat{\gamma}(e') \cong \hat{\gamma}'(e') : \tau$.

Proof. See Lemma 51.5 on page 471. □

52.2 Extensional Equivalence

Definition 52.3. Extensional equivalence, $e \sim e' : \tau$, *between closed expressions of type* τ *is defined by induction on* τ *as follows:*

$$e \sim e' : \mathtt{nat} \quad \textit{iff} \quad e \simeq e'$$

$$e \sim e' : \tau_1 \rightarrow \tau_2 \quad \textit{iff} \quad e_1 \sim e_1' : \tau_1 \textit{ implies } e(e_1) \sim e'(e_1') : \tau_2$$

Formally, extensional equivalence is defined as in Chapter 51, except that the definition of Kleene equivalence is altered to account for non-termination. Extensional equivalence is extended to open terms by substitution. Specifically, we define $e \sim e' : \tau\ [\Gamma]$ to mean that $\hat{\gamma}(e) \sim \widehat{\gamma}'(e') : \tau$ whenever $\gamma \sim \gamma' : \Gamma$.

Lemma 52.3 (Strictness)**.** *If* $e : \tau$ *and* $e' : \tau$ *are both divergent, then* $e \sim e' : \tau$.

Proof. By induction on the structure of τ. If $\tau = \mathtt{nat}$, then the result follows immediately from the definition of Kleene equivalence. If $\tau = \tau_1 \rightarrow \tau_2$, then $e(e_1)$ and $e'(e_1')$ diverge, so by induction $e(e_1) \sim e'(e_1') : \tau_2$, as required. □

Lemma 52.4 (Converse Evaluation)**.** *Suppose that* $e \sim e' : \tau$. *If* $d \mapsto e$, *then* $d \sim e' : \tau$, *and if* $d' \mapsto e'$, *then* $e \sim d' : \tau$.

52.3 Extensional and Observational Equivalence Coincide

As a technical convenience, we enrich $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ with *bounded recursion*, with abstract syntax $\mathtt{fix}^m[\tau](x.e)$ and concrete syntax $\mathtt{fix}^m\ x{:}\tau\ \mathtt{is}\ e$, where $m \geq 0$. The statics of bounded recursion is the same as for general recursion:

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{fix}^m[\tau](x.e) : \tau} . \tag{52.1a}$$

The dynamics of bounded recursion is defined as follows:

$$\frac{}{\mathtt{fix}^0[\tau](x.e) \mapsto \mathtt{fix}^0[\tau](x.e)} \tag{52.2a}$$

$$\frac{}{\mathtt{fix}^{m+1}[\tau](x.e) \mapsto [\mathtt{fix}^m[\tau](x.e)/x]e} \tag{52.2b}$$

If m is positive, the recursive bound is decremented so that subsequent uses of it will be limited to one fewer unrolling. If m reaches zero, the expression steps to itself so that the computation diverges with no result.

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 52.4 on page 484 below.

Theorem 52.5 (Fixed Point Induction)**.** *Suppose that $x : \tau \vdash e : \tau$. If*

$$(\forall m \geq 0)\ \mathit{fix}^m\ x{:}\tau\ \mathtt{is}\ e \sim \mathit{fix}^m\ x{:}\tau\ \mathtt{is}\ e' : \tau,$$

then $\mathtt{fix}\ x{:}\tau\ \mathtt{is}\ e \sim \mathtt{fix}\ x{:}\tau\ \mathtt{is}\ e' : \tau$.

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 judgement $\mathcal{A} : \rho \rightsquigarrow \tau$ is a special case of the general typing judgment for contexts.) Define extensional equivalence of applicative contexts, written $\mathcal{A} \approx \mathcal{A}' : \rho \rightsquigarrow \tau$, by induction on the structure of $\mathcal{A}$ as follows:

1. $\circ \approx \circ : \rho \rightsquigarrow \rho$;
2. if $\mathcal{A} \approx \mathcal{A}' : \rho \rightsquigarrow \tau_2 \rightarrow \tau$ and $e_2 \sim e_2' : \tau_2$, then $\mathcal{A}(e_2) \approx \mathcal{A}'(e_2') : \rho \rightsquigarrow \tau$.

We prove by induction on the structure of τ, if $\mathcal{A} \approx \mathcal{A}' : \rho \rightsquigarrow \tau$ and

$$\text{for every } m \geq 0,\ \mathcal{A}\{\mathtt{fix}^m\ x{:}\rho\ \mathtt{is}\ e\} \sim \mathcal{A}'\{\mathtt{fix}^m\ x{:}\rho\ \mathtt{is}\ e'\} : \tau, \tag{52.3}$$

then

$$\mathcal{A}\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e\} \sim \mathcal{A}'\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e'\} : \tau. \tag{52.4}$$

Choosing $\mathcal{A} = \mathcal{A}' = \circ$ (so that $\rho = \tau$) completes the proof.

If $\tau = \mathtt{nat}$, then assume that $\mathcal{A} \approx \mathcal{A}' : \rho \rightsquigarrow \mathtt{nat}$ and (52.3). By Definition 52.3 on page 480, we are to show

$$\mathcal{A}\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e\} \simeq \mathcal{A}'\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e'\}.$$

By Corollary 52.14 on page 487 there exists $m \geq 0$ such that

$$\mathcal{A}\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e\} \simeq \mathcal{A}\{\mathtt{fix}^m\,x{:}\rho\,\mathtt{is}\,e\}.$$

By (52.3) we have

$$\mathcal{A}\{\mathtt{fix}^m\,x{:}\rho\,\mathtt{is}\,e\} \simeq \mathcal{A}'\{\mathtt{fix}^m\,x{:}\rho\,\mathtt{is}\,e'\}.$$

By Corollary 52.14 on page 487

$$\mathcal{A}'\{\mathtt{fix}^m\,x{:}\rho\,\mathtt{is}\,e'\} \simeq \mathcal{A}'\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e'\}.$$

The result follows by transitivity of Kleene equivalence.

If $\tau = \tau_1 \rightharpoonup \tau_2$, then by Definition 52.3 on page 480, it is enough to show

$$\mathcal{A}\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e\}(e_1) \sim \mathcal{A}'\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e'\}(e_1') : \tau_2$$

whenever $e_1 \sim e_1' : \tau_1$. Let $\mathcal{A}_2 = \mathcal{A}(e_1)$ and $\mathcal{A}_2' = \mathcal{A}'(e_1')$. It follows from (52.3) that for every $m \geq 0$

$$\mathcal{A}_2\{\mathtt{fix}^m\,x{:}\rho\,\mathtt{is}\,e\} \sim \mathcal{A}_2'\{\mathtt{fix}^m\,x{:}\rho\,\mathtt{is}\,e'\} : \tau_2.$$

Noting that $\mathcal{A}_2 \approx \mathcal{A}_2' : \rho \rightsquigarrow \tau_2$, we have by induction

$$\mathcal{A}_2\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e\} \sim \mathcal{A}_2'\{\mathtt{fix}\,x{:}\rho\,\mathtt{is}\,e'\} : \tau_2,$$

as required. □

Lemma 52.6 (Reflexivity). *If $\Gamma \vdash e : \tau$, then $e \sim e : \tau\ [\Gamma]$.*

Proof. The proof proceeds along the same lines as the proof of Theorem 51.8 on page 473. The main difference is the treatment of general recursion, which is proved by fixed point induction. Consider Rule (13.1g). Assuming $\gamma \sim \gamma' : \Gamma$, we are to show that

$$\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,\hat{\gamma}(e) \sim \mathtt{fix}\,x{:}\tau\,\mathtt{is}\,\widehat{\gamma'}(e) : \tau.$$

By Theorem 52.5 on page 481 it is enough to show that, for every $m \geq 0$,

$$\texttt{fix}^{m}\, x{:}\tau\, \texttt{is}\, \hat{\gamma}(e) \sim \texttt{fix}^{m}\, x{:}\tau\, \texttt{is}\, \widehat{\gamma'}(e) : \tau.$$

We proceed by an inner induction on m. When $m = 0$ the result is immediate, since both sides of the desired equivalence diverge. Assuming the result for m, and applying Lemma 52.4 on page 480, it is enough to show that $\hat{\gamma}(e_1) \sim \widehat{\gamma'}(e_1) : \tau$, where

$$e_1 = [\texttt{fix}^{m}\, x{:}\tau\, \texttt{is}\, \hat{\gamma}(e)/x]\hat{\gamma}(e), \text{ and} \tag{52.5}$$

$$e_1' = [\texttt{fix}^{m}\, x{:}\tau\, \texttt{is}\, \widehat{\gamma'}(e)/x]\widehat{\gamma'}(e). \tag{52.6}$$

But this follows directly from the inner and outer inductive hypotheses. For by the outer inductive hypothesis, if

$$\texttt{fix}^{m}\, x{:}\tau\, \texttt{is}\, \hat{\gamma}(e) \sim \texttt{fix}^{m}\, x{:}\tau\, \texttt{is}\, \widehat{\gamma'}(e) : \tau,$$

then

$$[\texttt{fix}^{m}\, x{:}\tau\, \texttt{is}\, \hat{\gamma}(e)/x]\hat{\gamma}(e) \sim [\texttt{fix}^{m}\, x{:}\tau\, \texttt{is}\, \widehat{\gamma'}(e)/x]\widehat{\gamma'}(e) : \tau.$$

But the hypothesis holds by the inner inductive hypothesis, from which the result follows. □

Symmetry and transitivity of eager extensional equivalence are easily established by induction on types, noting that Kleene equivalence is symmetric and transitive. Eager extensional equivalence is therefore an equivalence relation.

Lemma 52.7 (Congruence)**.** *If $\mathcal{C}_0 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma_0 \triangleright \tau_0)$, and $e \sim e' : \tau\ [\Gamma]$, then $\mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_0\ [\Gamma_0]$.*

Proof. By induction on the derivation of the typing of $\mathcal{C}_0$, following along similar lines to the proof of Lemma 52.6 on the preceding page. □

Logical equivalence is consistent, by definition. Consequently, it is contained in observational equivalence.

Theorem 52.8. *If $e \sim e' : \tau\ [\Gamma]$, then $e \cong e' : \tau\ [\Gamma]$.*

Proof. By consistency and congruence of extensional equivalence. □

Lemma 52.9. *If $e \cong e' : \tau$, then $e \sim e' : \tau$.*

Proof. By induction on the structure of τ. If $\tau = \texttt{nat}$, then the result is immediate, since the empty expression context is a program context. If $\tau = \tau_1 \rightharpoonup \tau_2$, then suppose that $e_1 \sim e_1' : \tau_1$. We are to show that $e(e_1) \sim e'(e_1') : \tau_2$. By Theorem 52.8 on the preceding page $e_1 \cong e_1' : \tau_1$, and hence by Lemma 52.2 on page 480 $e(e_1) \cong e'(e_1') : \tau_2$, from which the result follows by induction. □

Theorem 52.10. *If $e \cong e' : \tau\ [\Gamma]$, then $e \sim e' : \tau\ [\Gamma]$.*

Proof. Assume that $e \cong e' : \tau\ [\Gamma]$. Suppose that $\gamma \sim \gamma' : \Gamma$. By Theorem 52.8 on the preceding page we have $\gamma \cong \gamma' : \Gamma$, and so by Lemma 52.2 on page 480 we have

$$\hat{\gamma}(e) \cong \hat{\gamma}'(e') : \tau.$$

Therefore by Lemma 52.9 on the preceding page we have

$$\hat{\gamma}(e) \sim \hat{\gamma}'(e') : \tau.$$

□

Corollary 52.11. *$e \cong e' : \tau\ [\Gamma]$ iff $e \sim e' : \tau\ [\Gamma]$.*

52.4 Compactness

The principle of fixed point induction is derived from a critical property of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, called *compactness*. This property states that only finitely many unwindings of a fixed point expression are needed in a complete evaluation of a program. While 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 52.13 on page 486) makes use of the stack machine for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ defined in Chapter 31, augmented with the following transitions for bounded recursive expressions:

$$\frac{}{k \triangleright \texttt{fix}^{0}\, x{:}\tau\, \texttt{is}\, e \mapsto k \triangleright \texttt{fix}^{0}\, x{:}\tau\, \texttt{is}\, e} \tag{52.7a}$$

$$\frac{}{k \triangleright \texttt{fix}^{m+1}\, x{:}\tau\, \texttt{is}\, e \mapsto k \triangleright [\texttt{fix}^{m}\, x{:}\tau\, \texttt{is}\, e/x]e} \tag{52.7b}$$

It is straightforward to extend the proof of correctness of the stack machine (Corollary 31.4 on page 271) to account for bounded recursion.

To get a feel for what is involved in the compactness proof, consider first the factorial function, f, in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$:

$$\texttt{fix}\, f\texttt{:nat} \rightharpoonup \texttt{nat is}\, \lambda\, \texttt{(}x\texttt{:nat. ifz}\, x\, \{\texttt{z} \Rightarrow \texttt{s(z)} \mid \texttt{s(}x'\texttt{)} \Rightarrow x * f\texttt{(}x'\texttt{)}\}\texttt{)}.$$

Obviously evaluation of $f\texttt{(}\overline{n}\texttt{)}$ 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,

$$\texttt{fix}^m\, f\texttt{:nat} \rightharpoonup \texttt{nat is}\, \lambda\, \texttt{(}x\texttt{:nat. ifz}\, x\, \{\texttt{z} \Rightarrow \texttt{s(z)} \mid \texttt{s(}x'\texttt{)} \Rightarrow x * f\texttt{(}x'\texttt{)}\}\texttt{)}.$$

Call this expression $f^{(m)}$. It follows from the definition of f that if $f\texttt{(}\overline{n}\texttt{)} \mapsto^* \overline{p}$, then $f^{(m)}\texttt{(}\overline{n}\texttt{)} \mapsto^* \overline{p}$ 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 $\tau = \texttt{nat} \rightharpoonup \texttt{(nat} \rightharpoonup \texttt{nat)}$, given by the expression

$$\texttt{fix}\, p\texttt{:}\tau\, \texttt{is}\, \lambda\, \texttt{(}x\texttt{:nat. ifz}\, x\, \{\texttt{z} \Rightarrow id \mid \texttt{s(}x'\texttt{)} \Rightarrow s \circ \texttt{(}p\texttt{(}x'\texttt{))}\}\texttt{)},$$

where $id = \lambda\, \texttt{(}y\texttt{:nat.}\, y\texttt{)}$ is the identity, $e' \circ e = \lambda\, \texttt{(}x\texttt{:}\tau\texttt{.}\, e'\texttt{(}e\texttt{(}x\texttt{)))}$ is composition, and $s = \lambda\, \texttt{(}x\texttt{:nat. s(}x\texttt{))}$ is the successor function. The application $a\texttt{(}\overline{n}\texttt{)}$ terminates after three transitions, regardless of the value of n, resulting in a λ-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 $a^{(m)}\texttt{()}$ 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 $x\,.\,e_x$. Define $f^{(\omega)} = \texttt{fix}\, x\texttt{:}\tau\, \texttt{is}\, e_x$, and $f^{(m)} = \texttt{fix}^m\, x\texttt{:}\tau\, \texttt{is}\, e_x$, and 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 "passive" occurrences of a recursive expression to be raised without affecting the outcome of evaluation.

Lemma 52.12. *If* $[f^{(m)}/y]k \triangleright [f^{(m)}/y]e \mapsto^* \epsilon \triangleleft \overline{n}$*, where* $e \neq y$*, then* $[f^{(m+1)}/y]k \triangleright [f^{(m+1)}/y]e \mapsto^* \epsilon \triangleleft \overline{n}$*.*

Proof. By induction on the definition of the transition judgement for $\mathcal{K}\{\texttt{nat} \rightharpoonup\}$. □

Theorem 52.13 (Compactness)**.** *Suppose that $y : \tau \vdash e : \mathtt{nat}$ where $y \notin f^{(\omega)}$. If $[f^{(\omega)}/y]e \mapsto^* \overline{n}$, then there exists $m \geq 0$ such that $[f^{(m)}/y]e \mapsto^* \overline{n}$.*

Proof. We prove simultaneously the stronger statements that if

$$[f^{(\omega)}/y]k \triangleright [f^{(\omega)}/y]e \mapsto^* \epsilon \triangleleft \overline{n},$$

then for some $m \geq 0$,

$$[f^{(m)}/y]k \triangleright [f^{(m)}/y]e \mapsto^* \epsilon \triangleleft \overline{n},$$

and

$$[f^{(\omega)}/y]k \triangleleft [f^{(\omega)}/y]e \mapsto^* \epsilon \triangleleft \overline{n}$$

then for some $m \geq 0$,

$$[f^{(m)}/y]k \triangleleft [f^{(m)}/y]e \mapsto^* \epsilon \triangleleft \overline{n}.$$

(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 31.4 on page 271).

We proceed by induction on transition. Suppose that the initial state is

$$[f^{(\omega)}/y]k \triangleright f^{(\omega)},$$

which arises when $e = y$, and the transition sequence is as follows:

$$[f^{(\omega)}/y]k \triangleright f^{(\omega)} \mapsto [f^{(\omega)}/y]k \triangleright [f^{(\omega)}/x]e_x \mapsto^* \epsilon \triangleleft \overline{n}.$$

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

$$[f^{(m)}/y]k \triangleright [f^{(m)}/x]e_x \mapsto^* \epsilon \triangleleft \overline{n}.$$

By Lemma 52.12 on the preceding page

$$[f^{(m+1)}/y]k \triangleright [f^{(m)}/x]e_x \mapsto^* \epsilon \triangleleft \overline{n}$$

and we need only observe that

$$[f^{(m+1)}/y]k \triangleright f^{(m+1)} \mapsto [f^{(m+1)}/y]k \triangleright [f^{(m)}/x]e_x$$

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 $z \neq y$

$$[f^{(\omega)}/y]k \triangleright \mathtt{fix}\, z{:}\tau\, \mathtt{is}\, d_\omega \mapsto [f^{(\omega)}/y]k \triangleright [\mathtt{fix}\, z{:}\tau\, \mathtt{is}\, d_\omega/z]d_\omega \mapsto^* \epsilon \triangleleft \overline{n}.$$

where $d_\omega = [f^{(\omega)}/y]d$. By induction there exists $m \geq 0$ such that

$$[f^{(m)}/y]k \triangleright [\mathtt{fix}\, z : \tau\, \mathtt{is}\, d_m/z]d_m \mapsto^* \epsilon \triangleleft \overline{n},$$

where $d_m = [f^{(m)}/y]d$. But then by Lemma 52.12 on page 485 we have

$$[f^{(m+1)}/y]k \triangleright [\mathtt{fix}\, z : \tau\, \mathtt{is}\, d_{m+1}/z]d_{m+1} \mapsto^* \epsilon \triangleleft \overline{n},$$

where $d_{m+1} = [f^{(m+1)}/y]d$, from which the result follows directly. □

Corollary 52.14. *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 52.13 on the facing page to obtain m, and note that the required Kleene equivalence follows. □

52.5 Co-Natural Numbers

In Chapter 13 we considered a variation of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ with the co-natural numbers, `conat`, as base type. This is achieved by specifying that $\mathtt{s}(e)$ `val` regardless of the form of e, so that the successor does not evaluate its argument. Using general recursion we may define the infinite number, ω, by $\mathtt{fix}\, x : \mathtt{conat}\, \mathtt{is}\, \mathtt{s}(x)$, which consists of an infinite stack of successors. Since the successor is intepreted lazily, ω evaluates to a value, namely $\mathtt{s}(\omega)$, its own successor. It follows that the principle of mathematical induction is not valid for the co-natural numbers. For example, the property of being equivalent to a finite numeral is satisfied by zero and is closed under successor, but fails for ω.

In this section we sketch the modifications to the preceding development for the co-natural numbers. The main difference is that the definition of extensional equivalence at type `conat` must be formulated to account for laziness. Rather than being defined *inductively* as the strongest relation closed under specified conditions, we define it *coinductively* as the weakest relation consistent two analogous conditions. We may then show that two expressions are related using the principle of *proof by coinduction*.

If `conat` is to continue to serve as the observable outcome of a computation, then we must alter the meaning of Kleene equivalence to account for laziness. We adopt the principle that we may observe of a computation only its outermost form: it is either zero or the successor of some other computation. More precisely, we define $e \simeq e'$ iff (a) if $e \mapsto^* \mathtt{z}$, then $e' \mapsto^* \mathtt{z}$, and *vice versa*; and (b) if $e \mapsto^* \mathtt{s}(e_1)$, then $e' \mapsto^* \mathtt{s}(e_1')$, and *vice versa*. Note well that we do not require anything of e_1 and e_1' in the second clause. This means that $\overline{1} \simeq \overline{2}$, yet we retain consistency in that $\overline{0} \not\simeq \overline{1}$.

Corollary 52.14 on the previous page can be proved for the co-natural numbers by essentially the same argument.

The definition of extensional equivalence at type $\mathtt{conat}$ is defined to be the *weakest* equivalence relation, $\mathcal{E}$, between closed terms of type $\mathtt{conat}$ satisfying the following *$\mathtt{conat}$-consistency conditions*: if $e\ \mathcal{E}\ e' : \mathtt{conat}$, then

1. If $e \mapsto^* \mathtt{z}$, then $e' \mapsto^* \mathtt{z}$, and *vice versa*.

2. If $e \mapsto^* \mathtt{s}(e_1)$, then $e' \mapsto^* \mathtt{s}(e_1')$ with $e_1\ \mathcal{E}\ e_1' : \mathtt{conat}$, and *vice versa*.

It is immediate that if $e \sim e' : \mathtt{conat}$, then $e \simeq e'$, and so extensional equivalence is consistent. It is also strict in that if e and e' are both divergent expressions of type $\mathtt{conat}$, then $e \sim e' : \mathtt{conat}$—simply because the preceding two conditions are vacuously true in this case.

This is an example of the more general principle of *proof by $\mathtt{conat}$-coinduction*. To show that $e \sim e' : \mathtt{conat}$, it suffices to exhibit a relation, $\mathcal{E}$, such that

1. $e\ \mathcal{E}\ e' : \mathtt{conat}$, and

2. $\mathcal{E}$ satisfies the $\mathtt{conat}$-consistency conditions.

If these requirements hold, then $\mathcal{E}$ is contained in extensional equivalence at type $\mathtt{conat}$, and hence $e \sim e' : \mathtt{conat}$, as required.

As an application of $\mathtt{conat}$-coinduction, let us consider the proof of Theorem 52.5 on page 481. The overall argument remains as before, but the proof for the type $\mathtt{conat}$ must be altered as follows. Suppose that $\mathcal{A} \approx \mathcal{A}' : \rho \rightsquigarrow \mathtt{conat}$, and let $a = \mathcal{A}\{\mathtt{fix}\, x{:}\rho\, \mathtt{is}\, e\}$ and $a' = \mathcal{A}'\{\mathtt{fix}\, x{:}\rho\, \mathtt{is}\, e'\}$. Writing $a^{(m)} = \mathcal{A}\{\mathtt{fix}^m\, x{:}\rho\, \mathtt{is}\, e\}$ and $a'^{(m)} = \mathcal{A}'\{\mathtt{fix}^m\, x{:}\rho\, \mathtt{is}\, e'\}$, assume that

$$\text{for every } m \geq 0,\ a^{(m)} \sim a'^{(m)} : \mathtt{conat}.$$

We are to show that

$$a \sim a' : \mathtt{conat}.$$

Define the functions p_n for $n \geq 0$ on closed terms of type $\mathtt{conat}$ by the following equations:

$$\begin{aligned} p_0(d) &= d \\ p_{(n+1)}(d) &= \begin{cases} d' & \text{if } p_n(d) \mapsto^* \mathtt{s}(d') \\ \textit{undefined} & \text{otherwise} \end{cases} \end{aligned}$$

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 ${a'_n}^{(m)} = p_n(a_n^{(m)})$. Define $\mathcal{E}$ to be the strongest relation such that $a_n \; \mathcal{E} \; a'_n : \texttt{conat}$ for all $n \geq 0$. We will show that the relation $\mathcal{E}$ satisfies the conat-consistency conditions, and so it is contained in extensional equivalence. Since $a \; \mathcal{E} \; a' : \texttt{conat}$ (by construction), the result follows immediately.

To show that $\mathcal{E}$ is conat-consistent, suppose that $a_n \; \mathcal{E} \; a'_n : \texttt{conat}$ for some $n \geq 0$. We have by Corollary 52.14 on page 487 $a_n \simeq a_n^{(m)}$, for some $m \geq 0$, and hence, by the assumption, $a_n \simeq {a'_n}^{(m)}$, and so by Corollary 52.14 on page 487 again, ${a'_n}^{(m)} \simeq a'_n$. Now if $a_n \mapsto^* \texttt{s}(b_n)$, then $a_n^{(m)} \mapsto^* \texttt{s}(b_n^{(m)})$ for some $b_n^{(m)}$, and hence there exists ${b'_n}^{(m)}$ such that ${a'_n}^{(m)} \mapsto^* {b'_n}^{(m)}$, and so there exists b'_n such that $a'_n \mapsto^* \texttt{s}(b'_n)$. But $b_n = p_{n+1}(a)$ and $b'_n = p_{n+1}(a')$, and we have $b_n \; \mathcal{E} \; b'_n : \texttt{conat}$ by construction, as required.

52.6 Exercises

1. Call-by-value variant, with recursive functions.

Chapter 53

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 23. 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 $\mathcal{L}\{\rightarrow\forall\}$ under a call-by-name interpretation.

53.1 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 $\forall(t\,.\,t \rightarrow t)$. What function could it be? When instantiated at a type τ it should evaluate to a function g of type $\tau \rightarrow \tau$ that, when further applied to a value v of type τ returns a value v' of type τ. Since f is polymorphic, g cannot depend on v, so v'

must be v. In other words, g must be the identity function at type τ, 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 τ, 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.

Let N be the type of polymorphic Church numerals introduced in Chapter 23, namely $\forall(t.t \to (t \to t) \to t)$. What are the values of this type? Given any type τ, and values $z : \tau$ and $s : \tau \to \tau$, the expression

$$f[\tau](z)(s)$$

must yield a value of type τ. Moreover, it must behave uniformly with respect to the choice of τ. What values could it yield? The only way to build a value of type τ 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.

53.2 Observational Equivalence

The definition of observational equivalence given in Chapters 51 and 52 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 $\mathcal{L}\{\to\forall\}$, 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 $\mathcal{L}\{\to\forall\}$ 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 $\mathcal{L}\{\to\forall\}$ 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 equivalence is defined for complete programs by requiring that $e \simeq e'$ iff either (a) $e \mapsto^* \mathbf{tt}$ and $e' \mapsto^* \mathbf{tt}$; or (b) $e \mapsto^* \mathbf{ff}$ and $e' \mapsto^* \mathbf{ff}$.

This is obviously an equivalence relation, and it is immediate that **tt** $\not\simeq$ **ff**, since 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 equivalence at the answer type, **2**.

To define observational equivalence, we must first define the concept of an expression context for $\mathcal{L}\{\rightarrow\forall\}$ as an expression with a "hole" in it. More precisely, we may give an inductive definition of the judgement

$$\mathcal{C} : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow (\Delta'; \Gamma' \triangleright \tau'),$$

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 judgement, and the verification of its properties, as an exercise for the reader.)

Definition 53.1. *Two expressions of the same type are* observationally equivalent, *written* $e \cong e' : \tau\ [\Delta; \Gamma]$, *iff* $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$ *whenever* $\mathcal{C} : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathbf{2})$.

Lemma 53.1. *Observational equivalence is the coarsest consistent congruence.*

Proof. The composition of a program context with another context is itself a program context. It is consistent by virtue of the empty context being a program context. □

Lemma 53.2.

1. *If* $e \cong e' : \tau\ [\Delta, t; \Gamma]$ *and* ρ type, *then* $[\rho/t]e \cong [\rho/t]e' : [\rho/t]\tau\ [\Delta; [\rho/t]\Gamma]$.

2. *If* $e \cong e' : \tau\ [\emptyset; \Gamma, x : \sigma]$ *and* $d : \sigma$, *then* $[d/x]e \cong [d/x]e' : \tau\ [\emptyset; \Gamma]$. *Moreover, if* $d \cong d' : \sigma$, *then* $[d/x]e \cong [d'/x]e : \tau\ [\emptyset; \Gamma]$ *and* $[d/x]e' \cong [d'/x]e' : \tau\ [\emptyset; \Gamma]$.

Proof. 1. Let $\mathcal{C} : (\Delta; [\rho/t]\Gamma \triangleright [\rho/t]\tau) \rightsquigarrow (\emptyset \triangleright \mathbf{2})$ be a program context. We are to show that

$$\mathcal{C}\{[\rho/t]e\} \simeq \mathcal{C}\{[\rho/t]e'\}.$$

Since $\mathcal{C}$ is closed, this is equivalent to

$$[\rho/t]\mathcal{C}\{e\} \simeq [\rho/t]\mathcal{C}\{e'\}.$$

Let $\mathcal{C}'$ be the context $\Lambda(t.\mathcal{C}\{\circ\})[\rho]$, 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 $\mathcal{C}'\{e\} \simeq [\rho/t]\mathcal{C}\{e\}$, and $\mathcal{C}'\{e'\} \simeq [\rho/t]\mathcal{C}\{e'\}$, from which the result follows.

2. By an argument essentially similar to that for Lemma 51.5 on page 471.

□

53.3 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 $\mathcal{L}\{\rightarrow\forall\}$ is somewhat more complex than for $\mathcal{L}\{\mathtt{nat}\rightarrow\}$. 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 ρ. 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 51 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 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 ρ and ρ', and treat the type variable t as standing for *any relation* (of some form) between ρ and ρ'. One 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. This permits us to consider a notion of "equivalence" between two expressions of different type—they are equivalent, *modulo* a relation between the interpretations of their free type variables.

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 53.2 (Admissibility). *A relation R between expressions of types ρ and ρ' is* admissible*, written $R : \rho \leftrightarrow \rho'$, iff it satisfies two requirements:*

1. *Respect for observational equivalence: if $R(e, e')$ and $d \cong e : \rho$ and $d' \cong e' : \rho'$, then $R(d, d')$.*
2. *Closure under converse evaluation: if $R(e, e')$, then if $d \mapsto e$, then $R(d, e')$ and if $d' \mapsto e'$, then $R(e, d')$.*

The second of these conditions will turn out to be a consequence of the first, but we are not yet in a position to establish this fact.

The judgement $\delta : \Delta$ states that δ is a *type substitution* that assigns a closed type to each type variable $t \in \Delta$. A type substitution, δ, induces a substitution function, $\hat{\delta}$, on types given by the equation

$$\hat{\delta}(\tau) = [\delta(t_1), \dots, \delta(t_n)/t_1, \dots, t_n]\tau,$$

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 δ and δ' be two type substitutions of closed types to the type variables in Δ. A *relation assignment*, η, between δ and δ' is an assignment of an admissible relation $\eta(t) : \delta(t) \leftrightarrow \delta'(t)$ to each $t \in \Delta$. The judgement $\eta : \delta \leftrightarrow \delta'$ states that η is a relation assignment between δ and δ'.

Logical equivalence is defined in terms of its generalization, called *parametric logical equivalence*, written $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$, defined as follows.

Definition 53.3 (Parametric Logical Equivalence)**.** *The relation $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$ is defined by induction on the structure of τ by the following conditions:*

$$\begin{array}{lll}
e \sim e' : t\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & \eta(t)(e,e') \\
e \sim e' : \mathbf{2}\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & e \simeq e' \\
e \sim e' : \tau_1 \rightarrow \tau_2\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & e_1 \sim e_1' : \tau_1\ [\eta : \delta \leftrightarrow \delta']\ \textit{implies} \\
 & & e\,(e_1) \sim e'\,(e_1') : \tau_2\ [\eta : \delta \leftrightarrow \delta'] \\
e \sim e' : \forall\,(t\,.\,\tau)\ [\eta : \delta \leftrightarrow \delta'] & \textit{iff} & \textit{for every}\ \rho,\ \rho',\ \textit{and every}\ R : \rho \leftrightarrow \rho', \\
 & & e\,[\rho] \sim e'\,[\rho'] : \tau\ [\eta[t \mapsto R] : \delta[t \mapsto \rho] \leftrightarrow \delta'[t \mapsto \rho']]
\end{array}$$

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*, γ, for a context Γ, written $\gamma : \Gamma$, is an substitution of a closed expression $\gamma(x) : \Gamma(x)$ to each variable $x \in \mathit{dom}(\Gamma)$. 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 Γ consists of the variables $x_1, \ldots, x_n$. The relation $\gamma \sim \gamma' : \Gamma\ [\eta : \delta \leftrightarrow \delta']$ is defined to hold iff $\mathit{dom}(\gamma) = \mathit{dom}(\gamma') = \mathit{dom}(\Gamma)$, and $\gamma(x) \sim \gamma'(x) : \Gamma(x)\ [\eta : \delta \leftrightarrow \delta']$ for every variable, x, in their common domain.

Definition 53.4 (Logical Equivalence)**.** *The expressions $\Delta; \Gamma \vdash e : \tau$ and $\Delta; \Gamma \vdash e' : \tau$ are* logically equivalent, *written $e \sim e' : \tau\ [\Delta; \Gamma]$ iff for every assigment δ and δ' of closed types to type variables in Δ, and every relation assignment $\eta : \delta \leftrightarrow \delta'$, if $\gamma \sim \gamma' : \Gamma\ [\eta : \delta \leftrightarrow \delta']$, then $\hat{\gamma}(\hat{\delta}(e)) \sim \widehat{\gamma'}(\widehat{\delta'}(e')) : \tau\ [\eta : \delta \leftrightarrow \delta']$.*

When e, e', and τ are closed, then this definition states that $e \sim e' : \tau$ iff $e \sim e' : \tau\ [\emptyset : \emptyset \leftrightarrow \emptyset]$, so that logical equivalence is indeed a special case of its generalization.

Lemma 53.3 (Closure under Converse Evaluation)**.** *Suppose that $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$. If $d \mapsto e$, then $d \sim e' : \tau$, and if $d' \mapsto e'$, then $e \sim d' : \tau$.*

Proof. By induction on the structure of τ. 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 53.4 (Respect for Observational Equivalence)**.** *Suppose that $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$. If $d \cong e : \hat{\delta}(\tau)$ and $d' \cong e' : \widehat{\delta'}(\tau)$, then $d \sim d' : \tau\ [\eta : \delta \leftrightarrow \delta']$.*

Proof. By induction on the structure of τ, relying on the definition of admissibility, and the congruence property of observational equivalence. For example, if $\tau = \forall(t\,.\,\sigma)$, then we are to show that for every $R : \rho \leftrightarrow \rho'$,

$$d\,[\rho] \sim d'\,[\rho'] : \sigma\ [\eta[t \mapsto R] : \delta[t \mapsto \rho] \leftrightarrow \delta'[t \mapsto \rho']].$$

Since observational equivalence is a congruence, $d\,[\rho] \cong e\,[\rho] : [\rho/t]\hat{\delta}(\sigma)$, $d'\,[\rho] \cong e'\,[\rho] : [\rho'/t]\widehat{\delta'}(\sigma)$. From the assumption it follows that

$$e\,[\rho] \sim e'\,[\rho'] : \sigma\ [\eta[t \mapsto R] : \delta[t \mapsto \rho] \leftrightarrow \delta'[t \mapsto \rho']],$$

from which the result follows by induction. □

Corollary 53.5. *The relation $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$ is an admissible relation between closed types $\hat{\delta}(\tau)$ and $\widehat{\delta'}(\tau)$.*

Proof. By Lemmas 53.3 on the facing page and 53.4 on the preceding page. □

Corollary 53.6. *If $e \sim e' : \tau\ [\Delta;\Gamma]$, and $d \cong e : \tau\ [\Delta;\Gamma]$ and $d' \cong e' : \tau\ [\Delta;\Gamma]$, then $d \sim d' : \tau\ [\Delta;\Gamma]$.*

Proof. By Lemma 53.2 on page 493 and Corollary 53.5. □

Lemma 53.7 (Compositionality)**.** *Suppose that*

$$e \sim e' : \tau\ [\eta[t \mapsto R] : \delta[t \mapsto \hat{\delta}(\rho)] \leftrightarrow \delta'[t \mapsto \widehat{\delta'}(\rho)]],$$

where $R : \hat{\delta}(\rho) \leftrightarrow \widehat{\delta'}(\rho)$ is such that $R(d, d')$ holds iff $d \sim d' : \rho\ [\eta : \delta \leftrightarrow \delta']$. Then $e \sim e' : [\rho/t]\tau\ [\eta : \delta \leftrightarrow \delta']$.

Proof. By induction on the structure of τ. When $\tau = t$, the result is immediate from the definition of the relation R. When $\tau = t' \neq t$, the result holds vacuously. When $\tau = \tau_1 \to \tau_2$ or $\tau = \forall(u\,.\,\tau)$, where without loss of generality $u \neq t$ and $u \notin \rho$, the result follows by induction. □

Despite the strong conditions on polymorphic types, logical equivalence is not overly restrictive—every expression satisfies its constraints. This result is sometimes called the *parametricity theorem*.

Theorem 53.8 (Parametricity)**.** *If $\Delta;\Gamma \vdash e : \tau$, then $e \sim e : \tau\ [\Delta;\Gamma]$.*

Proof. By rule induction on the statics of $\mathcal{L}\{\to\forall\}$ given by Rules (23.2).

We consider two representative cases here.

Rule (23.2d) 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 ρ, ρ', and $R : \rho \leftrightarrow \rho'$,

$$[\rho/t]\hat{\gamma}(\hat{\delta}(e)) \sim [\rho'/t]\widehat{\gamma'}(\widehat{\delta'}(e)) : \tau\ [\eta_* : \delta_* \leftrightarrow \delta'_*],$$

where $\eta_* = \eta[t \mapsto R]$, $\delta_* = \delta[t \mapsto \rho]$, and $\delta'_* = \delta'[t \mapsto \rho']$. Since

$$\Lambda(t.\hat{\gamma}(\hat{\delta}(e)))[\rho] \mapsto^* [\rho/t]\hat{\gamma}(\hat{\delta}(e))$$

and

$$\Lambda(t.\widehat{\gamma'}(\widehat{\delta'}(e)))[\rho'] \mapsto^* [\rho'/t]\widehat{\gamma'}(\widehat{\delta'}(e)),$$

the result follows by Lemma 53.3 on page 496.

Rule (23.2e) Suppose $\delta : \Delta$, $\delta' : \Delta$, $\eta : \delta \leftrightarrow \delta'$, and $\gamma \sim \gamma' : \Gamma\ [\eta : \delta \leftrightarrow \delta']$. By induction we have

$$\hat{\gamma}(\hat{\delta}(e)) \sim \widehat{\gamma'}(\widehat{\delta'}(e)) : \forall(t.\tau)\ [\eta : \delta \leftrightarrow \delta']$$

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 $d \sim d' : \rho\ [\eta : \delta \leftrightarrow \delta']$. By Corollary 53.5 on the previous page, this relation is admissible.

By the definition of logical equivalence at polymorphic types, we obtain

$$\hat{\gamma}(\hat{\delta}(e))[\hat{\rho}] \sim \widehat{\gamma'}(\widehat{\delta'}(e))[\hat{\rho}'] : \tau\ [\eta[t \mapsto R] : \delta[t \mapsto \hat{\rho}] \leftrightarrow \delta'[t \mapsto \hat{\rho}']].$$

By Lemma 53.7 on the preceding page

$$\hat{\gamma}(\hat{\delta}(e))[\hat{\rho}] \sim \widehat{\gamma'}(\widehat{\delta'}(e))[\hat{\rho}'] : [\rho/t]\tau\ [\eta : \delta \leftrightarrow \delta']$$

But

$$\begin{aligned} \hat{\gamma}(\hat{\delta}(e))[\hat{\rho}] &= \hat{\gamma}(\hat{\delta}(e))[\hat{\delta}(\rho)] && (53.1)\\ &= \hat{\gamma}(\hat{\delta}(e[\rho])), && (53.2) \end{aligned}$$

and similarly

$$\begin{aligned} \widehat{\gamma'}(\widehat{\delta'}(e))[\hat{\rho}'] &= \widehat{\gamma'}(\widehat{\delta'}(e))[\widehat{\delta'}(\rho)] && (53.3)\\ &= \widehat{\gamma'}(\widehat{\delta'}(e[\rho])), && (53.4) \end{aligned}$$

from which the result follows.

□

Corollary 53.9. *If $e \cong e' : \tau\ [\Delta;\Gamma]$, then $e \sim e' : \tau\ [\Delta;\Gamma]$.*

Proof. By Theorem 53.8 on page 497 $e \sim e : \tau\ [\Delta;\Gamma]$, and hence by Corollary 53.6 on page 497, $e \sim e' : \tau\ [\Delta;\Gamma]$. □

Lemma 53.10 (Congruence)**.** *If $e \sim e' : \tau\ [\Delta;\Gamma]$ and $\mathcal{C} : (\Delta;\Gamma \triangleright \tau) \rightsquigarrow (\Delta';\Gamma' \triangleright \tau')$, then $\mathcal{C}\{e\} \sim \mathcal{C}\{e'\} : \tau\ [\Delta';\Gamma']$.*

Proof. By induction on the structure of $\mathcal{C}$, following along very similar lines to the proof of Theorem 53.8 on page 497. □

Lemma 53.11 (Consistency)**.** *Logical equivalence is consistent.*

Proof. Follows immediately from the definition of logical equivalence. □

Corollary 53.12. *If $e \sim e' : \tau\ [\Delta;\Gamma]$, then $e \cong e' : \tau\ [\Delta;\Gamma]$.*

Proof. By Lemma 53.11 Logical equivalence is consistent, and by Lemma 53.10, it is a congruence, and hence is contained in observational equivalence. □

Corollary 53.13. *Logical and observational equivalence coincide.*

Proof. By Corollaries 53.9 and 53.12. □

If $d : \tau$ and $d \mapsto e$, then $d \sim e : \tau$, and hence by Corollary 53.12, $d \cong e : \tau$. 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.

Corollary 53.14 (Extensionality)**.**

1. *$e \cong e' : \tau_1 \to \tau_2$ iff for all $e_1 : \tau_1$, $e(e_1) \cong e'(e_1) : \tau_2$.*

2. *$e \cong e' : \forall(t.\tau)$ iff for all ρ, $e[\rho] \cong e'[\rho] : [\rho/t]\tau$.*

Proof. The forward direction is immediate in both cases, since observational equivalence is a congruence, by definition. The backward direction is proved similarly in both cases, by appeal to Theorem 53.8 on page 497. In the first case, by Corollary 53.13 it suffices to show that $e \sim e' : \tau_1 \to \tau_2$. To this end suppose that $e_1 \sim e_1' : \tau_1$. We are to show that $e(e_1) \sim e'(e_1') : \tau_2$. By the assumption we have $e(e_1') \cong e'(e_1') : \tau_2$. By parametricity we have $e \sim e : \tau_1 \to \tau_2$, and hence $e(e_1) \sim e(e_1') : \tau_2$. The result then follows by Lemma 53.4 on page 496. In the second case, by Corollary 53.13 on the preceding page it is sufficient to show that $e \sim e' : \forall(t.\tau)$. Suppose that $R : \rho \leftrightarrow \rho'$ for some closed types ρ and ρ'. It suffices to show that $e[\rho] \sim e'[\rho'] : \tau\ [\eta : \delta \leftrightarrow \delta']$, where $\eta(t) = R$, $\delta(t) = \rho$, and $\delta'(t) = \rho'$. By the assumption we have $e[\rho'] \cong e'[\rho'] : [\rho/t]\tau$. By parametricity $e \sim e : \forall(t.\tau)$, and hence $e[\rho] \sim e'[\rho'] : \tau\ [\eta : \delta \leftrightarrow \delta']$. The result then follows by Lemma 53.4 on page 496. □

Lemma 53.15 (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 $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta]$ iff $e \cong e' : \hat{\delta}(\tau)$.*

Proof. The backward direction follows immediately from Theorem 53.8 on page 497 and respect for observational equivalence. The forward direction is proved by induction on the structure of τ, appealing to Corollary 53.14 on the previous page to establish observational equivalence at function and polymorphic types. □

53.4 Parametricity Properties

The parametricity theorem enables us to deduce properties of expressions of $\mathcal{L}\{\rightarrow\forall\}$ 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 $\forall(t.t \rightarrow t)$ behaves like the identity function.

Theorem 53.16. *Let $e : \forall(t.t \rightarrow t)$ be arbitrary, and let id be $\Lambda(t.\lambda\,(x{:}t.x))$. Then $e \cong id : \forall(t.t \rightarrow t)$.*

Proof. By Corollary 53.13 on the preceding page it is sufficient to show that $e \sim id : \forall(t.t \rightarrow t)$. Let ρ and ρ' 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*, is to say

$$e[\rho](e_0)\ R\ e_0'.$$

It suffices to show that $e[\rho](e_0) \cong e_0 : \rho$, for then the result follows by the admissibility of R and the assumption $e_0\ R\ e_0'$.

By Theorem 53.8 on page 497 we have $e \sim e : \forall(t\,.\,t \to t)$. Let the relation $S : \rho \leftrightarrow \rho$ be defined by $d\ S\ d'$ iff $d \cong e_0 : \rho$ and $d' \cong e_0 : \rho$. This is clearly admissible, and we have $e_0\ S\ e_0$. It follows that

$$e[\rho](e_0)\ S\ e[\rho](e_0),$$

and so, by the definition of the relation S, $e[\rho](e_0) \cong e_0 : \rho$.

□

In Chapter 23 we showed that product, sum, and natural numbers types are all definable in $\mathcal{L}\{\to\forall\}$. The proof of definability in each case consisted of showing that the type and its associated introduction and elimination forms are encodable in $\mathcal{L}\{\to\forall\}$. 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 $\mathcal{L}\{\to\forall\}$ 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 $\mathcal{L}\{\to\forall\}$, as defined in Chapter 23 by the following equations:

$$\begin{aligned} \mathtt{unit} &= \forall(r\,.\,r \to r) \\ \langle\rangle &= \Lambda(r\,.\,\lambda\,(x{:}r\,.\,x)) \end{aligned}$$

It is easy to see that $\langle\rangle : \mathtt{unit}$ 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 53.16 on the facing page! We say that the type unit is *strongly definable* within $\mathcal{L}\{\to\forall\}$.

Continuing in this vein, let us examine the definition of the binary product type in $\mathcal{L}\{\to\forall\}$, also given in Chapter 23:

$$\begin{aligned} \tau_1 \times \tau_2 &= \forall(r\,.\,(\tau_1 \to \tau_2 \to r) \to r) \\ \langle e_1, e_2\rangle &= \Lambda(r\,.\,\lambda\,(x{:}\tau_1 \to \tau_2 \to r\,.\,x(e_1)(e_2))) \\ e \cdot \mathtt{l} &= e[\tau_1](\lambda\,(x{:}\tau_1\,.\,\lambda\,(y{:}\tau_2\,.\,x))) \\ e \cdot \mathtt{r} &= e[\tau_2](\lambda\,(x{:}\tau_1\,.\,\lambda\,(y{:}\tau_2\,.\,y))) \end{aligned}$$

It is easy to check that $\langle e_1, e_2\rangle \cdot \mathtt{l} \cong e_1 : \tau_1$ and $\langle e_1, e_2\rangle \cdot \mathtt{r} \cong e_2 : \tau_2$ 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 $\mathcal{L}\{\rightarrow\forall\}$. We will make use of a lemma governing the behavior of the elements of the product type whose proof relies on Theorem 53.8 on page 497.

Lemma 53.17. *If $e : \tau_1 \times \tau_2$, then $e \cong \langle e_1, e_2 \rangle : \tau_1 \times \tau_2$ for some $e_1 : \tau_1$ and $e_2 : \tau_2$.*

Proof. Expanding the definitions of pairing and the product type, and applying Corollary 53.13 on page 499, we let ρ and ρ' be arbitrary closed types, and let $R : \rho \leftrightarrow \rho'$ be an admissible relation between them. Suppose further that

$$h \sim h' : \tau_1 \rightarrow \tau_2 \rightarrow t\ [\eta : \delta \leftrightarrow \delta'],$$

where $\eta(t) = R$, $\delta(t) = \rho$, and $\delta'(t) = \rho'$ (and are each undefined on $t' \neq t$). We are to show that for some $e_1 : \tau_1$ and $e_2 : \tau_2$,

$$e\,[\rho]\,(h) \sim h'(e_1)(e_2) : t\ [\eta : \delta \leftrightarrow \delta'],$$

which is to say

$$e\,[\rho]\,(h)\ R\ h'(e_1)(e_2).$$

Now by Theorem 53.8 on page 497 we have $e \sim e : \tau_1 \times \tau_2$. Define the relation $S : \rho \leftrightarrow \rho'$ by $d\ S\ d'$ iff the following conditions are satisfied:

1. $d \cong h(d_1)(d_2) : \rho$ for some $d_1 : \tau_1$ and $d_2 : \tau_2$;
2. $d' \cong h'(d_1')(d_2') : \rho'$ for some $d_1' : \tau_1$ and $d_2' : \tau_2$;
3. $d\ R\ d'$.

This is clearly an admissible relation. Noting that

$$h \sim h' : \tau_1 \rightarrow \tau_2 \rightarrow t\ [\eta' : \delta \leftrightarrow \delta'],$$

where $\eta'(t) = S$ and 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 $e \cdot \mathtt{l} \cong e_1 : \tau_1$ and $e \cdot \mathtt{r} \cong e_2 : \tau_2$. We wish to show that $e \cong \langle e_1, e_2 \rangle : \tau_1 \times \tau_2$. From Lemma 53.17 it is easy to deduce that $e \cong \langle e \cdot \mathtt{l}, e \cdot \mathtt{r} \rangle : \tau_1 \times \tau_2$ by congruence and direct calculation. Hence, by congruence we have $e \cong \langle e_1, e_2 \rangle : \tau_1 \times \tau_2$.

By a similar line of reasoning we may show that the Church encoding of the natural numbers given in Chapter 23 strongly defines the natural numbers in that the following properties hold:

1. $\mathtt{natiter}\,\mathtt{z}\,\{\mathtt{z}\Rightarrow e_0 \mid \mathtt{s}(x)\Rightarrow e_1\} \cong e_0 : \rho$.
2. $\mathtt{natiter}\,\mathtt{s}(e)\,\{\mathtt{z}\Rightarrow e_0 \mid \mathtt{s}(x)\Rightarrow e_1\} \cong [\mathtt{natiter}\,e\,\{\mathtt{z}\Rightarrow e_0 \mid \mathtt{s}(x)\Rightarrow e_1\}/x]e_1 : \rho$.
3. Suppose that $x : \mathtt{nat} \vdash r(x) : \rho$. If
   (a) $r(\mathtt{z}) \cong e_0 : \rho$, and
   (b) $r(\mathtt{s}(e)) \cong [r(e)/x]e_1 : \rho$,

   then for every $e : \mathtt{nat}$, $r(e) \cong \mathtt{natiter}\,e\,\{\mathtt{z}\Rightarrow e_0 \mid \mathtt{s}(x)\Rightarrow e_1\} : \rho$.

The first two equations, which constitute weak definability, are easily established by calculation, using the definitions given in Chapter 23. The third property, the unicity of the iterator, is proved using parametricity by showing that every closed expression of type 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 53.18. *If $e : \mathtt{nat}$, then either $e \cong \mathtt{z} : \mathtt{nat}$, or there exists $e' : \mathtt{nat}$ such that $e \cong \mathtt{s}(e') : \mathtt{nat}$. Consequently, there exists $n \geq 0$ such that $e \cong \overline{n} : \mathtt{nat}$.*

Proof. By Theorem 53.8 on page 497 we have $e \sim e : \mathtt{nat}$. Define the relation $R : \mathtt{nat} \leftrightarrow \mathtt{nat}$ to be the strongest relation such that $d\ R\ d'$ iff either $d \cong \mathtt{z} : \mathtt{nat}$ and $d' \cong \mathtt{z} : \mathtt{nat}$, or $d \cong \mathtt{s}(d_1) : \mathtt{nat}$ and $d' \cong \mathtt{s}(d_1') : \mathtt{nat}$ 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. □

53.5 Representation Independence, Revisited

In Section 24.4 on page 213 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, E, between their representation types that is *preserved* by the operations of the type. The relation E

may be thought of as expressing the "equivalence" of the two representations; checking that each operation preserves E amounts to checking that the result of performing that operation on equivalent representations yields equivalent results.

As an example, we argued in Section 24.4 on page 213 that two implementations of a queue abstraction are similar. In the one the queue is represented by a list of elements in reverse arrival order (the latest element to arise is the head of the list). Enqueueing an element is easy; simply add it to the front of the list. Dequeueing an element requires reversing the list, removing the first element, and reversing the rest to obtain the new queue. In the other the queue is represented by a pair of lists, with the "back half" representing the latest arrivals in reverse order of arrival time, and the "front half" representing the oldest arrivals in order of arrival (the next one to depart the queue is at the head of the list). Enqueueing remains easy; the element is added to the "back half" as the first element. Dequeueing breaks into two cases. If the "front half" is non-empty, simply remove the head element and return the queue consisting of the "back half" as-is together with the tail of the "front half". If, on the other hand, the "front half" is empty, then the queue is reorganizing by reversing the "back half" and making it the new "front half", leaving the new "back half" empty. These two representations of queues are related by the relation E such that $q\,E\,(b, f)$ iff q is b followed by the reversal of f. It is easy to check that the operations of the queue preserve this relationship.

In Chapter 24 we asserted without proof that the existence of such a relation was sufficient to ensure that the behavior of any client is insensitive to the choice of either implementation. The proof of this intuitively plausible result relies on parametricity. One way to explain this is via the definition of existential types in $\mathcal{L}\{\rightarrow\forall\}$ described in Section 24.3 on page 212. According to that definition, the client, e, of an abstract type $\exists(t.\tau)$ is a polymorphic function of type $\forall(t.\tau \rightarrow \sigma)$, where σ, the result type of the computation, does not involve the type variable t. Being polymorphic, the client enjoys the parametricity property given by Theorem 53.8 on page 497. Specifically, suppose that ρ_1 and ρ_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, ρ_1 is the type of lists of elements of the queue, ρ_2 is the type of a pair of lists of elements, and R is the relation E 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

$$e_1 \sim e_2 : \tau\;[\eta : \delta_1 \leftrightarrow \delta_2], \tag{53.5}$$

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 (53.5) states that the two implementations are similar in that they preserve the relation R between the representation types. By Theorem 53.8 on page 497 it follows that the client, e, satisfies

$$e \sim e : \sigma \; [\eta : \delta_1 \leftrightarrow \delta_2].$$

But since σ is a closed type (in particular, does not involve t), this is equivalent to

$$e \sim e : \sigma \; [\emptyset : \emptyset \leftrightarrow \emptyset].$$

But then by Lemma 53.15 on page 500 we have

$$e\,[\rho_1]\,(e_1) \cong e\,[\rho_2]\,(e_2) : \sigma.$$

That is, the client behavior is not affected by the change of representation.

53.6 Exercises

Part XX

Appendices

Appendix A

Mathematical Preliminaries

A.1 Finite Sets and Maps

A.2 Families of Sets

History of versions

Version	Date	Author(s)	Changes
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.