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.
# Contents

Preface

## Judgements and Rules

### 1 Inductive Definitions

1.1 Objects and Judgements                          3
1.2 Inference Rules                                 4
1.3 Derivations                                    6
1.4 Rule Induction                                 7
1.5 Iterated and Simultaneous Inductive Definitions 10
1.6 Defining Functions by Rules                    12
1.7 Modes                                         13
1.8 Foundations                                    14
1.9 Exercises                                      15

## Hypothetical Judgements

2.1 Derivability                                  17
2.2 Admissibility                                20
2.3 Hypothetical Inductive Definitions            22
2.4 Exercises                                    23

## Parametric Judgements

3.1 Parameters and Objects                        25
3.2 Rule Schemes                                  26
3.3 Parametric Derivability                       27
3.4 Parametric Inductive Definitions              28
3.5 Exercises                                    30
4 Transition Systems 31
  4.1 Transition Systems ........................................... 31
  4.2 Iterated Transition ........................................ 32
  4.3 Simulation and Bisimulation ................................. 33
  4.4 Exercises .................................................... 34

II Levels of Syntax 35

5 Concrete Syntax 37
  5.1 Strings Over An Alphabet .................................... 37
  5.2 Lexical Structure ............................................ 38
  5.3 Context-Free Grammars ...................................... 42
  5.4 Grammatical Structure ..................................... 43
  5.5 Ambiguity .................................................... 45
  5.6 Exercises .................................................... 46

6 Abstract Syntax Trees 47
  6.1 Abstract Syntax Trees ....................................... 47
  6.2 Variables and Substitution ................................ 48
  6.3 Exercises .................................................... 51

7 Binding and Scope 53
  7.1 Abstract Binding Trees .................................... 54
  7.1.1 Structural Induction With Binding and Scope ..... 56
  7.1.2 Apartness ................................................ 57
  7.1.3 Renaming of Bound Parameters ......................... 57
  7.1.4 Substitution ............................................. 58
  7.2 Exercises .................................................... 59

8 Parsing 61
  8.1 Parsing Into Abstract Syntax Trees ....................... 61
  8.2 Parsing Into Abstract Binding Trees ..................... 64
  8.3 Syntactic Conventions .................................... 66
  8.4 Exercises .................................................... 66

III Static and Dynamic Semantics 67

9 Static Semantics 69
  9.1 Type System ................................................ 69
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.2</td>
<td>Structural Properties</td>
<td>72</td>
</tr>
<tr>
<td>9.3</td>
<td>Exercises</td>
<td>74</td>
</tr>
<tr>
<td>10</td>
<td>Dynamic Semantics</td>
<td>75</td>
</tr>
<tr>
<td>10.1</td>
<td>Structural Semantics</td>
<td>75</td>
</tr>
<tr>
<td>10.2</td>
<td>Contextual Semantics</td>
<td>78</td>
</tr>
<tr>
<td>10.3</td>
<td>Equational Semantics</td>
<td>80</td>
</tr>
<tr>
<td>10.4</td>
<td>Exercises</td>
<td>84</td>
</tr>
<tr>
<td>11</td>
<td>Type Safety</td>
<td>85</td>
</tr>
<tr>
<td>11.1</td>
<td>Preservation</td>
<td>86</td>
</tr>
<tr>
<td>11.2</td>
<td>Progress</td>
<td>86</td>
</tr>
<tr>
<td>11.3</td>
<td>Run-Time Errors</td>
<td>88</td>
</tr>
<tr>
<td>11.4</td>
<td>Exercises</td>
<td>90</td>
</tr>
<tr>
<td>12</td>
<td>Evaluation Semantics</td>
<td>91</td>
</tr>
<tr>
<td>12.1</td>
<td>Evaluation Semantics</td>
<td>91</td>
</tr>
<tr>
<td>12.2</td>
<td>Relating Transition and Evaluation Semantics</td>
<td>92</td>
</tr>
<tr>
<td>12.3</td>
<td>Type Safety, Revisited</td>
<td>94</td>
</tr>
<tr>
<td>12.4</td>
<td>Cost Semantics</td>
<td>95</td>
</tr>
<tr>
<td>12.5</td>
<td>Environment Semantics</td>
<td>96</td>
</tr>
<tr>
<td>12.6</td>
<td>Exercises</td>
<td>97</td>
</tr>
<tr>
<td>IV</td>
<td>Function Types</td>
<td>99</td>
</tr>
<tr>
<td>13</td>
<td>Function Definitions and Values</td>
<td>101</td>
</tr>
<tr>
<td>13.1</td>
<td>First-Order Functions</td>
<td>102</td>
</tr>
<tr>
<td>13.2</td>
<td>Higher-Order Functions</td>
<td>103</td>
</tr>
<tr>
<td>13.3</td>
<td>Evaluation Semantics and Definitional Equivalence</td>
<td>105</td>
</tr>
<tr>
<td>13.4</td>
<td>Dynamic Scope</td>
<td>107</td>
</tr>
<tr>
<td>13.5</td>
<td>Exercises</td>
<td>109</td>
</tr>
<tr>
<td>14</td>
<td>Gödel’s System T</td>
<td>111</td>
</tr>
<tr>
<td>14.1</td>
<td>Statics</td>
<td>111</td>
</tr>
<tr>
<td>14.2</td>
<td>Dynamics</td>
<td>113</td>
</tr>
<tr>
<td>14.3</td>
<td>Definability</td>
<td>114</td>
</tr>
<tr>
<td>14.4</td>
<td>Non-Definability</td>
<td>116</td>
</tr>
<tr>
<td>14.5</td>
<td>Exercises</td>
<td>118</td>
</tr>
</tbody>
</table>

**October 16, 2009**  
**Draft**  
**18:42**
## CONTENTS

### 15 Plotkin’s PCF
- 15.1 Statics .............................................. 121
- 15.2 Dynamics ........................................... 122
- 15.3 Definability ........................................ 124
- 15.4 Co-Natural Numbers .......................... 126
- 15.5 Exercises .......................................... 126

### V Finite Data Types

#### 16 Product Types
- 16.1 Nullary and Binary Products ............ 129
- 16.2 Finite Products ................................. 131
- 16.3 Primitive and Mutual Recursion .......... 133
- 16.4 Exercises ........................................ 134

#### 17 Sum Types
- 17.1 Binary and Nullary Sums ................. 135
- 17.2 Finite Sums .................................... 137
- 17.3 Uses for Sum Types ......................... 139
  - 17.3.1 Void and Unit ............................ 139
  - 17.3.2 Booleans .................................. 139
  - 17.3.3 Enumerations ......................... 140
  - 17.3.4 Options .................................. 140
- 17.4 Exercises ...................................... 142

#### 18 Pattern Matching
- 18.1 A Pattern Language ......................... 144
- 18.2 Statics .......................................... 145
- 18.3 Dynamics ...................................... 146
- 18.4 Exhaustiveness and Redundancy .......... 149
- 18.5 Exercises ...................................... 152

### VI Infinite Data Types

#### 19 Inductive and Co-Inductive Types
- 19.1 Motivating Examples ....................... 155
- 19.2 Generic Programming ....................... 158
  - 19.2.1 Positive Type Operators .............. 159
  - 19.2.2 Action of a Positive Type Operator 160

---

18:42 DRAFT October 16, 2009
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.3</td>
<td>Static Semantics</td>
<td>161</td>
</tr>
<tr>
<td>19.3.1</td>
<td>Types</td>
<td>162</td>
</tr>
<tr>
<td>19.3.2</td>
<td>Expressions</td>
<td>163</td>
</tr>
<tr>
<td>19.4</td>
<td>Dynamic Semantics</td>
<td>163</td>
</tr>
<tr>
<td>19.5</td>
<td>Exercises</td>
<td>164</td>
</tr>
<tr>
<td>20</td>
<td>General Recursive Types</td>
<td>165</td>
</tr>
<tr>
<td>20.1</td>
<td>Solving Type Isomorphisms</td>
<td>166</td>
</tr>
<tr>
<td>20.2</td>
<td>Recursive Data Structures</td>
<td>168</td>
</tr>
<tr>
<td>20.3</td>
<td>Self-Reference</td>
<td>169</td>
</tr>
<tr>
<td>20.4</td>
<td>Exercises</td>
<td>171</td>
</tr>
</tbody>
</table>

#### VII Dynamic Types

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>The Untyped $\lambda$-Calculus</td>
<td>175</td>
</tr>
<tr>
<td>21.1</td>
<td>The $\lambda$-Calculus</td>
<td>175</td>
</tr>
<tr>
<td>21.2</td>
<td>Definability</td>
<td>177</td>
</tr>
<tr>
<td>21.3</td>
<td>Scott’s Theorem</td>
<td>180</td>
</tr>
<tr>
<td>21.4</td>
<td>Untyped Means Uni-Typed</td>
<td>181</td>
</tr>
<tr>
<td>21.5</td>
<td>Exercises</td>
<td>183</td>
</tr>
<tr>
<td>22</td>
<td>Dynamic Typing</td>
<td>185</td>
</tr>
<tr>
<td>22.1</td>
<td>Dynamically Typed PCF</td>
<td>185</td>
</tr>
<tr>
<td>22.2</td>
<td>Critique of Dynamic Typing</td>
<td>189</td>
</tr>
<tr>
<td>22.3</td>
<td>Hybrid Typing</td>
<td>190</td>
</tr>
<tr>
<td>22.4</td>
<td>Optimization of Dynamic Typing</td>
<td>192</td>
</tr>
<tr>
<td>22.5</td>
<td>Static “Versus” Dynamic Typing</td>
<td>194</td>
</tr>
<tr>
<td>22.6</td>
<td>Dynamic Typing From Recursive Types</td>
<td>195</td>
</tr>
<tr>
<td>22.7</td>
<td>Exercises</td>
<td>196</td>
</tr>
</tbody>
</table>

#### VIII Variable Types

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>Girard’s System F</td>
<td>199</td>
</tr>
<tr>
<td>23.1</td>
<td>System F</td>
<td>200</td>
</tr>
<tr>
<td>23.2</td>
<td>Polymorphic Definability</td>
<td>203</td>
</tr>
<tr>
<td>23.2.1</td>
<td>Products and Sums</td>
<td>204</td>
</tr>
<tr>
<td>23.2.2</td>
<td>Natural Numbers</td>
<td>205</td>
</tr>
<tr>
<td>23.3</td>
<td>Parametricity</td>
<td>206</td>
</tr>
<tr>
<td>23.4</td>
<td>Restricted Forms of Polymorphism</td>
<td>207</td>
</tr>
</tbody>
</table>
## CONTENTS

### 29 Continuations 251
- 29.1 Informal Overview .................................................. 251
- 29.2 Semantics of Continuations ........................................ 253
- 29.3 Coroutines .............................................................. 255
- 29.4 Exercises ............................................................... 259

### X Types and Propositions 261

#### 30 Constructive Logic 263
- 30.1 Constructive Semantics ............................................... 264
- 30.2 Constructive Logic ...................................................... 265
  - 30.2.1 Rules of Provability ............................................... 266
  - 30.2.2 Rules of Proof ...................................................... 268
- 30.3 Propositions as Types ................................................ 269
- 30.4 Exercises ............................................................... 271

#### 31 Classical Logic 273
- 31.1 Classical Logic ........................................................ 274
- 31.2 Deriving Elimination Forms ......................................... 279
- 31.3 Dynamics of Proofs .................................................... 280
- 31.4 Exercises ............................................................... 281

### XI Subtyping 283

#### 32 Subtyping 285
- 32.1 Subsumption ............................................................ 286
- 32.2 Varieties of Subtyping ................................................. 286
  - 32.2.1 Numeric Types ....................................................... 286
  - 32.2.2 Product Types ....................................................... 287
  - 32.2.3 Sum Types ........................................................... 289
- 32.3 Variance ................................................................. 290
  - 32.3.1 Product Types ....................................................... 290
  - 32.3.2 Sum Types ........................................................... 290
  - 32.3.3 Function Types ...................................................... 291
  - 32.3.4 Recursive Types .................................................... 292
- 32.4 Safety for Subtyping ................................................ 294
- 32.5 Exercises ............................................................... 297

OCTOBER 16, 2009 DRAFT 18:42
33 Singleton and Dependent Kinds 299
  33.1 Informal Overview ............................................ 300

XII Symbols 303

34 Symbols 305
  34.1 Statics .......................................................... 305
  34.2 Scoped Dynamics ............................................... 307
  34.3 Unscoped Dynamics ............................................ 308
  34.4 Safety ............................................................ 309
  34.5 Exercises ........................................................ 310

35 Fluid Binding 311
  35.1 Statics .......................................................... 311
  35.2 Dynamics ........................................................ 312
  35.3 Type Safety ....................................................... 314
  35.4 Subtleties of Fluid Binding .................................... 315
  35.5 Dynamic Fluids ............................................... 317
  35.6 Exercises ........................................................ 318

36 Dynamic Classification 319
  36.1 Statics .......................................................... 320
  36.2 Dynamics ........................................................ 321
  36.3 Defining Classification ........................................ 322
  36.4 Exercises ........................................................ 323

XIII Storage Effects 325

37 Reynolds's IA 327
  37.1 Commands ......................................................... 328
    37.1.1 Statics ....................................................... 329
    37.1.2 Dynamics ................................................... 330
    37.1.3 Safety ....................................................... 331
  37.2 Some Programming Idioms ..................................... 333
  37.3 References to Assignables ..................................... 334
  37.4 Typed Commands and Assignables .............................. 336
  37.5 Exercises ........................................................ 337
## CONTENTS

### 38 Mutable Cells 339

38.1 Modal Formulation 341
   38.1.1 Syntax 341
   38.1.2 Statics 342
   38.1.3 Dynamics 343

38.2 Integral Formulation 344
   38.2.1 Statics 345
   38.2.2 Dynamics 345

38.3 Safety 346

38.4 Integral versus Modal Formulation 348

38.5 Exercises 350

### XIV Laziness 351

### 39 Eagerness and Laziness 353

39.1 Eager and Lazy Dynamics 353

39.2 Eager and Lazy Types 356

39.3 Self-Reference 357

39.4 Suspension Type 358

39.5 Exercises 360

### 40 Lazy Evaluation 361

40.1 Need Dynamics 362

40.2 Safety 365

40.3 Lazy Data Structures 367

40.4 Suspensions By Need 369

40.5 Exercises 369

### XV Parallelism 371

### 41 Speculation 373

41.1 Speculative Evaluation 373

41.2 Speculative Parallelism 374

41.3 Exercises 376

### 42 Work-Efficient Parallelism 377

42.1 Nested Parallelism 377

42.2 Cost Semantics 380

42.3 Vector Parallelism 384

October 16, 2009 Draft 18:42
CONTENTS

XVIII  Modalities  429

48  Monads  431
  48.1  The Lax Modality  432
  48.2  Exceptions  433
  48.3  Derived Forms  435
  48.4  Monadic Programming  436
  48.5  Exercises  438

49  Comonads  439
  49.1  A Comonadic Framework  440
  49.2  Comonadic Effects  443
    49.2.1  Exceptions  443
    49.2.2  Fluid Binding  445
  49.3  Exercises  447

XIX  Equivalence  449

50  Equational Reasoning for T  451
  50.1  Observational Equivalence  452
  50.2  Extensional Equivalence  456
  50.3  Extensional and Observational Equivalence Coincide  457
  50.4  Some Laws of Equivalence  460
    50.4.1  General Laws  461
    50.4.2  Extensionality Laws  461
    50.4.3  Induction Law  462
  50.5  Exercises  462

51  Equational Reasoning for PCF  463
  51.1  Observational Equivalence  463
  51.2  Extensional Equivalence  464
  51.3  Extensional and Observational Equivalence Coincide  465
  51.4  Compactness  468
  51.5  Co-Natural Numbers  471
  51.6  Exercises  473

52  Parametricity  475
  52.1  Overview  475
  52.2  Observational Equivalence  476
  52.3  Logical Equivalence  478

OCTOBER 16, 2009  DRAFT  18:42
xxi CONTENTS

52.4 Parametricity Properties .......................... 484
52.5 Exercises ......................................... 487

XX Working Drafts of Chapters 489

A Polarization 491
A.1 Polarization ........................................ 492
A.2 Focusing ........................................... 493
A.3 Statics .............................................. 494
A.4 Dynamics .......................................... 497
A.5 Safety ............................................. 498
A.6 Definability ........................................ 499
A.7 Exercises .......................................... 499
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 Objects and Judgements

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

- \( n \text{ nat} \)  
  - \( n \) is a natural number
- \( n = n_1 + n_2 \)  
  - \( n \) is the sum of \( n_1 \) and \( n_2 \)
- \( a \text{ ast} \)  
  - \( a \) is an abstract syntax tree
- \( \tau \text{ type} \)  
  - \( \tau \) is a type
- \( e : \tau \)  
  - expression \( e \) has type \( \tau \)
- \( e \Downarrow v \)  
  - expression \( e \) has value \( v \)

A judgement states that one or more 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 objects constituting an instance are its subjects.

We will use the meta-variable \( P \) to stand for an unspecified judgement form, and the meta-variables \( a, b, \) and \( c \) to stand for unspecified objects. We write \( a P \) for the judgement asserting that \( P \) holds of \( a \). When it is not 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.

We are being intentionally vague about the universe of objects that may be involved in an inductive definition. The rough-and-ready rule is that any sort of finite construction of objects from other objects is permissible. In particular, we shall make frequent use of the construction of composite objects of the form \( o(a_1, \ldots, a_n) \), where \( a_1, \ldots, a_n \) are objects and \( o \) is an \( n \)-argument operator. This construction includes as a special case the formation of \( n \)-tuples, \((a_1, \ldots, a_n)\), in which the tupling operator is left implicit. (In Chapters 6 and 7 we will formalize these and richer forms of objects, called abstract syntax trees.)

### 1.2 Inference Rules

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

\[
\begin{array}{c}
J_1 \\
\vdots
\end{array} \\
\begin{array}{c}
J_k \\
J
\end{array}
\]  

(1.1)

in which \( J \) and \( J_1, \ldots, J_k \) 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 \( J_1, \ldots, J_k \). When \( k \) is zero, a rule states that its conclusion holds unconditionally. Bear in mind that there may be, in general, many rules with the same conclusion, each specifying sufficient conditions for the conclusion. Consequently, if the conclusion of a rule holds, then it is not necessary that the premises hold, for it might have been derived by another rule.

For example, the following rules constitute an inductive definition of the judgement \( a \text{ nat} \):

\[
\begin{array}{c}
\text{zero nat} \\
\hline
a \text{ nat}
\end{array}
\]  

(1.2a)

\[
\begin{array}{c}
\text{succ}(a) \text{ nat} \\
\hline
a \text{ nat}
\end{array}
\]  

(1.2b)

These rules specify that \( a \text{ nat} \) holds whenever either \( a \) is \( \text{zero} \), or \( a \) is \( \text{succ}(b) \) where \( b \text{ nat} \). Taking these rules to be exhaustive, it follows that
1.2 Inference Rules

\( a \text{ nat iff } a \) is a natural number written in unary.

Similarly, the following rules constitute an inductive definition of the judgement \( a \text{ tree} \):

1.3a)

\[
\begin{array}{c}
\text{empty tree} \\
\hline
\end{array}
\]

1.3b)

\[
\begin{array}{c}
a_1 \text{ tree} \quad a_2 \text{ tree} \\
\hline
\text{node}(a_1; a_2) \text{ tree}
\end{array}
\]

These rules specify that \( a \) tree holds if either \( a \) is \( \text{empty} \), or \( a \) is \( \text{node}(a_1; a_2) \), 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 \( a = b \text{ nat} \) defining equality of \( a \text{ nat} \) and \( b \text{ nat} \) is inductively defined by the following rules:

1.4a)

\[
\begin{array}{c}
\text{zero} = \text{zero nat}
\end{array}
\]

1.4b)

\[
\begin{array}{c}
a = b \text{ nat} \\
\hline
\text{succ}(a) = \text{succ}(b) \text{ nat}
\end{array}
\]

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 object, \( a \), or is instead intended as a rule scheme specifying a rule for each choice of objects in the rule. (In Chapter 3 we will remove this ambiguity by introducing parameterization of rules by objects.)

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

\[
\frac{I_1 \ldots I_k}{J}
\]

is an inference rule and \( \nabla_1, \ldots, \nabla_k \) are derivations of its premises, then

\[
\frac{\nabla_1 \ldots \nabla_k}{J}
\]

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

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

\[
\begin{array}{c}
\frac{\text{zero nat}}{\text{succ(zero) nat}} \\
\frac{\text{succ(succ(zero)) nat}}{\text{succ(succ(succ(zero))) nat}}
\end{array}
\]

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

\[
\begin{array}{c}
\frac{\text{empty tree}}{\text{node(empty; empty) tree}} \\
\frac{\text{empty tree}}{\text{empty tree}} \\
\frac{\text{empty tree}}{\text{node(node(empty; empty); empty) tree}}
\end{array}
\]

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 $P$ holds of a judgement $J$ whenever $J$ is derivable, it is enough to show that $P$ is closed under, or respects, the rules defining $J$. Writing $P(J)$ to mean that the prop-
1.4 Rule Induction

Property \( P \) holds of the judgement \( J \), we say that \( P \) respects the rule

\[
\frac{J_1 \ldots J_k}{J}
\]

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

In practice the premises and conclusion of the rule involve objects that are universally quantified in the inductive step corresponding to that rule. Thus to show that a property \( P \) is closed under a rule of the form

\[
\frac{a_1 J_1 \ldots a_k J_k}{a J}
\]

we must show that for every \( a, a_1, \ldots, a_k \), if \( P(a_1 J_1), \ldots, P(a_k J_k) \), then \( P(a J) \).

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 is both (a) closed under those rules, and (b) sufficient for any other property also closed under those rules. The former property 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.

If \( P(J) \) is closed under a set of rules defining a judgement form, then so is the conjunction of \( P \) with the judgement itself. This means that when showing \( P \) to be closed under a rule, we may inductively assume not only that \( P(J_i) \) holds for each of the premises \( J_i \), but also that \( J_i \) itself holds as well. We shall generally take advantage of this without explicit mentioning that we are doing so.

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

1. \( P(\text{zero nat}) \).
2. for every \( a \), if \( P(a \text{ nat}) \), then \( P(\text{succ}(a) \text{ nat}) \).

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 \( P(a \text{ tree}) \) whenever \( a \text{ tree} \), it is enough to show

18:42 DRAFT October 16, 2009
1.4 Rule Induction

1. $P(\text{empty tree}).$

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

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 $a \text{ nat}$, then $a = a \text{ nat}$.

Proof. By rule induction on Rules (1.2):

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

Rule (1.2b) Assume that $a = a \text{ nat}$. It follows that $\text{succ}(a) = \text{succ}(a) \text{ nat}$ 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 $\text{succ}(a) \text{ nat}$, then $a \text{ nat}$.

Proof. It is instructive to re-state the lemma in a form more suitable for inductive proof: if $b \text{ nat}$ and $b$ is $\text{succ}(a)$ for some $a$, then $a \text{ nat}$. We proceed by rule induction on Rules (1.2).

Rule (1.2a) Vacuously true, since $\text{zero}$ is not of the form $\text{succ}(\_)$.

Rule (1.2b) We have that $b$ is $\text{succ}(b')$, and we may assume both that the lemma holds for $b'$ and that $b' \text{ nat}$. The result follows directly, since if $\text{succ}(b') = \text{succ}(a)$ for some $a$, then $a$ is $b'$.

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

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

October 16, 2009 Draft 18:42
1.5 Iterated and Simultaneous Inductive Definitions

Proof. It is instructive to re-state the lemma in a form more directly amenable to proof by rule induction. We are to show that if $b_1 = b_2 \text{ nat}$ then if $b_1$ is $\text{succ}(a_1)$ and $b_2$ is $\text{succ}(a_2)$, then $a_1 = a_2 \text{ nat}$. We proceed by rule induction on Rules (1.4):

**Rule (1.4a)** Vacuously true, since $\text{zero}$ is not of the form $\text{succ}(-)$.

**Rule (1.4b)** Assuming the result for $b_1 = b_2 \text{ nat}$, and hence that the premise $b_1 = b_2 \text{ nat}$ holds as well, we are to show that if $\text{succ}(b_1)$ is $\text{succ}(a_1)$ and $\text{succ}(b_2)$ is $\text{succ}(a_2)$, then $a_1 = a_2 \text{ nat}$. Under these assumptions we have $b_1 = a_1$ and $b_2 = a_2$, and so $a_1 = a_2 \text{ nat}$ is just the premise of the rule. (We make no use of the inductive hypothesis to complete this step of the proof.)

Both proofs rely on some natural assumptions about the universe of objects; see Section 1.8 on page 14 for further discussion.

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

$$
\frac{I_1 \quad \cdots \quad I_k}{J}
$$

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

$$
\frac{}{\text{nil list}} \quad (1.8a)
$$

$$
\frac{a \text{ nat} \quad b \text{ list}}{\text{cons}(a; b) \text{ list}} \quad (1.8b)
$$

The first premise of Rule (1.8b) is an instance of the judgement form $a \text{ nat}$, which was defined previously, whereas the premise $b \text{ list}$ 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 allows 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:

\[
\begin{align*}
\text{zero} &\quad \text{even} & (1.9a) \\
\frac{a \text{ odd}}{\text{succ}(a) \text{ even}} & (1.9b) \\
\frac{a \text{ even}}{\text{succ}(a) \text{ odd}} & (1.9c)
\end{align*}
\]

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

1. $P(\text{zero even})$;
2. If $P(a \text{ odd})$, then $P(\text{succ}(a) \text{ even})$;
3. If $P(a \text{ even})$, then $P(\text{succ}(a) \text{ odd})$.

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 $P$ by (1) $P(a \text{ even})$ iff $a$ nat, and (2) $P(a \text{ odd})$ iff $a$ nat. The principle of rule induction for Rules (1.9) states that it is sufficient to show the following facts:

1. $\text{zero nat}$, which is derivable by Rule (1.2a).
2. If $a$ nat, then $\text{succ}(a)$ nat, which is derivable by Rule (1.2b).
3. If $a$ nat, then $\text{succ}(a)$ 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 \( \text{sum}(a; b; c) \), with the intended meaning that \( c \) is the sum of \( a \) and \( b \), as follows:

\[
\frac{b \text{ nat}}{\text{sum}(\text{zero}; b; b)} \quad (1.10a)
\]

\[
\frac{\text{sum}(a; b; c)}{\text{sum}(\text{succ}(a); b; \text{succ}(c))} \quad (1.10b)
\]

The rules define a ternary (three-place) relation, \( \text{sum}(a; b; c) \), 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 \( a \text{ nat} \) and \( b \text{ nat} \), there exists a unique \( c \text{ nat} \) such that \( \text{sum}(a; b; c) \).

**Proof.** The proof decomposes into two parts:

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

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

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

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

**Rule (1.2b)** Assuming \( P(a \text{ nat}) \), we are to show \( P(\text{succ}(a) \text{ nat}) \). That is, we assume that if \( b \text{ nat} \) then there exists \( c \text{ nat} \) such that \( \text{sum}(a; b; c) \), and are to show that if \( b' \text{ nat} \), then there exists \( c' \text{ nat} \) such that \( \text{sum}(\text{succ}(a); b'; c') \). To this end, suppose that \( b' \text{ nat} \). Then by induction there exists \( c \text{ nat} \) such that \( \text{sum}(a; b'; c) \). Taking \( c' = \text{succ}(c) \), and applying Rule (1.10b), we obtain \( \text{sum}(\text{succ}(a); b'; c') \), as required.

For uniqueness, we prove that if \( \text{sum}(a; b; c_1), \text{then if sum}(a; b; c_2), \text{then c_1 = c_2 nat} \) by rule induction based on Rules (1.10).
1.7 Modes

Rule (1.10a) We have $a = \text{zero}$ and $c_1 = b$. By an inner induction on the same rules, we may show that if $\text{sum}(\text{zero}; b; c_2)$, then $c_2$ is $b$. By Lemma 1.1 on page 9 we obtain $b = b \text{ nat}$.

Rule (1.10b) We have that $a = \text{succ}(a')$ and $c_1 = \text{succ}(c'_1)$, where $\text{sum}(a'; b; c'_1)$. By an inner induction on the same rules, we may show that if $\text{sum}(a; b; c_2)$, then $c_2 = \text{succ}(c'_2) \text{ nat}$ where $\text{sum}(a'; b; c'_2)$. By the outer inductive hypothesis $c'_1 = c'_2 \text{ nat}$ and so $c_1 = c_2 \text{ nat}$.

\[\square\]

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 $\text{sum}(a; b; c)$ has mode $(\forall, \forall, \exists)$. The notation arises from the form of the proposition it expresses: for all $a \text{ nat}$ and for all $b \text{ nat}$, there exists $c \text{ nat}$ such that $\text{sum}(a; b; c)$. If we wish to further specify that $c$ is uniquely determined by $a$ and $b$, we would say that the judgement $\text{sum}(a; b; c)$ has mode $(\forall, \forall, \exists!)$, corresponding to the proposition for all $a \text{ nat}$ and for all $b \text{ nat}$, there exists a unique $c \text{ nat}$ such that $\text{sum}(a; b; c)$. If we wish only to specify that the sum is unique, if it exists, then we would say that the addition judgement has mode $(\forall, \forall, \exists^1)$, corresponding to the proposition for all $a \text{ nat}$ and for all $b \text{ nat}$ there exists at most one $c \text{ nat}$ such that $\text{sum}(a; b; c)$.

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 $(\forall, 3 \leq 1, \forall)$, 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 $(3 \leq 1, \forall, \forall)$, which is just another way of stating that addition of natural numbers has a partial inverse.
1.8 Foundations

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:

\[
\begin{align*}
\frac{a \text{ nat}}{a + \text{zero} = a \text{ nat}} \\
\frac{a + b = c \text{ nat}}{a + \text{succ}(b) = \text{succ}(c) \text{ nat}}
\end{align*}
\]  

(1.11a)

(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 \(a + b = c \text{ nat}\).

1.8 Foundations

An inductively judgement form, such as \(a \text{ nat}\), may be seen as isolating a class of objects, \(a\), satisfying criteria specified by a collection of rules. While intuitively clear, this description is vague in that it does not specify what sorts of things may appear as the subjects of a judgement. Just what is \(a\)? And what, exactly, are the objects \(\text{zero}\) and \(\text{succ}(a)\) used in the definition of the judgement \(a \text{ nat}\)? More generally, what sorts of objects are permissible in an inductive definition?

One answer to these questions is to fix in advance a particular set, \(\mathcal{U}\), to serve as the universe of discourse over which all judgements are defined. The universe must be rich enough to contain all objects of interest, and must be specified clearly enough to avoid concerns about its existence. Standard practice is to define \(\mathcal{U}\) to be a particular set that can be shown to exist using the standard axioms of set theory, and to specify how the various objects of interest are constructed as elements of this set.

But what should we demand of \(\mathcal{U}\) to serve as a suitable universe of discourse? At the very least it should include labeled finitary trees, which are trees of finite height each of whose nodes has finitely many children and is labeled with an operator drawn from some infinite set. An object such as \(\text{succ} (\text{succ}(\text{zero}))\) is a finitary tree with nodes labeled \(\text{zero}\) having no children and nodes labeled \(\text{succ}\) having one child. Similarly, a finite tuple \((a_1, \ldots, a_n)\) may be thought of as a tree whose node is labeled by an \(n\)-tuple operator. Finitary trees will suffice for our work, but it is common
to consider also regular trees, which are finitary trees in which a child of a node may also be an ancestor of it, and infinitary trees, which admit nodes with infinitely many children.

The standard way to show that the universe, \( \mathcal{U} \), exists (that is, is properly defined) is to construct it explicitly from the axioms of set theory. This requires that we fix the representation of trees as particular sets, using well-known, but notoriously unenlightening, methods.\(^1\) Instead we shall simply take it as given that this can be done, and take \( \mathcal{U} \) to be a suitably rich universe including at least the finitary trees. In particular we assume that \( \mathcal{U} \) comes equipped with operations that allow us to construct finitary trees as elements of \( \mathcal{U} \), and to deconstruct such elements of \( \mathcal{U} \) into an operator and finitely many children.

The advantage of working within set theory is that it settles any worries about the existence of the universe, \( \mathcal{U} \). However, it is important to keep in mind that accepting the axioms of set theory is far more dubious, foundationally speaking, than just accepting the existence of finitary trees without recourse to encoding them as sets. Moreover, there is a significant disadvantage to working with sets, namely that abstract sets have no intrinsic computational content, and hence are of no use to implementation. Yet it is intuitively clear that finitary trees can be readily implemented on a computer by means that have nothing to do with their set-theoretic encodings. Thus we are better off just taking \( \mathcal{U} \) as our starting point, from both a foundational and computational perspective.

### 1.9 Exercises

1. Give an inductive definition of the judgement \( \text{max}(a; b; c) \), where \( a \text{ nat} \), \( b \text{ nat} \), and \( c \text{ nat} \), 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 \( \text{hgt}(a; b) \).

\[
\begin{align*}
\text{hgt}(\text{empty}; \text{zero}) & \\
\text{hgt}(a_1; b_1) & \quad \text{hgt}(a_2; b_2) & \quad \text{max}(b_1; b_2; b) \\
\hline
\text{hgt}(\text{node}(a_1; a_2); \text{succ}(b)) & \\
\end{align*}
\]

\(^1\)Perhaps you have seen the definition of the natural number 0 as the empty set, \( \emptyset \), and the number \( n + 1 \) as the set \( n \cup \{ n \} \), or the definition of the ordered pair \( \langle a, b \rangle \) as the set \( \{ a, \{ a, b \} \} \). Similar coding tricks can be used to represent any finitary tree.
1.9 Exercises

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

3. Give an inductive definition of the judgement “$\nabla$ 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 categorical judgement is an unconditional assertion about some object of the universe. The inductively defined judgements given in Chapter 1 are all categorical. 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 not only categorical, but also 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 categorical, 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

$$
\frac{J_1 \quad \cdots \quad J_k}{J_i}
$$
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 \(\Gamma\) or \(\Delta\), 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 \(\Gamma\). 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 \(\Gamma\). The derivability judgement \(J_1, \ldots, J_n \vdash_{\mathcal{R}} J\) is sometimes expressed by saying that the rule

\[
\frac{J_1 \quad \ldots \quad J_n}{J}
\]

(2.1)

is derivable from the rules \(\mathcal{R}\).

For example, consider the derivability judgement

\[
a \text{nat} \vdash_{(1.2)} \text{succ(succ(a)) nat}
\]

(2.2)

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

\[
\frac{a \text{nat}}{\text{succ}(a) \text{ nat}} \quad \frac{\text{succ}(a) \text{ nat}}{\text{succ(succ(a)) nat}},
\]

(2.3)

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

\[
\frac{a \text{ nat}}{\text{succ(succ(a)) nat}}
\]

(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** (Uniformity). If \(\Gamma \vdash_{\mathcal{R}} J\), then \(\Gamma \vdash_{\mathcal{R} \cup R'} J\).

**Proof.** Any derivation of \(J\) from \(\mathcal{R}[\Gamma]\) is also a derivation from \((\mathcal{R} \cup R')[\Gamma]\), since the presence of additional rules does not influence the validity of the derivation.

\(\square\)
2.1 Derivability

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.

**Exchange** If $\Gamma_1, J_1, J_2, \Gamma_2 \vdash_\mathcal{R} J$, then $\Gamma_1, J_2, J_1, \Gamma_2 \vdash_\mathcal{R} J$. The relative ordering of the axioms is immaterial.

**Contraction** If $\Gamma, J, J \vdash_\mathcal{R} K$, then $\Gamma, J \vdash_\mathcal{R} K$. We may use a hypothesis as many times as we like in a derivation.

**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.

These properties may be summarized by saying that the derivability hypothetical judgement is structural.

**Theorem 2.2.** For any rule set, $\mathcal{R}$, the derivability judgement $\Gamma \vdash_\mathcal{R} J$ is structural.

**Proof.** Reflexivity follows directly from the meaning of derivability. Weakening follows directly from uniformity. Exchange and contraction follow from the treatment of the rules, $\mathcal{R}$, as a finite set, for which order does not matter and replication is immaterial. Transitivity is proved by rule induction on the first premise.

In view of the structural properties of exchange and contraction, we regard the hypotheses, $\Gamma$, of a derivability judgement as a finite set of assumptions, so that the order and multiplicity of hypotheses does not matter. In particular, when writing $\Gamma$ as the union $\Gamma_1 \Gamma_2$ of two sets of hypotheses, a hypothesis may occur in both $\Gamma_1$ and $\Gamma_2$. This is obvious when $\Gamma_1$ and $\Gamma_2$ are given, but when decomposing a given $\Gamma$ into two parts, it is well to remember that the same hypothesis may occur in both parts of the decomposition.
2.2 Admissibility

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

$$
\frac{J_1 \ldots J_n}{J}
$$

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

For example, the admissibility judgement

$$
succ(a) \text{ nat} \models (1.2) a \text{ nat}
$$

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

$$
succ(a) \text{ nat} \frac{a \text{ nat}}{}
$$

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

$$
succ(\text{junk}) \text{ nat}
$$

(where junk is some object for which $\text{junk} \text{ nat}$ 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 $\text{junk} \text{ nat}$. 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 given by Theorem 2.2 on the preceding page ensure that derivability is stronger than admissibility.

Theorem 2.3. If $\Gamma \vdash_R J$, then $\Gamma \models_R J$.

Proof. Repeated application of the transitivity of derivability shows that if $\Gamma \vdash_R J$ and $\vdash_R \Gamma$, then $\vdash_R J$. 

$\square$
To see that the converse fails, observe that there is no composition of rules such that
\[ \text{succ}(\text{junk})\text{ nat} \vdash (1.2) \text{junk nat}, \]
yet the admissibility judgement
\[ \text{succ}(\text{junk})\text{ nat} \models (1.2) \text{junk nat} \]
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 \( \nabla \) 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: \( J \models_R J \).

**Weakening** If \( J \) is derivable from the original rules assuming that each of the judgements in \( \Gamma \) are derivable from these rules, then \( J \) must also be derivable assuming that \( \Gamma \) and also \( K \) are derivable from the original rules: if \( \Gamma \models_R J \), then \( \Gamma, K \models_R J \).

**Exchange** The order of assumptions in an iterated implication does not matter.

**Contraction** Assuming the same thing twice is the same as assuming it once.

**Transitivity** If \( \Gamma, K \models_R J \) and \( \Gamma \models_R K \), then \( \Gamma \models_R J \). If the assumption \( K \) is used, then we may instead appeal to the assumed derivability of \( K \).

**Theorem 2.4.** The admissibility judgement \( \Gamma \models_R J \) is structural.

*Proof.* Follows immediately from the definition of admissibility as stating that if the hypotheses are derivable relative to \( R \), then so is the conclusion.

Just as with derivability, we may, in view of the properties of exchange and contraction, regard the hypotheses, \( \Gamma \), of an admissibility judgement as a finite set, for which order and multiplicity does not matter.
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 form
\[
\Gamma; \Gamma_1 \vdash J_1 \ldots \Gamma; \Gamma_n \vdash J_n \quad \Gamma \vdash J.
\] (2.9)

The hypotheses \( \Gamma \) are the global hypotheses of the rule, and the hypotheses \( \Gamma_i \) are the local hypotheses of the \( i \)th premise of the rule. Informally, this rule states that \( J \) is a derivable consequence of \( \Gamma \) whenever each \( J_i \) is a derivable consequence of \( \Gamma \), augmented with the additional hypotheses \( \Gamma_i \). Thus, one way to show that \( J \) is derivable from \( \Gamma \) is to show, in turn, that each \( J_i \) is derivable from \( \Gamma; \Gamma_i \). The derivation of each premise involves a “context switch” in which we extend the global hypotheses with the local hypotheses of that premise, establishing a new set of global hypotheses for use within that derivation.

Often a hypothetical rule is given for each choice of global context, without restriction. In that case the rule is said to be pure, because it applies irrespective of the context in which it is used. A pure rule, being stated uniformly for all global contexts, may be given in implicit form, as follows:
\[
\Gamma_1 \vdash J_1 \ldots \Gamma_n \vdash J_n \quad \Gamma \vdash J.
\] (2.10)

This formulation omits explicit mention of the global context in order to focus attention on the local aspects of the inference.

Sometimes it is necessary to restrict the global context of an inference, so that it applies only if a specified side condition is satisfied. Such rules are said to be impure. Impure rules generally have the form
\[
\Gamma; \Gamma_1 \vdash J_1 \ldots \Gamma; \Gamma_n \vdash J_n \quad \Psi \quad \Gamma \vdash J.
\] (2.11)

where the condition, \( \Psi \), limits the applicability of this rule to situations in which it is true. For example, \( \Psi \) may restrict the global context of the inference to be empty, so that no instances involving global hypotheses are permissible.
2.4 Exercises

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, $\Gamma$, and a basic judgement, $J$. A collection of hypothetical rules, $\mathcal{R}$, defines the strongest formal derivability judgement closed under rules $\mathcal{R}$, which, by a slight abuse of notation, we write as $\Gamma \vdash_{\mathcal{R}} J$.

Since $\Gamma \vdash_{\mathcal{R}} J$ is the strongest judgement closed under $\mathcal{R}$, the principle of hypothetical rule induction is valid for reasoning about it. Specifically, to show that $\mathcal{P}(\Gamma \vdash J)$ whenever $\Gamma \vdash_{\mathcal{R}} J$, it is enough to show, for each rule (2.9) in $\mathcal{R}$,

if $\mathcal{P}(\Gamma \Gamma_1 \vdash J_1)$ and ... and $\mathcal{P}(\Gamma \Gamma_n \vdash J_n)$, 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 many cases we wish to ensure that the formal derivability relation defined by a collection of hypothetical rules is structural. This amounts to showing that the following structural rules be admissible:

\begin{align*}
\Gamma, J &\vdash J \\
\Gamma \vdash J &
\Gamma, K \vdash J \\
\Gamma \vdash K &
\Gamma, K \vdash J \\
\Gamma \vdash J
\end{align*}

(2.12a) \hspace{1cm} (2.12b) \hspace{1cm} (2.12c)

In the common case that the rules of a hypothetical inductive definition are pure, the structural rules (2.12b) and (2.12c) may be easily shown admissible by rule induction. However, it is typically necessary to include Rule (2.12a) explicitly, perhaps in a restricted form, to ensure reflexivity.

2.4 Exercises

1. Prove that if all rules in a hypothetical inductive definition are pure, then the structural rules of weakening (Rule (2.12b)) and transitivity (Rule (2.12c)) are admissible.

2. Define $\Gamma' \vdash \Gamma$ to mean that $\Gamma' \vdash J_i$ for each $J_i$ in $\Gamma$. Show that $\Gamma \vdash J$ if whenever $\Gamma' \vdash \Gamma$, it follows that $\Gamma' \vdash J$. \textit{Hint:} from left to right, appeal to transitivity of entailment; from right to left, consider the case of $\Gamma' = \Gamma$.
3. 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

Parametric Judgements

Basic judgements express properties of objects of the universe of discourse. Hypothetical judgements express entailments between judgements, or reasoning under hypotheses. Parametric judgements express entailments among properties of objects involving parameters, abstract symbols serving as atomic objects in an expanded universe.

Parameters have a variety of uses: as atomic symbols with no properties other than their identity, and as variables given meaning by substitution, the replacement of a parameter by an object. We shall make frequent use of parametric judgements throughout this book. Parametric inductive definitions, which generalize hypothetical inductive definitions to permit introduction of parameters in an inference, are of particular importance in our work.

3.1 Parameters and Objects

We assume given an infinite set of parameters, which we will consider to be abstract atomic objects that are distinct from all other objects and that can be distinguished from one another (that is, we can tell whether any two given parameters are the same or different).\(^1\) It follows that if we are given an object possibly containing a parameter, \(x\), we can rename \(x\) to another parameter, \(x'\), within that object.

To account for parameters we consider the family \(U[\mathcal{X}]\) of expansions of the universe of discourse with parameters drawn from the finite set \(\mathcal{X}\).

\(^1\)Parameters are sometimes called symbols, atoms, or names to emphasize their atomic, featureless character.
(The expansion $U[\emptyset]$ may be identified with the universe, $U$, of finitary trees discussed in Chapter 1.) The elements of $U[\mathcal{X}]$ are finitary trees in which the parameters, $\mathcal{X}$, may occur as leaves. We assume that parameters are distinct from operators so that there can be no confusion between a parameter and an operator that has no children.

Expansion of the universe is monotone in that if $\mathcal{X} \subseteq \mathcal{Y}$, then $U[\mathcal{X}] \subseteq U[\mathcal{Y}]$, for a tree possibly involving parameters from $\mathcal{X}$ is surely a tree possibly involving parameters from $\mathcal{Y}$. A bijection $\pi : \mathcal{X} \leftrightarrow \mathcal{X}'$ between parameter sets induces a renaming $\pi^* : U[\mathcal{X}] \leftrightarrow U[\mathcal{X}']$ that, intuitively, replaces each occurrence of $x \in \mathcal{X}$ by $\pi(x) \in \mathcal{X}'$ in any element of $U[\mathcal{X}]$, yielding an element of $U[\mathcal{X}']$.

### 3.2 Rule Schemes

The concept of an inductive definition extends naturally to any fixed expansion of the universe by parameters, $\mathcal{X}$. A collection of rules defined over the expansion $U[\mathcal{X}]$ determines the strongest judgement over that expansion closed under these rules. Extending the notation of Chapter 2, we write $\Gamma \vdash_{X} J$ to mean that $J$ is derivable from $R[\Gamma]$ over the expansion $U[\mathcal{X}]$.

It is often useful to consider rules that are defined over an arbitrary expansion of the universe by parameters. Recall Rule (1.2b) from Chapter 1:

\[
\frac{a \text{ nat}}{\text{succ}(a) \text{ nat}} \tag{3.1}
\]

As discussed in Chapter 1, this is actually a rule scheme that stands for an infinite family of rules, called its instances, one for each choice of element $a$ of the universe, $U$. We extend the concept of rule scheme so that it applies to any expansion of the universe by parameters, so that for each choice of $\mathcal{X}$, we obtain a family of instances of the rule scheme, one for each element $a$ of the expansion $U[\mathcal{X}]$.

We will generally gloss over the distinction between a rule and a rule scheme, so that for example Rules (1.2) may be considered over any expansion of the universe by parameters without explicit mention. A collection of such rules defines the strongest judgement over a given expansion closed under all instances of the rule schemes over this expansion. Consequently, we may reason by rule induction as described in Chapter 1 over any expansion of the universe by simply reading the rules as applying to objects in that expansion.
3.3 Parametric Derivability

It will be useful to consider a generalization of the derivability judgement that specifies the parameters, as well as the hypotheses, of a judgement. To a first approximation, the parametric derivability judgement

\[ \mathcal{X} \mid \Gamma \vdash_{\mathcal{R}} J \]  

means simply that \( \Gamma \vdash_{\mathcal{R}}^X J \). That is, the judgement \( J \) is derivable from hypotheses \( \Gamma \) over the expansion \( \mathcal{U}[X] \). For example, the parametric judgement

\[ \{ x \} \mid x \text{ nat} \vdash_{(1.2)} \text{succ(succ}(x)) \text{ nat} \]  

is valid with respect to Rules (1.2), because the judgement \( \text{succ(succ}(x)) \text{ nat} \) is derivable from the assumption \( x \text{ nat} \) in the expansion of the universe with parameter \( x \).

This is the rough-and-ready interpretation of the parametric judgement. However, the full meaning is slightly stronger than that. In addition to the condition just specified, we also demand that the judgement hold for all renamings of the parameters \( X \), so that the validity of the judgement cannot depend on the exact choice of parameters. To ensure this we define the meaning of the parametric judgement (3.2) to be given by the following condition:

\[ \forall \pi : X \leftrightarrow X' \quad \pi^\uparrow \Gamma \vdash^{X'}_{\pi^\uparrow \mathcal{R}} \pi^\uparrow J. \]

Evidence for the judgement (3.2) consists of a parametric derivation, \( \nabla_{X'} \), of the judgement \( \pi^\uparrow J \) from rules \( \pi^\uparrow \mathcal{R}[\pi^\uparrow \Gamma] \) for some bijection \( \pi : X \leftrightarrow X' \).

For example, judgement (3.3) is valid with respect to Rules (1.2) since, for every \( x' \), the judgement

\[ \text{succ(succ}(x')) \text{ nat} \]

is derivable from Rules (1.2) expanded with the axiom \( x' \text{ nat} \). Evidence for this consists of the parametric derivation, \( \nabla_{x'} \),

\[
\frac{\text{nat}}{x' \text{ nat}} \quad \frac{\text{nat}}{\text{succ}(x') \text{ nat}} \quad \frac{\text{nat}}{\text{succ(succ}(x')) \text{ nat}} \]

composed of Rules (1.2) and the axiom \( x' \text{ nat} \).

Parametric derivability enjoys two structural properties in addition to those enjoyed by the derivability judgement itself:
3.4 Parametric Inductive Definitions

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

**Renaming** If $\mathcal{X}, x \mid \Gamma \vdash R$ with $x \notin \mathcal{X}$, then for every $x' \notin \mathcal{X}$,

$$\mathcal{X}, x' \mid [x \mapsto \cdot] \Gamma \vdash [x \mapsto x'] \Gamma \vdash R \vdash [x \mapsto \cdot] J,$$

and conversely.

Proliferation implies 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 fresh parameters.

**Theorem 3.1.** The parametric derivability judgement is structural.

*Proof.* Both properties follow directly from the definition of parametric derivability.

In view of Theorem 3.1 we may tacitly assume that the fresh parameters of a judgement are disjoint from the ambient parameters. For if not, we may simply rename them to ensure that it is so, and appeal to the renaming property to obtain the desired judgement. In practice we tacitly assume that the fresh parameters have already been renamed apart from the ambient parameters, so that evidence for judgement (3.4) may be considered to be a parametric derivation $\nabla_x$ with parameter $x$.

### 3.4 Parametric Inductive Definitions

A *parametric inductive definition* is a generalization of a hypothetical inductive definition to permit expansion not only of the set of rules, but also of the set of parameters, in each premise of a rule. A *parametric rule* has the form

$$\frac{\mathcal{X} \mid \Gamma \vdash J_1 \ldots \mathcal{X} \mid \Gamma \vdash J_n}{\mathcal{X} \mid \Gamma \vdash J}.$$  \hspace{1cm} (3.5)

The set, $\mathcal{X}$, is the set of *global parameters* of the inference, and, for each $1 \leq i \leq n$, the set $\mathcal{X}_i$ is the set of *fresh local parameters* of the $i$th premise. The local parameters are *fresh* in the sense that, by suitable renaming, they may be chosen to be disjoint from the global parameters of the inference. The pair $\mathcal{X} \mid \Gamma$ is called the *global context* of the rule, and each pair $\mathcal{X}_i \mid \Gamma_i$ is called the *local context* of the $i$th premise of the rule.
3.4 Parametric Inductive Definitions

A parametric rule is *pure* if it is stated for all choices of global context. A pure rule may be written in *implicit form*,

\[
\frac{X_1 | \Gamma_1 \vdash J_1 \ldots \ X_n | \Gamma_n \vdash J_n}{J},
\]

with the understanding that it stands for the infinite family of rules of the form Rule (3.5) for all choices of global context \( \mathcal{Y} | \Gamma \). An *impure* parametric rule is one that is stated only for certain choices of global context, for example by insisting that the global parameters be empty.

A parametric inductive definition may be regarded as an ordinary inductive definition of the *formal parametric judgement* \( X | \Gamma \vdash J \). If \( \mathcal{R} \) is a collection of parametric derivability rules, we abuse notation slightly by writing \( X | \Gamma \vdash_R J \) to mean that the formal parametric judgement \( X | \Gamma \vdash J \) is derivable from rules \( \mathcal{R} \).

The principle of rule induction for a parametric inductive definition states that to show \( \mathcal{P}(X | \Gamma \vdash J) \) whenever \( X | \Gamma \vdash_R J \), it is enough to show that \( \mathcal{P} \) is closed under the rules \( \mathcal{R} \). Specifically, for each rule in \( \mathcal{R} \) of the form (3.5), we must show that

\[
\text{if } \mathcal{P}(X | \Gamma_1 \vdash J_1) \ldots \mathcal{P}(X | \Gamma_n \vdash J_n) \text{ then } \mathcal{P}(X | \Gamma \vdash J).
\]

Because the meaning of the parametric judgement is independent of the choice of parameter names, any property \( \mathcal{P} \) of a parametric judgement must not depend on the choice of local parameter names.

To ensure that a formal parametric judgement is structural, the following rules must be admissible relative to the rules that define it:

\[
\frac{X | \Gamma, J \vdash J}{X | \Gamma, J \vdash J}
\]

\[
\frac{X | \Gamma \vdash J \quad X | \Gamma, J \vdash K}{X | \Gamma \vdash K}
\]

\[
\frac{X | \Gamma \vdash J}{X, x | \Gamma \vdash J}
\]

\[
\frac{X | \Gamma \vdash K}{X | \Gamma, J \vdash K}
\]

\[
\frac{X, x' | [x \mapsto x'] | \Gamma \vdash [x \mapsto x']^+ J^+ \quad X, x | \Gamma \vdash J}{X, x' | \Gamma \vdash J}
\]
3.5 Exercises

The admissibility of Rule (3.7a) is, in practice, ensured by explicitly including it in a limited form sufficient to ensure that it holds for the general case.

The admissibility of Rules (3.7c) and (3.7d) are assured if each of the parametric rules is pure. For then we may simply assimilate the parameter $x$ to the global parameters, and the hypothesis $J$ to the global hypotheses, without disrupting the validity of the derivation.

The admissibility of Rule (3.7e) is ensured by requiring that a rule be stated for all choices of local parameters provided that they are disjoint from the global parameters. This is called the renaming convention. In a proof by rule induction the naming convention allows us to choose the local parameters to be as fresh as required in a given situation, without explicit mention of having done so. In particular, this ensures that Rule (3.7e) is admissible. When constructing a derivation we need not provide a separate derivation for each choice of local parameters, but rather can provide only one derivation using some choice of fresh local parameters, for we may then transform this single derivation into the required family of derivations by simply renaming the chosen parameters in the given derivation.

Examples of parametric inductive definitions are given in Chapters 6 and 7, and will be used heavily throughout the book.

3.5 Exercises

1. Investigate parametric admissibility.

2. Prove structurality.

3. Explore identification convention.
Chapter 4

Transition Systems

Transition systems are used to describe the execution behavior of programs by defining an abstract computing device with a set, $S$, of states that are related by a transition judgement, $\rightarrow$. The transition judgement describes how the state of the machine evolves during execution.

4.1 Transition Systems

An (ordinary) transition system is specified by the following judgements:

1. $s$ state, asserting that $s$ is a state of the transition system.

2. $s$ final, where $s$ state, asserting that $s$ is a final state.

3. $s$ initial, where $s$ state, asserting that $s$ is an initial state.

4. $s \rightarrow s'$, where $s$ state and $s'$ state, asserting that state $s$ may transition to state $s'$.

We require that if $s$ final, then for no $s'$ do we have $s \rightarrow s'$. In general, a state $s$ for which there is no $s' \in S$ such that $s \rightarrow s'$ is said to be stuck, which may be indicated by writing $s \not\rightarrow$. All final states are stuck, but not all stuck states need be final!

A transition sequence is a sequence of states $s_0, \ldots, s_n$ such that $s_0$ initial, and $s_i \rightarrow s_{i+1}$ for every $0 \leq i < n$. A transition sequence is maximal iff $s_n \not\rightarrow$, 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. A transition system is deterministic iff for every
4.2 Iterated Transition

A labelled transition system over a set of labels, I, is a generalization of a transition system in which the single transition judgement, \( s \rightarrow s' \) is replaced by an \( I \)-indexed family of transition judgements, \( s \rightarrow^i s' \), where \( s \) and \( s' \) are states of the system. In typical situations the family of transition relations is given by a simultaneous inductive definition in which each rule may make reference to any member of the family.

It is often necessary to consider families of transition relations in which there is a distinguished unlabelled transition, \( s \rightarrow s' \), in addition to the indexed transitions. It is sometimes convenient to regard this distinguished transition as labelled by a special, anonymous label not otherwise in \( I \). For historical reasons this distinguished label is often designated by \( \tau \) or \( \epsilon \), but we will simply use an unadorned arrow. The unlabelled form is often called a silent transition, in contrast to the labelled forms, which announce their presence with a label.

4.2 Iterated Transition

Let \( s \rightarrow s' \) be a transition judgement, whether drawn from an indexed set of such judgements or not.

The iteration of transition judgement, \( s \rightarrow^* s' \), is inductively defined by the following rules:

\[
\begin{align*}
\text{Iterated Transition} & : s \rightarrow^* s \\
& \text{Iterated Transition} & : s \rightarrow s' \quad s' \rightarrow^* s'' \\
& \quad \Rightarrow s \rightarrow^* s''
\end{align*}
\]

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

The principle of rule induction for these rules states that to show that \( P(s, s') \) holds whenever \( s \rightarrow^* s' \), it is enough to show these two properties of \( P \):

1. \( P(s, s) \).
2. if \( s \rightarrow s' \) 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 \( \rightarrow^* \) is reflexive and transitive.
4.3 Simulation and Bisimulation

The \textit{n-times iterated} transition judgement, \( s \mapsto^n s' \), where \( n \geq 0 \), is inductively defined by the following rules.

\[
\begin{align*}
 s & \mapsto^n s \\
 s \mapsto s' & \quad s' \mapsto^n s'' \\
 s & \mapsto^{n+1} s''
\end{align*}
\]  

(4.2a)

(4.2b)

**Theorem 4.1.** For all states \( s \) and \( s' \), \( s \mapsto^* s' \) iff \( s \mapsto^k s' \) for some \( k \geq 0 \).

Finally, we write \( s \downarrow \) to indicate that there exists some \( s' \) final such that \( s \mapsto^* s' \).

4.3 Simulation and Bisimulation

A \textit{strong simulation} between two transition systems \( \mapsto_1 \) and \( \mapsto_2 \) is given by a binary relation, \( s_1 S s_2 \), between their respective states such that if \( s_1 S s_2 \), then \( s_1 \mapsto_1 s'_1 \) implies \( s_2 \mapsto_2 s'_2 \) for some state \( s'_2 \) such that \( s'_1 S s'_2 \). Two states, \( s_1 \) and \( s_2 \), are \textit{strongly similar} iff there is a strong simulation, \( S \), such that \( s_1 S s_2 \). Two transition systems are strongly similar iff each initial state of the first is strongly similar to an initial state of the second. Finally, two transition systems are \textit{strongly bisimilar} iff there is a single relation \( S \) such that both \( S \) and its converse are strong simulations.

A strong simulation between two labelled transition systems over the same set, \( I \), of labels consists of a relation \( S \) between states such that for each \( i \in I \) the relation \( S \) is a strong simulation between \( i \mapsto_1 \) and \( i \mapsto_2 \). That is, if \( s_1 S s_2 \), then \( s_1 \mapsto_1 s'_1 \) implies \( s_2 \mapsto_2 s'_2 \) for some \( s'_2 \) such that \( s'_1 S s'_2 \). In other words the simulation must preserve labels, and not just transitions.

The requirements for strong simulation are rather stringent: every step in the first system must be mimicked by a similar step in the second, up to the simulation relation in question. This means, in particular, that a sequence of steps in the first system can only be simulated by a sequence of steps of the same length in the second—there is no possibility of performing “extra” work to achieve the simulation.

A \textit{weak simulation} between transition systems is a binary relation between states such that if \( s_1 S s_2 \), then \( s_1 \mapsto_1 s'_1 \) implies \( s_2 \mapsto_2 s'_2 \) for some \( s'_2 \) such that \( s'_1 S s'_2 \). That is, every step in the first may be matched by zero or more steps in the second. A \textit{weak bisimulation} is such that both
it and its converse are weak simulations. We say that states $s_1$ and $s_2$ are weakly (bi)similar iff there is a weak (bi)simulation $S$ such that $s_1 S s_2$.

The corresponding notion of weak simulation for labelled transitions involves the silent transition. The idea is that to weakly simulate the labelled transition $s_1 \xrightarrow{\ell} s'_1$, we do not wish to permit multiple labelled transitions between related states, but rather to permit any number of unlabelled transitions to accompany the labelled transition. A relation between states is a weak simulation iff it satisfies both of the following conditions whenever $s_1 S s_2$:

1. If $s_1 \xrightarrow{\ell} s'_1$, then $s_2 \xrightarrow{\ell} s'_2$ for some $s'_2$ such that $s'_1 S s'_2$.

2. If $s_1 \xrightarrow{\ell} s'_1$, then $s_2 \xrightarrow{\ell} s'_2 \xrightarrow{\ell} s''_2$ for some $s''_2$ such that $s'_1 S s'_2$.

That is, every silent transition must be mimicked by zero or more silent transitions, and every labelled transition must be mimicked by a corresponding labelled transition, preceded and followed by any number of silent transitions. As before, a weak bisimulation is a relation between states such that both it and its converse are weak simulations. Finally, two states are weakly (bi)similar iff there is a weak (bi)simulation between them.

### 4.4 Exercises

1. Prove that $S$ is a weak simulation for the ordinary transition system $\rightarrow$ iff $S$ is a strong simulation for $\rightarrow^*$. 

18:42 Draft October 16, 2009
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 $L\{\text{num str}\}$, that supports elementary arithmetic on the natural numbers and simple computations on strings. In addition, $L\{\text{num str}\}$ 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 $c \text{ char}$ to indicate that $c$ is a character, and let $\Sigma$ stand for a finite set of such judgements, which is sometimes called an alphabet. The judgement $\Sigma \vdash s \text{ str}$, defining the strings over the alphabet $\Sigma$, is inductively defined by the following rules:

\begin{align*}
\Sigma \vdash \epsilon \text{ str} \\
\Sigma \vdash c \text{ char} & \quad \Sigma \vdash s \text{ str} \\
\Sigma \vdash c \cdot s \text{ str}
\end{align*}

(5.1a) (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 $\Sigma$ when it is clear from context.
5.2 Lexical Structure

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

1. \( \epsilon P \), and
2. if \( s P \) and \( c \) char, then \( c \cdot s P \).

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 \( s_1 \cdot s_2 = s \) str, stating that \( s \) is the result of concatenating the strings \( s_1 \) and \( s_2 \).

\[
\begin{align*}
\epsilon \cdot s &= s \text{ str} \\
(s_1 \cdot s_2) \cdot s_2 &= c \cdot s \text{ str}
\end{align*}
\]  

(5.2a) (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.

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. Concatenation 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 \( ab \cdot cd \), \( a \cdot bc \cdot d \), or \( ab \cdot cd \), or even \( \epsilon \cdot abcd \) or \( abcd \cdot \epsilon \), 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.\footnote{In some languages white space is significant, in which case it must be converted to symbolic form for subsequent processing.}

The character representation of symbols is, in most cases, conveniently described using regular expressions. The lexical structure of $L\{\text{num str}\}$ is specified as follows:

- **Item** \( \text{itm} ::= \text{kwd} | \text{id} | \text{num} | \text{lit} | \text{spl} \)
- **Keyword** \( \text{kwd} ::= \text{l} \cdot \text{e} \cdot \text{t} \cdot \epsilon | \text{b} \cdot \epsilon \cdot \epsilon | \text{i} \cdot \text{n} \cdot \epsilon \)
- **Identifier** \( \text{id} ::= \text{ltr} (\text{ltr} | \text{dig})^{*} \)
- **Numeral** \( \text{num} ::= \text{dig dig}^{*} \)
- **Literal** \( \text{lit} ::= \text{qum (ltr | dig)}^{*} \text{qum} \)
- **Special** \( \text{spl} ::= + | \ast | \hat{ } | ( | ) | | \)
- **Letter** \( \text{ltr} ::= \text{a} | \text{b} | \ldots \)
- **Digit** \( \text{dig} ::= 0 | 1 | \ldots \)
- **Quote** \( \text{qum} ::= " \)

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:

\[
\frac{s \text{ str}}{\text{ID}[s] \text{ tok}} \quad (5.3a) \\
\frac{n \text{ nat}}{\text{NUM}[n] \text{ tok}} \quad (5.3b) \\
\frac{s \text{ str}}{\text{LIT}[s] \text{ tok}} \quad (5.3c) \\
\frac{\text{LET tok}}{} \quad (5.3d)
\]
Lexical analysis is inductively defined by the following judgement forms:

- \( s \text{ charstr} \leftarrow t \text{ tokstr} \) — Scan input
- \( s \text{ itm} \leftarrow t \text{ tok} \) — Scan an item
- \( s \text{ kwv} \leftarrow t \text{ tok} \) — Scan a keyword
- \( s \text{ id} \leftarrow t \text{ tok} \) — Scan an identifier
- \( s \text{ num} \leftarrow t \text{ tok} \) — Scan a number
- \( s \text{ spl} \leftarrow t \text{ tok} \) — Scan a symbol
- \( s \text{ lit} \leftarrow t \text{ tok} \) — Scan a string literal
- \( s \text{ whs} \) — Skip white space

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, \( s \text{ whs} \) states that the string \( s \) consists only of “white space”, and \( s \text{ lord} \) states that \( s \) is either an alphabetic letter or a digit, and so forth.

\[
\epsilon \text{ charstr} \leftarrow \epsilon \text{ tokstr} \quad (5.4a)
\]
5.2 Lexical Structure

\[
s = s_1^*s_2^*s_3 \quad s_1 \text{whs} \quad s_2 \text{itm} \longrightarrow t \quad s_3 \text{charstr} \longrightarrow ts \quad \text{tokstr}
\]

\[
s \text{charstr} \longrightarrow t \cdot ts \quad \text{tokstr}
\]  

(5.4b)

\[
s \text{kwd} \longrightarrow t\quad \text{tok}
\]

\[
s \text{itm} \longrightarrow t\quad \text{tok}
\]  

(5.4c)

\[
s \text{id} \longrightarrow t\quad \text{tok}
\]

\[
s \text{itm} \longrightarrow t\quad \text{tok}
\]  

(5.4d)

\[
s \text{num} \longrightarrow t\quad \text{tok}
\]

\[
s \text{itm} \longrightarrow t\quad \text{tok}
\]  

(5.4e)

\[
s \text{lit} \longrightarrow t\quad \text{tok}
\]

\[
s \text{itm} \longrightarrow t\quad \text{tok}
\]  

(5.4f)

\[
s \text{spl} \longrightarrow t\quad \text{tok}
\]

\[
s \text{itm} \longrightarrow t\quad \text{tok}
\]  

(5.4g)

\[
s = l \cdot t \cdot e\quad \text{str}
\]

\[
s \text{kwd} \longrightarrow \text{LET tok}
\]  

(5.4h)

\[
s = b \cdot e \cdot t\quad \text{str}
\]

\[
s \text{kwd} \longrightarrow \text{BE tok}
\]  

(5.4i)

\[
s = i \cdot n \cdot e\quad \text{str}
\]

\[
s \text{kwd} \longrightarrow \text{IN tok}
\]  

(5.4j)

\[
s = s_1^*s_2 \quad s_1 \text{lttr} \quad s_2 \text{lord}
\]

\[
s \text{id} \longrightarrow \text{ID}[s] \quad \text{tok}
\]  

(5.4k)

\[
s = s_1^*s_2 \quad s_1 \text{dig} \quad s_2 \text{dgs} \quad s \text{num} \longrightarrow n \quad \text{nat}
\]

\[
s \text{num} \longrightarrow \text{NUM}[n] \quad \text{tok}
\]  

(5.4l)

\[
s = s_1^*s_2^*s_3 \quad s_1 \text{qum} \quad s_2 \text{lord} \quad s_3 \text{qum}
\]

\[
s \text{lit} \longrightarrow \text{LIT}[s_2] \quad \text{tok}
\]  

(5.4m)

\[
s = + \cdot e\quad \text{str}
\]

\[
s \text{spl} \longrightarrow \text{ADD tok}
\]  

(5.4n)

\[
s = \ast \cdot e\quad \text{str}
\]

\[
s \text{spl} \longrightarrow \text{MUL tok}
\]  

(5.4o)

\[
s = ^\cdot e\quad \text{str}
\]

\[
s \text{spl} \longrightarrow \text{CAT tok}
\]  

(5.4p)

October 16, 2009

Draft

18:42
By convention Rule (5.4k) applies only if none of Rules (5.4h) to (5.4j) apply. Technically, Rule (5.4k) has implicit premises that rule out keywords as possible identifiers.

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 $\alpha$ 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,

$$
A ::= \alpha_1 \\
\vdots \\
A ::= \alpha_n,
$$

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

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

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

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

(5.5)

we associate an inference rule

$$\frac{s_1' A_1 \ldots s_n' A_n}{s_1 s_2' \ldots s_n s_{n+1}' A}.$$ (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:

$$\frac{s_1' A_1 \ldots s_n' A_n}{s_1 \hat{s}_1' \hat{s}_2' \ldots \hat{s}_n' \hat{s}_{n+1}' A}.$$ (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 not invertible, 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 $L\{\text{num str}\}$ may be specified by a context-free grammar over the tokens defined in Section 5.2 on page 38. The grammar has only one syntactic class, $\text{exp}$, which is defined by the following compound production:

- **Expression** $ \text{exp} ::= \text{num} | \text{lit} | \text{id} | \text{LP} \text{exp} \text{RP} | \text{exp} \text{ADD} \text{exp} | \text{exp} \text{MUL} \text{exp} | \text{exp} \text{CAT} \text{exp} | \text{VB} \text{exp} \text{VB} | \text{LET} \text{id} \text{BE} \text{exp} \text{IN} \text{exp}$

- **Number** $ \text{num} ::= \text{NUM}[n] \quad (n \text{ nat})$

- **String** $ \text{lit} ::= \text{LIT}[s] \quad (s \text{ str})$

- **Identifier** $ \text{id} ::= \text{ID}[s] \quad (s \text{ str})$

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:

$$\frac{s \text{ num}}{s \text{ exp}}$$ (5.8a)
To emphasize the role of string concatenation, we may rewrite Rule (5.8e), for example, as follows:

\[
\begin{align*}
  s &\text{ exp} \\
  &\text{ MUL} \\
  &s_1 \text{ exp} \quad s_2 \text{ exp} \\
  &\text{ ADD} \\
  &s_1 \text{ exp} \quad s_2 \text{ exp} \\
  &\text{ MUL} \\
  &s_1 \text{ exp} \quad s_2 \text{ exp} \\
  &\text{ CAT} \\
  &s_1 \text{ exp} \quad s_2 \text{ exp} \\
  &\text{ LET} \\
  &s_1 \text{ id} \quad s_2 \text{ exp} \quad s_3 \text{ exp} \\
  &\text{ BE} \\
  &s_1 \text{ exp} \quad s_2 \text{ IN} \quad s_3 \text{ exp} \\
  &\text{ NUM} \\
  &n \text{ nat} \\
  &\text{ num} \\
  &\text{ LIT} \\
  &s \text{ str} \\
  &\text{ lit} \\
  &\text{ ID} \\
  &s \text{ str} \\
  &\text{ id}
\end{align*}
\]

That is, \( s \text{ exp} \) is derivable if \( s \) is the concatenation of \( s_1 \), the multiplication sign, and \( s_2 \), where \( s_1 \text{ exp} \) and \( s_2 \text{ exp} \).
Apart from subjective matters of readability, a principal goal of concrete syntax design is to eliminate 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

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

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

\[
\text{NUM}[1] \text{ ADD } \wedge \text{NUM}[2] \text{ MUL } \text{NUM}[3]
\]

and

\[
\text{NUM}[1] \text{ ADD } \wedge \text{NUM}[2] \text{ MUL } \text{NUM}[3],
\]

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 43 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

\[
\text{NUM}[1] \text{ ADD } \text{NUM}[2] \text{ ADD } \text{NUM}[3],
\]

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 eliminate ambiguity the grammar of $L\{\text{num str}\}$ given in Section 5.4 on page 43 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 sub-classes.

\[
\begin{align*}
\text{Factor} & \quad \text{fct} ::= \num | \lit | \id | \LP \text{prg} \RP \\
\text{Term} & \quad \text{trm} ::= \text{fct} | \text{fct} \text{MUL} \text{trm} | \text{VB} \text{fct} \text{VB} \\
\text{Expression} & \quad \text{exp} ::= \text{trm} | \text{trm} \text{ADD} \text{exp} | \text{trm} \text{CAT} \text{exp} \\
\text{Program} & \quad \text{prg} ::= \text{exp} | \text{LET} \id \text{BE} \text{exp} \text{IN} \text{prg}
\end{align*}
\]

The effect of this grammar is to ensure that \texttt{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 Trees

The concrete syntax of a language defines its linear representation as strings of symbols. The string representation of a program is convenient for keyboard input and network transmission, but is all-but-useless for analysis of the properties of programming languages. The abstract syntax of a language dispenses with the linear representation in favor of exposing the hierarchical structure of programs, making clear which phrases are constituents of which others. Phrases are represented as abstract syntax trees, or ast’s, that may involve variables serving as placeholders for other ast’s.

6.1 Abstract Syntax Trees

An abstract syntax tree, or ast for short, is an ordered tree in which nodes are labelled by operators. Each operator has an arity specifying the number of children of any node that it labels. A signature, \( \Omega \), is a finite set of judgements of the form \( \text{ar}(o) = k \), which specifies that the operator \( o \) has arity \( k \geq 0 \). A signature may specify at most one arity for each operator.

The class of closed abstract syntax trees over a signature, \( \Omega \), is inductively defined by the following rules:

\[
\Omega \vdash \text{ar}(o) = k \\
\frac{a_1 \text{ ast} \quad \ldots \quad a_k \text{ ast}}{o(a_1, \ldots, a_k) \text{ ast}}
\]  

(6.1a)

One may read this as specifying one rule for each operator, \( o \), such that \( \Omega \vdash \text{ar}(o) = k \). When \( k \) is zero, Rule (6.1a) has no premises (other than the arity judgement), and hence forms the basis for the induction. The ast \( o() \) is usually abbreviated to \( o \) for operators of arity zero.
The rule set \( A[\Omega] \) consists of the expansion of Rules (6.1) with judgements of \( \Omega \) as axioms (rules without premises).

For example, the abstract syntax of closed arithmetic expressions (without variables) may be specified by the following signature:

\[
\begin{align*}
\text{ar}(\text{num}[n]) &= 0 \quad (n \text{ nat}) \\
\text{ar}(\text{str}[s]) &= 0 \quad (s \text{ str}) \\
\text{ar}(\text{plus}) &= 2 \\
\text{ar}(\text{times}) &= 2 \\
\text{ar}(\text{cat}) &= 2 \\
\text{ar}(\text{len}) &= 1
\end{align*}
\]

Accounting for the binding and scope of variables goes beyond the expressive capabilities of ast’s; this will be rectified in Chapter 7 using an enriched form of ast’s.

The principle of structural induction is the principle of rule induction specialized to \( A[\Omega] \) for some signature \( \Omega \). Specifically, to show that \( P(a \text{ ast}) \), it is enough to show, for each operator \( o \) such that \( \Omega \vdash \text{ar}(o) = k \) for some \( k \),

\[
\text{if } P(a_1 \text{ ast}), \ldots, P(a_k \text{ ast}), \text{ then } P(o(a_1, \ldots, a_k) \text{ ast}).
\]

When \( k \) is zero, this reduces to showing that \( P(o) \).

For example, consider the following rules defining the height of a closed abstract syntax tree over some signature \( \Omega \):

\[
\begin{align*}
\text{hgt}(a_1) &= h_1 \quad \ldots \quad \text{hgt}(a_k) = h_k \\
\max(h_1, \ldots, h_k) &= h \quad \text{ar}(o) = k \\
\text{hgt}(o(a_1, \ldots, a_k)) &= h + 1
\end{align*}
\]

There is one rule for each \( k \) such that \( \Omega \vdash \text{ar}(o) = k \) for some operator \( o \). Let \( H[\Omega] \) consist of rules \( A[\Omega] \) and Rules (6.1).

We may prove by structural induction that every ast has a unique height. For an operator \( o \) of arity \( k \), we may assume by induction that, for each \( 1 \leq i \leq k \), there is a unique \( h_i \) such that \( \text{hgt}(a_i) = h_i \). We may show separately that the maximum, \( h \), of these is uniquely determined, and hence that the overall height, \( h + 1 \), is also uniquely determined.
A variable in an ast is a placeholder for a fixed, but unspecified, ast. Given an ast and a designated variable, we may substitute an ast for all occurrences of that variable in that ast.

Fix a signature, Ω, of operators, let $\mathcal{X} = \{x_1, \ldots, x_m\}$ be a finite set of parameters, and let $\Gamma$ be the finite set of hypotheses $x_1 \text{ ast}, \ldots, x_m \text{ ast}$. The parametric judgement

$$\mathcal{X} \mid \Gamma \vdash_{\mathcal{A}[\Omega]} a \text{ ast} \quad (6.3)$$

states that $a$ is an ast in which any of the parameters in $\mathcal{X}$ may be used as atomic ast’s. Once we define substitution these atomic ast’s will function as variables that may be replaced with other ast’s.

The parametric judgement (6.3) may be directly defined by the following rules:

$$\vdash_{\mathcal{A}[\Omega]} a \text{ ast} \quad (6.4a)$$

$$\Omega \vdash \text{ar}(o) = k \quad \frac{\mathcal{X} \mid \Gamma \vdash a_1 \text{ ast} \ldots \mathcal{X} \mid \Gamma \vdash a_k \text{ ast}}{\mathcal{X} \mid \Gamma \vdash o(a_1, \ldots, a_k) \text{ ast}} \quad (6.4b)$$

t is easy to check that the judgement $\mathcal{X} \mid \Gamma \vdash a \text{ ast}$ defined by these rules is structural.

The principle of structural induction extends to ast’s with variables. To prove that $\mathcal{P}(\mathcal{X} \mid \Gamma \vdash a \text{ ast})$ holds whenever $\mathcal{X} \mid \Gamma \vdash a \text{ ast}$, it is enough to show these two facts:

1. $\mathcal{P}(\mathcal{X}, x \mid \Gamma, x \text{ ast} \vdash x \text{ ast})$ for every $\mathcal{X}$ and parameter $x \notin \mathcal{X}$.

2. If $\Omega \vdash \text{ar}(o) = k$, and $\mathcal{P}(\mathcal{X} \mid \Gamma \vdash a_i \text{ ast})$ for each $1 \leq i \leq k$, then $\mathcal{P}(\mathcal{X} \mid \Gamma \vdash o(a_1, \ldots, a_k))$

As discussed in Chapter 3 we consider only properties $\mathcal{P}$ that are independent of the names of the parameters.

The definition of the height of an ast may be extended to ast’s with variables. Let $\mathcal{X}$ and $\Gamma$ be as above. The parametric judgement $\mathcal{X} \mid \Gamma \vdash \text{hgt}(a) = m$ is inductively defined by the following rules:

$$\vdash_{\mathcal{A}[\Omega]} \text{hgt}(x) = 1 \quad (6.5a)$$

$$\Omega \vdash \text{ar}(o) = k \quad \frac{\text{max}(h_1, \ldots, h_k) = h \quad \mathcal{X} \mid \Gamma \vdash \text{hgt}(a_1) = h_1 \ldots \mathcal{X} \mid \Gamma \vdash \text{hgt}(a_k) = h_k}{\mathcal{X} \mid \Gamma \vdash \text{hgt}(o(a_1, \ldots, a_n)) = h + 1} \quad (6.5b)$$
Let $\mathcal{H}[\Omega]$ be the extension of rules $\mathcal{A}[\Omega]$ with Rules (6.5). A simple structural induction shows that every ast with variables has a height.

**Theorem 6.1.** Let $\mathcal{X} = \{x_1, \ldots, x_m\}$, and $\Gamma = x_1 \text{ ast}, \ldots, x_m \text{ ast}$. If $\mathcal{X} \vdash \mathcal{A}[\Omega] a \text{ ast}$, then there exists a unique $h$ such that $\mathcal{X} \vdash \mathcal{H}[\Omega] hgt(a) = h$.

**Proof.** By structural induction on $a$, which is to say by rule induction on Rules (6.4). For Rule (6.4a) the unique $p$ is provided by Rule (6.5a). For Rule (6.4b) the result follows by induction and the unicity of the maximum. 

Substitution is the process of replacing all occurrences of a variable in an ast with another ast. Substitution is defined by a parametric inductive definition of the judgment $\mathcal{X} \mid \Gamma \vdash [a/x]b = c \text{ ast}$, which states that the result of substituting $a$ for $x$ in $b$ is $c$.

\[
\frac{}{\mathcal{X} \mid \Gamma \vdash [a/x]x = a \text{ ast}} \tag{6.6a}
\]

\[
\frac{x \neq y}{\mathcal{X}, y \mid \Gamma, y \text{ ast} \vdash [a/x]y = y \text{ ast}} \tag{6.6b}
\]

\[
\frac{\Omega \vdash \text{ar}(o) = k \quad \mathcal{X} \mid \Gamma \vdash [a/x]b_1 = c_1 \text{ ast} \quad \ldots \quad \mathcal{X} \mid \Gamma \vdash [a/x]b_k = c_k \text{ ast}}{\mathcal{X} \mid \Gamma \vdash [a/x]o(b_1, \ldots, b_k) = o(c_1, \ldots, c_k) \text{ ast}} \tag{6.6c}
\]

Let $\mathcal{S}[\Omega]$ be the expansion of rules $\mathcal{A}[\Omega]$ with Rules (6.6).

**Theorem 6.2.** Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ and let $\Gamma$ be $x_1 \text{ ast}, \ldots, x_n \text{ ast}$. If $\mathcal{X} \mid \Gamma \vdash \mathcal{A}[\Omega] \text{ a ast}$, and $\mathcal{X}, x \mid \Gamma, x \text{ ast} \vdash \mathcal{A}[\Omega] \text{ b ast}$, where $x \notin \mathcal{X}$, then there exists a unique $c$ such that $\mathcal{X} \mid \Gamma \vdash \mathcal{S}[\Omega] [a/x]b = c \text{ ast}$.

**Proof.** By structural induction on $b$. There are three cases to consider, corresponding to the inferences:

1. $\mathcal{X}, x \mid \Gamma, x \text{ ast} \vdash x \text{ ast}$;
2. $\mathcal{X}, x, y \mid \Gamma, x \text{ ast}, y \text{ ast} \vdash y \text{ ast}$, where $y \notin \mathcal{X}, x$ and so $y \neq x$.
3. $\mathcal{X}, x \mid \Gamma, x \text{ ast} \vdash o(b_1, \ldots, b_k) \text{ ast}$, given that $\mathcal{X}, x \mid \Gamma, x \text{ ast} \vdash b_i \text{ ast}$ for each $1 \leq i \leq k.$
The first two cases are covered by Rules (6.6a) and (6.6b); the third is covered by induction and Rule (6.6c).

In view of this theorem we write \([a/x]b\) for the unique \(c\) given by the theorem, provided that we are in a context in which the premises of the theorem are understood to hold.

**Corollary 6.3.** The structural rule of substitution

\[
\frac{\cal X \mid \Gamma \vdash a \ast \quad \cal X, x \mid \Gamma, x \ast \vdash b \ast}{\cal X \mid \Gamma \vdash [a/x]b \ast}
\]

is admissible for Rules (6.4).
6.3 Exercises
Abstract syntax trees expose the hierarchical structure of syntax, dispensing with the details of how one might represent pieces of syntax on a page or a computer screen. Abstract binding trees, or abt’s, enrich this representation with the concepts of binding and scope. In just about every language there is a means of associating a meaning to an identifier within a specified range of significance (perhaps the whole program, but often limited regions of it).

Abstract binding trees enrich abstract syntax trees with a means of introducing a fresh, or new, parameter within a specified scope. Uses of the parameter within that scope serve as references to the point at which the parameter is bound—it’s so-called binding site. Since bound parameters are merely references to a binding site, the name of the parameter does not matter, provided only that it does not conflict with any other parameters currently within scope. It is in this sense that a bound parameter is said to be “new” or “fresh”.

In this chapter we introduce the concept of an abstract binding tree, including the relation of \( \alpha \)-equivalence, which expresses the irrelevance of the choice of bound parameters, and the operation of capture-avoiding substitution, which ensures that parameters are not confused by substitution. While intuitively clear, the precise formalization of these concepts requires some care; experience has shown that it is surprisingly easy to get them wrong.

All of the programming languages that we shall study are represented as abstract binding trees. Consequently, we will re-use the machinery developed in this chapter many times, avoiding considerable redundancy and consolidating the effort required to make precise the notions of binding and
7.1 Abstract Binding Trees

The concepts of binding and scope are formalized by the concept of an abstract binding tree, or abt. An abt is an ast with an additional construct, called an abstractor, that introduces, or binds, a parameter for use within an abt, called the scope of the abstractor. Occurrences of the parameter within its scope are references to the abstractor at which it is bound. In this sense bound parameters behave like pronouns in natural language. Whenever we use a pronoun such as “it” in a sentence, it is understood to be a reference to an object that was specified in the context in which it occurs.

An abstractor has the form $x.a$, where $x$ is a parameter and $a$ is an abt. Such an abstractor binds the parameter, $x$, for use within its scope, the abt $a$. The parameter $x$ is meaningful only within $a$, and is, in a sense to be made precise shortly, distinct from any other parameters whose scope includes $a$. It is in this sense that an abstractor is said to introduce a “new” or “fresh” parameter for use within its scope. Making this precise requires some technical machinery, but the rough-and-ready rule is to consider each abstractor to bind a distinct parameter that serves as a reference to that binding site wherever it occurs.

As with abstract syntax trees, the definition of abstract binding trees is relative to a signature assigning arities to operators. However, to account for binding and scope, the concept of arity is generalized to be a finite sequence of natural numbers $(n_1, \ldots, n_k)$, where $k$ and each $n_i$ are natural numbers. The number $k$ determines the number of children of a node labelled with that operator, and, for each $1 \leq i \leq k$, the number $n_i$ specifies the number of parameters bound by that operator in the $i$th argument position. This number is called the valence of that argument. Only abstractors have positive valence; variables and operators form abt’s of valence zero. Since ast’s do not bind parameters, the abt arity $(0, 0, \ldots, 0)$ of length $k$ corresponds to the ast arity $k$—it specifies an operator with $k$ arguments that binds no variables in any argument.

A signature, $\Omega$, consists of a finite set of judgements $ar(o) = (n_1, \ldots, n_k)$ specifying the arity of some finite set of operators. The well-formed abt’s over a signature $\Omega$ are specified by a parametric judgement of the form

$$\{ x_1, \ldots, x_m \} \mid x_1 \text{ abt}^0, \ldots, x_m \text{ abt}^0 \vdash a \text{ abt}^n$$

(7.1)

stating that $a$ is an abt of valence $n$, with free variables $x_1, \ldots, x_m$. Let $\mathcal{X}$
range over parameter sets \{ x_1, \ldots, x_m \}, and let \( \Gamma \) range over finite sets of hypotheses of the form \( x_1 \ abt^0, \ldots, x_m \ abt^0 \). The judgement (7.1) is inductively defined by the following rules:

\[
\begin{align*}
\text{(7.2a)} \\
\overline{\text{\( X, x \mid \Gamma, x \ abt^0 \vdash x \ abt^0 \)}}
\end{align*}
\]

\[
\begin{align*}
\text{ar}(o) &= (n_1, \ldots, n_k) \\
\overline{\text{\( X \mid \Gamma \vdash a_1 \ abt^{n_1} \quad \ldots \quad X \mid \Gamma \vdash a_k \ abt^{n_k} \)}} & \text{(7.2b)}
\end{align*}
\]

\[
\begin{align*}
\overline{\text{\( X \mid \Gamma \vdash o(a_1, \ldots, a_k) \ abt^0 \)}} \\
\overline{\text{\( X, a \mid \Gamma \vdash a \ abt^n \)}} \\
\overline{\text{\( X \mid \Gamma \vdash x. a \ abt^{n+1} \)}} & \text{(7.2c)}
\end{align*}
\]

Rule (7.2c) specifies that an abstractor, \( x. a \), is an abt of valence \( n + 1 \) provided that \( a \) is an abt of valence \( n \) under the assumption that \( x \) is a “fresh” parameter of valence zero. The freshness of the parametric \( x \) is assured by the renaming convention discussed in Chapter 3. If \( x \in X \), then the premise of the rule is implicitly renamed to the judgement

\[
\overline{\text{\( X, x' \mid \Gamma, x' \ abt^0 \vdash [x \mapsto x']^+ (a) \ abt^n , \)}}
\]

where \( x' \notin X \), ensuring freshness.

For example, the language \( L\{\text{num, str}\} \) may be represented as abstract binding trees over the following signature:

\[
\begin{align*}
\text{ar}(\text{num}[n]) &= () \\
\text{ar}(\text{str}[s]) &= () \\
\text{ar}(\text{plus}) &= (0,0) \\
\text{ar}(\text{times}) &= (0,0) \\
\text{ar}(\text{cat}) &= (0,0) \\
\text{ar}(\text{len}) &= (0) \\
\text{ar}(\text{let}) &= (0,1)
\end{align*}
\]

Only the \texttt{let} operator binds a parameter, and then only in its second argument. An abt formed from the operator \texttt{let} must have the form

\[
\texttt{let}(a, x. b)
\]

where the first argument is an abt of valence zero, and the second is an abstractor of valence one. This specifies that the parameter, \( x \), is available for use within \( b \), but not within \( a \), and is distinct from all other parameters that may be within scope wherever this abt occurs.
7.1 Abstract Binding Trees

7.1.1 Structural Induction With Binding and Scope

The principle of structural induction for abstract syntax trees extends to abstract binding trees. For a fixed signature, $\Omega$, to show that $P(X | \Gamma \vdash a \ abt^n)$ whenever $X | \Gamma \vdash a \ abt^n$, it suffices to show that $P$ is closed under Rules (7.2). Specifically,

1. $P(X, x | \Gamma, x \ abt^0 \vdash x \ abt^0)$.

2. If $\Omega \vdash ar(o) = (n_1, \ldots, n_k)$ and $P(X | \Gamma \vdash a_1 \ abt^{n_1}), \ldots, P(X | \Gamma \vdash a_k \ abt^{n_k})$, then $P(X | \Gamma \vdash o(a_1, \ldots, a_k) \ abt^0)$.

3. If $P(X, x | \Gamma, x \ abt^0 \vdash a \ abt^n)$, then $P(X | \Gamma \vdash x . a \ abt^{n+1})$.

By the renaming convention discussed in Chapter 3 the inductive hypothesis for abstractors holds for all choices of fresh local parameters. This means that we may tacitly choose the parameter, $x$, to be any parameter not occurring in $X$. In practice we simply assume that $x$ has been so chosen, but in technical detail we must in general rename $x$ to some other parameter $x' \notin X$ in the case that $x \in X$.

As an example, the following rules, $H[\Omega]$, define the height of an abstract binding tree over a signature $\Omega$:

\[
\frac{X, x | \Gamma, x \ abt^0 \vdash \text{hgt}(x) = 1}{X, x | \Gamma, x \ abt^0 \vdash \text{hgt}(x) = 1} \quad (7.3a)
\]

\[
\begin{align*}
X | \Gamma \vdash \text{hgt}(a_1) &= h_1 \\
& \ldots \\
X | \Gamma \vdash \text{hgt}(a_k) &= h_k \\
\text{max}(h_1, \ldots, h_k) &= h \\
X | \Gamma \vdash \text{hgt}(o(a_1; \ldots; a_k)) &= h + 1
\end{align*} \quad (7.3b)
\]

\[
\frac{X, x | \Gamma, x \ abt^0 \vdash \text{hgt}(a) = h}{X | \Gamma \vdash \text{hgt}(x . a) = h + 1} \quad (7.3c)
\]

A straightforward structural induction shows that every well-formed abt has a height.

**Theorem 7.1.** If $X | \Gamma \vdash a \ abt^n$, then there exists a unique $h$ such that $X | \Gamma \vdash \text{hgt}(a) = h$.

Observe that this property respects renaming of parameters, since all are assigned unit height.
7.1 Abstract Binding Trees

7.1.2 Apartness

The parameter set, $X$, in the judgement $X | \Gamma \vdash a \text{ abt}^n X$ implies that the only parameters that may occur in $a$ are those in $X$. Occasionally it is useful to determine which parameters (among those that may) actually do, or do not, occur unbound in an abt.

The judgement $X, x | \Gamma, x \text{ abt}^0 \vdash x \notin a \text{ abt}^n$ states that $x$ lies apart from abt $a$. It is inductively defined by the following rules:

$$X, x | \Gamma, x \text{ abt}^0 \vdash x \notin y \text{ abt}^0$$  \hspace{1cm} (7.4a)

$$X, x | \Gamma, x \text{ abt}^0 \vdash x \notin a_1 \text{ abt}^{n_1} \ldots \Gamma, x | \Gamma, x \text{ abt}^0 \vdash x \notin a_k \text{ abt}^{n_k}$$  \hspace{1cm} (7.4b)

$$X, x, y | \Gamma, x \text{ abt}^0 \vdash x \notin y \text{ abt}^0$$  \hspace{1cm} (7.4c)

By the renaming convention the parameters $x$ and $y$ in the premise Rule (7.4c) may be assumed to be distinct from each other and to not occur in $X$.

We say that a parameter, $x$, lies within, or is free in, an abt, $a$, written $x \in a \text{ abt}$, iff it is not the case that $x \notin a \text{ abt}$.

7.1.3 Renaming of Bound Parameters

Two abt’s are said to be $\alpha$-equivalent iff they differ at most in the choice of bound parameter names. The judgement $X' | \Gamma \vdash a =_\alpha b \text{ abt}^n$ is inductively defined by the following rules:

$$X, x | \Gamma, x \text{ abt}^0 \vdash x =_\alpha x \text{ abt}^0$$  \hspace{1cm} (7.5a)

$$X | \Gamma \vdash a_1 =_\alpha b_1 \text{ abt}^{n_1} \ldots \Gamma | \Gamma \vdash a_k =_\alpha b_k \text{ abt}^{n_k}$$  \hspace{1cm} (7.5b)

$$X, z | \Gamma, z \text{ abt}^0 \vdash [x \rightarrow z]^\dagger(a) =_\alpha [y \rightarrow z]^\dagger(b) \text{ abt}^{n+1}$$  \hspace{1cm} (7.5c)

In Rule (7.5c) we tacitly assume that $z \notin X$.

We write $\Gamma \vdash a =_\alpha b$, or even just $a =_\alpha b$, for $X' | \Gamma \vdash a =_\alpha b \text{ abt}^n$ the parameters and valence are clear from context.
Lemma 7.2. The following instance of \( \alpha \)-equivalence, called \( \alpha \)-conversion, is derivable:

\[
\mathcal{X} \mid \Gamma \vdash x . a =_\alpha y . [x \mapsto y]^\dagger (a) \ \text{abt}^{n+1} \quad (y \notin \mathcal{X})
\]

Theorem 7.3. \( \alpha \)-equivalence is reflexive, symmetric, and transitive.

Proof. Reflexivity and symmetry are immediately obvious from the form of the definition. Transitivity is proved by a simultaneous induction on the derivations of \( \mathcal{X} \mid \Gamma \vdash a =_\alpha b \ \text{abt}^n \) and \( \mathcal{X} \mid \Gamma \vdash b =_\alpha c \ \text{abt}^n \). The most interesting case is when both derivations end with Rule (7.5c). We have \( a = x . a' \), \( b = y . b' \), \( c = z . c' \), and \( n = m + 1 \) for some \( m \). By the renaming convention we also have

\[
\mathcal{X}, u \mid \Gamma, u \ \text{abt}^0 \vdash [x \mapsto u]^\dagger (a') =_\alpha [y \mapsto u]^\dagger (b') \ \text{abt}^m
\]

where \( u \notin \mathcal{X} \), and

\[
\mathcal{X}, u \mid \Gamma, u \ \text{abt}^0 \vdash [y \mapsto u]^\dagger (b') =_\alpha [z \mapsto u]^\dagger (c') \ \text{abt}^m,
\]

where \( u \notin \mathcal{X} \). The result then follows immediately by an application of Rule (7.5c). \( \square \)

7.1.4 Substitution

Substitution is the process of replacing all occurrences (if any) of a free parameter in an abt by another abt in such a way that the scopes of parameters are properly respected. The judgement \( \mathcal{X} \mid \Gamma \vdash [a/x]b = c \ \text{abt}^n \) is inductively defined by the following rules:

\[
\begin{align*}
\frac{\mathcal{X} \mid \Gamma \vdash [a/x]x = a \ \text{abt}^0}{\mathcal{X}} \quad (7.6a) \\
\frac{x \neq y}{\mathcal{X}, y \mid \Gamma, y \ \text{abt}^0 \vdash [a/x]y = y \ \text{abt}^0} \quad (7.6b) \\
\frac{\mathcal{X} \mid \Gamma \vdash [a/x]b_1 = c_1 \ \text{abt}^{n_1} \ldots \mathcal{X} \mid \Gamma \vdash [a/x]b_k = c_k \ \text{abt}^{n_k}}{\mathcal{X} \mid \Gamma \vdash [a/x]o(b_1, \ldots, b_k) = o(c_1, \ldots, c_k) \ \text{abt}^0} \quad (7.6c)
\end{align*}
\]
In Rule (7.6d) we may assume (by the renaming convention) that \( y \not\in \chi \), so that if the free parameters of \( a \) are drawn from \( \chi \), then \( y \not\in a \) abt. This latter condition is called avoidance of capture, for if \( y \in a \) abt, and \( x \in b \) abt, then occurrences of \( y \) in \( c \) would refer improperly to the abstractor \( y.a \), rather than to the surrounding binding site.

The penalty for avoiding capture during substitution is that the result of performing a substitution is only determined up to \( \alpha \)-equivalence. To see this, let us re-state Rule (7.6d) with the use of the renaming convention made explicit:

\[
\chi, y' \mid \Gamma, y \ abt^0 \vdash [a/x]b = c \ abt^n \quad x \neq y \quad y' \not\in \chi
\]

\[
\chi \mid \Gamma \vdash [a/x]y.b = y'.[y\mapsto\rightarrow y']^\dagger (c) \ abt^n
\]

Since \( y'.[y\mapsto\rightarrow y']^\dagger (c) \) is \( \alpha \)-equivalent \( y.c \), we see that the result of substitution is determined only up to the names of bound variables.

**Theorem 7.4.** If \( \chi \mid \Gamma \vdash a \ abt^0 \) and \( \chi, x \mid \Gamma, x \ abt^0 \vdash b \ abt^n \), then there exists \( \chi \mid \Gamma \vdash c \ abt^n \) such that \( \chi \mid \Gamma \vdash [a/x]b = c \ abt^n \). If \( \chi \mid \Gamma \vdash [a/x]b = c \ abt^n \) and \( \chi \mid \Gamma \vdash [a/x]b = c' \ abt^n \), then \( \chi \mid \Gamma \vdash c =_\alpha c' \ abt^n \).

**Proof.** The first part is proved by rule induction on \( \chi \mid \Gamma, x \ abt^0 \vdash b \ abt^n \), in each case constructing the required derivation of the substitution judgement. The second part is proved by simultaneous rule induction on the two premises, deriving the desired equivalence in each case.

Even though the result is not uniquely determined, we abuse notation and write \( [a/x]b \) for any \( c \) such that \( [a/x]b = c \), with the understanding that \( c \) is determined only up to choice of names of bound parameters.

### 7.2 Exercises

1. Suppose that `let` is an operator of arity \((0,1)\) and that `plus` is an operator of arity \((0,0)\). Determine whether or not each of the following
\( \alpha \)-equivalences are valid.

\[
\begin{align*}
\text{let}(x,x.x) &= \alpha \text{let}(x,y.y) & (7.8a) \\
\text{let}(y,x.x) &= \alpha \text{let}(y,y.y) & (7.8b) \\
\text{let}(x,x.x) &= \alpha \text{let}(y,y.y) & (7.8c) \\
\text{let}(x,x.\text{plus}(x,y)) &= \alpha \text{let}(x,z.\text{plus}(z,y)) & (7.8d) \\
\text{let}(x,x.\text{plus}(x,y)) &= \alpha \text{let}(x,y.\text{plus}(y,y)) & (7.8e)
\end{align*}
\]

2. Prove that apartness respects \( \alpha \)-equivalence.

3. Prove that substitution respects \( \alpha \)-equivalence.
Chapter 8

Parsing

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.

8.1 Parsing Into Abstract Syntax Trees

The process of translation from concrete to abstract syntax is called parsing. We will define parsing as a judgement between the concrete and abstract syntax of \( L\{\text{num str}\} \) given in Chapter 6. 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.

The parsing judgements for \( L\{\text{num str}\} \) follow the unambiguous gram-
mar given in Chapter 5:

- \( s \text{ prg} \leftarrow a \text{ ast} \) Parse as a program
- \( s \text{ exp} \leftarrow a \text{ ast} \) Parse as an expression
- \( s \text{ trm} \leftarrow a \text{ ast} \) Parse as a term
- \( s \text{ fct} \leftarrow a \text{ ast} \) Parse as a factor
- \( s \text{ num} \leftarrow a \text{ ast} \) Parse as a number
- \( s \text{ lit} \leftarrow a \text{ ast} \) Parse as a literal
- \( s \text{ id} \leftarrow a \text{ ast} \) Parse as an identifier

These judgements are inductively defined simultaneously by the following rules:

- \[ \frac{n \text{ nat}}{\text{NUM}[n] \text{ num} \leftarrow \text{num}[n] \text{ ast}} \] (8.1a)
- \[ \frac{s \text{ str}}{\text{LIT}[s] \text{ lit} \leftarrow \text{str}[s] \text{ ast}} \] (8.1b)
- \[ \frac{s \text{ str}}{\text{ID}[s] \text{ id} \leftarrow \text{id}[s] \text{ ast}} \] (8.1c)
- \[ \frac{s \text{ num} \leftarrow a \text{ ast}}{s \text{ fct} \leftarrow a \text{ ast}} \] (8.1d)
- \[ \frac{s \text{ lit} \leftarrow a \text{ ast}}{s \text{ fct} \leftarrow a \text{ ast}} \] (8.1e)
- \[ \frac{s \text{ id} \leftarrow a \text{ ast}}{s \text{ fct} \leftarrow a \text{ ast}} \] (8.1f)
- \[ \frac{s \text{ prg} \leftarrow a \text{ ast}}{\text{LP}s \text{ RP fct} \leftarrow a \text{ ast}} \] (8.1g)
- \[ \frac{s \text{ fct} \leftarrow a \text{ ast}}{s \text{ trm} \leftarrow a \text{ ast}} \] (8.1h)
- \[ \frac{s_1 \text{ fct} \leftarrow a_1 \text{ ast}}{s_2 \text{ trm} \leftarrow a_2 \text{ ast}} \]
- \[ s_1 \text{ MUL} s_2 \text{ trm} \leftarrow \text{times}(a_1; a_2) \text{ ast} \] (8.1i)
- \[ \frac{s \text{ fct} \leftarrow a \text{ ast}}{\text{VB}s \text{ VB trm} \leftarrow \text{len}(a) \text{ ast}} \] (8.1j)
- \[ \frac{s \text{ trm} \leftarrow a \text{ ast}}{s \text{ exp} \leftarrow a \text{ ast}} \] (8.1k)
8.1 Parsing Into Abstract Syntax Trees

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 8.1.** If \( s \) prg \( \longrightarrow a \) ast, then \( s \) prg and \( a \) ast, and similarly for the other parsing judgements.

**Proof.** By rule induction on Rules (8.1).

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

**Theorem 8.2.** If \( s \) prg, then there is a unique \( a \) such that \( s \) prg \( \longrightarrow a \) ast, 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 8.3.** If \( a \) ast, then there exists a (not necessarily unique) string \( s \) such that \( s \) prg and \( s \) prg \( \longrightarrow a \) ast. 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.
8.2 Parsing Into Abstract Binding Trees

In this section we revise the parser given in Section 8.1 on page 61 to translate from token strings to abstract binding trees to make explicit the binding and scope of identifiers in a program. We will work over the signature given in Chapter 7 defining the abt representation of \( L\{\text{num str}\} \).

The revised parsing judgement, \( s \overset{\text{prg}}{\leftrightarrow} a \ abt \), between strings \( s \) and abt's \( a \), is defined by a collection of rules similar to those given in Section 8.1 on page 61. 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

\[
\text{ID}[s_1] \ id \leftarrow x_1 \ abt, \ldots, \text{ID}[s_n] \ id \leftarrow x_n \ abt \vdash \ s \overset{\text{prg}}{\leftrightarrow} a \ abt,
\]

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

\[
\Gamma, \text{ID}[s] \ id \leftarrow x \ abt \vdash \text{ID}[s] \ id \leftarrow x \ abt.
\]

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

\[
\Gamma \vdash s_1 \overset{\text{id}}{\leftrightarrow} x \ abt \quad \Gamma \vdash s_2 \overset{\text{exp}}{\leftrightarrow} a_2 \ abt \\
\Gamma, s_1 \overset{\text{id}}{\leftrightarrow} x \ abt \vdash s_3 \overset{\text{prg}}{\leftrightarrow} a_3 \ abt \\
\Gamma \vdash \text{LET} s_1 \overset{\text{BE}}{\leftrightarrow} s_2 \overset{\text{IN}}{\leftrightarrow} s_3 \overset{\text{prg}}{\leftrightarrow} \text{let}(a_2; x. a_3) \ abt
\]

Unfortunately, this approach does not quite work properly! If an inner \texttt{let} expression binds the same identifier as an outer \texttt{let} expression, there is an ambiguity in how to parse occurrences of that identifier. Parsing such nested \texttt{let}'s will introduce two hypotheses, say \text{ID}[] \ id \leftarrow x_1 \ abt and \text{ID}[] \ id \leftarrow x_2 \ abt, for the same identifier \text{ID}[]. By the structural property of exchange, we may choose arbitrarily which to apply to any particular occurrence of \text{ID}[], and hence we may parse different occurrences differently.

To rectify this we must 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
8.2 Parsing Into Abstract Binding Trees

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

$$\Gamma \vdash \text{prg} \leftrightarrow \text{abt}$$

is reduced to the categorical judgement

$$\text{prg} \leftrightarrow \text{abt}[\sigma]$$

where $\sigma$ 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:

$$\begin{align*}
  s_1 \text{id} & \leftrightarrow x [\sigma] \\
  s_2 \text{exp} & \leftrightarrow a_2 \text{abt}[\sigma] \\
  \sigma' = \sigma[s_1 \mapsto x] \\
  s_3 \text{prg} & \leftrightarrow a_3 \text{abt}[\sigma']
\end{align*}$$

(8.3)

$$\text{LET } s_1 \text{ BE } s_2 \text{ IN } s_3 \text{ prg} \leftrightarrow \text{let}(a_2; x.a_3) \text{ abt}[\sigma]$$

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.

$$\sigma(\text{ID}[s]) = x$$

(8.4)

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

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

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

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

To specify a language we shall use a concise tabular notation for simultaneously specifying both its abstract and concrete syntax. Officially, the language is always a collection of abt's, but when writing examples we shall often use the concrete notation for the sake of concision and clarity. Our method of specifying the concrete syntax is sufficient for our purposes, but leaves out niggling details such as precedences of operators or the use of bracketing to disambiguate.

The method is best illustrated by example. Here is a specification of the syntax of $L\{\text{num,str}\}$ presented in the tabular style that we shall use throughout the book:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$\text{num}$</td>
<td>$\text{num}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{str}$</td>
<td>$\text{str}$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$x$</td>
<td>$x$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{num}[n]$</td>
<td>$n$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{str}[s]$</td>
<td>&quot;s&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{plus}(e_1;e_2)$</td>
<td>$e_1 + e_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{times}(e_1;e_2)$</td>
<td>$e_1 \times e_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{cat}(e_1;e_2)$</td>
<td>$e_1 \cdot e_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{len}(e)$</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{let}(e_1;x.e_2)$</td>
<td>$\text{let } x \text{ be } e_1 \text{ in } e_2$</td>
</tr>
</tbody>
</table>

This specification is to be understood as defining two judgments, $\tau_{type}$ and $\tau_{exp}$, which specify two syntactic categories, one for types, the other for expressions. The abstract syntax column uses patterns ranging over abt’s to determine the arities of the operators for that syntactic category. The concrete syntax column specifies the typical notational conventions used in examples. In this manner Table (8.3) defines two signatures, $\Omega_{type}$ and $\Omega_{expr}$, that specify the operators for types and expressions, respectively. The signature for types specifies that $\text{num}$ and $\text{str}$ are two operators of arity (). The signature for expressions specifies two families of operators, $\text{num}[n]$ and $\text{str}[s]$, of arity (), three operators of arity (0, 0) corresponding to addition, multiplication, and concatenation, one operator of arity (0) for length, and one operator of arity (0, 1) for let-binding expressions to identifiers.

8.4 Exercises
Part III

Static and Dynamic Semantics
Chapter 9

Static Semantics

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 static semantics 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 static semantics is considered to be improperly defined, and the language is deemed unsafe for execution.

In this chapter we present the static semantics of the language $L\{\text{num str}\}$ as an illustration of the methodology that we shall employ throughout this book.

9.1 Type System
Recall that the abstract syntax of $\mathcal{L}\{\text{num str}\}$ is given by Grammar (8.3), which we repeat here for convenience:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$ ::=</td>
<td>$\text{num}$</td>
<td>num</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{str}$</td>
<td>str</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$ ::=</td>
<td>$x$</td>
<td>$x$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{num}[n]$</td>
<td>$n$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{str}[s]$</td>
<td>&quot;s&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{plus}(e_1;e_2)$</td>
<td>$e_1 + e_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{times}(e_1;e_2)$</td>
<td>$e_1 \ast e_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{cat}(e_1;e_2)$</td>
<td>$e_1 ^ e_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{len}(e)$</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{let}(e_1;x.e_2)$</td>
<td>$\text{let } x \text{ be } e_1 \text{ in } e_2$</td>
</tr>
</tbody>
</table>

According to the conventions discussed in Chapter 8, this grammar defines two judgements, $\tau$ type defining the category of types, and $e$ expr defining the category of expressions.

The role of a static semantics 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 $\text{plus}(x;\text{num}[n])$ 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 static semantics of $\mathcal{L}\{\text{num str}\}$ consists of an inductive definition of parametric hypothetical judgements of the form

$$\mathcal{X} \mid \Gamma \vdash e : \tau,$$

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

The rules defining the static semantics of $\mathcal{L}\{\text{num str}\}$ are as follows:

$$\Gamma, x : \tau \vdash x : \tau \quad (9.1a)$$

$$\Gamma \vdash \text{str}[s] : \text{str} \quad (9.1b)$$
9.1 Type System

\[ \Gamma \vdash \text{num} [n] : \text{num} \]  \hspace{1cm} (9.1c)

\[ \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash \text{plus}(e_1; e_2) : \text{num}} \]  \hspace{1cm} (9.1d)

\[ \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash \text{times}(e_1; e_2) : \text{num}} \]  \hspace{1cm} (9.1e)

\[ \frac{\Gamma \vdash e_1 : \text{str} \quad \Gamma \vdash e_2 : \text{str}}{\Gamma \vdash \text{cat}(e_1; e_2) : \text{str}} \]  \hspace{1cm} (9.1f)

\[ \frac{\Gamma \vdash e : \text{str}}{\Gamma \vdash \text{len}(e) : \text{num}} \]  \hspace{1cm} (9.1g)

\[ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let}(e_1; x.e_2) : \tau_2} \]  \hspace{1cm} (9.1h)

In Rule (9.1h) we tacitly assume that the variable, \( x \), is not already declared in \( \Gamma \). This condition may always be met by choosing a suitable representative of the \( \alpha \)-equivalence class of the \texttt{let} expression.

Rules (9.1) 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 \( L\{\text{num str}\} \), the introductory forms for the type \( \text{num} \) are the numerals, \( \text{num}[n] \), and for the type \( \text{str} \) are the literals, \( \text{str}[s] \). 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 \( \text{num} \) are addition and multiplication, and for the type \( \text{str} \) 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 9.1** (Unicity of Typing). For every typing context \( \Gamma \) and expression \( e \), there exists at most one \( \tau \) such that \( \Gamma \vdash e : \tau \).

**Proof.** By rule induction on Rules (9.1). \( \square \)
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 9.2** (Inversion for Typing). Suppose that \( \Gamma \vdash e : \tau \). If \( e = \text{plus}(e_1; e_2) \), then \( \tau = \text{num} \), \( \Gamma \vdash e_1 : \text{num} \), and \( \Gamma \vdash e_2 : \text{num} \), and similarly for the other constructs of the language.

**Proof.** These may all be proved by induction on the derivation of the typing judgement \( \Gamma \vdash e : \tau \). \( \square \)

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

## 9.2 Structural Properties

The static semantics enjoys the structural properties of the parametric hypothetical judgement.

**Lemma 9.3** (Weakening). If \( \Gamma \vdash e' : \tau' \), then \( \Gamma, x : \tau \vdash e' : \tau' \) for any \( x \notin \text{dom}(\Gamma) \) and any \( \tau \) type.

**Proof.** By induction on the derivation of \( \Gamma \vdash e' : \tau' \). We will give one case here, for rule (9.1h). We have that \( e' = \text{let}(e_1; z.e_2) \), where by the conventions on parameters we may assume \( z \) is chosen such that \( z \notin \text{dom}(\Gamma) \) 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 (9.1h). \( \square \)

**Lemma 9.4** (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 (9.1h). As in the preceding case, \( e' = \text{let}(e_1; z.e_2) \), where \( z \) may be chosen so that \( z \neq x \) and \( z \notin \text{dom}(\Gamma) \). We have by induction
9.2 Structural Properties

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

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

It follows by Rule (9.1h) that $\Gamma \vdash [e/x]\text{let}(e_1; z.e_2) : \tau,$ as desired. \qed

From a programming point of view, Lemma 9.3 on the preceding page allows us to use an expression in any context that binds its free variables: if $e$ is well-typed in a context $\Gamma,$ then we may “import” it into any context that includes the assumptions $\Gamma.$ In other words the introduction of new variables beyond those required by an expression, $e,$ does not invalidate $e$ itself; it remains well-formed, with the same type.\footnote{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.} More significantly, Lemma 9.4 on the facing 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 9.4 on the preceding 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 9.5 (Decomposition).** If $\Gamma \vdash [e/x]e' : \tau',$ then for every type $\tau$ such that $\Gamma \vdash e : \tau,$ we have $\Gamma, x : \tau \vdash e' : \tau'.$

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

OCTOBER 16, 2009  DRAFT  18:42
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 $e'$.

9.3 Exercises

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

18:42 Draft October 16, 2009
Chapter 10

Dynamic Semantics

The dynamic semantics of a language specifies how programs are to be executed. One important method for specifying dynamic semantics is called structural semantics, which consists of a collection of rules defining a transition system whose states are expressions with no free variables. Contextual semantics may be viewed as an alternative presentation of the structural semantics of a language. Another important method for specifying dynamic semantics, called evaluation semantics, is the subject of Chapter 12.

10.1 Structural Semantics

A structural semantics for $\mathcal{L}\{\text{num,str}\}$ consists of a transition system whose states are closed expressions, all of which are initial states. The final states are the closed values, as defined by the following rules:

\begin{align*}
\text{num}[n] \text{ val} \quad (10.1a) \\
\text{str}[s] \text{ val} \quad (10.1b)
\end{align*}

The transition judgement, $e \mapsto e'$, is also inductively defined.

\begin{align*}
n_1 + n_2 = n \text{ nat} \\
\text{plus}(\text{num}[n_1];\text{num}[n_2]) \mapsto \text{num}[n] \quad (10.2a)
\end{align*}

\begin{align*}
e_1 \mapsto e_1' \\
\text{plus}(e_1;e_2) \mapsto \text{plus}(e_1';e_2) \quad (10.2b)
\end{align*}
\[ e_1 \text{ val} \quad e_2 \mapsto e'_2 \]
\[ \text{plus}(e_1; e_2) \mapsto \text{plus}(e_1; e'_2) \]  
(10.2c)

\[ s_1 \hat{\ast} s_2 = s \text{ str} \]
\[ \text{cat}(\text{str}[s_1]; \text{str}[s_2]) \mapsto \text{str}[s] \]  
(10.2d)

\[ e_1 \mapsto e'_1 \]
\[ \text{cat}(e_1; e_2) \mapsto \text{cat}(e'_1; e_2) \]  
(10.2e)

\[ e_1 \text{ val} \quad e_2 \mapsto e'_2 \]
\[ \text{cat}(e_1; e_2) \mapsto \text{cat}(e_1; e'_2) \]  
(10.2f)

\[ \text{let}(e_1; x. e_2) \mapsto [e_1/x]e_2 \]  
(10.2g)

We have omitted rules for multiplication and computing the length of a string, which follow a similar pattern. Rules (10.2a), (10.2d), and (10.2g) 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 (10.2) exhibit structure arising from the principle of introduction and elimination discussed in Chapter 9. The instruction transitions express the inversion principle, which states that eliminatory forms are inverse to introductory forms. For example, Rule (10.2a) 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 (10.2g) 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 occurrence. To avoid repeated work in the latter case, we may instead specify a by-value interpretation of binding by the following rules:

\[ e_1 \text{ val} \]
\[ \text{let}(e_1; x. e_2) \mapsto [e_1/x]e_2 \]  
(10.3a)

\(^1\)The justification for the terminology “by name” is obscure, but as it is very well-established we shall stick with it.
10. Structural Semantics

\[
\begin{align*}
e_1 & \mapsto e_1' \\
\text{let}(e_1; x. e_2) & \mapsto \text{let}(e_1'; x. e_2)
\end{align*}
\]

(10.3b)

Rule (10.3b) is an additional search rule specifying that we may evaluate \( e_1 \) before \( e_2 \). Rule (10.3a) ensures that \( e_2 \) is not evaluated until evaluation of \( e_1 \) is complete.

A derivation sequence in a structural semantics 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.

\[
\begin{align*}
\text{let}(\text{plus}(\text{num}[1]; \text{num}[2]); x. \text{plus}(\text{plus}(x; \text{num}[3]); \text{num}[4])) & \mapsto \text{let}(\text{num}[3]; x. \text{plus}(\text{plus}(x; \text{num}[3]); \text{num}[4])) \\
& \mapsto \text{plus}(\text{plus}(\text{num}[3]; \text{num}[3]); \text{num}[4]) \\
& \mapsto \text{plus}(\text{num}[6]; \text{num}[4]) \\
& \mapsto \text{num}[10]
\end{align*}
\]

Each step in this sequence of transitions is justified by a derivation according to Rules (10.2). For example, the third transition in the preceding example is justified by the following derivation:

\[
\begin{align*}
\text{plus}(\text{num}[3]; \text{num}[3]) & \mapsto \text{num}[6] \quad (10.2a) \\
\text{plus}(\text{plus}(\text{num}[3]; \text{num}[3]); \text{num}[4]) & \mapsto \text{plus}(\text{num}[6]; \text{num}[4]) \quad (10.2b)
\end{align*}
\]

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

The principle of rule induction for the structural semantics of \( \mathcal{L}\{\text{num str}\} \) 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 (10.2). For example, we may show by rule induction that structural semantics of \( \mathcal{L}\{\text{num str}\} \) is determinate.

**Lemma 10.1** (Determinacy). If \( e \mapsto e' \) and \( e \mapsto e'' \), then \( e' \) and \( e'' \) are \( \alpha \)-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. \( \square \)
10.2 Contextual Semantics

A variant of structural semantics, called contextual semantics, 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 $L\{ \text{num str} \}$ is defined by the following rules, together with similar rules for multiplication of numbers and the length of a string.

\[
\begin{align*}
\text{plus}(\text{num}[m]; \text{num}[n]) & \leadsto \text{num}[p] \\
\text{cat}(\text{str}[s]; \text{str}[t]) & \leadsto \text{str}[u] \\
\text{let}(e_1; x. e_2) & \leadsto [e_1/x]e_2
\end{align*}
\]

The judgement $E\, \text{ectxt}$ 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.)

\[
\begin{align*}
\circ \, \text{ectxt} & \\
E_1 \, \text{ectxt} & \quad \text{plus}(E_1; e_2) \, \text{ectxt} \\
\text{val}(e_1) & \quad \text{E}_2 \, \text{ectxt} \\
\text{plus}(e_1; E_2) & \, \text{ectxt}
\end{align*}
\]

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 semantics. For example, Rule (10.5c) states that in an expression $\text{plus}(e_1; e_2)$, 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, $e_2$. 

18:42 Draft October 16, 2009
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:

\[
\begin{align*}
  e &= \circ\{e\} \\
  e_1 &= \mathcal{E}_1\{e\} \\
  \text{plus}(e_1; e_2) &= \text{plus}(\mathcal{E}_1; e_2)\{e\} \\
  e_1 \text{ val} & \quad e_2 = \mathcal{E}_2\{e\} \\
  \text{plus}(e_1; e_2) &= \text{plus}(e_1; \mathcal{E}_2)\{e\}
\end{align*}
\] (10.6)

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 dynamic semantics for \( \mathcal{L}\{\text{num str}\} \) is defined using contextual semantics by a single rule:

\[
\begin{align*}
  e &= \mathcal{E}\{e_0\} \\
  e_0 & \rightsquigarrow e'_0 \\
  e' &= \mathcal{E}\{e'_0\}
\end{align*}
\] (10.7)

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 semantics define the same transition relation. For the sake of the proof, let us write \( e \rightsquigarrow_s e' \) for the transition relation defined by the structural semantics (Rules (10.2)), and \( e \rightsquigarrow_c e' \) for the transition relation defined by the contextual semantics (Rules (10.7)).

**Theorem 10.2.** \( e \rightsquigarrow_s e' \) if, and only if, \( e \rightsquigarrow_c e' \).

**Proof.** From left to right, proceed by rule induction on Rules (10.2). 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 (10.2a), take \( \mathcal{E} = \circ \), and observe that \( e \rightsquigarrow e' \). For Rule (10.2b), 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 \( \mathcal{E} = \text{plus}(\mathcal{E}_1; e_2) \), and observe that \( e = \text{plus}(\mathcal{E}_1; e_2)\{e_0\} \) and \( e' = \text{plus}(\mathcal{E}_1; e_2)\{e'_0\} \) with \( e_0 \rightsquigarrow e'_0 \).
From right to left, observe that if \( e \mapsto_c e' \), then there exists an evaluation context \( \mathcal{E} \) such that \( e = \mathcal{E}\{e_0\} \), \( e' = \mathcal{E}\{e'_0\} \), and \( e_0 \mapsto e'_0 \). We prove by induction on Rules (10.6) that \( e \mapsto s e' \). For example, for Rule (10.6a), \( e_0 \) is \( e \), \( e'_0 \) is \( e' \), and \( e \mapsto e' \). Hence \( e \mapsto s e' \). For Rule (10.6b), we have that \( \mathcal{E} = \text{plus}(\mathcal{E}_1; e_2) \), \( e_1 = \mathcal{E}_1\{e_0\} \), \( e'_1 = \mathcal{E}_1\{e'_0\} \), and \( e_1 \mapsto s e'_1 \). Therefore \( e \) is \( \text{plus}(e_1; e_2) \), \( e' \) is \( \text{plus}(e'_1; e_2) \), and therefore by Rule (10.2b), \( e \mapsto s e' \).

Since the two transition judgements coincide, contextual semantics may be seen as an alternative way of presenting a structural semantics. It has two advantages over structural semantics, one relatively superficial, one rather less so. The superficial advantage stems from writing Rule (10.7) in the simpler form

\[
\begin{align*}
\frac{e_0 \mapsto e'_0}{\mathcal{E}\{e_0\} \mapsto \mathcal{E}\{e'_0\}}.
\end{align*}
\]

(10.8)

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 15, is that the transition judgement in contextual semantics applies only to closed expressions of a fixed type, whereas structural semantics transitions are necessarily defined over expressions of every type.

### 10.3 Equational Semantics

Another formulation of the dynamic semantics of a language is based on regarding computation as a form of equational deduction, much in the style of elementary algebra. For example, in algebra we may show that the polynomials \( x^2 + 2x + 1 \) and \( (x + 1)^2 \) are equivalent by a simple process of calculation and re-organization using the familiar laws of addition and multiplication. The same laws are sufficient to determine the value of any polynomial, given the values of its variables. So, for example, we may plug in 2 for \( x \) in the polynomial \( x^2 + 2x + 1 \) and calculate that \( 2^2 + 2 \cdot 2 + 1 = 9 \), 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 definition, or computational, equivalence of expressions in \( \mathcal{L}\{\text{num} \, \text{str}\} \), which we write as \( \forall \mathcal{L}_e \mid \Gamma \vdash e \equiv e' : \tau \), where \( \Gamma \) consists of one assumption of the form \( x : \tau \) for each
10.3 Equational Semantics

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 $\Gamma$.

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

\[
\frac{}{\Gamma \vdash e \equiv e : \tau} \tag{10.9a}
\]

\[
\frac{}{\Gamma \vdash e' \equiv e : \tau} \quad \frac{}{\Gamma \vdash e \equiv e' : \tau} 
\]

\[
\frac{\Gamma \vdash e \equiv e' : \tau \quad \Gamma \vdash e' \equiv e'' : \tau}{\Gamma \vdash e \equiv e'' : \tau} \tag{10.9c}
\]

\[
\frac{\Gamma \vdash e_1 \equiv e_1' : \text{num} \quad \Gamma \vdash e_2 \equiv e_2' : \text{num}}{\Gamma \vdash \text{plus}(e_1; e_2) \equiv \text{plus}(e_1'; e_2') : \text{num}} \tag{10.9d}
\]

\[
\frac{\Gamma \vdash e_1 \equiv e_1' : \text{str} \quad \Gamma \vdash e_2 \equiv e_2' : \text{str}}{\Gamma \vdash \text{cat}(e_1; e_2) \equiv \text{cat}(e_1'; e_2') : \text{str}} \tag{10.9e}
\]

\[
\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \text{let}(e_1; x. e_2) \equiv \text{let}(e_1'; x. e_2') : \tau_2} \tag{10.9f}
\]

\[
n_1 + n_2 = n \text{ nat} 
\frac{}{\Gamma \vdash \text{plus}(\text{num}[n_1]; \text{num}[n_2]) \equiv \text{num}[n] : \text{num}} \tag{10.9g}
\]

\[
s_1 \hat{s}_2 = s \text{ str} 
\frac{}{\Gamma \vdash \text{cat}(\text{str}[s_1]; \text{str}[s_2]) \equiv \text{str}[s] : \text{str}} \tag{10.9h}
\]

\[
\frac{}{\Gamma \vdash \text{let}(e_1; x. e_2) \equiv [e_1/x]e_2 : \tau} \tag{10.9i}
\]

Rules (10.9a) through (10.9c) state that definitional equivalence is an equivalence relation. Rules (10.9d) through (10.9f) state that it is a congruence relation, which means that it is compatible with all expression-forming constructs in the language. Rules (10.9g) through (10.9i) specify the meanings of the primitive constructs of $\mathcal{L}\{\text{num str}\}$. For the sake of concision, Rules (10.9) may be characterized as defining the strongest congruence closed under Rules (10.9g), (10.9h), and (10.9i).
Rules (10.9) 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

\[ \text{let } x \text{ be } 1 + 2 \text{ in } x + 3 + 4 \equiv 10 : \text{num} \]

by applying Rules (10.9). 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 (10.9). A prototypical example is the putative equivalence

\[ x : \text{num}, y : \text{num} \vdash x_1 + x_2 \equiv x_2 + x_1 : \text{num}, \]  

which, intuitively, expresses the commutativity of addition. Although we shall not prove this here, this equivalence is not derivable from Rules (10.9). And yet we may derive all of its closed instances,

\[ n_1 + n_2 \equiv n_2 + n_1 : \text{num}, \]  

where \( n_1 \) nat and \( n_2 \) nat are particular numbers.

The “gap” between a general law, such as Equation (10.10), and all of its instances, given by Equation (10.11), 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 \textit{semantic}, or \textit{observational}, equivalence, since it expresses relationships that hold by virtue of the semantics of the expressions involved.\footnote{This rather vague concept of equivalence is developed rigorously in Chapter 50.} Semantic equivalence is a \textit{synthetic judgment}, one that requires proof. It is to be distinguished from definitional equivalence, which expresses an \textit{analytic judgement}, one that is self-evident based solely on the dynamic semantics of the operations involved. As such definitional equivalence may be thought of as \textit{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.

\textbf{Theorem 10.3.} \( e \equiv e' : \tau \) iff there exists \( e_0 \) \textit{val} such that \( e \mapsto^* e_0 \) and \( e' \mapsto^* e_0 \).
10.3 Equational Semantics  

**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 \( e_0 \) such that
\[
[e_1, \ldots, e_n / x_1, \ldots, x_n]e \mapsto^* e_0
\]
and
\[
[e_1, \ldots, e_n / x_1, \ldots, x_n]e' \mapsto^* e_0.
\]
This is proved by rule induction on Rules (10.9).

The formulation of definitional equivalence for the by-value semantics of binding requires a bit of additional machinery. The key idea is motivated by the modifications required to Rule (10.9i) 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:

\[
\frac{e_1 \text{ val}}{\Gamma \vdash \text{let}(e_1; x. e_2) \equiv [e_1/x]e_2 : \tau}
\]  

(10.12)

This is almost correct, except that the judgement \( e \) \text{ val} is defined only for closed expressions, whereas \( e_1 \) might well involve free variables in \( \Gamma \). What is required is to extend the judgement \( e \) \text{ val} to the hypothetical judgement

\[
\Gamma, x_1 \text{ val}, \ldots, x_n \text{ val} \vdash e \text{ val}
\]

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, \( \Xi \), of such hypotheses as part of definitional equivalence, writing \( \Xi \Gamma \vdash e \equiv e' : \tau \), and modifying Rule (10.9f) as follows:

\[
\frac{\Xi \Gamma \vdash e_1 \equiv e'_1 : \tau_1 \quad \Xi \text{ val} \Gamma, x : \tau_1 \vdash e_2 \equiv e'_2 : \tau_2}{\Xi \Gamma \vdash \text{let}(e_1; x. e_2) \equiv \text{let}(e'_1; x. e'_2) : \tau_2}
\]  

(10.13)

The other rules are correspondingly modified to simply carry along \( \Xi \) is an additional set of hypotheses of the inference.
10.4 Exercises

1. For the structural operational semantics of $L\{\text{num str}\}$, prove that if $e \mapsto e_1$ and $e \mapsto e_2$, then $e_1 =_a e_2$.

2. Formulate a variation of $L\{\text{num str}\}$ with both a by-name and a by-value $\text{let}$ construct.
Chapter 11

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 \( \mathcal{L}\{\text{num str}\} \) 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 static and the dynamic semantics. The static semantics may be seen as predicting that the value of an expression will have a certain form so that the dynamic semantics 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 \( \mathcal{L}\{\text{num str}\} \) may be stated as follows:

**Theorem 11.1 (Type Safety).**  
1. If \( e : \tau \) and \( e \rightarrow e' \), then \( e' : \tau \).
2. If \( e : \tau \), then either \( e \text{ val} \), or there exists \( e' \) such that \( e \rightarrow e' \).

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 \( e \rightarrow e' \). 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.

11.1 Preservation

The preservation theorem for \( L_{\{\text{num, str}\}} \) defined in Chapters 9 and 10 is proved by rule induction on the transition system (rules (10.2)).

**Theorem 11.2** (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 (10.2b),

\[
\begin{align*}
e_1 & \mapsto e'_1 \\
\text{plus}(e_1; e_2) & \mapsto \text{plus}(e'_1; e_2)
\end{align*}
\]

Assume that \( \text{plus}(e_1; e_2) : \tau \). By inversion for typing, we have that \( \tau = \text{num} \), \( e_1 : \text{num} \), and \( e_2 : \text{num} \). By induction we have that \( e'_1 : \text{num} \), and hence \( \text{plus}(e'_1; e_2) : \text{num} \). The case for concatenation is handled similarly.

Now consider rule (10.2g),

\[
\begin{align*}
e_1 & \text{val} \\
\text{let}(e_1; x.e_2) & \mapsto [e_1/x]e_2
\end{align*}
\]

Assume that \( \text{let}(e_1; x.e_2) : \tau_2 \). By the inversion lemma 9.2 on page 72, \( e_1 : \tau_1 \) for some \( \tau_1 \) such that \( x : \tau_1 \vdash e_2 : \tau_2 \). By the substitution lemma 9.4 on page 72 \( [e_1/x]e_2 : \tau_2 \), 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.

11.2 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 11.3** (Canonical Forms). If \( e \text{ val and } e : \tau \), then

\[
\]
11.2 Progress

1. If $\tau = \text{num}$, then $e = \text{num}[n]$ for some number $n$.

2. If $\tau = \text{str}$, then $e = \text{str}[s]$ for some string $s$.

Proof. By induction on rules (9.1) and (10.1).

Progress is proved by rule induction on rules (9.1) defining the static semantics of the language.

**Theorem 11.4 (Progress).** If $e : \tau$, then either $e \text{ val}$, or there exists $e'$ such that $e \rightarrow e'$.

Proof. The proof proceeds by induction on the typing derivation. We will consider only one case, for rule (9.1d),

\[
\begin{array}{c}
e_1 : \text{num} \quad e_2 : \text{num} \\
\text{plus}(e_1;e_2) : \text{num}
\end{array}
\]

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

By induction we have that either $e_1 \text{ val}$, or there exists $e_1'$ such that $e_1 \rightarrow e_1'$. In the latter case it follows that $\text{plus}(e_1;e_2) \rightarrow \text{plus}(e_1';e_2)$, as required. In the former we also have by induction that either $e_2 \text{ val}$, or there exists $e_2'$ such that $e_2 \rightarrow e_2'$. In the latter case we have that $\text{plus}(e_1;e_2) \rightarrow \text{plus}(e_1';e_2')$, as required. In the former, we have, by the Canonical Forms Lemma 11.3 on the facing page, $e_1 = \text{num}[n_1]$ and $e_2 = \text{num}[n_2]$, and hence

\[
\text{plus}(\text{num}[n_1];\text{num}[n_2]) \rightarrow \text{num}[n_1 + n_2].
\]

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 static and dynamic semantics are coherent, and that no ill-defined states can ever be encountered while evaluating a well-typed expression.
11.3 Run-Time Errors

Suppose that we wish to extend \( L\{\text{num,str}\} \) with, say, a quotient operation that is undefined for a zero divisor. The natural typing rule for quotients is given by the following rule:

\[
\frac{e_1 : \text{num} \quad e_2 : \text{num}}{\text{div}(e_1;e_2) : \text{num}}.
\]

But the expression \( \text{div}(\text{num}[3];\text{num}[0]) \) 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 \textit{checked} from \textit{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 dynamic semantics 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 dynamic semantics for quotient \textit{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:

\[
\frac{\text{val}}{\text{div}(e_1;\text{num}[0]) \ err}{(11.1a)}
\]
Rule (11.1a) 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.

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 11.5 (Progress With Error).** If \( e : \tau \), then either \( e \text{ err} \), or \( e \text{ val} \), or there exists \( e' \) such that \( e \rightsquigarrow e' \).

**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. □

A disadvantage of this approach to the formalization of error checking is that it appears to require a special set of evaluation rules to check for errors. An alternative is to fold in error checking with evaluation by enriching the language with a special error expression, \( \text{error} \), which signals that an error has arisen. Since an error condition aborts the computation, the static semantics assigns an arbitrary type to \( \text{error} \):

\[
\text{error} : \tau
\]  

This rule destroys the unicity of typing property (Lemma 9.1 on page 71). This can be restored by introducing a special error expression for each type, but we shall not do so here for the sake of simplicity.

The dynamic semantics is augmented with rules that provoke a checked error (such as division by zero), plus rules that propagate the error through other language constructs.

\[
\begin{align*}
\text{div}(e_1; \text{num}[0]) & \rightsquigarrow \text{error} & (11.3a) \\
\text{plus}(\text{error}; e_2) & \rightsquigarrow \text{error} & (11.3b)
\end{align*}
\]
\[
\begin{array}{c}
e_1 \text{val} \\
\text{plus}(e_1; \text{error}) \rightarrow \text{error}
\end{array}
\]  
(11.3c)

There are similar error propagation rules for the other constructs of the language. By defining \( e \ err \) to hold exactly when \( e = \text{error} \), the revised progress theorem continues to hold for this variant semantics.

### 11.4 Exercises

1. Complete the proof of preservation.

2. Complete the proof of progress.
Chapter 12

Evaluation Semantics

In Chapter 10 we defined the dynamic semantics of $\mathcal{L}\{\text{num str}\}$ using the method of structural semantics. 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 \textit{evaluation semantics}, or \textit{ES}, presents the dynamic semantics as a relation between a phrase and its value, without detailing how it is to be determined in a step-by-step manner. Two variants of evaluation semantics are also considered, namely \textit{environment semantics}, which delays substitution, and \textit{cost semantics}, which records the number of steps that are required to evaluate an expression.

\subsection{12.1 Evaluation Semantics}

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

\begin{align*}
\text{num}[n] & \Downarrow \text{num}[n] & (12.1a) \\
\text{str}[s] & \Downarrow \text{str}[s] & (12.1b) \\
\frac{e_1 \Downarrow \text{num}[n_1] \quad e_2 \Downarrow \text{num}[n_2] \quad n_1 + n_2 = n \text{ nat}}{\text{plus}(e_1; e_2) \Downarrow \text{num}[n]} & (12.1c) \\
\frac{e_1 \Downarrow \text{str}[s_1] \quad e_2 \Downarrow \text{str}[s_2] \quad s_1 \hat{s_2} = s \text{ str}}{\text{cat}(e_1; e_2) \Downarrow \text{str}[s]} & (12.1d)
\end{align*}
12.2 Relating Transition and Evaluation Semantics

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 (12.1f) 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 \( P(e \Downarrow v) \) holds, it is enough to show that \( P \) is closed under Rules (12.1):

1. Show that \( P(\text{num}[n] \Downarrow \text{num}[n]) \).
2. Show that \( P(\text{str}[s] \Downarrow \text{str}[s]) \).
3. Show that \( P(\text{plus}(e_1; e_2) \Downarrow \text{num}[n]), \text{if } P(e_1 \Downarrow \text{num}[n_1]), P(e_2 \Downarrow \text{num}[n_2]), \text{and } n_1 + n_2 = n \text{ nat} \).
4. Show that \( P(\text{cat}(e_1; e_2) \Downarrow \text{str}[s]), \text{if } P(e_1 \Downarrow \text{str}[s_1]), P(e_2 \Downarrow \text{str}[s_2]), \text{and } s_1 \hat{s}_2 = s \text{ str} \).
5. Show that \( P(\text{let}(e_1; x \cdot e_2) \Downarrow v_2), \text{if } P([e_1/x]e_2 \Downarrow v_2) \).

This induction principle is not the same as structural induction on \( e \text{ exp} \), because the evaluation rules are not syntax-directed!

**Lemma 12.1.** If \( e \Downarrow v, \text{ then } v \text{ val} \).

*Proof.* By induction on Rules (12.1). All cases except Rule (12.1f) 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.

12.2 Relating Transition and Evaluation Semantics

We have given two different forms of dynamic semantics for \( L\{\text{num str}\} \). 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 transition
12.2 Relating Transition and Evaluation Semantics

semantics describes a step-by-step process of execution, whereas the evaluation semantics suppresses the intermediate states, focusing attention on the initial and final states alone. This suggests that the appropriate correspondence is between complete execution sequences in the transition semantics and the evaluation judgement in the evaluation semantics. (We will consider only numeric expressions, but analogous results hold also for string-valued expressions.)

**Theorem 12.2.** 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 12.3.** If $e \Downarrow v$, then $e \mapsto^* v$.

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

\[
\begin{align*}
\text{plus}(e_1; e_2) & \mapsto^* \text{plus}(\text{num}[n_1]; e_2) \\
& \mapsto^* \text{plus}(\text{num}[n_1]; \text{num}[n_2]) \\
& \mapsto \text{num}[n_1 + n_2]
\end{align*}
\]

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

For the converse, recall from Chapter 4 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 12.4.** 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 $\text{plus}(e_1; e_2) \mapsto \text{plus}(e'_1; e_2)$, where $e_1 \mapsto e'_1$. Suppose further that $\text{plus}(e'_1; e_2) \Downarrow v$, so that $e'_1 \Downarrow \text{num}[n_1], e_2 \Downarrow \text{num}[n_2], n_1 + n_2 = n \text{ nat}$, and $v$ is $\text{num}[n]$. By induction $e_1 \Downarrow \text{num}[n_1]$, and hence $\text{plus}(e_1; e_2) \Downarrow \text{num}[n]$, as required.
12.3 Type Safety, Revisited

The type safety theorem for $\mathcal{L}\{\text{num str}\}$ (Theorem 11.1 on page 85) 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 dynamic semantics. But what if we had instead specified the dynamic semantics 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 semantics, 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 $\mathcal{L}\{\text{num str}\}$, it demands much more than just progress — it requires that every expression evaluate to a value! If $\mathcal{L}\{\text{num str}\}$ were extended to admit operations that may result in an error (as discussed in Section 11.3 on page 88), 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 semantics, but only by reference to a transition semantics. Another point of view is to instrument the semantics 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 class of errors solely to prove that they cannot arise! Nevertheless, we will press on to show how a semblance of type safety can be established using evaluation semantics.

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:

\[
\frac{}{\text{plus}(\text{str}[s]; e_2) \uparrow} \quad (12.2a) \\
\frac{e_1 \text{ val}}{\text{plus}(e_1; \text{str}[s]) \uparrow} \quad (12.2b)
\]
These rules explicitly check for the misapplication of addition to a string; similar rules govern each of the primitive constructs of the language.

**Theorem 12.5.** If \( e \uparrow \), then there is no \( \tau \) such that \( e : \tau \).

**Proof.** By rule induction on Rules (12.2). For example, for Rule (12.2a), we observe that \( \text{str}[s] : \text{str} \), and hence \( \text{plus}(\text{str}[s]; e_2) \) is ill-typed. \( \square \)

**Corollary 12.6.** 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 12.5 remains valid; there is nothing to ensure that we have included sufficiently many checks for run-time type errors. We can prove that the ones we define cannot arise in a well-typed program, but we cannot prove that we have covered all possible cases. By contrast the transition semantics 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 static semantics 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 semantics and the implementation.

### 12.4 Cost Semantics

A structural semantics provides a natural notion of *time complexity* for programs, namely the number of steps required to reach a final state. An evaluation semantics, 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 semantics*.

Evaluation judgements have the form \( e \psi^k v \), with the meaning that \( e \) evaluates to \( v \) in \( k \) steps.

\[
\text{num}[n] \psi^0 \text{num}[n]
\]  
(12.3a)
Theorem 12.7. 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 semantics. From right to left proceed by induction on $k$, with an inner rule induction on the definition of the transition semantics. \qed

12.5 Environment Semantics

Both the transition semantics and the evaluation semantics given earlier rely on substitution to replace let-bound variables by their bindings during evaluation. This approach maintains the invariant that only closed expressions are ever considered. However, in practice, we do not perform substitution, but rather record the bindings of variables in a data structure where they may be retrieved on demand. In this section we show how this can be expressed for a by-value interpretation of binding using hypothetical judgements. It is also possible to formulate an environment semantics for the by-name interpretation, at the cost of some additional complexity (see Chapter 40 for a full discussion of the issues involved).

The basic idea is to consider hypotheses of the form $x \Downarrow v$, where $x$ is a variable and $v$ is a closed value, such that no two hypotheses govern the same variable. Let $\Theta$ range over finite sets of such hypotheses, which we call an environment. We will consider judgements of the form $\Theta \vdash e \Downarrow v$, where $\Theta$ is an environment governing some finite set of variables.

\begin{align*}
\frac{e_1 \Downarrow^k_1 \text{num}[n_1] \quad e_2 \Downarrow^k_2 \text{num}[n_2]}{\text{plus}(e_1; e_2) \Downarrow^{k_1+k_2+1} \text{num}[n_1+n_2]} & \quad (12.3b) \\
\frac{\text{str}[s] \Downarrow^0 \text{str}[s]}{} & \quad (12.3c) \\
\frac{e_1 \Downarrow^k_1 s_1 \quad e_2 \Downarrow^k_2 s_2}{\text{cat}(e_1; e_2) \Downarrow^{k_1+k_2+1} \text{str}[s_1^s_2]} & \quad (12.3d) \\
\frac{[e_1/x]e_2 \Downarrow^k_2 v_2}{\text{let}(e_1; x, e_2) \Downarrow^{k_2+1} v_2} & \quad (12.3e)
\end{align*}
Rule (12.4a) is an instance of the general reflexivity rule for hypothetical judgements. The \texttt{let} rule augments the environment with a new assumption governing the bound variable, which may be chosen to be distinct from all other variables in \(\Theta\) to avoid multiple assumptions for the same variable.

The environment semantics implements evaluation by deferred substitution.

\textbf{Theorem 12.8.} \(x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v \iff [v_1, \ldots, v_n/x_1, \ldots, x_n]e \Downarrow v.\)

\textit{Proof.} The left to right direction is proved by induction on the rules defining the evaluation semantics, making use of the definition of substitution and the definition of the evaluation semantics for closed expressions. The converse is proved by induction on the structure of \(e\), again making use of the definition of substitution. Note that we must induct on \(e\) in order to detect occurrences of variables \(x_i\) in \(e\), which are governed by a hypothesis in the environment semantics. \(\square\)

### 12.6 Exercises

1. Prove that if \(e \Downarrow v\), then \(v\) \texttt{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 transition semantics.

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

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

Function Types
Chapter 13

Function Definitions and Values

In the language $\mathcal{L}\{\text{num str}\}$ we may perform calculations such as the doubling of a given expression, but we cannot express the concept of doubling itself. The general concept may be expressed by abstracting away from the expression being doubled, leaving behind just the pattern of doubling some fixed, but unspecified, number, represented by a variable. Specific instances of doubling are recovered by substituting an expression for the variable. A function is an expression with a designated free variable.

We consider two methods for permitting a function to be used more than once in an expression (for example, to double several different numbers). One method is through the introduction of function definitions, which give names to functions. An instance of the function is obtained by applying the function name to another expression, its argument. Each function has a domain and a range type, which in $\mathcal{L}\{\text{num str}\}$ must be either num or str. A function whose domain and range are base type is said to be a first-order function. A language in which functions are first-order and confined to function definitions is said to have second-class functions, since they are not values in the same sense as numbers or strings.

A more general method for supporting functions is as first-class values of function type whose domain and range are arbitrary types, including function types. A language with function types is said to be higher-order, in contrast to first-order, since it allows functions to be passed as arguments to and returned as results from other functions. Higher-order languages are surprisingly powerful, and they are, correspondingly, remarkably subtle, and have led to notorious design errors in programming languages.
13.1 First-Order Functions

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$\text{fun}[\tau_1; \tau_2](x_1.e_2; f.e)$</td>
<td>$\text{fun}\ f(x_1:\tau_1):\tau_2 = e_2 \text{ in } e$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{call}<a href="e">f</a>$</td>
<td>$f(e)$</td>
</tr>
</tbody>
</table>

The variable $f$ ranges over a distinguished class of variables, called function names. The expression $\text{fun}[\tau_1; \tau_2](x_1.e_2; f.e)$ binds the function name $f$ within $e$ to the pattern $x_1.e_2$, which has parameter $x_1$ and definition $e_2$. The domain and range of the function are, respectively, the types $\tau_1$ and $\tau_2$. The expression $\text{call}[f](e)$ instantiates the abstractor bound to $f$ with the argument $e$.

The static semantics of $\mathcal{L}\{\text{num str fun}\}$ consists of judgements of the form $\Gamma \vdash e : \tau$, where $\Gamma$ consists of hypotheses of one of two forms:

1. $x : \tau$, declaring the type of a variable $x$ to be $\tau$;
2. $f(\tau_1) : \tau_2$, declaring that $f$ is a function name with domain $\tau_1$ and range $\tau_2$.

The second form of assumption is sometimes called a function header, since it resembles the concrete syntax of the first part of a function definition. The static semantics is defined in terms of these hypotheses by the following rules:

\[
\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2 \quad \Gamma, f(\tau_1) : \tau_2 \vdash e : \tau \\
\Gamma \vdash \text{fun}[\tau_1; \tau_2](x_1.e_2; f.e) : \tau
\]

(13.1a)

\[
\Gamma, f(\tau_1) : \tau_2 \vdash e : \tau_1 \quad \Gamma, f(\tau_1) : \tau_2 \vdash \text{call}[f](e) : \tau_2
\]

(13.1b)

The structural property of substitution takes an unusual form that matches the form of the hypotheses governing function names. The operation of function substitution, written $\llbracket x.e / f \rrbracket e'$, is inductively defined similarly to ordinary substitution, but bearing in mind that the function name, $f$, may only occur within $e'$ as part of a function call. The rule governing such occurrences is given as follows:

\[
\llbracket x.e / f \rrbracket \text{call}[f](e') = \text{let}(e', x.e)
\]

(13.2)

That is, at call sites to $f$, we bind $x$ to $e'$ within $e$ to instantiate the pattern substituted for $f$. 

18:42   

DRAFT  

OCTOBER 16, 2009
Lemma 13.1. If $\Gamma, f \tau_1 \vdash e : \tau$ 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 dynamic semantics of $\mathcal{L}\{\text{num str fun}\}$ is easily defined using function substitution:

$$\text{fun}[\tau_1; \tau_2] (x_1. e_2; f. e) \mapsto [x_1. e_2 / f]e \quad (13.3)$$

Observe that the use of function substitution eliminates all applications of $f$ within $e$, so that no rule is required for evaluating them. This rule imposes either a call-by-name or a call-by-value application discipline according to whether the let binding is given a by-name or a by-value interpretation.

The safety of $\mathcal{L}\{\text{num str fun}\}$ may be proved separately, but it may also be obtained as a corollary of the safety of the more general language of higher-order functions, which we discuss next.

13.2 Higher-Order Functions

The syntactic and semantic similarity between variable definitions and function definitions in $\mathcal{L}\{\text{num str fun}\}$ 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 $\lambda$-abstraction, written $\text{lam}[\tau] (x. e)$. Correspondingly, we must generalize application to have the form $\text{ap}(e_1; e_2)$, where $e_1$ is any expression, and not just a function name. These are, respectively, the introduction and elimination forms for the function type, $\text{arr}(\tau_1; \tau_2)$, whose elements are functions with domain $\tau_1$ and range $\tau_2$.

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$\text{arr}(\tau_1; \tau_2)$</td>
<td>$\tau_1 \rightarrow \tau_2$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$\text{lam}[\tau] (x. e)$</td>
<td>$\lambda(x : \tau. e)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{ap}(e_1; e_2)$</td>
<td>$e_1(e_2)$</td>
</tr>
</tbody>
</table>
The static semantics of $\mathcal{L}\{\text{num str} \rightarrow\}$ is given by extending Rules (9.1) with the following rules:

$$
\begin{align*}
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \text{arr}(\tau_1; \tau_2)} \tag{13.4a} \\
\frac{\Gamma \vdash e_1 : \text{arr}(\tau_2; \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{ap}(e_1; e_2) : \tau} \tag{13.4b}
\end{align*}
$$

**Lemma 13.2 (Inversion).** Suppose that $\Gamma \vdash e : \tau$.

1. If $e = \lambda x : \tau_1 . e$, then $\tau = \text{arr}(\tau_1; \tau_2)$ and $\Gamma, x : \tau_1 \vdash e : \tau_2$.

2. If $e = \text{ap}(e_1; e_2)$, then there exists $\tau_2$ such that $\Gamma \vdash e_1 : \text{arr}(\tau_2; \tau)$ 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 13.3 (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 dynamic semantics of $\mathcal{L}\{\text{num str} \rightarrow\}$ extends that of $\mathcal{L}\{\text{num str}\}$ with the following additional rules:

$$
\begin{align*}
\lambda x : \tau . e &\rightarrow \text{val} \tag{13.5a} \\
\frac{e_1 \rightarrow e'_1}{\text{ap}(e_1; e_2) \rightarrow \text{ap}(e'_1; e_2)} \tag{13.5b} \\
\text{ap}(\lambda x : \tau_2 . e_1; e_2) &\rightarrow [e_2/x]e_1 \tag{13.5c}
\end{align*}
$$

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 13.4 (Preservation).** If $e : \tau$ and $e \rightarrow e'$, then $e' : \tau$.

18:42 DRAFT OCTOBER 16, 2009
Proof. The proof is by induction on rules (13.5), which define the dynamic semantics of the language.

Consider rule (13.5c),

\[
\text{ap}(\lambda \tau \cdot (x \cdot e_1); e_2) \rightarrow [e_2/x]e_1.
\]

Suppose that \(\text{ap}(\lambda \tau \cdot (x \cdot e_1); e_2) : \tau_1\). By Lemma 13.2 on the preceding page \(e_2 : \tau_2\) and \(x : \tau_2 \vdash e_1 : \tau_1\), so by Lemma 13.3 on the facing page \([e_2/x]e_1 : \tau_1\).

The other rules governing application are handled similarly. ☐

**Lemma 13.5 (Canonical Forms).** If \(e \hspace{1pt}\downarrow\downarrow\) and \(e : \text{arr}(\tau_1; \tau_2)\), then \(e = \lambda \tau \cdot (x \cdot e_2)\) 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 \(e \hspace{1pt}\downarrow\downarrow\). ☐

**Theorem 13.6 (Progress).** If \(e : \tau\), then either \(e\) is a value, or there exists \(e'\) such that \(e \rightarrow e'\).

Proof. The proof is by induction on rules (13.4). Note that since we consider only closed terms, there are no hypotheses on typing derivations.

Consider rule (13.4b). By induction either \(e_1 \downarrow\downarrow\) or \(e_1 \rightarrow e_1'\). In the latter case we have \(\text{ap}(e_1; e_2) \rightarrow \text{ap}(e_1'; e_2)\). In the former case, we have by Lemma 13.5 that \(e_1 = \lambda \tau \cdot (x \cdot e)\) for some \(x\) and \(e\). But then \(\text{ap}(e_1; e_2) \rightarrow [e_2/x]e\). ☐

### 13.3 Evaluation Semantics and Definitional Equivalence

An inductive definition of the evaluation judgement \(e \downarrow v\) for \(\mathcal{L}\{\text{num str} \rightarrow\}\) is given by the following rules:

\[
\lambda \tau \cdot (x \cdot e) \downarrow \lambda \tau \cdot (x \cdot e) \quad (13.6a)
\]

\[
e_1 \downarrow \lambda \tau \cdot (x \cdot e) \quad [e_2/x]e \downarrow v \\
\text{ap}(e_1; e_2) \downarrow v \quad (13.6b)
\]

It is easy to check that if \(e \downarrow v\), then \(v\) \hspace{1pt}\val\hspace{1pt}, and that if \(e\) \hspace{1pt}\val\hspace{1pt}, then \(e \downarrow e\).
Theorem 13.7. $e \downarrow v$ iff $e \rightarrow^* v$ and $v$ val.

Proof. In the forward direction we proceed by rule induction on Rules (13.6). The proof makes use of a pasting lemma stating that, for example, if $e_1 \rightarrow^* e_1'$, then $\text{ap}(e_1; e_2) \rightarrow^* \text{ap}(e_1'; e_2)$, and similarly for the other constructs of the language.

In the reverse direction we proceed by rule induction on Rules (4.1). The proof relies on a converse evaluation lemma, which states that if $e \rightarrow e'$ and $e' \downarrow v$, then $e \downarrow v$. This is proved by rule induction on Rules (13.5). □

Definitional equivalence for the call-by-name semantics of $L\{\text{num str }\rightarrow\}$ is defined by a straightforward extension to Rules (10.9).

$$
\begin{align*}
\Gamma \vdash \text{ap}(\lambda [\tau_1] (x . e_2); e_1) & \equiv [e_1/x] e_2 : \tau_2 \\
\Gamma \vdash e_1 & \equiv e_1' : \tau_2 \quad \Gamma \vdash e_2 & \equiv e_2' : \tau_2 \\
\Gamma \vdash \text{ap}(e_1; e_2) & \equiv \text{ap}(e_1'; e_2') : \tau \\
\Gamma, x : \tau_1 \vdash e_2 & \equiv e_2' : \tau_2 \\
\Xi \Gamma \vdash \lambda [\tau_1] (x . e_2) & \equiv \lambda [\tau_1] (x . e_2') : \tau_1 \rightarrow \tau_2
\end{align*}
$$

(13.7a) (13.7b) (13.7c)

Definitional equivalence for call-by-value requires a small bit of additional machinery. The main idea is to restrict Rule (13.7a) 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 (13.7a) 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 $\Xi$ is the finite set of hypotheses $x_1 \text{ val}, \ldots, x_k \text{ val}$ governing the variables in scope at that point. We write $\Xi \vdash e \text{ val}$ to indicate that $e$ is a value under these hypotheses, so that, for example, $\Xi, x \text{ val} \vdash x \text{ val}$.

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:

$$
\Xi, x \text{ val} \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2 \\
\Xi \Gamma \vdash \lambda [\tau_1] (x . e_2) \equiv \lambda [\tau_1] (x . e_2') : \tau_1 \rightarrow \tau_2
$$

(13.8a)
13.4 Dynamic Scope

The dynamic semantics of function application given by Rules (13.5) is defined for closed expressions (those without free variables). Variables are never encountered during evaluation, because a closed expression will have been substituted for it before it is needed during evaluation. This accurately reflects the meaning of a variable as an unknown whose value may be specified by substitution. This treatment of variables is called static scope, or static binding, because it respects the statically determined scoping rules defined in Chapter 7.

Another evaluation strategy for $L\{\rightarrow\}$ is sometimes considered as an alternative to static binding, called dynamic scope, or dynamic binding. The semantics of a dynamically scoped version of $L\{\rightarrow\}$ is given by the same rules as for static binding, but altered in two crucial respects. First, evaluation is defined for open terms (those with free variables), as well as for closed terms. It is, however, an error to evaluate a variable; as with static scope, we must arrange that its binding is determined before its value is needed. Second, the binding of a variable is specified by a special form of substitution that incurs, rather than avoids, capture of free variables.

To avoid confusion, we will use the term replacement to refer to the capture-incuring form of substitution, which we write as $[x \leftarrow e_1]e_2$. As an example of replacement, let $e$ be the expression $\lambda(x:\sigma).y$ (with a free variable $y$), and let $e'$ be the expression $\lambda(y:\tau).f(y)$, where $f$ is a variable. The result of the substitution $[e/f]e'$ is the expression

$$\lambda(y':\tau.\lambda(x:\sigma.y)(y')),$$

in which the bound variable, $y$, has been renamed to $y'$ to avoid confusion with the free variable, $y$, in $e$. The variable $y$ remains free in the result. In contrast, the result of the replacement $[f \leftarrow e]e'$ is the expression

$$\lambda(y:\tau.\lambda(x:\sigma.y)(y)),$$

which has no free variables because the free $y$ in $e$ is captured by the binding for $y$ in $e'$.

The implications of these alterations to the semantics of $L\{\rightarrow\}$ are far-reaching. An immediate question suggested by the foregoing example is
whether typing is preserved by replacement (as distinct from substitution). The answer is no! In the example if $\sigma \neq \tau$, then the result of replacement is not well-typed, even though both $e$ and $e'$ are well-typed (assuming $y : \tau$ and $f : \tau \rightarrow \tau'$). For this reason, dynamic scope is usually only considered feasible for languages with only one type, so that such considerations do not arise.\footnote{See Chapter 22 for a discussion of useful programming languages with but one type.} An alternative is to consider a much richer type system that accounts for the types of the free variables in an expression; this possibility is explored in Chapter 35.

Setting aside these concerns, there is a further problem with dynamic scope that merits careful consideration, since it is closely tied to its purported advantages. The idea of dynamic scope is to make it convenient to parameterize a function by the values of one or more variables, without having to pass them as additional arguments. So, for example, a function $\lambda (x : \sigma . e)$ with $y$ free is to be regarded as a family of functions, one for each choice of the parameter $y$. Using replacement, rather than substitution, allows the specification of a value for $y$ to be determined by the context in which the function is used, rather than the context in which the function is introduced. (This is what gives rise to the terminology “dynamic scope.”) Thus, in the example above, the meaning of the expression $e$ is not fixed until after the replacement of $f$ by $e$ in $e'$, at which point $y$ is tied to the argument of the function $e'$. Whatever that turns out to be will determine the particular instance of $e$ that will be used.

The chief difficulty with dynamic scope is that the names of bound variables matter. For example, consider the expression $e''$ given by $\lambda (y' : \tau . f (y'))$. The expression $e'$ is $\alpha$-equivalent to $e''$; all we have done is to rename the bound variable from $y$ to $y'$. The principles of binding and scope described in Chapter 7 state that these two expressions should be interchangeable in all situations, and indeed they are under static scope. However, with dynamic scope they behave quite differently. In particular, the replacement $[x \leftarrow e]e''$ results in the expression

$$\lambda (y' : \tau . \lambda (x : \sigma . y) (y')),$$

which differs from the replacement $[x \leftarrow e]e'$, even though $e'$ and $e''$ are $\alpha$-equivalent.

From a programmer’s perspective, the author of the expression $e$ must be aware of the parameter naming conventions used by the author of $e'$ (or $e''$). This does violence to any form of modularity or separation of concerns; the two pieces of code must be written in conjunction with each other, and
this intimate relationship must be maintained as the code evolves. Experience shows that this is an impossible demand. For this reason, together with the difficulties with typing, dynamic scoping of variables is often treated with skepticism. However, there are other means of supporting essentially the same functionality, but without doing violence to the fundamental principles of binding and scope explained in Chapter 7. This concept, called fluid binding, is the subject of Chapter 35.

13.5 Exercises
Chapter 14

Gödel’s System T

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

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$\text{nat}$</td>
<td>$\text{nat}$</td>
</tr>
<tr>
<td></td>
<td>$\text{arr}(\tau_1; \tau_2)$</td>
<td>$\tau_1 \rightarrow \tau_2$</td>
<td>$\tau_1 \rightarrow \tau_2$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$x$</td>
<td>$x$</td>
</tr>
<tr>
<td></td>
<td>$z$</td>
<td>$z$</td>
<td>$z$</td>
</tr>
<tr>
<td></td>
<td>$s(e)$</td>
<td>$s(e)$</td>
<td>$s(e)$</td>
</tr>
<tr>
<td></td>
<td>$\text{natrec}(e; e_0; x. y. e_1)$</td>
<td>$\text{natrec} e { z \Rightarrow e_0</td>
<td>s(x) \text{ with } y \Rightarrow e_1 }$</td>
</tr>
<tr>
<td></td>
<td>$\lambda [\tau](x. e)$</td>
<td>$\lambda (x: \tau. e)$</td>
<td>$\lambda (x: \tau. e)$</td>
</tr>
<tr>
<td></td>
<td>$\text{ap}(e_1; e_2)$</td>
<td>$e_1(e_2)$</td>
<td>$e_1(e_2)$</td>
</tr>
</tbody>
</table>

We write $\pi$ for the expression $s(\ldots s(z))$, in which the successor is applied $n \geq 0$ times to zero. The expression

$$\text{natrec}(e; e_0; x. y. e_1)$$

is called \textit{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 \textit{iteration}, written $\text{natiter}(e; e_0; y. e_1)$, 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 16) at our disposal. To define primitive recursion from iteration we simultaneously compute the predecessor while iterating the specified computation.

The static semantics of $\mathcal{L}\{\text{nat} \rightarrow\}$ is given by the following typing rules:

\begin{align}
\Gamma, x : \text{nat} & \vdash x : \text{nat} \quad (14.1a) \\
\Gamma & \vdash z : \text{nat} \quad (14.1b) \\
\Gamma & \vdash e : \text{nat} \quad (14.1c) \\
\Gamma & \vdash s(e) : \text{nat} \\
\Gamma & \vdash e_0 : \tau \quad \Gamma, x : \text{nat}, y : \tau \vdash e_1 : \tau \\
\Gamma & \vdash \text{natrec}(e; e_0; x. y. e_1) : \tau \quad (14.1d)
\end{align}
14.2 Dynamics

\[ \Gamma, x : \sigma \vdash e : \tau \]
\[ \Gamma \vdash \text{lam} [\sigma] (x.e) : \text{arr} (\sigma; \tau) \] (14.1e)

\[ \Gamma \vdash e_1 : \text{arr} (\tau_2; \tau) \]
\[ \Gamma \vdash e_2 : \tau_2 \]
\[ \Gamma \vdash \text{ap} (e_1; e_2) : \tau \] (14.1f)

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

**Lemma 14.1.** If \( \Gamma \vdash e : \tau \) and \( \Gamma, x : \tau \vdash e' : \tau' \), then \( \Gamma \vdash [e/x]e' : \tau' \).

14.2 Dynamics

The dynamic semantics of \( L\{\text{nat} \to\} \) adopts a call-by-name interpretation of function application, and requires that the successor operation evaluate its argument (so that values of type \( \text{nat} \) are numerals).

The closed values of \( L\{\text{nat} \to\} \) are determined by the following rules:

\[ \text{z } \text{val} \] (14.2a)

\[ \frac{e \text{ val}}{	ext{s(e) val}} \] (14.2b)

\[ \frac{\text{lam}[\tau](x.e) \text{ val}}{\text{val}} \] (14.2c)

The dynamic semantics of \( L\{\text{nat} \to\} \) is given by the following rules:

\[ e \mapsto e' \quad \text{s(e) } \mapsto \text{s(e')} \] (14.3a)

\[ e_1 \mapsto e' \quad \text{ap(e_1; e_2) } \mapsto \text{ap(e'_1; e_2)} \] (14.3b)

\[ \text{ap(lam}[\tau](x.e); e_2) \mapsto [e_2/x]e \] (14.3c)

\[ e \mapsto e' \quad \text{natrec(e; e_0; x.y.e_1) } \mapsto \text{natrec(e'; e_0; x.y.e_1)} \] (14.3d)

\[ \text{natrec(z; e_0; x.y.e_1) } \mapsto e_0 \] (14.3e)
14.3 Definability

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

\[
e_f(n) \equiv f(n) : \text{nat}.
\]  

That is, the numeric function \( f : \mathbb{N} \to \mathbb{N} \) is definable iff there is an expression \( e_f \) of type \( \text{nat} \to \text{nat} \) 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 \( \mathcal{L}\{\text{nat} \to \} \), written \( \Gamma \vdash e \equiv e' : \tau \), is the strongest congruence containing these axioms:

\[
\Gamma \vdash \text{ap}(\lambda\tau(x.e_2);e_1) \equiv [e_1/x]e_2 : \tau 
\]  

\[
\Gamma \vdash \text{natrec}(z;e_0;x.y.e_1) \equiv e_0 : \tau 
\]  

\[
\Gamma \vdash \text{natrec}(s(e);e_0;x.y.e_1) \equiv [e,\text{natrec}(e;e_0;x.y.e_1)/x,y]e_1 : \tau
\]  

Rules (14.3e) and (14.3f) 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 14.2 (Canonical Forms). If \( e : \tau \) and \( e \text{ val} \), then

1. If \( \tau = \text{nat} \), then \( e = s(s(\ldots z)) \) 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 14.3 (Safety). 1. If \( e : \tau \) and \( e \mapsto e' \), then \( e' : \tau \).

2. If \( e : \tau \), then either \( e \text{ val} \) or \( e \mapsto e' \) for some \( e' \).
For example, the doubling function, \( d(n) = 2 \times n \), is definable in \( L\{\text{nat} \rightarrow \} \) by the expression \( e_d : \text{nat} \rightarrow \text{nat} \) given by
\[
\lambda (x: \text{nat}. \text{natrec } x \{ z \Rightarrow z | s(u) \text{ with } v \Rightarrow s(s(v)) \}).
\]
To check that this defines the doubling function, we proceed by induction on \( n \in \mathbb{N} \). For the basis, it is easy to check that
\[
e_d (\overline{0}) \equiv \overline{0} : \text{nat}.
\]
For the induction, assume that
\[
e_d (\overline{n}) \equiv \overline{d(n)} : \text{nat}.
\]
Then calculate using the rules of definitional equivalence:
\[
e_d (\overline{n + 1}) \equiv s(s(e_d (\overline{n})))
\equiv s(s(2 \times n))
\equiv 2 \times (n + 1)
\equiv \overline{d(n + 1)}.
\]
As another example, consider the following function, called Ackermann’s function, defined by the following equations:
\[
A(0, n) = n + 1
\]
\[
A(m + 1, 0) = A(m, 1)
\]
\[
A(m + 1, n + 1) = A(m, A(m + 1, n)).
\]
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 \( \text{nat} \rightarrow \text{nat} \) 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 \( L\{\text{nat} \rightarrow \} \) 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
\[
it : (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}
\]
to be the λ-abstraction

\[ \lambda(f: \text{nat} \rightarrow \text{nat}. \lambda(n: \text{nat}. \text{natrec } n \{z \mapsto \text{id} \mid s(\_ ) \text{ with } g \mapsto f \circ g \}), \]

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

\[ \text{it}(f)(m)(m) \equiv f^{(m)}(m) : \text{nat}, \]

where the latter expression is the \( n \)-fold composition of \( f \) starting with \( m \). We may then define the Ackermann function

\[ e_a : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \]

to be the expression

\[ \lambda(m: \text{nat}. \text{natrec } m \{z \mapsto \text{suc} \mid s(\_ ) \text{ with } f \mapsto \lambda(n: \text{nat}. \text{it}(f)(n)(f(1))) \}). \]

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

\[ e_a(0)(m) \equiv s(m) \quad (14.6) \]
\[ e_a(m + 1)(0) \equiv e_a(m)(1) \quad (14.7) \]
\[ e_a(m + 1)(n + 1) \equiv e_a(m)(e_a(s(m))(n)) \quad (14.8) \]

That is, the Ackermann function is definable in \( L\{\text{nat} \rightarrow \} \).

### 14.4 Non-Definability

It is impossible to define an infinite loop in \( L\{\text{nat} \rightarrow \} \).

**Theorem 14.4.** If \( e : \tau \), then there exists \( v \text{ val such that } e \equiv v : \tau. \)

**Proof.** See Corollary 50.9 on page 459. \( \square \)

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

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

The essence of Gödel-numbering is captured by the following simple construction on abstract syntax trees. (The generalization to abstract binding trees is slightly more difficult, the main complication being to ensure that $\alpha$-equivalent expressions are assigned the same Gödel number.) Recall that a general ast, $a$, has the form $o(a_1, \ldots, a_k)$, where $o$ is an operator of arity $k$. Fix an enumeration of the operators so that every operator has an index $i \in \mathbb{N}$, and let $m$ be the index of $o$ in this enumeration. Define the Gödel number $\langle a \rangle$ of $a$ to be the number $2^m 3^{n_1} 5^{n_2} \ldots p_k^{n_k}$, where $p_k$ is the $k$th prime number (so that $p_0 = 2$, $p_1 = 3$, and so on), and $n_1, \ldots, n_k$ are the Gödel numbers of $a_1, \ldots, a_k$, respectively. This 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 $f_{\text{univ}} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ such that, for any $e : \text{nat} \rightarrow \text{nat}$, $f_{\text{univ}}(\langle e \rangle)(m) = n$ iff $e(m) \equiv n : \text{nat}$. The determinacy of the dynamic semantics, together with Theorem 14.4 on the preceding page, ensure that $f_{\text{univ}}$ is a well-defined function. It is called the universal function for $\mathcal{L} \{ \text{nat} \rightarrow \}$ because it specifies the behavior of any expression $e$ of type $\text{nat} \rightarrow \text{nat}$. Using the universal function, let us define an auxiliary mathematical function, called the diagonal function, $d : \mathbb{N} \rightarrow \mathbb{N}$, by the equation $d(m) = f_{\text{univ}}(m)(m)$. This function is chosen so that $d(\langle e \rangle) = n$ iff $e(\langle e \rangle) \equiv n : \text{nat}$. (The motivation for this definition will be apparent in a moment.)

The function $d$ is not definable in $\mathcal{L} \{ \text{nat} \rightarrow \}$. Suppose that $d$ were defined by the expression $e_d$, so that we have $e_d(\langle e \rangle) \equiv e(\langle e \rangle) : \text{nat}$. Let $e_D$ be the expression

$$\lambda(x: \text{nat}. s(e_d(x)))$$
of type nat → nat. We then have
\[ e_D(\overline{e_D}) \equiv s(e_d(\overline{e_D})) \]
\[ \equiv s(e_D(\overline{e_D})). \]

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

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

### 14.5 Exercises

1. Explore variant dynamic semantics for \( L\{nat \rightarrow\} \), both separately and in combination, in which the successor does not evaluate its argument, and in which functions are called by value.
Chapter 15

Plotkin’s PCF

The language \( \mathcal{L}\{\text{nat} \rightarrow\} \), also known as Plotkin’s PCF, integrates functions and natural numbers using general recursion, a means of defining self-referential expressions. In contrast to \( \mathcal{L}\{\text{nat} \rightarrow\} \) expressions in \( \mathcal{L}\{\text{nat} \rightarrow\} \) 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 \( \mathcal{L}\{\text{nat} \rightarrow\} \), the language \( \mathcal{L}\{\text{nat} \rightarrow\} \) 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 \( \mathcal{L}\{\text{nat} \rightarrow\} \) is the fixed point characterization of recursive definitions. In ordinary mathematical practice one may define a function \( f \) by recursion equations such as these:

\[
\begin{align*}
  f(0) &= 1 \\
  f(n + 1) &= (n + 1) \times f(n)
\end{align*}
\]

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 \( f : \mathbb{N} \rightarrow \mathbb{N} \) 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 \( \text{fix}(F) \), where \( \text{fix} \) 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, \( \phi \) on the natural numbers, is an approximation to a total function, \( f \), if \( \phi(m) = n \) implies that \( f(m) = n \). Let \( \bot : \mathbb{N} \to \mathbb{N} \) be the totally undefined partial function—\( \bot(n) \) is undefined for every \( n \in \mathbb{N} \). Intuitively, this is the “worst” approximation to the desired solution, \( f \), of the recursion equations given above. Given any approximation, \( \phi \), of \( f \), we may “improve” it by considering \( \phi' = F(\phi) \). Intuitively, \( \phi' \) is defined on 0 and on \( m + 1 \) for every \( m \geq 0 \) on which \( \phi \) is defined. Continuing in this manner, \( \phi'' = F(\phi') = F(F(\phi)) \) is an improvement on \( \phi' \), and hence a further improvement on \( \phi \). If we start with \( \bot \) as the initial approximation to \( f \), then pass to the limit

\[ \lim_{i \geq 0} F^{(i)}(\bot), \]

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 \( \mathcal{L}\{\text{nat} \to \} \)—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 may 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 programmer’s to ensure that the functions defined by recursion are
total—all of our loops terminate.

15.1 Statics

The abstract binding syntax of $\mathcal{L}\{\text{nat} \to \}$ is given by the following gram-
mar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$ ::=</td>
<td>nat</td>
<td>nat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{parr}(\tau_1;\tau_2)$</td>
<td>$\tau_1 \to \tau_2$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$ ::=</td>
<td>$x$</td>
<td>$x$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z$</td>
<td>$z$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$s(e)$</td>
<td>$s(e)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{ifz}(e; e_0; x.e_1)$</td>
<td>$\text{ifz} e { z \Rightarrow e_0 \mid s(x) \Rightarrow e_1 }$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{lam}<a href="x.e">\tau</a>$</td>
<td>$\lambda(x: \tau.e)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{ap}(e_1; e_2)$</td>
<td>$e_1(e_2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{fix}<a href="x.e">\tau</a>$</td>
<td>$\text{fix } x: \tau e$</td>
</tr>
</tbody>
</table>

The expression $\text{fix}[\tau](x.e)$ is called general recursion; it is discussed in
more detail below. The expression $\text{ifz}(e; e_0; x.e_1)$ branches according to
whether $e$ evaluates to $z$ or not, binding the predecessor to $x$ in the case
that it is not.

The static semantics of $\mathcal{L}\{\text{nat} \to \}$ is inductively defined by the follow-
ing rules:

\[
\frac{}{\Gamma, x : \tau \vdash x : \tau} \quad (15.1a)
\]

\[
\frac{}{\Gamma \vdash z : \text{nat}} \quad (15.1b)
\]

\[
\frac{\Gamma \vdash e : \text{nat}}{\Gamma \vdash s(e) : \text{nat}} \quad (15.1c)
\]

\[
\frac{\Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \text{nat} \vdash e_1 : \tau}{\Gamma \vdash \text{ifz}(e; e_0; x.e_1) : \tau} \quad (15.1d)
\]
\( \Gamma, x : \tau_1 \vdash e : \tau_2 \)
\[
\Gamma \vdash \text{lam}[(x. e) : \text{parr}(\tau_1; \tau_2)]
\]
\( \Gamma \vdash e_1 : \text{parr}(\tau_2; \tau) \quad \Gamma \vdash e_2 : \tau_2 \)
\[
\Gamma \vdash \text{ap}(e_1; e_2) : \tau
\]
\( \Gamma, x : \tau \vdash e : \tau \)
\[
\Gamma \vdash \text{fix}[\tau](x. e) : \tau
\]

Rule (15.1g) reflects the self-referential nature of general recursion. To show that \( \text{fix}[\tau](x. e) \) has type \( \tau \), we assume that it is the case by assigning that type to the variable, \( x \), which stands for the recursive expression itself, and checking that the body, \( e \), has type \( \tau \) under this very assumption.

The structural rules, including in particular substitution, are admissible for the static semantics.

**Lemma 15.1.** If \( \Gamma, x : \tau \vdash e' : \tau' \), \( \Gamma \vdash e : \tau \), then \( \Gamma \vdash [e/x]e' : \tau' \).

### 15.2 Dynamics

The dynamic semantics of \( \mathcal{L}\{\text{nat} \rightarrow \} \) 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:

\[
\frac{}{z \ val} \quad (15.2a)
\]
\[
\frac{}{e \ val} \quad (15.2b)
\]
\[
\frac{}{s(e) \ val} \quad (15.2c)
\]
\[
\frac{}{\text{lam}[\tau](x. e) \ val} \quad (15.2c)
\]

The transition judgement \( e \mapsto e' \) is defined by the following rules:

\[
\frac{}{s(e) \mapsto s(e')} \quad (15.3a)
\]
\[
\frac{}{e \mapsto e'} \quad (15.3a)
\]
\[
\frac{}{\text{if}z(e; e_0; x. e_1) \mapsto \text{if}z(e'; e_0; x. e_1)} \quad (15.3b)
\]
15.2 Dynamics

\[
\text{if}z (z; e_0; x.e_1) \mapsto e_0
\]  (15.3c)

\[
s(e) \text{ val}
\]

\[
\text{if}z (s(e); e_0; x.e_1) \mapsto [e/x]e_1
\]  (15.3d)

\[
e_1 \mapsto e'_1
\]

\[
ap(e_1; e_2) \mapsto ap(e'_1; e_2)
\]  (15.3e)

\[
ap(lam[\tau] (x.e); e_2) \mapsto [e_2/x]e
\]  (15.3f)

\[
\text{fix}[\tau] (x.e) \mapsto [\text{fix}[\tau] (x.e)/x]e
\]  (15.3g)

Rule (15.3g) implements self-reference by substituting the recursive expression itself for the variable \(x\) in its body. This is called *unwinding* the recursion.

**Theorem 15.2 (Safety).**

1. If \(e : \tau\) and \(e \mapsto e'\), then \(e' : \tau\).

2. If \(e : \tau\), then either \(e \text{ val}\) 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 (15.3g). Suppose that \(\text{fix}[\tau] (x.e) : \tau\). By inversion of typing we have \(\text{fix}[\tau] (x.e) \vdash [\text{fix}[\tau] (x.e)/x]e : \tau\), 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 (15.1g) the result follows immediately since we may make progress by unwinding the recursion. \(\square\)

Definitional equivalence for \(\mathcal{L}\{\text{nat} \to \}\), written \(\Gamma \vdash e_1 \equiv e_2 : \tau\), is defined to be the strongest congruence containing the following axioms:

\[
\Gamma \vdash \text{if}z (\tau; z; e_0 . x.e_1) \equiv e_0 : \tau
\]  (15.4a)

\[
\Gamma \vdash \text{if}z (\tau; s(e); e_0 . x.e_1) \equiv [e/x]e_1 : \tau
\]  (15.4b)

\[
\Gamma \vdash \text{fix}[\tau] (x.e) \equiv [\text{fix}[\tau] (x.e)/x]e : \tau
\]  (15.4c)

\[
\Gamma \vdash \text{ap}(\text{lam}[\tau] (x.e_2); e_1) \equiv [e_1/x]e_2 : \tau
\]  (15.4d)

These rules are sufficient to calculate the value of any closed expression of type \text{nat}: if \(e : \text{nat}\), then \(e \equiv \overline{n} : \text{nat}\) iff \(e \mapsto^* \overline{n}\).
15.3 Definability

General recursion is a very flexible programming technique that permits a wide variety of functions to be defined within $L\{\text{nat} \to \}$ 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 $\text{fun} \ x : \tau_1 : \tau_2 \ is \ e$ 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

$\text{fun} \ x : \tau_1 : \tau_2 \ is \ e \mapsto \text{fun} \ x : \tau_1 : \tau_2 \ is \ e, e_1 / x, y \ e.$

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 $L\{\text{nat} \to \}$ using a combination of recursion and functions, writing

$\text{fix} \ x : \tau_1 \to \tau_2 \ is \ \lambda (y : \tau_1. e)$

for $\text{fun} \ x : \tau_1 : \tau_2 \ is \ e.$ 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 $L\{\text{nat} \to \}$ is defined in $L\{\text{nat} \to \}$ using recursive functions by taking the expression

$\text{natrec} e \{z \Rightarrow e_0 | s(x) \ with \ y \Rightarrow e_1\}$

to stand for the application, $e'(e)$, where $e'$ is the general recursive function

$\text{fun} f (u : \text{nat}) : \tau \ is \ \text{ifz} u \ {z \Rightarrow e_0 | s(x) \Rightarrow \ [f(x)/y]e_1}.$

The static and dynamic semantics of primitive recursion are derivable in $L\{\text{nat} \to \}$ using this expansion.

In general, functions definable in $L\{\text{nat} \to \}$ are partial in that they may be undefined for some arguments. A partial (mathematical) function, $\phi : \mathbb{N} \to \mathbb{N}$, is definable in $L\{\text{nat} \to \}$ iff there is an expression $e_\phi : \text{nat} \to \text{nat}$ such that $\phi(m) = n \text{ iff } e_\phi(m) \equiv n : \text{nat}$. So, for example, if $\phi$ is the totally undefined function, then $e_\phi$ is any function that loops without returning whenever it is called.
It is informative to classify those partial functions \( \phi \) that are definable in \( L\{\text{nat} \to \} \). These are the so-called \textit{partial recursive functions}, which are defined to be the primitive recursive functions augmented by the \textit{minimization} operation: given \( \phi \), define \( \psi(m) \) to be the least \( n \geq 0 \) such that (1) for \( m < n \), \( \phi(m) \) is defined and non-zero, and (2) \( \phi(n) = 0 \). If no such \( n \) exists, then \( \psi(m) \) is undefined.

**Theorem 15.3.** A partial function \( \phi \) on the natural numbers is definable in \( L\{\text{nat} \to \} \) iff it is partial recursive.

**Proof sketch.** Minimization is readily definable in \( L\{\text{nat} \to \} \), so it is at least as powerful as the class of partial recursive functions. Conversely, we may, with considerable tedium, define an evaluator for expressions of \( L\{\text{nat} \to \} \) as a partial recursive function, using Gödel-numbering to represent expressions as numbers. Consequently, \( L\{\text{nat} \to \} \) does not exceed the power of the class of partial recursive functions.

Church’s Law states that the partial recursive functions coincide with the class 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 \( L\{\text{nat} \to \} \) is as powerful as any other programming language with respect to the class of definable functions on the natural numbers.

The universal function, \( \phi_{\text{univ}} \) for \( L\{\text{nat} \to \} \) is the partial function on the natural numbers defined by

\[
\phi_{\text{univ}}(\langle e \rangle)(m) = n \text{ iff } e(m) \equiv n : \text{nat}.
\]

In contrast to \( L\{\text{nat} \to \} \), the universal function \( \phi_{\text{univ}} \) for \( L\{\text{nat} \to \} \) is partial (may be undefined for some inputs). It is, in essence, an interpreter that, given the code \( \langle e \rangle \) of a closed expression of type \( \text{nat} \to \text{nat} \), simulates the dynamic semantics to calculate the result, if any, of applying it to the \( m \), obtaining \( n \). 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 \( L\{\text{nat} \to \} \). In contrast, we proved in Chapter 14 that the analogous function is \textit{not} definable in \( L\{\text{nat} \to \} \) using the technique of diagonalization. It is instructive to examine why that argument does not apply in the present setting. As in Section 14.4 on page 116, we may derive the equivalence

\[
e_D(\langle e_D \rangle) \equiv s(e_D(\langle e_D \rangle))
\]

\(^1\)See Chapter 21 for further discussion of Church’s Law.
for \( L\{\text{nat} \rightarrow \} \). The difference, however, is that this equation is not inconsistent! Rather than being contradictory, it is merely a proof that the expression \( e_D(⌜e_D\⌝) \) does not terminate when evaluated, for if it did, the result would be a number equal to its own successor, which is impossible.

### 15.4 Co-Natural Numbers

The evaluation strategy for the successor operation specified by Rules (15.3) ensures that the type \( \text{nat} \) is interpreted standardly as the type of natural numbers. This means that if \( e : \text{nat} \) and \( e \\text{ val} \), then \( e \) is definitionally equivalent to a numeral. In contrast the lazy interpretation of successor, obtained by omitting Rule (15.3a), and requiring that \( s(e) \text{ val} \) for any \( e \), ruins this correspondence. The expression

\[
\omega = \text{fix } x : \text{nat is } s(x)
\]

evaluates to \( s(\omega) \), which is a value of type \( \text{nat} \). The “number” \( \omega \) may be thought of as an infinite stack of successors, which is therefore larger than any finite stack of successors starting with zero. In other words \( \omega \) is larger than any (finite) natural number, and hence can be regarded as an infinite “natural number.”

Of course it is stretching the terminology to refer to \( \omega \) as a number, much less as a natural number. Rather, we should say that the lazy interpretation of the successor operation gives rise to a distinct type, called the lazy natural numbers, or the co-natural numbers. The latter terminology arises from considering the co-natural numbers as “dual” to the ordinary natural numbers in the following sense. The standard natural numbers are inductively defined as the least type such that if \( e \equiv z : \text{nat} \) or \( e \equiv s(e') : \text{nat} \) for some \( e' : \text{nat} \), then \( e : \text{nat} \). Dually, the co-natural numbers may be regarded as the largest type such that if \( e : \text{conat} \), then either \( e \equiv z : \text{conat} \), or \( e \equiv s(e') : \text{nat} \) for some \( e' : \text{conat} \). The difference is that \( \omega : \text{conat} \), because \( \omega \) is definitionally equivalent to its own successor, whereas it is not the case that \( \omega : \text{nat} \), according to these definitions.

The duality between the natural numbers and the co-natural numbers is developed further in Chapter 19, wherein we consider the concepts of inductive and co-inductive types. Eagerness and laziness in general is discussed further in Chapter 40.

### 15.5 Exercises
Part V

Finite Data Types
Chapter 16

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 \( i \)th component is an element of the type \( \tau_i \), for each \( i \in I \). The components are accessed by \( I \)-indexed projection operations, generalizing the binary case. Special cases of the finite product include \( n \)-tuples, indexed by sets of the form \( I = \{ 0, \ldots, n - 1 \} \), and labelled tuples, or records, indexed by finite sets of symbols. Similarly to binary products, finite products admit both an eager and a lazy interpretation.

16.1 Nullary and Binary Products
The abstract syntax of products is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau )</td>
<td>( \text{unit} )</td>
<td>( \text{unit} )</td>
</tr>
<tr>
<td></td>
<td>( \text{prod}(\tau_1; \tau_2) )</td>
<td>( \tau_1 \times \tau_2 )</td>
<td></td>
</tr>
<tr>
<td>Expr</td>
<td>( e )</td>
<td>( \text{triv} )</td>
<td>( \langle \rangle )</td>
</tr>
<tr>
<td></td>
<td>( \text{pair}(e_1; e_2) )</td>
<td>( \langle e_1, e_2 \rangle )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{proj}<a href="e">l</a> )</td>
<td>( e \cdot 1 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{proj}<a href="e">r</a> )</td>
<td>( e \cdot r )</td>
<td></td>
</tr>
</tbody>
</table>

The type \( \text{prod}(\tau_1; \tau_2) \) is sometimes called the binary product of the types \( \tau_1 \) and \( \tau_2 \), and the type \( \text{unit} \) is correspondingly called the nullary product (of no types). We sometimes speak loosely of product types in such as way as to cover both the binary and nullary cases. The introductory form for the product type is called pairing, and its eliminatory forms are called projections. For the unit type the introductory form is called the unit element, or null tuple. There is no eliminatory form, there being nothing to extract from a null tuple.

The static semantics of product types is given by the following rules.

\[
\Gamma \vdash \text{triv} : \text{unit} \tag{16.1a}
\]

\[
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{pair}(e_1; e_2) : \text{prod}(\tau_1; \tau_2) \tag{16.1b}
\]

\[
\Gamma \vdash e : \text{prod}(\tau_1; \tau_2) \\
\Gamma \vdash \text{proj}[l](e) : \tau_1 \tag{16.1c}
\]

\[
\Gamma \vdash e : \text{prod}(\tau_1; \tau_2) \\
\Gamma \vdash \text{proj}[r](e) : \tau_2 \tag{16.1d}
\]

The dynamic semantics of product types is specified by the following rules:

\[
\text{triv\ val} \tag{16.2a}
\]

\[
\{e_1\ \text{val}\} \quad \{e_2\ \text{val}\} \\
\text{pair}(e_1; e_2)\ \text{val} \tag{16.2b}
\]

\[
\left\{ \begin{array}{l} 
\quad e_1 \mapsto e'_1 \\
\quad \text{pair}(e_1; e_2) \mapsto \text{pair}(e'_1; e_2) 
\end{array} \right\} \tag{16.2c}
\]
The syntax of finite product types is given by the following grammar:

\[
\begin{align*}
\text{Type} & \quad \tau & \quad ::= & \quad \prod [I] (i \mapsto \tau_i) & \quad \prod_{i \in I} \tau_i \\
\text{Expr} & \quad e & \quad ::= & \quad \text{tuple}[I] (i \mapsto e_i) & \quad \langle e_i \rangle_{i \in I} \\
& & & \quad \text{proj}[I] [i] (e) & \quad e \cdot i
\end{align*}
\]

For \( I \) a finite index set of size \( n \geq 0 \), the syntactic form \( \prod [I] (i \mapsto \tau_i) \) specifies an \( n \)-argument operator of arity \((0, 0, \ldots, 0)\) whose \( i \)th argument is the type \( \tau_i \). 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 \( \text{tuple}[I] (i \mapsto e_i) \) 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 static semantics of finite products is given by the following rules:

\[
\begin{align*}
\forall i \in I & \quad \Gamma \vdash e_i : \tau_i \\
\Gamma & \vdash \text{tuple}[I](i \mapsto e_i) : \text{prod}[I](i \mapsto \tau_i) \\
\Gamma & \vdash e : \text{prod}[I](i \mapsto e_i) \\
& \quad j \in I \\
\Gamma & \vdash \text{proj}[I][j](e) : \tau_j
\end{align*}
\] (16.3a)

In Rule (16.3b) the index \( j \in I \) is a particular element of the index set \( I \), whereas in Rule (16.3a), the index \( i \) ranges over the index set \( I \).

The dynamic semantics of finite products is given by the following rules:

\[
\begin{align*}
\{ (\forall i \in I) e_i \text{ val} \} \\
\text{tuple}[I](i \mapsto e_i) \text{ val}
\end{align*}
\] (16.4a)

\[
\begin{align*}
\{ & \quad e_j \mapsto e'_j \quad (\forall i \neq j) e'_i = e_i \\
& \quad \text{tuple}[I](i \mapsto e_i) \mapsto \text{tuple}[I](i \mapsto e'_i) \\
\} \\
\text{tuple}[I](i \mapsto e_i) \mapsto & \quad \text{tuple}[I](i \mapsto e'_i)
\end{align*}
\] (16.4b)

\[
\begin{align*}
e \mapsto e' & \\
\text{proj}[I][j](e) & \mapsto \text{proj}[I][j](e')
\end{align*}
\] (16.4c)

\[
\begin{align*}
tuple[I](i \mapsto e_i) \text{ val} \\
\text{proj}[I][j](\text{tuple}[I](i \mapsto e_i)) & \mapsto e_j
\end{align*}
\] (16.4d)

Rule (16.4b) 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 16.2 (Safety).** If \( e : \tau \), then either \( e \text{ val} \) 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 16.1 on page 129. \(\square\)

We may define nullary and binary products as particular instances of finite products by choosing an appropriate index set. The type \text{unit} may be defined as the product \( \prod_{\varnothing} \varnothing \) of the empty family over the empty index set, taking the expression \( \langle \rangle \) to be the empty tuple, \( \langle \varnothing \rangle \in \varnothing \). 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 \( \tau_1 \) and \( \tau_2 \). 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 \( e \cdot 1 \) and \( e \cdot 2 \) 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 \( \langle l_i : e_i \rangle \) for each \( 0 \leq i < n \). If \( e \) is such a tuple, then \( e \cdot l \) projects the component of \( e \) labeled by \( l \in L \).

### 16.3 Primitive and Mutual Recursion

Using products we may simplify the primitive recursion construct described in Chapter 14 by avoiding to pass the predecessor itself to the inductive step separately from passing the result of the recursive call on the predecessor. Writing \( \text{natiter} e \{ z \mapsto e_0 \mid s(x) \mapsto e_1 \} \) for this slightly simplified form of the recursor, we may define the original form, \( \text{natrec} e \{ z \mapsto e_0 \mid s(x_1) \text{ with } x_2 \mapsto e_1 \} \), to be the expression \( e' \cdot r \), where \( e' \) is the expression

\[
\text{natiter} e \{ z \mapsto \langle z, e_0 \rangle \mid s(x) \mapsto \langle s(x \cdot 1), [x \cdot 1, x \cdot r / x_0, x_1 | e_1] \rangle \}.
\]

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 static and dynamic semantics 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:

\[
\begin{align*}
E(0) &= 1 \\
O(0) &= 0 \\
E(n + 1) &= O(n) \\
O(n + 1) &= E(n)
\end{align*}
\]

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 \( L \{ \text{nat} \rightarrow \} \), 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 $e_o$ of type $nat \rightarrow nat \rightarrow nat$ so that $e_o(0)$ represents E and $e_o(1)$ 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, $\tau_{EO}$, given by

$$\prod \langle even : nat \rightarrow nat, odd : nat \rightarrow nat \rangle.$$ 

From this we will obtain the required mutually recursive functions as the projections $e_{EO} \cdot even$ and $e_{EO} \cdot odd$.

To effect the mutual recursion the expression $e_{EO}$ is defined to be

$$\text{fix this} : \tau_{EO} \text{ is } \langle even : e_E, odd : e_O \rangle,$$

where $e_E$ is the expression

$$\lambda (x : nat. \text{if} z x \{ z \Rightarrow s(z) \mid s(y) \Rightarrow \text{this} \cdot \text{odd}(y) \}),$$

and $e_O$ is the expression

$$\lambda (x : nat. \text{if} z x \{ z \Rightarrow z \mid s(y) \Rightarrow \text{this} \cdot \text{even}(y) \}).$$

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 so-called 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.

### 16.4 Exercises
Chapter 17

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.

17.1 Binary and Nullary Sums

The abstract syntax of sums is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>τ</td>
<td>::=</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>void</td>
<td>void</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sum(τ₁; τ₂)</td>
<td>τ₁ + τ₂</td>
</tr>
<tr>
<td>Expr</td>
<td>e</td>
<td>::=</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>abort<a href="e">τ</a></td>
<td>abort_τ e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in[l]<a href="e">τ</a></td>
<td>l · e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in[r]<a href="e">τ</a></td>
<td>r · e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>case(e; x₁; e₁; x₂; e₂)</td>
<td>case_e { l · x₁ ⇒ e₁</td>
</tr>
</tbody>
</table>

The type void is the nullary sum type, whose values are selected from a choice of zero alternatives — there are no values of this type, and so no introductory forms. The eliminatory form, abort[τ](e), aborts the computation in the event that e evaluates to a value, which it cannot do. The type
$\tau = \text{sum}(\tau_1; \tau_2)$ is the \textit{binary sum}. The elements of the sum type are \textit{labelled} to indicate whether they are drawn from the left or the right summand, either $\text{in}[l][\tau](e)$ or $\text{in}[r][\tau](e)$. A value of the sum type is eliminated by case analysis on the label of the value.

The static semantics of sum types is given by the following rules.

\[
\Gamma \vdash e : \text{void} \\
\Gamma \vdash \text{abort}[\tau](e) : \tau
\]  

(17.1a)

\[
\Gamma \vdash e : \tau_1 \quad \tau = \text{sum}(\tau_1; \tau_2) \\
\Gamma \vdash \text{in}[l][\tau](e) : \tau
\]  

(17.1b)

\[
\Gamma \vdash e : \tau_2 \quad \tau = \text{sum}(\tau_1; \tau_2) \\
\Gamma \vdash \text{in}[r][\tau](e) : \tau
\]  

(17.1c)

\[
\Gamma \vdash e : \text{sum}(\tau_1; \tau_2) \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau \\
\Gamma \vdash \text{case}(e; x_1. e_1; x_2. e_2) : \tau
\]  

(17.1d)

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 dynamic semantics of sums is given by the following rules:

\[
e \mapsto e' \\
\text{abort}[\tau](e) \mapsto \text{abort}[\tau](e')
\]  

(17.2a)

\[
\{ e \ \text{val} \} \\
\text{in}[l][\tau](e) \ \text{val}
\]  

(17.2b)

\[
\{ e \ \text{val} \} \\
\text{in}[r][\tau](e) \ \text{val}
\]  

(17.2c)

\[
\{ e \mapsto e' \} \\
\{ \text{in}[l][\tau](e) \mapsto \text{in}[l][\tau](e') \}
\]  

(17.2d)

\[
\{ e \mapsto e' \} \\
\{ \text{in}[r][\tau](e) \mapsto \text{in}[r][\tau](e') \}
\]  

(17.2e)

\[
e \mapsto e' \\
\text{case}(e; x_1. e_1; x_2. e_2) \mapsto \text{case}(e'; x_1. e_1; x_2. e_2)
\]  

(17.2f)

\[
\{ e \ \text{val} \} \\
\text{case(in}[l][\tau](e); x_1. e_1; x_2. e_2) \mapsto [e/x_1]e_1
\]  

(17.2g)
17.2 Finite Sums

\[
\begin{align*}
\{ e \text{ val} \} \\
\text{case}(\text{in}[r]\tau)(e);x_1.e_1;x_2.e_2) \rightarrow [e/x_2]e_2
\end{align*}
\]

(17.2h)

The bracketed premises and rules are to be included for an eager semantics, and excluded for a lazy semantics.

The coherence of the static and dynamic semantics is stated and proved as usual.

**Theorem 17.1 (Safety).**
1. If \( e : \tau \) and \( e \rightarrow e' \), then \( e' : \tau \).
2. If \( e : \tau \), then either \( e \text{ val} \) or \( e \rightarrow e' \) for some \( e' \).

**Proof.** The proof proceeds along standard lines, by induction on Rules (17.2) for preservation, and by induction on Rules (17.1) for progress. \(\square\)

17.2 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:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau )</td>
<td>( \text{sum}[I](i \mapsto \tau_i) )</td>
<td>( \sum_{i \in I} \tau_i )</td>
</tr>
<tr>
<td>Expr</td>
<td>( e )</td>
<td>( \text{in}[I]<a href="e">j</a> )</td>
<td>( j \cdot e )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{case}[I](e;i \mapsto x_i.e_i) )</td>
<td>( \text{case } e { i \cdot x_i \Rightarrow e_i }_i \in I )</td>
</tr>
</tbody>
</table>

The abstract binding tree representation of the finite case expression involves an \( I \)-indexed family of abstractors \( x_i.e_i \), but is otherwise similar to the binary form. 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 static semantics of finite sums is defined by the following rules:

\[
\Gamma \vdash e : \tau_j \quad j \in I \\
\Gamma \vdash \text{in}[I][j](e) : \text{sum}[I](i \mapsto \tau_i) \quad (17.3a)
\]

\[
\Gamma \vdash e : \text{sum}[I](i \mapsto \tau_i) \quad (\forall i \in I) \quad \Gamma, x_i : \tau_i \vdash e_i : \tau \\
\Gamma \vdash \text{case}[I](e;i \mapsto x_i.e_i) : \tau \quad (17.3b)
\]

These rules generalize to the finite case the static semantics for nullary and binary sums given in Section 17.1 on page 135.
The dynamic semantics of finite sums is defined by the following rules:

\[
\begin{align*}
\{e \text{ val}\} & \quad \text{in}\ [I]\ [j]\ (e) \text{ val} \\
\{e \mapsto e'\} & \quad \text{in}\ [I]\ [j]\ (e) \mapsto \text{in}\ [I]\ [j]\ (e') \\
e \mapsto e' & \quad \text{case}\ [I]\ (e; i \mapsto x_i. e_i) \mapsto \text{case}\ [I]\ (e'; i \mapsto x_i. e_i) \\
\text{in}\ [I]\ [j]\ (e) \text{ val} & \quad \text{case}\ [I]\ (\text{in}\ [I]\ [j]\ (e); i \mapsto x_i. e_i) \mapsto [e / x_j]e_j
\end{align*}
\]

These again generalize the dynamic semantics of binary sums given in Section 17.1 on page 135.

**Theorem 17.2** (Safety). If \( e : \tau \), then either \( e \text{ val} \) 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 17.1 on page 135.

As with products, nullary and binary sums are special cases of the finite form. The type \( \text{void} \) may be defined to be the sum type \( \sum_{\in \emptyset} \emptyset \) of the empty family of types. The expression \( \text{abort} (e) \) may corresponding be defined as the empty case analysis, \( \text{case} e \{ \emptyset \} \). Similarly, the binary sum type \( \tau_1 + \tau_2 \) may be defined as the sum \( \sum_{i \in I} \tau_i \), where \( I = \{ l, r \} \) is the two-element index set. The binary sum injections \( l \cdot e \) and \( r \cdot e \) are defined to be their counterparts, \( l \cdot e \) and \( r \cdot e \), respectively. Finally, the binary case analysis,

\[
\text{case} e \{ 1 \cdot x_1 \Rightarrow e_1 | r \cdot x_r \Rightarrow e_r \},
\]

is defined to be the case analysis, \( \text{case} e \{ i \cdot x_i \Rightarrow \tau_i \}_{i \in I} \). It is easy to check that the static and dynamic semantics of sums given in Section 17.1 on page 135 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.
17.3 Uses for 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.

17.3.1 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 \( e : \text{unit} \), then if \( e \) evaluates to a value, it must be \( \text{unit} \) — in other words, \( e \) has no interesting value (but it could diverge). On the other hand, if \( e : \text{void} \), then \( e \) must not yield a value; if it were to have a value, it would have to be a value of type \( \text{void} \), 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!

17.3.2 Booleans

Perhaps the simplest example of a sum type is the familiar type of Booleans, whose syntax is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau ) ::= bool</td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>Expr</td>
<td>( e ) ::= tt</td>
<td>tt</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ff</td>
<td>ff</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if(( e; e_1; e_2 ))</td>
<td>if ( e ) then ( e_1 ) else ( e_2 )</td>
</tr>
</tbody>
</table>

The values of type \( \text{bool} \) are \( \text{tt} \) and \( \text{ff} \). The expression \( \text{if}(e; e_1; e_2) \) branches on the value of \( e : \text{bool} \). We leave a precise formulation of the static and dynamic semantics of this type as an exercise for the reader.

The type \( \text{bool} \) is definable in terms of binary sums and nullary products:

\[
\text{bool} = \text{sum(unit;unit)} \quad \quad \quad \quad (17.5a)
\]

\[
\text{tt} = \text{in}[l][\text{bool}](\text{triv}) \quad \quad \quad \quad (17.5b)
\]

\[
\text{ff} = \text{in}[r][\text{bool}](\text{triv}) \quad \quad \quad \quad (17.5c)
\]

\[
\text{if}(e;e_1;e_2) = \text{case}(e;x_1.e_1;x_2.e_2) \quad \quad \quad \quad (17.5d)
\]
In the last equation above the variables $x_1$ and $x_2$ are chosen arbitrarily such that $x_1 \not\in e_1$ and $x_2 \not\in e_2$. (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 dynamic semantics of the type $\text{bool}$ is engendered by these definitions.

### 17.3.3 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 $\text{suit}$, whose elements are $\spadesuit$, $\heartsuit$, $\diamondsuit$, and $\clubsuit$, has as elimination form the case analysis

$$\text{case } e \{ \spadesuit \Rightarrow e_0 | \heartsuit \Rightarrow e_1 | \diamondsuit \Rightarrow e_2 | \clubsuit \Rightarrow e_3 \},$$

which distinguishes among the four suits. Such finite enumerations are easily representable as sums. For example, we may define $\text{suit} = \sum_{e \in I} \text{unit}$, where $I = \{ \spadesuit, \heartsuit, \diamondsuit, \clubsuit \}$ 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.

### 17.3.4 Options

Another use of sums is to define the option types, which have the following syntax:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$\text{opt}(\tau)$</td>
<td>$\tau \text{ opt}$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{just}(e)$</td>
<td>$\text{just}(e)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{ifnull}[\tau](e; e_1; x; e_2)$</td>
<td>$\text{check } e{\text{null } \Rightarrow e_1</td>
</tr>
</tbody>
</table>

The type $\text{opt}(\tau)$ represents the type of “optional” values of type $\tau$. The introductory forms are null, corresponding to “no value”, and $\text{just}(e)$, corresponding to a specified value of type $\tau$. The elimination form discriminates between the two possibilities.
The option type is definable from sums and nullary products according to the following equations:

\[
\text{opt}(\tau) = \text{sum}(\text{unit}; \tau) \\
\text{null} = \text{in}[1][\text{opt}(\tau)](\text{triv}) \\
\text{just}(e) = \text{in}[r][\text{opt}(\tau)](e) \\
\text{ifnull}[\tau](e; e_1; x_2.e_2) = \text{case}(e; e_1; x_2.e_2)
\]

We leave it to the reader to examine the static and dynamic semantics 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 \( \text{null}: \tau \rightarrow \text{bool} \), that yields \( \text{true} \) if its argument is null, and \( \text{false} \) otherwise. This allows the programmer to take steps to avoid using null as a value of the type it purports to inhabit. Consequently, programs are riddled with conditionals of the form

\[
\text{if null}(e) \text{ then...error... else...proceed...}
\]

Despite this, “null pointer” exceptions at run-time are rampant, in part because it is quite easy to overlook the need for such a test, and in part because detection of a null pointer leaves little recourse other than abortion of the program.

The underlying problem may be traced to the failure to distinguish the type \( \tau \) from the type \( \text{opt}(\tau) \). Rather than think of the elements of type \( \tau \) as pointers, and thereby have to worry about the null pointer, one instead distinguishes between a genuine value of type \( \tau \) and an optional value of type \( \tau \). An optional value of type \( \tau \) may or may not be present, but, if it is, the underlying value is truly a value of type \( \tau \) (and cannot be null). The elimination form for the option type,

\[
\text{ifnull}[\tau](e; e_{\text{error}}; x.e_{\text{ok}})
\]
propagates the information that \( e \) is present into the non-null branch by binding a genuine value of type \( \tau \) to the variable \( x \). The case analysis effects a change of type from “optional value of type \( \tau \)” to “genuine value of type \( \tau \)”, so that within the non-null branch no further null checks, explicit or implicit, are required. Observe that such a change of type is not achieved by the simple Boolean-valued test exemplified by expression (17.7); the advantage of option types is precisely that it does so.

### 17.4 Exercises

1. Formulate general \( n \)-ary sums in terms of nullary and binary sums.

2. Explain why it makes little sense to consider self-referential sum types.
Chapter 18

Pattern Matching

Pattern matching is a natural and convenient generalization of the elimination forms for product and sum types. For example, rather than write

\[
\text{let } x \text{ be } e \text{ in } x \cdot 1 + x \cdot r
\]

to add the components of a pair, \( e \), of natural numbers, we may instead write

\[
\text{match } e \{ x, y, \langle x, y \rangle \Rightarrow x + y \},
\]

using pattern matching to name the components of the pair and refer to them directly. The first argument to the \texttt{match} expression is called the \texttt{match value} and the second argument consist of a finite sequence of \texttt{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 \texttt{match} expression. Each rule consists of a \texttt{pattern}, possibly involving variables, and an \texttt{expression} that may involve those variables (as well as any others currently in scope). The value of the \texttt{match} is determined by considering each rule in the order given to determine the first rule whose pattern matches the \texttt{match value}. If such a rule is found, the value of the \texttt{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 \texttt{match value}.

Pattern matching becomes more interesting, and useful, when combined with sums. The patterns \( 1 \cdot x \) and \( r \cdot x \) 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 \texttt{match} expresses the computation that, when given a pair of type \((\textit{unit} + \textit{unit}) \times \textit{nat}\), either doubles or squares its second component.
depending on the form of its first component:

\[
\text{match } e \{ x . \langle l \cdot \langle \rangle, x \rangle \Rightarrow x + x \mid y . \langle r \cdot \langle \rangle, y \rangle \Rightarrow y * y \}. \tag{18.1}
\]

It is an instructive exercise to express the same computation using only the primitives for sums and products given in Chapters 16 and 17.

In this chapter we study a simple language, \( L \{ \text{pat} \} \), of pattern matching over eager product and sum types.

### 18.1 A Pattern Language

The main challenge in formalizing \( L \{ \text{pat} \} \) is to manage properly the binding and scope of variables. The key observation is that a rule, \( p \Rightarrow e \), binds variables in both the pattern, \( p \), and the expression, \( e \), simultaneously. Each rule in a sequence of rules may bind a different number of variables, independently of the preceding or succeeding rules. This gives rise to a somewhat unusual abstract syntax for sequences of rules that permits each rule to have a different valence. For example, the abstract syntax for expression (18.1) is given by

\[
\text{match } e \{ r_1; r_2 \},
\]

where \( r_1 \) is the rule

\[
x . \langle l \cdot \langle \rangle, x \rangle \Rightarrow x + x
\]

and \( r_2 \) is the rule

\[
y . \langle r \cdot \langle \rangle, y \rangle \Rightarrow y * y.
\]

The salient point is that each rule binds its own variables, in both the pattern and the expression.

The abstract syntax of \( L \{ \text{pat} \} \) is defined by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>( e )</td>
<td>match( (e; rs) )</td>
<td>match ( e { rs } )</td>
</tr>
<tr>
<td>Rules</td>
<td>( rs )</td>
<td>rules [ n ] (( r_1 ); \ldots; ( r_n ))</td>
<td>( r_1 \mid \ldots \mid r_n )</td>
</tr>
<tr>
<td>Rule</td>
<td>( r )</td>
<td>( x_1, \ldots, x_k ).rule( (p; e) )</td>
<td>( x_1, \ldots, x_n ).p ( \Rightarrow e )</td>
</tr>
<tr>
<td>Pattern</td>
<td>( p )</td>
<td>wild</td>
<td>( - )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>triv ( \langle \rangle )</td>
<td>( \langle \rangle )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pair( (p_1; p_2) )</td>
<td>( \langle p_1, p_2 \rangle )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in[ l ](( p ))</td>
<td>( 1 \cdot p )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in[ r ](( p ))</td>
<td>( r \cdot p )</td>
</tr>
</tbody>
</table>
The operator \( \text{rules}[n] \) has arity \((k_1, \ldots, k_n)\), where \( n \geq 0 \) and, for each \( 1 \leq i \leq n \), the \( i \)th rule has valence \( k_i \geq 0 \). Correspondingly, the \( i \)th rule consists of an abstractor binding \( k_i \) variables in the pattern and expression. A pattern is either a variable, a \textit{wild card} pattern, a \textit{unit pattern} matching only the trivial element of the \textit{unit} type, a \textit{pair pattern}, or a \textit{choice pattern}.

18.2 Statics

The static semantics of \( \mathcal{L}\{\text{pat}\} \) makes use of a \textit{linear} hypothetical judgement of the form

\[
x_1 : \tau_1, \ldots, x_k : \tau_k \vdash p : \tau.
\]

The meaning of this judgement is almost the same as that of the ordinary judgement

\[
x_1 : \tau_1, \ldots, x_k : \tau_k \vdash p : \tau,
\]

except that the hypotheses are treated specially so as to ensure that each variable is used exactly once in the pattern. This is achieved by dropping the usual structural rules of weakening and contraction, and limiting the combination of assumptions \( \Lambda_1 \Lambda_2 \) to disjoint sets of assumptions, which is written \( \Lambda_1 \# \Lambda_2 \).

The pattern typing judgement \( \Lambda \vdash p : \tau \) is inductively defined by the following rules:

\[
\begin{align*}
\frac{x : \tau \vdash x : \tau}{\varnothing \vdash \cdot : \tau} & \quad \text{(18.2a)} \\
\frac{\varnothing \vdash \cdot : \tau}{\varnothing \vdash \langle \rangle : \text{unit}} & \quad \text{(18.2b)} \\
\frac{\Lambda_1 \vdash p_1 : \tau_1 \quad \Lambda_2 \vdash p_2 : \tau_2 \quad \Lambda_1 \# \Lambda_2}{\Lambda_1 \Lambda_2 \vdash \langle p_1, p_2 \rangle : \tau_1 \times \tau_2} & \quad \text{(18.2d)} \\
\frac{\Lambda_1 \vdash p : \tau_1}{\Lambda_1 \vdash 1 \cdot p : \tau_1 + \tau_2} & \quad \text{(18.2e)} \\
\frac{\Lambda_2 \vdash p : \tau_2}{\Lambda_2 \vdash r \cdot p : \tau_1 + \tau_2} & \quad \text{(18.2f)}
\end{align*}
\]

Rule (18.2a) states that a variable is a pattern of type \( \tau \) provided that \( x : \tau \) is the \textit{only} assumption of the judgement. Rule (18.2d) expresses the formation of a pair pattern from patterns for its components, and imposes the
requirement that the variables used in the two sub-patterns must be disjoint, ensuring thereby that no variable may be used more than once in a pattern.

The judgment $x_1, \ldots, x_k. p \Rightarrow e : \tau > \tau'$ states that the rule $x_1, \ldots, x_k. p \Rightarrow e$ matches a value of type $\tau$ against the pattern $p$, binding the variables $x_1, \ldots, x_k$, and yields a value of type $\tau'$.

\[
\begin{array}{c}
\Lambda \vdash p : \tau \\
\Gamma \vdash e : \tau' \\
\Lambda = x_1 : \tau_1, \ldots, x_k : \tau_k \\
\Gamma \# \Lambda
\end{array}
\]  
\hspace{2cm} (18.3a)

\[
\begin{array}{c}
\Gamma \vdash r_1 : \tau > \tau' \\
\ldots \\
\Gamma \vdash r_n : \tau > \tau'
\end{array}
\]  
\hspace{2cm} (18.3b)

Rule (18.3a) makes use of the pattern typing judgement to determine both the type of the pattern, $p$, and also the types of its variables, $\Lambda$.\footnote{It may help to read the hypotheses, $\Lambda$, as an “output,” rather than as an “input,” of the judgement, in contrast to the usual reading of a hypothetical judgement.} These variables are available for use within $e$, along with any other variables that may be in scope, without restriction. In Rule (18.3b) if the parameter, $n$, is zero, then the rule states that the empty sequence has an arbitrary domain and range, since it matches no value and yields no result.

Finally, the typing rule for the \texttt{match} expression is given as follows:

\[
\begin{array}{c}
\Gamma \vdash e : \tau \\
\Gamma \vdash rs : \tau > \tau'
\end{array}
\]  
\hspace{2cm} (18.4)

The \texttt{match} expression has type $\tau'$ if the rules transform any value of type $\tau$, the type of the \texttt{match} expression, to a value of type $\tau'$.

\subsection*{18.3 Dynamics}

The dynamics of pattern matching is defined using substitution to “guess” the bindings of the pattern variables. The dynamics is given by the judgements $e \mapsto e'$, representing a step of computation, and $e \lor$, representing the checked condition of pattern matching failure.

\[
\begin{array}{c}
e \mapsto e' \\
\text{match } e \{ rs \} \mapsto \text{match } e' \{ rs \}
\end{array}
\]  
\hspace{2cm} (18.5a)

\[
\begin{array}{c}
\text{match } e \{ \} \lor
\end{array}
\]  
\hspace{2cm} (18.5b)
Rule (18.5b) specifies that evaluation results in a checked error once all rules are exhausted. Rules (18.5c) 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 18.1** (Preservation). If $e \mapsto e'$ and $e : \tau$, then $e' : \tau$.

**Proof.** By a straightforward induction on the derivation of $e \mapsto e'$, making use of the evident substitution lemma for the statics. □

The formulation of pattern matching given in Rules (18.5) does not define how pattern matching is to be accomplished, rather it simply checks whether there is substitution for the variables in the pattern that results in the candidate value. This streamlines the presentation of the dynamics and the proof of preservation, but could be considered “too slick” in that it does not show how to find such a substitution or to determine that none exists. This gap may be filled by introducing two judgements. The first,

$$e_1 \triangleleft x_1, \ldots, e_k \triangleleft x_k \vdash p \triangleleft e,$$

where $e$ val and $e_i$ val for each $1 \leq i \leq k$, is a linear hypothetical judgement stating that $[e_1, \ldots, e_k/x_1, \ldots, x_k]p = e$. The second, $e \perp p$, where $e$ val, states that $e$ fails to match the pattern $p$.

The pattern matching judgement is defined by the following rules, writing $\Theta$ for the assumptions governing variables:

$$x \triangleleft e \vdash x \triangleleft e$$ (18.6a)

$$\Theta \vdash \_ \triangleleft e$$ (18.6b)

$$\Theta \vdash \langle \rangle \triangleleft \langle \rangle$$ (18.6c)

---

**OCTOBER 16, 2009**

**DRAFT** 18:42
The rules for a pattern mismatch are as follows:

\[
\begin{align*}
\Theta \vdash p \triangleleft e & \quad \Theta \vdash p \triangleleft e \\
\Theta \vdash \langle p_1, p_2 \rangle \triangleleft \langle e_1, e_2 \rangle & \quad (18.6d)
\end{align*}
\]

\[
\begin{align*}
\Theta \vdash p \triangleleft e & \quad \Theta \vdash 1 \cdot p \triangleleft 1 \cdot e \\
\Theta \vdash \langle e_1, e_2 \rangle \triangleleft \langle p_1, p_2 \rangle & \quad (18.6e)
\end{align*}
\]

\[
\begin{align*}
\Theta \vdash p \triangleleft e & \quad \Theta \vdash r \cdot p \triangleleft r \cdot e \\
\Theta \vdash e \triangleleft p & \quad (18.6f)
\end{align*}
\]

The salient property of these judgements is that they are complementary.

**Theorem 18.2.** Suppose that \( e : \tau, x_1 : \tau_1, \ldots, x_k : \tau_k \vdash p : \tau \), and \( e \ \text{val} \). Then either there exists \( e_1, \ldots, e_k \) such that \( x_1 \triangleleft e_1, \ldots, x_k \triangleleft e_k \vdash p \triangleleft e \), or \( e \perp p \).

**Proof.** By rule induction on Rules (18.2), making use of the canonical forms lemma to characterize the shape of \( e \) based on its type. \( \square \)
18.4 Exhaustiveness and Redundancy

While it is possible to state and prove a progress theorem for $L\{\text{pat}\}$ as defined in Section 18.1 on page 144, 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.

The statics of rules given in Section 18.1 on page 144 does not ensure exhaustiveness or irredundancy of rules. To do so we introduce a language of match conditions that identify a subset of the closed values of a type. With each rule we associate a match condition 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 condition of some rule in the sequence. A rule in a sequence is redundant if every value that satisfies its match condition also satisfies the match condition of some preceding rule.

The language of match conditions is defined by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond</td>
<td>$\xi$</td>
<td>$\xi ; ::= ; \text{any}[\tau]$</td>
<td>$\top_\tau$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{in}[l]\text{[sum}(\tau_1; \tau_2)\text{]}(\xi_1)$</td>
<td>$l \cdot \xi_1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{in}[r]\text{[sum}(\tau_1; \tau_2)\text{]}(\xi_2)$</td>
<td>$r \cdot \xi_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{triv}$</td>
<td>$\langle \rangle$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{pair}(\xi_1; \xi_2)$</td>
<td>$\langle \xi_1, \xi_2 \rangle$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{nil}[\tau]$</td>
<td>$\perp_\tau$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{alt}(\xi_1; \xi_2)$</td>
<td>$\xi_1 \lor \xi_2$</td>
</tr>
</tbody>
</table>

The judgement $\xi : \tau$ is defined by the following rules:

$$\top_\tau : \tau$$ \hspace{1cm} (18.8a)

$$\frac{\xi_1 : \tau_1}{l \cdot \xi_1 : \tau_1 + \tau_2}$$ \hspace{1cm} (18.8b)

$$\frac{\xi_1 : \tau_2}{r \cdot \xi_1 : \tau_1 + \tau_2}$$ \hspace{1cm} (18.8c)
18.4 Exhaustiveness and Redundancy

Informally, \( \xi : \tau \) means that \( \xi \) constrains values of type \( \tau \).

For \( \xi : \tau \), \( e : \tau \), and \( e \) val, we define the satisfaction judgement \( e \models \xi \) as follows:

\[
\begin{align*}
(\langle \rangle : \text{unit}) & \quad (18.8d) \\
\xi_1 : \tau_1 & \quad \xi_2 : \tau_2 \\
\langle \xi_1, \xi_2 \rangle : \tau_1 \times \tau_2 & \quad (18.8e) \\
\perp_\tau : \tau & \quad (18.8f) \\
\xi_1 : \tau & \quad \xi_2 : \tau \\
\xi_1 \lor \xi_2 : \tau & \quad (18.8g)
\end{align*}
\]

Finally, we instrument the statics of patterns and rules to associate a match condition that specifies the values that may be matched by that pattern or rule. This allows us to ensure that rules are both exhaustive and irredundant.
The judgement $\Gamma \vdash p : \tau$ augments the judgement $\Lambda \vdash p : \tau$ with a match constraint characterizing the set of values of type $\tau$ matched by the pattern $p$. It is inductively defined by the following rules:

$$x : \tau \vdash x : \tau [\uparrow_\tau]$$ (18.10a)

$$\emptyset \vdash : \tau [\uparrow_\tau]$$ (18.10b)

$$\emptyset \vdash \langle \rangle : \text{unit} [\langle \rangle]$$ (18.10c)

$$\begin{array}{c}
\Lambda_1 \vdash p : \tau_1 [\xi_1] \\
\Lambda_1 \vdash 1 \cdot p : \tau_1 + \tau_2 [1 \cdot \xi_1]
\end{array}$$ (18.10d)

$$\begin{array}{c}
\Lambda_2 \vdash p : \tau_2 [\xi_2] \\
\Lambda_2 \vdash r \cdot p : \tau_1 + \tau_2 [r \cdot \xi_2]
\end{array}$$ (18.10e)

$$\begin{array}{c}
\Lambda_1 \vdash p_1 : \tau_1 [\xi_1] \\
\Lambda_2 \vdash p_2 : \tau_2 [\xi_2] \\
\Lambda_1 \ # \ \Lambda_2
\end{array}$$ (18.10f)

Rules (18.10a) to (18.10b) specify that all values of the pattern type are matched. Rule (18.10c) specifies that the only value of type $\text{unit}$ is matched by the pattern. Rules (18.10d) to (18.10e) specify that the pattern matches only those values with the specified injection tag and whose argument is matched by the specified pattern. Rule (18.10f) specifies that the pattern matches only pairs whose components match the specified patterns.

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.

$$\begin{array}{c}
\Lambda \vdash p : \tau [\xi] \\
\Gamma \vdash e : \tau'
\end{array}$$ (18.11a)

$$\begin{array}{c}
\Gamma \vdash x_1, \ldots , x_k. p \Rightarrow e : \tau > \tau' [\xi]
\end{array}$$

$$\begin{array}{c}
\Gamma \vdash r_1 : \tau > \tau' [\xi_1] \\
\vdots \\
\Gamma \vdash r_n : \tau > \tau' [\xi_n]
\end{array}$$

$$\begin{array}{c}
(\forall 1 \leq i \leq n) \xi_i \neq \xi_1 \lor \ldots \lor \xi_{i-1}
\end{array}$$ (18.11b)

Rule (18.11b) ensures that each successive rule is irredudant relative to the preceding rules in that it demands that it not be the case that every value
satisfying $\xi_i$ satisfies some preceding $\xi_j$. That is, it requires that there be some value satisfying $\xi_i$ that does not satisfy some preceding $\xi_j$.

Finally, the typing rule for match expressions requires exhaustiveness:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash rs : \tau > \tau' [\xi] \quad \top_{\tau} \models \xi}{\Gamma \vdash \text{match } e \{ rs \} : \tau'} \quad (18.12)$$

The third premise ensures that every value of type $\tau$ satisfies the constraint $\xi$ representing the values matched by some rule in the given rule sequence.

The additional constraints on the statics are sufficient to ensure progress, because no well-formed match expression can fail to match a value of the specified type. If a given sequence of rules is inexhaustive, this can always be rectified by including a “default” rule of the form $x. x \Rightarrow e_x$, where $e_x$ handles the unmatched value $x$ gracefully, perhaps by raising an exception (see Chapter 28 for a discussion of exceptions).

**Theorem 18.3.** If $e : \tau$, then either $e \text{ val}$ or there exists $e'$ such that $e \rightarrow e'$.

### 18.5 Exercises
Part VI

Infinite Data Types
Chapter 19

Inductive and Co-Inductive Types

The inductive and the coinductive types are two important classes of recursive types. 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.

19.1 Motivating Examples

The most important example of an inductive type is the type of natural numbers as formalized in Chapter 14. 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, natiter e {z⇒e0 | s(x)⇒e1}, 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 \textit{nat}.

With a view towards deriving the type \textit{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 static semantics of this reformulation:

\[
\begin{align*}
\Gamma \vdash e : \text{unit} + \text{nat} \\
\Gamma \vdash \text{foldnat}(e) : \text{nat}
\end{align*}
\]

(19.1a)

\[
\begin{align*}
\Gamma, x : \text{unit} + \tau \vdash e_1 : \tau \\
\Gamma \vdash \text{recnat}[x.e_1](e_2) : \tau
\end{align*}
\]

(19.1b)

The expression \text{foldnat}(e) is the unique introductory form of the type \textit{nat}. Using this, the expression \(z\) is defined to be \text{foldnat}(1 \cdot \langle \rangle), \) and \(s(e)\) is defined to be \text{foldnat}(r \cdot e). The recursor, \text{recnat}[x.e_1](e_2), takes a consolidated basis and inductive step as argument, as well as the natural number of which to recur.

The dynamic semantics of the consolidated primitives is defined by the following rules:

\[
\begin{align*}
\text{fold}(e) \rightarrow \text{val} \\
\text{recnat}[x.e_1](e_2) \rightarrow \text{recnat}[x.e_1](e_2')
\end{align*}
\]

(19.2a)

\[
\begin{align*}
\left\{ e_b = 1 \cdot \langle \rangle, \ e_s = r \cdot \text{recnat}[x.e_1](y) \right\} \rightarrow \\
\text{recnat}[x.e_1](\text{foldnat}(e_2)) \\
\rightarrow \text{case } e_2 \{ 1 \cdot \langle \rangle \Rightarrow [e_b / x]e_1 \mid r \cdot y \Rightarrow [e_s / x]e_1 \}
\end{align*}
\]

(19.2c)

Observe that in the successor branch of Rule (19.2c) the recursor is applied to the predecessor before substituting the result into \(e_1\).

An illustrative example of a coinductive type is the type of \textit{streams} of natural numbers. A stream is an infinite sequence of natural numbers such that nan 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 \textit{dual} to the analogous property of the eliminator 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: $\text{hd}(e)$ returns the next, or head, element of the stream, and $\text{tl}(e)$ returns the tail of the stream, the stream resulting when the head element is removed. A stream is introduced by a generator, the dual of 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 static semantics of streams is given by the following rules:

$$
\Gamma \vdash e : \text{stream} \\
\Gamma \vdash \text{hd}(e) : \text{nat} \quad (19.3a)
$$

$$
\Gamma \vdash e : \text{stream} \\
\Gamma \vdash \text{tl}(e) : \text{stream} \quad (19.3b)
$$

$$
\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_1 : \text{nat} \quad \Gamma, x : \tau \vdash e_2 : \tau \\
\Gamma \vdash \text{strgen } e < \text{hd}(x) \Rightarrow e_1 \& \text{tl}(x) \Rightarrow e_2 > : \text{stream} \quad (19.3c)
$$

In Rule (19.3c) the current state of the stream is given by the expression $e$ of some type $\tau$, and the head and tail of the stream are determined by the expressions $e_1$ and $e_2$, respectively, as a function of the current state.

The dynamic semantics of the foregoing stream primitives is given by the following rules:

$$
\text{strgen } e < \text{hd}(x) \Rightarrow e_1 \& \text{tl}(x) \Rightarrow e_2 > \vdash e \mapsto e' \quad (19.4a)
$$

$$
\text{hd}(e) \mapsto \text{hd}(e') \quad (19.4b)
$$

$$
\text{hd} (\text{strgen } e < \text{hd}(x) \Rightarrow e_1 \& \text{tl}(x) \Rightarrow e_2 >) \mapsto [e/x]e_1 \quad (19.4c)
$$

$$
\text{tl} (\text{strgen } e < \text{hd}(x) \Rightarrow e_1 \& \text{tl}(x) \Rightarrow e_2 >) \mapsto \text{strgen } [e/x]e_2 < \text{hd}(x) \Rightarrow e_1 \& \text{tl}(x) \Rightarrow e_2 > \quad (19.4e)
$$

Rules (19.4c) and (19.4e) 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 static semantics:

\[
\begin{align*}
\Gamma & \vdash e : \text{stream} \\
\Gamma & \vdash \text{unfoldstream}(e) : \text{nat} \times \text{stream}
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : \tau & \vdash e_1 : \text{nat} \times \tau \quad \Gamma & \vdash e_2 : \tau \\
\Gamma & \vdash \text{genstream}[x. e_1](e_2) : \text{stream}
\end{align*}
\]

Rule (19.5a) states that a stream may be unfolded into a pair consisting of its head, a natural number, and its tail, another stream. The head, \( \text{hd}(e) \), and tail, \( \text{tl}(e) \), of a stream, \( e \), are defined to be the projections \( \text{unfoldstream}(e) \cdot 1 \) and \( \text{unfoldstream}(e) \cdot r \), respectively. Rule (19.5b) 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 dynamic semantics of these primitives is given by the following rules:

\[
\begin{align*}
\text{genstream}[x. e_1](e_2) & \entails \text{val} \\
\frac{e \mapsto e'}{\text{unfoldstream}(e) \mapsto \text{unfoldstream}(e')}
\end{align*}
\]

\[
\begin{align*}
\frac{e_1 = ([e_2/x]e_1) \cdot l \quad e_1 = ([e_2/x]e_1) \cdot r}{\text{unfoldstream}(\text{genstream}[x. e_1](e_2)) \mapsto (e_h, \text{genstream}[x. e_1](e_1))}
\end{align*}
\]

The generator is applied to the new state component of \([e_2/x]e_1\) in Rule (19.6c) to obtain the tail of the stream.

### 19.2 Generic Programming

A fully general account of inductive and coinductive types requires the concept of type-generic, or just generic, programming. A generic program is one whose behavior is determined by a type operator, which is a type expression with a distinguished type variable. The generic program induced by a type operator is one that performs a specified operation at each spot in a data structure corresponding to an occurrence of the type variable in the operator expression, and otherwise leaves all other parts of the data structure fixed.
19.2.1 Positive Type Operators

A type operator is an abstractor of the form $t \cdot \tau$ such that $\{ t \mid t \ \text{type} \vdash \tau$ type. A type operator is therefore a type, $\tau$, containing zero or more occurrences of a type variable, $t$. The occurrences of $t$ in $\tau$ mark the spots at which we wish to perform some computation, leaving the other spots alone. We may substitute another type, $\sigma$, for occurrences of $t$ in $\tau$ to obtain $[\sigma/t] \tau$, and we may perform a computation transforming values of type $\sigma_1$ into values of type $\sigma_2$ at occurrence of $t$ in $\tau$, thereby transforming a value of type $[\sigma_1/t] \tau$ into a value of type $[\sigma_2/t] \tau$. Such a transformation is always possible, provided that $t \cdot \tau$ is a positive type operator, one for which $t$ never occurs within the domain of a function type.

For example, given a transformation, $f$, from values of type $\sigma_1$ to values of type $\sigma_2$, the (positive) type operator $t \cdot \text{unit} + t$ induces a transformation from type $\text{unit} + \sigma_1$ to type $\text{unit} + \sigma_2$ that sends $l \cdot \langle \rangle$ to itself, and $r \cdot e$ to $r \cdot f(e)$. Similarly, the type operator $t \cdot \text{unit} + (\text{nat} \times t)$ induces a transformation from type $\text{unit} + (\text{nat} \times \sigma_1)$ to type $\text{unit} + (\text{nat} \times \sigma_2)$ that sends $l \cdot \langle \rangle$ to itself, and sends $r \cdot \langle e_1, e_2 \rangle$ to $r \cdot \langle e_1, f(e_2) \rangle$. No such transformation is possible for the (non-positive) type operator $t \cdot t \rightarrow t$, because to transform $\sigma_1 \rightarrow \sigma_2$ to $\sigma_2 \rightarrow \sigma_2$ requires a transformation from $\sigma_2$ to $\sigma_1$ as well as one in the other direction.

To make these ideas precise, we define a class of positive type operators to which we associate an action on both types and terms. The positivity judgement, $\Delta \vdash t \cdot \tau \ \text{pos}$ is inductively defined by the following rules:

\[
\frac{}{\Delta \vdash t \cdot t \ \text{pos}} \quad (19.7a)
\]

\[
\frac{\Delta \vdash t \cdot u \ \text{pos} \quad u \neq t}{\Delta \vdash t \cdot u \ \text{pos}} \quad (19.7b)
\]

\[
\frac{}{\Delta \vdash t \cdot \text{unit} \ \text{pos}} \quad (19.7c)
\]

\[
\frac{\Delta \vdash t \cdot \tau_1 \ \text{pos} \quad \Delta \vdash t \cdot \tau_2 \ \text{pos}}{\Delta \vdash t \cdot \tau_1 \times \tau_2 \ \text{pos}} \quad (19.7d)
\]

\[
\frac{}{\Delta \vdash t \cdot \text{void} \ \text{pos}} \quad (19.7e)
\]

\[
\frac{\Delta \vdash t \cdot \tau_1 \ \text{pos} \quad \Delta \vdash t \cdot \tau_2 \ \text{pos}}{\Delta \vdash t \cdot \tau_1 \ + \tau_2 \ \text{pos}} \quad (19.7f)
\]
In Rule (19.7g), the type variable \( t \) is excluded from the domain of the function type by demanding that it be well-formed without regard to \( t \).

The positivity judgement is preserved under substitution.

**Lemma 19.1.** If \( t . \sigma \text{ pos} \) and \( u . \tau \text{ pos} \), then \( t . [\sigma / u] \tau \text{ pos} \).

**Proof.** By rule induction on Rules (19.7). \(\square\)

### 19.2.2 Action of a Positive Type Operator

Positive type operators admit a *covariant action*, or *map* operation, that transforms types and expressions in tandem. Specifically, if \( t . \tau \text{ pos} \), then

1. If \( \sigma \text{ type} \), then \( \text{Map}[t . \tau](\sigma) \text{ type} \).

2. If \( x : \sigma_1 \vdash e : \sigma_2 \) and \( \text{map}[t . \tau](x . e) = x' . e' \), then \( x' : \text{Map}[t . \tau](\sigma_1) \vdash e' : \text{Map}[t . \tau](\sigma_2) \).

The action on types is given by substitution:

\[
\text{Map}[t . \tau](\sigma) := [\sigma / t] \tau.
\]

The action of the type operator \( t . \tau \) on an abstraction \( x . e \) transforms an element \( e_1 \) of type \( \text{Map}[t . \tau](\sigma_1) \) into an element of \( e_2 \) of type \( \text{Map}[t . \tau](\sigma_2) \). This is achieved by replacing each sub-expression, \( d \), of \( e_1 \) corresponding to an occurrence of \( t \) in \( \tau \) by the expression \( [d / x] e_2 \).

For example, consider the type operator \( t . \text{unit} + (\text{nat} \times t) \).

The action of this operator on \( x . e \) such that

\[
x : \sigma_1 \vdash e : \sigma_2
\]

is an abstractor \( x' . e' \) of type

\[
x' : \text{unit} + (\text{nat} \times \sigma_1) \vdash e' : \text{unit} + (\text{nat} \times \sigma_2)
\]

that behaves as described above.
The action of a positive type operator on an abstraction is defined by the judgement
\[ \text{map}[t\cdot\tau](x\cdot e) = x'\cdot e', \]
which is inductively defined by the following rules:

\[ \text{map}[t\cdot t](x\cdot e) = x\cdot e \] \hspace{1cm} (19.8a)

\[ u \neq t \quad \text{map}[t\cdot u](x\cdot e) = x\cdot x \] \hspace{1cm} (19.8b)

\[ \text{map}[t\cdot \text{unit}](x\cdot e) = x'\cdot \emptyset \] \hspace{1cm} (19.8c)

\[ \text{map}[t\cdot \tau_1](x\cdot e) = x_1'\cdot e_1' \]
\[ \text{map}[t\cdot \tau_2](x\cdot e) = x_2'\cdot e_2' \] \hspace{1cm} (19.8d)

\[ \text{map}[t\cdot \tau_1 \times \tau_2](x\cdot e) = x'\cdot [x_1'\cdot 1, x_2'\cdot r/ x_1'\cdot x_2'](e_1', e_2') \]

\[ \text{map}[t\cdot \text{void}](x\cdot e) = x'\cdot \text{abort}(x') \] \hspace{1cm} (19.8e)

\[ \text{map}[t\cdot \tau_1](x_1\cdot e) = x_1'\cdot e_1' \]
\[ \text{map}[t\cdot \tau_2](x_2\cdot e) = x_2'\cdot e_2' \] \hspace{1cm} (19.8f)

\[ \text{map}[t\cdot \tau_1 + \tau_2](x\cdot e) = x'\cdot \text{case } x'\{1\cdot x_1' \Rightarrow 1\cdot e_1' | r\cdot x_2' \Rightarrow r\cdot e_2'\} \]

\[ \text{map}[t\cdot \tau_1 \rightarrow \tau_2](x\cdot e) = x'\cdot \lambda(x_1':\tau_1, [x'(x_1')/x_2']e_2') \] \hspace{1cm} (19.8g)

**Lemma 19.2.** Suppose that \( t \cdot \tau \text{ pos. If } x : \sigma \vdash e : \sigma', \text{ and } \text{map}[t\cdot \tau](x\cdot e) = x'\cdot e' \), then \( x' : \text{Map}[t\cdot \tau](\sigma) \vdash e' : \text{Map}[t\cdot \tau](\sigma') \).

**Proof.** By rule induction on Rules (19.8).

---

### 19.3 Static Semantics

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 \( L\{\mu_1\mu_2\} \), which extends \( L\{\rightarrow \times +\} \) with inductive and co-inductive types.
19.3.1 Types

The syntax of inductive and coinductive types involves type variables, which are, of course, variables ranging over the class of types. The abstract syntax of inductive and coinductive types is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau )</td>
<td>( t )</td>
<td>( t )</td>
</tr>
<tr>
<td></td>
<td>( \text{ind}(t.\tau) )</td>
<td>( \mu_1(t.\tau) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{coi}(t.\tau) )</td>
<td>( \mu_1(t.\tau) )</td>
<td></td>
</tr>
</tbody>
</table>

The subscripts on the inductive and coinductive types stand for “initial” and “final”, respectively.

Type formation judgements have the form

\[ t_1 \text{ type}, \ldots, t_n \text{ type} \vdash \tau \text{ type}, \]

where \( t_1, \ldots, t_n \) are type names. We let \( \Delta \) range over finite sets of hypotheses of the form \( t \text{ type} \), where \( t \text{ name} \) is a type name. The type formation judgement is inductively defined by the following rules:

\[
\Delta, t \text{ type} \vdash t \text{ type} \quad (19.9a)
\]

\[
\Delta \vdash \text{unit type} \quad (19.9b)
\]

\[
\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type} \quad \Delta \vdash \text{prod}(\tau_1; \tau_2) \text{ type} \quad (19.9c)
\]

\[
\Delta \vdash \text{void type} \quad (19.9d)
\]

\[
\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type} \quad \Delta \vdash \text{sum}(\tau_1; \tau_2) \text{ type} \quad (19.9e)
\]

\[
\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type} \quad \Delta \vdash \text{arr}(\tau_1; \tau_2) \text{ type} \quad (19.9f)
\]

\[
\Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta \vdash l.\tau \text{ pos} \quad \Delta \vdash \text{ind}(l.\tau) \text{ type} \quad (19.9g)
\]

\[
\Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta \vdash l.\tau \text{ pos} \quad \Delta \vdash \text{coi}(l.\tau) \text{ type} \quad (19.10)
\]
19.3.2 Expressions

The abstract syntax of expressions for inductive and coinductive types is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>e</td>
<td>$\vdash$ fold<a href="$e$">$t.\tau$</a> fold($e$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\vdash$ rec[$t.\tau$]<a href="$e'$">$x.e$</a> rec<a href="$e'$">$x.e$</a></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\vdash$ unfold<a href="$e$">$t.\tau$</a> unfold($e$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\vdash$ gen[$t.\tau$]<a href="$e'$">$x.e$</a> gen<a href="$e'$">$x.e$</a></td>
<td></td>
</tr>
</tbody>
</table>

The expression rec[$x.e$]($e'$) is called a recursor, and the expression gen[$x.e$]($e'$) is called a generator. The expression fold($e$) is called a fold operation, or constructor, and unfold($e$) is called an unfold operation, or destructor.

The static semantics for inductive and coinductive types is given by the following typing rules:

\[
\begin{align*}
\Gamma &\vdash e : [\text{ind}(t.\tau)/t]\tau \\
\Gamma &\vdash \text{fold}[t.\tau](e) : \text{ind}(t.\tau) \\
\Gamma &\vdash e' : \text{ind}(t.\tau) \quad \Gamma, x : [\rho/t]\tau \vdash e : \rho \\
\Gamma &\vdash \text{rec}[t.\tau][x.e](e') : \rho \\
\Gamma &\vdash \text{unfold}[t.\tau](e) : [\text{coi}(t.\tau)/t]\tau \\
\Gamma &\vdash e' : \rho \quad \Gamma, x : \rho \vdash e : [\rho/t]\tau \\
\Gamma &\vdash \text{gen}[t.\tau][x.e](e') : \text{coi}(t.\tau) \\
\end{align*}
\]

The dynamic semantics of these constructs is given in terms of the action of a positive type operator, which we now define.

19.4 Dynamic Semantics

The dynamic semantics of inductive and coinductive types is given in terms of the covariant action of the associated type operator. The following rules specify a lazy dynamics for $\mathcal{L}\{\mu,\nu\}$:

\[
\begin{align*}
\text{fold}(e) &\rightarrow \text{val} \\
\end{align*}
\]

\[
\begin{align*}
e' &\rightarrow e'' \\
\text{rec}[x.e](e') &\rightarrow \text{rec}[x.e](e'') \\
\end{align*}
\]
\[
\text{map}[t \cdot \tau](x' \cdot \text{rec}[x \cdot e](x')) = x'' \cdot e''
\]
(19.12c)
\[
\text{rec}[x \cdot e](\text{fold}(e')) \mapsto [e'/x''|e''/x|e]
\]
\[
\text{gen}[x \cdot e](e') \mapsto \text{val}
\]
(19.12d)
\[
e \mapsto e'
\]
\[
\text{unfold}(e) \mapsto \text{unfold}(e')
\]
(19.12e)
\[
\text{map}[t \cdot \tau](x' \cdot \text{gen}[x \cdot e](x')) = x'' \cdot e''
\]
(19.12f)

Rule (19.12c) 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 (19.12f) is simply the dual of this rule for coinductive types.

**Lemma 19.3.** If \( e : \tau \) and \( e \mapsto e' \), then \( e' : \tau \).

*Proof.* By rule induction on Rules (19.12). \( \square \)

**Lemma 19.4.** If \( e : \tau \), then either \( e \text{ val} \) or there exists \( e' \) such that \( e \mapsto e' \).

*Proof.* By rule induction on Rules (19.11). \( \square \)

Although we shall not give the proof here, the language \( L\{\mu_i \mu_f\} \) is terminating, and all functions defined within it are total.

**Theorem 19.5.** If \( e : \tau \) in \( L\{\mu_i \mu_f\} \), then there exists \( e' \text{ val} \) such that \( e \mapsto^* e' \).

### 19.5 Exercises
Chapter 20

General Recursive Types

Inductive and coinductive types may be seen as initial and final solutions to certain forms of recursive type equations. Both the inductive type, \( \mu_i(t.\tau) \), and the coinductive type, \( \mu_f(t.\tau) \), are fixed points of the type operator \( t.\tau \). Thus both are solutions to the recursion equation \( t \cong \tau \) “up to isomorphism” in that both

\[ \mu_i(t.\tau) \cong [\mu_i(t.\tau) / t]\tau \]

and

\[ \mu_f(t.\tau) \cong [\mu_f(t.\tau) / t]\tau. \]

However, inductive and coinductive types provide solutions to type isomorphisms only for positive type operators. In many situations this restriction cannot be met. For example, to model self-reference we require a solution to the type isomorphism \( t \cong t \rightarrow \sigma \) for which the associated type operator \( t.\sigma \) is not positive.

In this chapter we study the language \( \mathcal{L}\{\mu\} \), which provides solutions to general type isomorphism equations, without positivity restrictions. The (general) 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

\[ x : \mu t.\tau \vdash \text{unfold}(x) : [\mu t.\tau / t]\tau \]

and

\[ x : [\mu t.\tau / t]\tau \vdash \text{fold}(x) : \mu t.\tau, \]

which are mutually inverse to each other.
Postulating solutions to arbitrary type isomorphism equations may seem suspicious, since we know by Cantor’s Theorem that isomorphisms such as \( X \cong \wp(X) \) do not exist, provided that we interpret types as sets and \( \wp(X) \) as the set of all subsets of \( X \). But rather than presenting a paradox, this observation simply means that types cannot be naively interpreted as sets of values. If we interpret types as classifying potentially undefined computations, rather than as fixed collections of well-defined values, then the proof of Cantor’s Theorem breaks down. Somewhat counterintuitively, the failure of Cantor’s Theorem is precisely what makes type theory so powerful. In particular, we may solve a rich variety of type isomorphisms that are impossible to solve in a set-theoretic setting.

### 20.1 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, \( \text{fold}(e) \) and \( \text{unfold}(e) \), 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 \( \mathcal{L}\{\mu\} \) extends \( \mathcal{L}\{\to\} \) with recursive types and their associated operations.

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau )</td>
<td>( t )</td>
<td>( t )</td>
</tr>
<tr>
<td></td>
<td>( \mid \text{rec}(t. \tau) )</td>
<td>( \mu t. \tau )</td>
<td></td>
</tr>
<tr>
<td>Expr</td>
<td>( e )</td>
<td>( \text{fold}<a href="e">t. \tau</a> )</td>
<td>( \text{fold}(e) )</td>
</tr>
<tr>
<td></td>
<td>( \mid \text{unfold}(e) )</td>
<td>( \text{unfold}(e) )</td>
<td></td>
</tr>
</tbody>
</table>

The expression \( \text{fold}(e) \) is the introductory form for the recursive type, and \( \text{unfold}(e) \) is its eliminatory form.

The static semantics of \( \mathcal{L}\{\mu\} \) consists of two forms of judgement. The first, called type formation, is a general hypothetical judgement of the form

\[ T \mid \Delta \vdash \tau \text{ type}, \]

where \( T = \{ t_1, \ldots, t_k \} \) and \( \Delta \) is \( t_1 \) type, \ldots, \( t_k \) type. As usual we drop explicit mention of \( T \), relying on typographical conventions to make clear which are the type variables of the judgement.

Type formation is inductively defined by the following rules:

\[ \Delta, t \text{ type} \vdash t \text{ type} \quad \text{(20.1a)} \]
The second form of judgement comprising the static semantics is the 
\textit{typing judgement}, which is a general hypothetical judgement of the form
\[ \mathcal{X} \mid \Gamma \vdash e : \tau, \]
where we assume that \( \tau \) type. The parameter set, \( \mathcal{X} \), is a finite set of variables, each of which is governed by a typing hypothesis in \( \Gamma \). We ordinarily suppress the parameter set, \( \mathcal{X} \), in favor of relying on the form of \( \Gamma \) to make clear what is intended.

Typing for \( \mathcal{L}\{\mu\} \) is inductively defined by the following rules:
\[ \Gamma \vdash e : [\text{rec}(t.\tau)/t]_\tau \]
\[ \Gamma \vdash \text{fold}[t.\tau](e) : \text{rec}(t.\tau) \quad (20.2a) \]
\[ \Gamma \vdash e : \text{rec}(t.\tau) \]
\[ \Gamma \vdash \text{unfold}(e) : [\text{rec}(t.\tau)/t]_\tau \quad (20.2b) \]

The dynamic semantics of \( \mathcal{L}\{\mu\} \) is specified by one axiom stating that the elimination form is inverse to the introduction form, together with rules specifying the order of evaluation (eager or lazy, according to whether the bracketed rules and premises are included or omitted):
\[ \{e \text{ val}\} \]
\[ \text{fold}[t.\tau](e) \text{ val} \quad (20.3a) \]
\[ \left\{ e \mapsto e' \quad \text{fold}[t.\tau](e) \mapsto \text{fold}[t.\tau](e') \right\} \quad (20.3b) \]
\[ e \mapsto e' \quad \text{unfold}(e) \mapsto \text{unfold}(e') \quad (20.3c) \]
\[ \text{fold}[t.\tau](e) \text{ val} \quad \text{unfold}(\text{fold}[t.\tau](e)) \mapsto e \quad (20.3d) \]

Definitional equivalence for \( \mathcal{L}\{\mu\} \) is the least congruence containing the following rule:
\[ \Gamma \vdash \text{unfold}(\text{fold}[t.\tau](e)) \equiv e : [\text{rec}(t.\tau)/t]_\tau \quad (20.4) \]

It is a straightforward exercise to prove type safety for \( \mathcal{L}\{\mu\} \).
Theorem 20.1 (Safety). 1. If \( e : \tau \) and \( e \to e' \), then \( e' : \tau \).

2. If \( e : \tau \), then either \( e \) \text{val}, or there exists \( e' \) such that \( e \to e' \).

20.2 Recursive Data Structures

One important application of recursive types is to the representation of data structures such as lists and trees whose size and content is determined during the course of execution of a program.

One example is the type of natural numbers, which we have taken as primitive in Chapter 15. We may instead treat \( \text{nat} \) as a recursive type by thinking of it as a solution (up to isomorphism) of the type equation \( t \cong 1 + t \), which is to say that every natural number is either zero or the successor of another natural number. More formally, we may define \( \text{nat} \) to be the recursive type

\[
\mu t. [z : \text{unit}, s : t], \tag{20.5}
\]

which specifies that

\( \text{nat} \cong [z : \text{unit}, s : \text{nat}] \).

The zero and successor operations are correspondingly defined by the following equations:

\[
\begin{align*}
z &= \text{fold}(z \cdot \langle \rangle) \\
s(e) &= \text{fold}(s \cdot e).
\end{align*}
\]

The conditional branch on zero is defined by the following equation:

\[
\text{ifz } e \{ z \Rightarrow e_0 \mid s(x) \Rightarrow e_1 \} = \\
\text{case unfold}(e) \{ z \Rightarrow e_0 \mid s \cdot x \Rightarrow e_1 \},
\]

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 in that they correctly simulate the dynamic semantics given in Chapter 15.

As another example, the type \( \text{nat list} \) of lists of natural numbers may be represented by the recursive type

\[
\mu t. [n : \text{unit}, c : \text{nat} \times t]
\]

so that we have the isomorphism

\( \text{nat list} \cong [n : \text{unit}, c : \text{nat} \times \text{nat list}] \).
The list formation operations are represented by the following equations:

\[
\text{nil} = \text{fold}(n \cdot \langle \rangle) \\
\text{cons}(e_1; e_2) = \text{fold}(c \cdot \langle e_1, e_2 \rangle) .
\]

A conditional branch on the form of the list may be defined by the following equation:

\[
\text{listcase } e \{ \text{nil} \Rightarrow e_0 \mid \text{cons}(x; y) \Rightarrow e_1 \} = \\
\text{case unfold}(e) \{ n \cdot \_ \Rightarrow e_0, \mid \text{c} \cdot \langle x, y \rangle \Rightarrow e_1 \},
\]

where we have used an underscore for a “don’t care” variable, and used pattern-matching syntax to bind the components of a pair.

There is a natural correspondence between this representation of lists and the conventional “blackboard notation” for linked lists. We may think of \text{fold} as an abstract heap-allocated pointer to a tagged cell consisting of either (a) the tag \text{n} with no associated data, or (b) the tag \text{c} attached to a pair consisting of a natural number and another list, which must be an abstract pointer of the same sort.

20.3 Self-Reference

In the general recursive expression, \text{fix}\[\tau\](x.e), the variable, x, stands for the expression itself. This is ensured by the unrolling transition

\[
\text{fix}\[\tau\](x.e) \mapsto \text{[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 \text{this} or \text{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 \tau, written \text{self}(\tau). The introduction form of this type is (a variant of) general recursion, written \text{self}\[\tau\](x.e), and the elimination form is an operation to unroll the recursion by one step,
written \texttt{unroll}(e). The static semantics of these constructs is given by the following rules:

\[
\begin{align*}
\Gamma, x : \text{self}(\tau) & \vdash e : \tau \quad (20.6a) \\
\Gamma & \vdash \text{self}[^\tau](x.e) : \text{self}(\tau) \\
\Gamma & \vdash e : \text{self}(\tau) \\
\Gamma & \vdash \text{unroll}(e) : \tau \\
\end{align*}
\]

The dynamic semantics is given by the following rule for unrolling the self-reference:

\[
\begin{align*}
\text{self}[^\tau](x.e) \mapsto \text{val} \quad (20.7a) \\
\text{unroll}(e) & \mapsto \text{unroll}(e') \\
\text{unroll}(\text{self}[^\tau](x.e)) & \mapsto [\text{self}[^\tau](x.e)/x]e \\
\end{align*}
\]

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 \text{self}(\tau) is definable from recursive types. As suggested earlier, the key is to consider a self-referential expression of type \tau to be a function of the expression itself. That is, we seek to define the type \text{self}(\tau) so that it satisfies the isomorphism

\[
\text{self}(\tau) \cong \text{self}(\tau) \rightarrow \tau.
\]

This means that we seek a fixed point of the type operator \texttt{t.t \rightarrow t}, where \texttt{t \notin t} is a type variable standing for the type in question. The required fixed point is just the recursive type

\[
\text{rec}(t.t \rightarrow \tau),
\]

which we take as the definition of \text{self}(\tau).

The self-referential expression \text{self}[^\tau](x.e) is then defined to be the expression

\[
\text{fold}(\lambda(x: \tau \text{self}.e)).
\]

We may easily check that Rule (20.6a) is derivable according to this definition. The expression \text{unroll}(e) is correspondingly defined to be the expression

\[
\text{unfold}(e)(e).
\]
It is easy to check that Rule (20.6b) is derivable from this definition. Moreover, we may check that the definitional equivalence

\[ \text{unroll} (\text{self} \ y \ is \ e) \equiv [\text{self} \ y \ is \ e] / y e \]

also holds by expanding the definitions and applying the rules of definitional equivalence for recursive types.

This completes the derivation of the type \( \text{self}(\tau) \) of self-referential expressions of type \( \tau \). Using this type we may define general recursion at any type \( \tau \) by simply inserting unrolling operations that are implicit in the semantics of general recursion. Specifically, we may define \( \text{fix} \ x : \tau \ is \ e \) to be the expression

\[ \text{unroll} (\text{self} \ y \ is \ [\text{unroll} (y) / x] e) \]

It is easy to check that this verifies the static semantics of general recursion given in Chapter 15. Moreover, it also validates the dynamic semantics, as evidenced by the following derivation:

\[
\text{fix} \ x : \tau \ is \ e = \text{unroll} (\text{self} \ y \ is \ [\text{unroll} (y) / x] e) \\
\equiv [\text{unroll} (\text{self} \ y \ is \ [\text{unroll} (y) / x] e) / x] e \\
= [\text{fix} \ x : \tau \ is \ e / x] e.
\]

By replacing \( x \) in \( e \) by \( \text{unroll} (e) \), and wrapping the entire self-referential expression similarly, we ensure that the self-reference is unrolled implicitly as in Chapter 15, rather than explicitly, as here.

One consequence of this derivation is that adding recursive types to a programming language is a non-conservative extension. For suppose that we add recursive types to a terminating language such as \( \mathcal{L}\{\text{nat} \to \} \) defined in Chapter 14. The foregoing argument shows that general recursion is definable in this extension, and hence that the termination property of the language has been destroyed. This is in contrast to extensions with, say, product and sum types, which do not disrupt the termination properties of the language. In short, adding new language features (new forms of type) can have subtle, and often surprising, consequences!

### 20.4 Exercises
Part VII

Dynamic Types
Chapter 21

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!

21.1 The λ-Calculus

The abstract syntax of $\mathcal{L}\{\lambda\}$ is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term</td>
<td>$u$</td>
<td>$x$</td>
<td>$x$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\lambda(x.,u)$</td>
<td>$\lambda x.,u$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{ap}(u_1;,u_2)$</td>
<td>$u_1(u_2)$</td>
</tr>
</tbody>
</table>

¹An apt description suggested by Dana Scott.
The second form of expression is called a $\lambda$-abstraction, and the third is called application.

The static semantics of $\mathcal{L}\{\lambda\}$ is defined by general hypothetical judgements of the form $x_1, \ldots, x_n \vdash u \ ok$, stating that $u$ is a well-formed expression involving the variables $x_1, \ldots, x_n$. (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:

$$\Gamma, x \ ok \vdash x \ ok \quad (21.1a)$$

$$\Gamma \vdash u_1 \ ok \quad \Gamma \vdash u_2 \ ok \quad \Gamma \vdash \text{ap}(u_1; u_2) \ ok \quad (21.1b)$$

$$\Gamma, x \ ok \vdash u \ ok \quad \Gamma \vdash \lambda(x. u) \ ok \quad (21.1c)$$

The dynamic semantics is given by the following rules:

$$\lambda(x. u) \ val \quad (21.2a)$$

$$\text{ap}(\lambda(x. u_1); u_2) \mapsto [u_2/x]u_1 \quad (21.2b)$$

$$u_1 \mapsto u'_1 \quad \text{ap}(u_1; u_2) \mapsto \text{ap}(u'_1; u_2) \quad (21.2c)$$

In the $\lambda$-calculus literature this judgement is called weak head reduction. The first rule is called $\beta$-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 21.1.** 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 dynamic semantics. Since every closed value of $\mathcal{L}\{\lambda\}$ is a $\lambda$-abstraction, every closed expression is either a value or can make progress. \qed

Definitional equivalence for $\mathcal{L}\{\lambda\}$ is a judgement of the form $\Gamma \vdash u \equiv u'$, where $\Gamma = x_1 \ ok, \ldots, x_n \ ok$ for some $n \geq 0$, and $e$ and $e'$ are terms.
21.2 Definability

having at most the variables \(x_1, \ldots, x_n\) free. It is inductively defined by the following rules:

\[
\begin{align*}
\Gamma, u \text{ ok} & \vdash u \equiv u \\
\Gamma & \vdash u \equiv u' \\
\Gamma & \vdash u' \equiv u \\
\Gamma & \vdash u \equiv u' \quad \Gamma & \vdash u' \equiv u'' \\
\Gamma & \vdash u \equiv u'' \\
\Gamma & \vdash e_1 \equiv e_1' \\
\Gamma & \vdash e_2 \equiv e_2' \\
\Gamma & \vdash \text{ap}(e_1; e_2) \equiv \text{ap}(e_1'; e_2') \\
\Gamma, x \text{ ok} & \vdash u \equiv u' \\
\Gamma & \vdash \lambda(x. u) \equiv \lambda(x. u') \\
\Gamma & \vdash \text{ap}(\lambda(x. u); e_1) \equiv \left[ e_1/x \right]e_2
\end{align*}
\]

(21.3a)
(21.3b)
(21.3c)
(21.3d)
(21.3e)
(21.3f)

We often write just \(u \equiv u'\) when the variables involved need not be emphasized or are clear from context.

21.2 Definability

Interest in the untyped \(\lambda\)-calculus stems from its surprising expressive power: 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 \(\lambda\)-calculus. This is certainly true for all known means of defining computable functions on the natural numbers. The force of Church’s Law is that it postulates that all future notions of computation will be equivalent in expressive power (measured by definability of functions on the natural numbers) to the \(\lambda\)-calculus. Church’s Law is therefore a scientific law in the same sense as, say, Newton’s Law of Universal Gravitation, which makes a prediction about all future measurements of the acceleration due to the gravitational field of a massive object.\(^2\)

\(^2\)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 true 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.
21.2 Definability

We will sketch a proof that the untyped \( \lambda \)-calculus is as powerful as the language PCF described in Chapter 15. The main idea is to show that the PCF primitives for manipulating the natural numbers are definable in the untyped \( \lambda \)-calculus. This means, in particular, that we must show that the natural numbers are definable as \( \lambda \)-terms in such a way that case analysis, which discriminates between zero and non-zero numbers, is definable. The principal difficulty is with computing the predecessor of a number, which requires a bit of cleverness. Finally, we show how to represent general recursion, completing the proof.

The first task is to represent the natural numbers as certain \( \lambda \)-terms, called the Church numerals.

\[
\bar{0} = \lambda b. \lambda s. b \quad (21.4a)
\]
\[
\bar{n+1} = \lambda b. \lambda s. s(\bar{n}(b)(s)) \quad (21.4b)
\]

It follows that

\[
\bar{n}(u_1)(u_2) \equiv u_2(\ldots(u_2(u_1))) ,
\]

the \( n \)-fold application of \( u_2 \) to \( u_1 \). That is, \( \bar{n} \) iterates its second argument (the induction step) \( n \) times, starting with its first argument (the basis).

Using this definition it is not difficult to define the basic functions of arithmetic. For example, successor, addition, and multiplication are defined by the following untyped \( \lambda \)-terms:

\[
succ = \lambda x. \lambda b. \lambda s. x(b)(s) \quad (21.5)
\]
\[
plus = \lambda x. \lambda y. y(x)(\text{succ}) \quad (21.6)
\]
\[
times = \lambda x. \lambda y. y(\bar{0})(\text{plus}(x)) \quad (21.7)
\]

It is easy to check that \( \text{succ}(\bar{n}) \equiv \bar{n+1} \), and that similar correctness conditions hold for the representations of addition and multiplication.

We may readily define \( \text{if}z(u; u_0; u_1) \) to be the application \( u(u_0)(\lambda _{\ldots}u_1) \), where the underscore stands for a dummy variable chosen apart from \( u_1 \). We can use this to define \( \text{if}z(u; u_0; x.u_1) \), provided that we can compute the predecessor of a natural number. Doing so requires a bit of ingenuity. We wish to find a term \( \text{pred} \) such that

\[
\text{pred}(\bar{0}) \equiv \bar{0} \quad (21.8)
\]
\[
\text{pred}(\bar{n+1}) \equiv \bar{n} . \quad (21.9)
\]

To compute the predecessor using Church numerals, we must show how to compute the result for \( \bar{n+1} \) as a function of its value for \( \bar{n} \). At first glance
21.2 Definability

this seems straightforward—just take the successor—until we consider the base case, in which we define the predecessor of 0 to be 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.

\[
\langle u_1, u_2 \rangle = \lambda f. f(u_1)(u_2) \\
u \cdot 1 = u(\lambda x. \lambda y. x) \\
u \cdot x = u(\lambda x. \lambda y. y)
\]

It is easy to check that under this encoding \(\langle u_1, u_2 \rangle \cdot 1 \equiv u_1\), and similarly for the second projection. We may now define the required term u representing the predecessor:

\[
u'_p = \lambda x. x(\langle 0, 0 \rangle) (\lambda y. \langle y \cdot x, s(y \cdot x) \rangle) \\
u_p = \lambda x. u(x) \cdot 1
\]

It is then easy to check that this gives us the required behavior. Finally, we may define \(\text{ifz}(u; u_0; x) u_1\) to be the untyped term

\[u(u_0)(\lambda x. [u_p(u)/x]u_1).\]

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 Y combinator:

\[
Y = \lambda F. (\lambda f. F(f(f))) (\lambda f. F(f(f))).
\]

Observe that

\[Y(F) \equiv F(Y(F)).\]

Using the Y combinator, we may define general recursion by writing \(Y(\lambda x. u)\), where x stands for the recursive expression itself.
21.3 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 \( u(\uparrow v) \equiv v \), where \( \uparrow v \) is the Gödel number of \( v \), and \( \uparrow v \) is its representation as a Church numeral. (See Chapter 14 for a discussion of Gödel-numbering.)

2. Any two non-trivial\(^3\) properties \( A_0 \) and \( A_1 \) of untyped terms that respect definitional equivalence are inseparable. This means that there is no decidable property \( B \) of untyped terms such that \( A_0 \ u \) implies that \( B \ u \) and \( A_1 \ u \) implies that it is not the case that \( B \ u \). In particular, if \( A_0 \) and \( A_1 \) are inseparable, then neither is decidable.

For a property \( B \) of untyped terms to respect definitional equivalence means that if \( B \ u \) and \( u \equiv u' \), then \( B \ u' \).

**Lemma 21.2.** For any \( u \) there exists \( v \) such that \( u(\uparrow v) \equiv v \).

*Proof Sketch.* The proof relies on the definability of the following two operations in the untyped λ-calculus:

1. \( \text{ap}(\uparrow u_1)(\uparrow u_2) \equiv \uparrow u_1(u_2) \).
2. \( \text{nm}(\bar{n}) \equiv \uparrow \bar{n} \).

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 \( \bar{n} \). Given these, we may find the required term \( v \) by defining \( v = w(\uparrow \bar{w}) \), where \( w = \lambda x. u(\text{ap}(x)(\text{nm}(x))) \). We have

\[
\begin{align*}
v &= w(\uparrow \bar{w}) \\
&\equiv u(\text{ap}(\uparrow \bar{w}))(\text{nm}(\uparrow \bar{w})) \\
&\equiv u(\uparrow w(\uparrow \bar{w})) \\
&\equiv u(\uparrow \bar{v}).
\end{align*}
\]

\(^3\)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.
21.4 Untyped Means Uni-Typed

The definition is very similar to that of $Y(u)$, 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 21.3.** Suppose that $A_0$ and $A_1$ 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 $w(⌜u⌝) ≡ 0$ or $w(⌜u⌝) ≡ 1$.
2. If $A_0 u$, then $w(⌜u⌝) ≡ 0$.
3. If $A_1 u$, then $w(⌜u⌝) ≡ 1$.

**Proof.** Suppose there is such an untyped term $w$. Let $v$ be the untyped term
$$\lambda x. ifz(w(x); u_1; u_0),$$
where $A_0 u_0$ and $A_1 u_1$. By Lemma 21.2 on the fac-
ing page there is an untyped term $t$ such that $v(⌜T⌝) ≡ t$. If $w(⌜T⌝) ≡ 0$, then $t ≡ v(⌜T⌝) ≡ u_1$, and so $A_1 t$, since $A_1$ respects definitional equiv-
alancl and $A_1 u_1$. But then $w(⌜T⌝) ≡ T$ by the defining properties of $w$, which is a contradiction. Similarly, if $w(⌜T⌝) ≡ 1$, then $A_0 t$, and hence
$w(⌜T⌝) ≡ 0$, again a contradiction.

**Corollary 21.4.** There is no algorithm to decide whether or not $u ≡ u'$.

**Proof.** For fixed $u$ consider the property $E_u u'$ defined by $u' ≡ u$. This is
non-vacuous and respects definitional equivalence, and hence is undecid-
able.

21.4 Untyped Means Uni-Typed

The untyped $\lambda$-calculus may be faithfully embedded in the typed language
$L\{\mu\}$, enriched with recursive types. This means that every untyped $\lambda$
term has a representation as an expression in $L\{\mu\}$ in such a way that ex-
ecution of the representation of a $\lambda$-term corresponds to execution of the
term itself. If the execution model of the $\lambda$-calculus is call-by-name, this
correspondence holds for the call-by-name variant of $L\{\mu\}$, and similarly
for call-by-value.

It is important to understand that this form of embedding is not a mat-
ter of writing an interpreter for the $\lambda$-calculus in $L\{\mu\}$ (which we could
surely do), but rather a direct representation of untyped $\lambda$-terms as certain typed expressions of $\mathcal{L}\{\mu\}$. 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 $\lambda$-calculus is really the uni-typed $\lambda$-calculus! It is not the absence of types that gives it its power, but rather that it has only one type, namely the recursive type

$$D = \mu t. t \to t.$$  

A value of type $D$ is of the form $\text{fold}(e)$ where $e$ is a value of type $D \to D$ — 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 $\lambda$-calculus.

This isomorphism leads to the following embedding, $u^\dagger$, of $u$ into $\mathcal{L}\{\mu\}$:

\begin{align*}
  x^\dagger &= x & (21.16a) \\
  \lambda x. u^\dagger &= \text{fold}(\lambda (x:D). u^\dagger)) & (21.16b) \\
  u_1(u_2)^\dagger &= \text{unfold}(u_1^\dagger)(u_2^\dagger) & (21.16c)
\end{align*}

Observe that the embedding of a $\lambda$-abstraction is a value, and that the embedding of an application exposes the function being applied by unrolling the recursive type. Consequently,

\begin{align*}
  \lambda x. u_1(u_2)^\dagger &= \text{unfold}(\text{fold}(\lambda (x:D). u_1^\dagger))) (u_2^\dagger) \\
  &= \lambda (x:D). u_1^\dagger) (u_2^\dagger) \\
  &= ([u_2^\dagger/x]u_1^\dagger) \\
  &= ([u_2/x]u_1)^\dagger.
\end{align*}

The last step, stating that the embedding commutes with substitution, is easily proved by induction on the structure of $u_1$. Thus $\beta$-reduction is faithfully implemented by evaluation of the embedded terms.
Thus we see that the canonical untyped language, $\mathcal{L}\{\lambda\}$, 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 22 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.

21.5 Exercises
Chapter 22

Dynamic Typing

We saw in Chapter 21 that an untyped language may be viewed as a unityped language in which the so-called untyped terms are terms of a distinguished recursive type. In the case of the untyped $\lambda$-calculus this recursive type has a particularly simple form, expressing that every term is isomorphic to a function. Consequently, no run-time errors can occur due to the misuse of a value—the only elimination form is application, and its first argument can only be a function. 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 $\lambda$-calculus (rather than defining them via Church encodings), then it is possible to incur a run-time error arising from attempting to apply a number to an argument, or to add a function to a number.

One school of thought in language design is to turn this vice into a virtue by embracing a model of computation that has multiple classes of value of a single type. Such languages are said to be dynamically typed, in supposed opposition to the statically typed languages we have studied thus far. In this chapter we show that the supposed opposition between static and dynamic languages is fallacious: dynamic typing is but a mode of use of static typing, and, moreover, it is profitably seen as such. Dynamic typing can hardly be in opposition to that of which it is a special case!

22.1 Dynamically Typed PCF

To illustrate dynamic typing we formulate a dynamically typed version of $L\{\text{nat} \to\}$, called $L\{\text{dyn}\}$. The abstract syntax of $L\{\text{dyn}\}$ is given by the
The following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>d</td>
<td>::= x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>num((\pi))</td>
<td>(\pi)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zero</td>
<td>zero</td>
</tr>
<tr>
<td></td>
<td></td>
<td>succ(d)</td>
<td>succ(d)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ifz(d; d_0; x . d_1)</td>
<td>ifzd {zero (\Rightarrow) d_0 \mid succ(x) (\Rightarrow) d_1}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fun((\lambda)(x . d))</td>
<td>(\lambda) x . d</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dap(d_1; d_2)</td>
<td>d_1(d_2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fix(x . d)</td>
<td>fix x is d</td>
</tr>
</tbody>
</table>

There are two classes of values in \(L\{\text{dyn}\}\), the numbers, which have the form \(\pi\), and the functions, which have the form \(\lambda\) x . d. The elimination forms of \(L\{\text{dyn}\}\) operate on classified values, and must check that their arguments are of the appropriate class at run-time. The expressions zero and succ(d) are not in themselves values, but rather are operations that evaluate to classified values, as we shall see shortly.

The concrete syntax of \(L\{\text{dyn}\}\) is somewhat deceptive, in keeping with common practice in dynamic languages. For example, the concrete syntax for a number is a bare numeral, \(\pi\), but in fact it is just a convenient notation for the classified value, \(\pi\), of class num. Similarly, the concrete syntax for a function is a bare \(\lambda\)-abstraction, \(\lambda\) x . d, which must be regarded as standing for the classified value \(\lambda\) x . d of class fun. It is the responsibility of the parser to translate the surface syntax into the abstract syntax, adding class information to values in the process.

The static semantics of \(L\{\text{dyn}\}\) is essentially the same as that of \(L\{\lambda\}\) given in Chapter 21; it merely checks that there are no free variables in the expression. The judgement

\[
x_1 \text{ ok}, \ldots x_n \text{ ok} \vdash d \text{ ok}
\]

states that \(d\) is a well-formed expression with free variables among those in the hypothesis list.

The dynamic semantics for \(L\{\text{dyn}\}\) 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

\[1\]The numerals, \(\pi\), are \(n\)-fold compositions of the form \(s(s(\ldots s(z) \ldots))\).
having classes labelling values is precisely to make this run-time check possible. One could argue that the required check may be made by inspection of the unlabelled value itself, but this is unrealistic. At run-time both numbers and functions might be represented by machine words, the former a two’s complement number, the latter an address in memory. But given an arbitrary word, one cannot determine whether it is a number or an address!

The value judgement, $d \text{ val}$, states that $d$ is a fully evaluated (closed) expression:

\[
\frac{}{\text{num}(n) \text{ val}} \quad (22.1a)
\]

\[
\frac{}{\text{fun}(\lambda (x \cdot d) \text{ val}} \quad (22.1b)
\]

The dynamic semantics makes use of judgements that check the class of a value, and recover the underlying $\lambda$-abstraction in the case of a function.

\[
\frac{}{\text{num}(n) \text{ is_num } n} \quad (22.2a)
\]

\[
\frac{}{\text{fun}(\lambda (x \cdot d) \text{ is_fun } \lambda (x \cdot d)} \quad (22.2b)
\]

The second argument of each of these judgements has a special status—it is not an expression of $L\{\text{dyn}\}$, 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.

\[
\frac{}{\text{num}(\_ ) \text{ isnt_fun}} \quad (22.3a)
\]

\[
\frac{}{\text{fun}(\_ ) \text{ isnt_num}} \quad (22.3b)
\]

The transition judgement, $d \mapsto d'$, and the error judgement, $d \text{ err}$, are defined simultaneously by the following rules.

\[
\frac{}{\text{zero} \mapsto \text{num}(z)} \quad (22.4a)
\]

\[
\frac{}{d \mapsto d' \quad \text{succ}(d) \mapsto \text{succ}(d')} \quad (22.4b)
\]

\[
\frac{d \text{ is_num } n}{\text{succ}(d) \mapsto \text{num}(s(n))} \quad (22.4c)
\]

\[
\frac{d \text{ isnt_num}}{\text{succ}(d) \text{ err}} \quad (22.4d)
\]
22.1 Dynamically Typed PCF

\[ d \mapsto d' \]
\[ \text{ifz}(d; d_0; x \cdot d_1) \mapsto \text{ifz}(d'; d_0; x \cdot d_1) \]  \hspace{1cm} (22.4e)

\[ \text{ifz}(d; d_0; x \cdot d_1) \mapsto d_0 \]  \hspace{1cm} (22.4f)

\[ \text{ifz}(d; d_0; x \cdot d_1) \mapsto \left[ \text{num}(\pi)/x \right] d_1 \]  \hspace{1cm} (22.4g)

\[ \text{ifz}(d; d_0; x \cdot d_1) \mapsto \text{err} \]  \hspace{1cm} (22.4h)

\[ d_1 \mapsto d'_1 \]
\[ \text{dap}(d_1; d_2) \mapsto \text{dap}(d'_1; d_2) \]  \hspace{1cm} (22.4i)

\[ \text{dap}(d_1; d_2) \mapsto [d_2/x] d \]  \hspace{1cm} (22.4j)

\[ \text{dap}(d_1; d_2) \mapsto \text{err} \]  \hspace{1cm} (22.4k)

\[ \text{fix}(x \cdot d) \mapsto [\text{fix}(x \cdot d)/x] d \]  \hspace{1cm} (22.4l)

Rule (22.4g) labels the predecessor with the class \text{num} to maintain the invariant that variables are bound to expressions of \( \mathcal{L}\{\text{dyn}\} \).

The language \( \mathcal{L}\{\text{dyn}\} \) enjoys essentially the same safety properties as \( \mathcal{L}\{\text{nat} \rightarrow \} \), except that there are more opportunities for errors to arise at run-time.

**Theorem 22.1.** If \( d \) \text{ ok}, then either \( d \) \text{ val}, or \( d \) \text{ err}, or there exists \( d' \) such that \( d \mapsto d' \).

**Proof.** By rule induction on Rules (22.4). The rules are designed so that if \( d \) \text{ 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.

\[ \Box \]
22.2 Critique of Dynamic Typing

The safety of $\mathcal{L}\{\text{dyn}\}$ is often promoted as an advantage of dynamic over static typing. Unlike static languages, essentially every piece of abstract syntax has a well-defined dynamic semantics. 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 $\mathcal{L}\{\text{dyn}\}$. To make this possible, the dynamic semantics of $\mathcal{L}\{\text{dyn}\}$ incurs significant overhead to enforce the classification of values.

Consider, for example, the addition function in $\mathcal{L}\{\text{dyn}\}$, whose specification is that, when passed two values of class $\text{num}$, returns their sum, which is also of class $\text{num}$:

$$\text{fun}(\lambda(x.\text{fix}\ p\ is\ fun(\lambda(y.\text{ifz}(y;x;y'.\text{succ}(\text{dap}(p;y'))))))).$$

The addition function may, deceptively, be written in concrete syntax as follows:

$$\lambda x.\text{fix}\ p\ is\ \lambda y.\text{ifz}\ y\{\text{zero} \Rightarrow x | \text{succ}(y') \Rightarrow \text{succ}(p(y'))\}.$$

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 $\text{fun}$. The semantics of the fixed point construct binds $p$ to this function. This means that the dynamic class check incurred by the application of $p$ in the recursive call is guaranteed to succeed. But $\mathcal{L}\{\text{dyn}\}$ offers no means of suppressing this redundant check, because it cannot express the invariant that $p$ is always bound to a value of class $\text{fun}$.

Second, observe that the result of applying the inner $\lambda$-abstraction is either $x$, the argument of the outer $\lambda$-abstraction, or the successor of a recursive call to the function itself. The successor operation checks that its argument is of class $\text{num}$, 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 $\text{num}$ once, and observe that it is otherwise a loop invariant that the result of applying the inner function is of this class. However, $\mathcal{L}\{\text{dyn}\}$ gives us no way to express this invariant; the

---

2This 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 $\text{num}$. 

---
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 \( \text{num} \), then the semantics of the conditional will ensure that all recursive calls have this class. And again there is no way to express this invariant in \( L\{\text{dyn}\} \), 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 \( L\{\text{dyn}\} \), because it cannot enforce the restrictions required to express the required invariants. For that we need a static type system!

### 22.3 Hybrid Typing

Consider the language \( L\{\text{nat dyn} \rightarrow \} \), which extends \( L\{\text{nat} \rightarrow \} \) (defined in Chapter 15) with the following additional constructs:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau )</td>
<td>( \text{dyn} )</td>
<td>( \text{dyn} )</td>
</tr>
<tr>
<td>Expr</td>
<td>( e )</td>
<td>( \text{new}<a href="e">\text{num}</a> )</td>
<td>( l ! e )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{cast}<a href="e">\text{int}</a> )</td>
<td>( e ? l )</td>
</tr>
<tr>
<td>Class</td>
<td>( l )</td>
<td>( \text{num} )</td>
<td>( \text{num} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{fun} )</td>
<td>( \text{fun} )</td>
</tr>
</tbody>
</table>

The type \( \text{dyn} \) represents the type of dynamically classified values. (Here we have only two classes of data, numbers and functions, but a richer language would have many more possibilities.) The \( \text{new} \) operation attaches a classifier to a value, and the \( \text{cast} \) operation checks the classifier and returns the associated value. It is important to note that the cast operation checks the \( \text{class} \), not the \( \text{type} \), of its argument.

The static semantics for \( L\{\text{nat dyn} \rightarrow \} \) extends that of \( L\{\text{nat} \rightarrow \} \) with the following additional rules:

\[
\frac{\Gamma \vdash e : \text{nat}}{\Gamma \vdash \text{new}[\text{num}](e) : \text{dyn}} \quad (22.5a)
\]
### 22.3 Hybrid Typing

\[
\begin{align*}
\Gamma \vdash e : \text{parr}(\text{dyn};\text{dyn}) \\
\Gamma \vdash \text{new}[\text{fun}](e) : \text{dyn}
\end{align*}
\]

(22.5b)

\[
\begin{align*}
\Gamma \vdash e : \text{dyn} \\
\Gamma \vdash \text{cast}[\text{num}](e) : \text{nat}
\end{align*}
\]

(22.5c)

\[
\begin{align*}
\Gamma \vdash e : \text{dyn} \\
\Gamma \vdash \text{cast}[\text{fun}](e) : \text{parr}(\text{dyn};\text{dyn})
\end{align*}
\]

(22.5d)

The static semantics ensures that class labels are applied to objects of the appropriate type, namely \text{num} for natural numbers, and \text{fun} for functions defined over labelled values.

The dynamic semantics of \( L\{\text{nat dyn} \rightarrow \} \) extends that of \( L\{\text{nat} \rightarrow \} \) with the following rules:

\[
\begin{align*}
&\frac{\text{e val}}{\text{new}[l](e) \text{ val}} \tag{22.6a} \\
&\frac{\text{e} \mapsto \text{e}' \quad \text{new}[l](e) \mapsto \text{new}[l](e')}{\text{new}[l](e) \mapsto \text{new}[l](e')} \tag{22.6b} \\
&\frac{\text{e} \mapsto \text{e}' \quad \text{cast}[l](e) \mapsto \text{cast}[l](e')}{\text{cast}[l](e) \mapsto \text{cast}[l](e')} \tag{22.6c} \\
&\frac{\text{e val} \quad \text{new}[l](e) \mapsto e}{\text{cast}[l](\text{new}[l](e)) \mapsto e} \tag{22.6d} \\
&\frac{\text{e val} \quad l \neq l' \quad \text{new}[l'](e) \mapsto e}{\text{err}} \tag{22.6e}
\end{align*}
\]

Casting compares the class of the object to the required class, returning the underlying object if these coincide, and signalling an error otherwise.

**Lemma 22.2 (Canonical Forms).** If \( e : \text{dyn} \text{ and e val} \), then \( e = \text{new}[l](e') \) for some class \( l \) and some \( e' \text{ val} \). If \( l = \text{num} \), then \( e' : \text{nat} \), and if \( l = \text{fun} \), then \( e' : \text{parr}(\text{dyn};\text{dyn}) \).

**Proof.** By a straightforward rule induction on static semantics of \( L\{\text{nat dyn} \rightarrow \} \).

\[\square\]

**Theorem 22.3 (Safety).** The language \( L\{\text{nat dyn} \rightarrow \} \) is safe:

1. If \( e : \tau \text{ and e} \mapsto \text{e}' \), then \( \text{e}' : \tau \).

2. If \( e : \tau \), then either \( e \text{ val} \), or \( e \text{ err} \), or \( e \mapsto \text{e}' \) for some \( \text{e}' \).

October 16, 2009 \quad Draft \quad 18:42
Proof. Preservation is proved by rule induction on the dynamic semantics, and progress is proved by rule induction on the static semantics, making use of the canonical forms lemma. The opportunities for run-time errors are the same as those for $\mathcal{L}^{\text{dyn}}$—a well-typed cast might fail at run-time if the class of the case does not match the class of the value.

22.4 Optimization of Dynamic Typing

The language $\mathcal{L}\{\text{nat dyn} \rightarrow\}$ combines static and dynamic typing by enriching $\mathcal{L}\{\text{nat} \rightarrow\}$ with the type, $\text{dyn}$, of classified values. It is, for this reason, called a hybrid language.\footnote{But as we shall see in Section 22.6 on page 195, it is simply a statically typed language with a distinguished recursive type.} The advantage of the hybrid type system is that it allows us to express invariants that are crucial to the optimization of programs in $\mathcal{L}\{\text{dyn}\}$.

Let us examine how this plays out in the case of the addition function, which may be rendered in $\mathcal{L}\{\text{nat dyn} \rightarrow\}$ by the expression

$$\text{fun } \lambda(x: \text{dyn} . \text{fix } p: \text{dyn} \Rightarrow \text{fun } \lambda(y: \text{dyn} . e_{x,p,y}),$$

where

$$x: \text{dyn}, p: \text{dyn}, y: \text{dyn} \vdash e_{x,p,y}: \text{dyn}$$

is defined to be the expression

$$\text{ifz } (y ? \text{num}) \{\text{zero }\Rightarrow x \mid \text{suc}(y) \Rightarrow \text{num }\Rightarrow (s((p ? \text{fun})(\text{num } y) ? \text{num}))\}.$$  

This is a re-formulation of the dynamic addition function given in Section 22.2 on page 189 in which we have made explicit the checking and imposition of classes on values. We will exploit the static type system of $\mathcal{L}\{\text{nat dyn} \rightarrow\}$ to optimize this dynamically typed implementation of addition in accordance with the specification given in Section 22.2 on page 189.

First, note that the body of the $\text{fix}$ expression is an explicitly labelled function. This means that when the recursion is unwound, the variable $p$ is bound to this value of type $\text{dyn}$. Consequently, the check that $p$ is labelled with class $\text{fun}$ is redundant, and can be eliminated. This is achieved by re-writing the function as follows:

$$\text{fun } \lambda(x: \text{dyn} . \text{fun } \text{fix } p: \text{dyn} \Rightarrow \text{fun } \lambda(y: \text{dyn} . e'_{x,p,y}),$$
where \( e'_{x,p,y} \) is the expression

\[
\text{ifz } (y ? \text{num}) \{ \text{zero } \Rightarrow x \mid \text{succ}(y') \Rightarrow \text{num} ! (s(p(\text{num} ! y') ? \text{num})) \}.
\]

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 \( \text{dyn} \rightarrow \text{dyn} \), reflecting that the body is now a “bare function”, rather than a labelled function value of type \( \text{dyn} \).

Next, observe that the parameter \( y \) of type \( \text{dyn} \) 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 \( \text{num} \) 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 \( \text{dyn} \rightarrow \text{dyn} \) to \( \text{nat} \rightarrow \text{dyn} \). 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 \( \text{num} \), and, if so, the underlying value is incremented and labelled with class \( \text{num} \). 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 \( \text{num} \). 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 \( \text{num} \)! However, one might reasonably insist that this is only a theoretical possibility—after all, we are defining the addition function, and its arguments might reasonably be restricted to have class \( \text{num} \). This can be achieved by replacing \( x \) by \( x ? \text{num} \), which checks that \( x \) is of class \( \text{num} \), and returns the underlying number.

Combining these optimizations we obtain the inner loop \( e''_x \) defined as follows:

\[
\text{fix } p : \text{nat} \rightarrow \text{nat} \text{ is } \lambda (y : \text{nat}). \text{ifz } y \{ \text{zero } \Rightarrow x \mid \text{succ}(y') \Rightarrow \text{num} ! (s(p(\text{num} ! y') ? \text{num})) \}.
\]

This function has type \( \text{nat} \rightarrow \text{nat} \), 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 \( \text{dyn} \). Thus we require a value of type \( \text{dyn} \rightarrow \text{dyn} \), but what we have at hand is a function of type \( \text{nat} \rightarrow \text{nat} \). This can be
converted to the required form by pre-composing with a cast to `num` and post-composing with a coercion to `num`:

\[
\text{fun } \lambda(x: \text{dyn}. \text{fun } \lambda(y: \text{dyn}. \text{num} (e''(y ? \text{num})))).
\]

The innermost \(\lambda\)-abstraction converts the function \(e''\) from type \(\text{nat} \rightarrow \text{nat}\) to type \(\text{dyn} \rightarrow \text{dyn}\) 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 \(\text{dyn}\).

\section*{22.5 Static “Versus” Dynamic Typing}

There have been many attempts to explain the distinction between dynamic and static typing, most 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 \(\lambda\)-abstractions, and the presence of classes on values. But it is based on a confusion of classes with types—the \textit{class} of a value (\texttt{num} or \texttt{fun}) is not its \textit{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. Thus, this supposed distinction between dynamic and static typing makes no sense, and is best disregarded.

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. While this description seems superficially accurate, it does not bear scrutiny. 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 \textit{class checking}, not \textit{type checking}, at run-time. For example, application checks that its first argument is labelled with \texttt{fun}; it does not type check the body of the function. Indeed, at no point does the dynamic semantics compute the \textit{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 \textit{heterogeneous} lists, whereas static languages admit only \textit{homogeneous} lists. (The distinction applies to other collections as well.) To see why this description is wrong, let us consider briefly how one might add lists to \(L\{\text{dyn}\}\). One would add two constructs, \texttt{nil}, representing the empty list, and \texttt{cons} \((d_1; d_2)\),
representing the non-empty list with head $d_1$ and tail $d_2$. The origin of the supposed distinction lies in the observation that each element of a list represented in this manner might have a different class. For example, one might form the list

$$\text{cons}(s(z);\text{cons}(\lambda x.x;\text{nil})),$$

whose first element is a number, and whose second element is a function. Such a list is said to be heterogeneous. In contrast static languages commit to a single type for each element of the list, and hence are said to be homogeneous. But here again the supposed distinction breaks down on close inspection, because it is based on the confusion of the type of a value with its class. Every labelled value has type dyn, so that the lists are type homogeneous. But since values of type dyn may have different classes, lists are class heterogeneous—regardless of whether the language is statically or dynamically typed!

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 22.4 on page 192.

### 22.6 Dynamic Typing From 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:

\[
\begin{align*}
\text{dyn} & = \mu t. \left[ \text{num}: \text{nat}, \text{fun}: t \rightarrow t \right] \\
\text{new}[\text{num}](e) & = \text{fold}(\text{num} \cdot e) \\
\text{new}[\text{fun}](e) & = \text{fold}(\text{fun} \cdot e)
\end{align*}
\]

\[4\] Here we have made use of a special expression error to signal an error condition. In a richer language we would use exceptions, which are introduced in Chapter 28.
cast[num]\(e\) = case unfold\(e\) \{num \cdot x \Rightarrow x | \text{fun} \cdot x \Rightarrow \text{error}\} \quad (22.10)

\[\text{cast[fun]}(e) = \text{case unfold}(e) \{\text{num} \cdot x \Rightarrow \text{error} | \text{fun} \cdot x \Rightarrow x\} \quad (22.11)\]

One may readily check that the static and dynamic semantics for the type \(\text{dyn}\) are derivable according to these definitions.

This encoding shows that we need not include a special-purpose type \(\text{dyn}\) in a statically typed language in order to admit dynamic typing. Instead, one may use the general concepts of recursive types and sum types to define special-purpose dynamically typed sub-languages on a per-program basis. For example, if we wish to admit strings into our dynamic sub-language, then we may simply expand the type definition above to admit a third summand for strings, and so on for any type we may wish to consider. Classes emerge as labels of the summands of a sum type, and recursive types ensure that we can represent class-heterogeneous aggregates. Thus, not only is dynamic typing a special case of static typing, but we need make no special provision for it in a statically typed language, since we already have need of recursive types independently of this particular application.

### 22.7 Exercises
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 $L\{\text{nat} \rightarrow\}$ there is a distinct identity function for each type $\tau$, namely $\lambda(x:\tau.x)$, even though the behavior is the same for each choice of $\tau$. Similarly, there is a distinct composition operator for each triple of types, namely

$$\circ_{\tau_1,\tau_2,\tau_3} = \lambda(f:\tau_2 \rightarrow \tau_3. \lambda(g:\tau_1 \rightarrow \tau_2. \lambda(x:\tau_1.f(g(x))))).$$

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 $\lambda$-calculus. Although motivated by a simple practical problem (how to avoid writing redundant code), the concept of polymorphism is central to an impressive variety of seemingly disparate concepts, including the concept of data abstraction (the subject of Chapter 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 \( L\{\rightarrow\forall\} \), 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:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type ( \tau )</td>
<td>::=</td>
<td>( t )</td>
<td>( t )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{arr}(\tau_1;\tau_2) )</td>
<td>( \tau_1 \rightarrow \tau_2 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{all}(t.\tau) )</td>
<td>( \forall(t.\tau) )</td>
</tr>
<tr>
<td>Expr ( e )</td>
<td>::=</td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{lam}<a href="x.e">\tau</a> )</td>
<td>( \lambda(x:\tau.e) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{ap}(e_1;e_2) )</td>
<td>( e_1(e_2) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{Lam}(t.e) )</td>
<td>( \Lambda(t.e) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{App}<a href="e">\tau</a> )</td>
<td>( e[\tau] )</td>
</tr>
</tbody>
</table>

The meta-variable \( t \) ranges over a class of type variables, and \( x \) ranges over a class of expression variables. The type abstraction, \( \text{Lam}(t.e) \), defines a generic, or polymorphic, function with type parameter \( t \) standing for an unspecified type within \( e \). The type application, or instantiation, \( \text{App}[\tau](e) \), 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, \( \text{all}(t.\tau) \), that determines the type, \( \tau \), of the result as a function of the argument, \( t \).

The static semantics of \( L\{\rightarrow\forall\} \) consists of two judgement forms, the type formation judgement,

\[
T \mid \Delta \vdash \tau \text{ type}
\]

and the typing judgement,

\[
T \mathcal{X} \mid \Delta \Gamma \vdash e : \tau.
\]

These are generic judgements over the parameter set \( T \) of type variables and the parameter set \( \mathcal{X} \) of expression variables. They are also hypothetical in a set \( \Delta \) of type assumptions of the form \( t \text{ type} \), where \( t \in T \), and typing assumptions of the form \( x : \tau \), where \( x \in T \) and \( \Delta \vdash \tau \text{ type} \). 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:

\[
\Delta, t \text{ type} \vdash t \text{ type} \quad (23.1a)
\]
The rules defining the typing judgement are as follows:

\[ \Delta, x : \tau \vdash x : \tau \]  
(23.2a)

\[ \Delta, t : \tau \vdash \lambda t.x : \tau \]  
(23.2b)

\[ \Delta, t : \tau \vdash \text{all}(t, \tau) : \tau \]  
(23.2c)

\[ \Delta, t : \tau \vdash \text{Lam}(t, e) : \text{all}(t, \tau) \]  
(23.2d)

\[ \Delta, t : \tau \vdash \text{App}(\tau)(e) : [\tau/t]\tau' \]  
(23.2e)

**Lemma 23.1** (Regularity). If \( \Delta, t : \tau \vdash e : \tau \), and if \( \Delta \vdash \tau_i \text{ type} \) for each assumption \( x_i : \tau_i \) in \( \Gamma \), then \( \Delta \vdash \tau \text{ type} \).

**Proof.** By induction on Rules (23.2). \( \square \)

The static semantics 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 \( \Delta, t : \tau \vdash \tau' \text{ type} \) and \( \Delta \vdash \tau \text{ type} \), then \( \Delta \vdash [\tau/t]\tau' \text{ type} \).

2. If \( \Delta, t : \tau \vdash e' : \tau' \) and \( \Delta \vdash \tau \text{ type} \), then \( \Delta [\tau/t] \Gamma \vdash [\tau/t]e' : [\tau/t]\tau' \).

3. If \( \Delta, x : \tau \vdash e' : \tau' \) and \( \Delta \vdash e : \tau \), then \( \Delta \Gamma [e/x]e' := \tau' \).

The second part of the lemma requires substitution into the context, \( \Gamma \), as well as into the term and its type, because the type variable \( t \) may occur freely in any of these positions.
Returning to the motivating examples from the introduction, the polymorphic identity function, \( I \), is written
\[
\Lambda (t. \lambda (x:t.x));
\]
it has the polymorphic type
\[
\forall (t.t \rightarrow t).
\]
Instances of the polymorphic identity are written \( I[\tau] \), where \( \tau \) is some type, and have the type \( \tau \rightarrow \tau \).

Similarly, the polymorphic composition function, \( C \), is written
\[
\Lambda (t_1. \Lambda (t_2. \Lambda (t_3. \lambda (f:t_2 \rightarrow t_3. \lambda (g:t_1 \rightarrow t_2. \lambda (x:t_1. f(g(x)))))))).
\]
The function \( C \) has the polymorphic type
\[
\forall (t_1. \forall (t_2. \forall (t_3. (t_2 \rightarrow t_3) \rightarrow (t_1 \rightarrow t_2) \rightarrow (t_1 \rightarrow t_3)))).
\]
Instances of \( C \) are obtained by applying it to a triple of types, writing \( C[\tau_1][\tau_2][\tau_3] \). Each such instance has the type
\[
(\tau_2 \rightarrow \tau_3) \rightarrow (\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_3).
\]

**Dynamic Semantics**

The dynamic semantics of \( \mathcal{L}\{\rightarrow \forall\} \) is given as follows:

\[
\frac{\text{lam}[\tau](x.e) \text{ val}}{\text{Lam}(t.e) \text{ val}} \quad (23.3a)
\]

\[
\frac{\text{app}(\text{lam}[\tau_1](x.e); e_2) \mapsto [e_2/x]e}{\text{App}[\tau](\text{Lam}(t.e)) \mapsto [\tau/t]e} \quad (23.3c)
\]

\[
\frac{e_1 \mapsto e'_1}{\text{app}(e_1; e_2) \mapsto \text{app}(e'_1; e_2)} \quad (23.3d)
\]

\[
\frac{e \mapsto e'}{\text{app}[\tau](e) \mapsto \text{app}[\tau](e')} \quad (23.3f)
\]

These rules endow \( \mathcal{L}\{\rightarrow \forall\} \) 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 \( \mathcal{L}\{\rightarrow \forall\} \), using familiar methods.
Lemma 23.3 (Canonical Forms). Suppose that $e : \tau$ and $e \text{ val}$, then

1. If $\tau = \text{arr}(\tau_1; \tau_2)$, then $e = \text{lam}[\tau_1](x.e_2)$ with $x : \tau_1 \vdash e_2 : \tau_2$.

2. If $\tau = \text{all}(t.\tau')$, then $e = \text{Lam}(t.e')$ with $t \text{ type } \vdash e' : \tau'$.

Proof. By rule induction on the static semantics. \qed

Theorem 23.4 (Preservation). If $e : \sigma$ and $e \mapsto e'$, then $e' : \sigma$.

Proof. By rule induction on the dynamic semantics. \qed

Theorem 23.5 (Progress). If $e : \sigma$, then either $e \text{ val}$ or there exists $e'$ such that $e \mapsto e'$.

Proof. By rule induction on the static semantics. \qed

23.2 Polymorphic Definability

The language $\mathcal{L}\{ightarrow \forall\}$ is astonishingly expressive. Not only are all finite products and sums definable in the language, but so are all inductive and coinductive types, including both the eager and the lazy natural numbers! This is most naturally expressed using definitional equivalence, which is defined to be the least congruence containing the following two axioms:

\[
\begin{align*}
\Delta \Gamma, x : \tau_1 \vdash e : \tau_2 \quad & \Delta \Gamma \vdash e_1 : \tau_1 \\
\Delta \Gamma \vdash \lambda(x : \tau. e_2)(e_1) & \equiv [e_1/x]e_2 : \tau_2
\end{align*}
\] (23.4a)

\[
\begin{align*}
\Delta, t \text{ type } \Gamma \vdash e : \tau \quad & \Delta \vdash \sigma \text{ type} \\
\Delta \Gamma \vdash \Lambda(t.e)[\sigma] & \equiv [\sigma/t]e : [\sigma/t]\tau
\end{align*}
\] (23.4b)

The remaining rules specify 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 $\mathcal{L}_{\arrow \forall}$ as follows:

$\text{unit} = \forall(r. r \rightarrow r)$

$\langle \rangle = \Lambda(r. \lambda(x: r.x))$

It is easy to check that the static semantics given in Chapter 16 is derivable. There being no elimination rule, there is no requirement on the dynamic semantics.

Binary products are definable in $\mathcal{L}_{\arrow \forall}$ by using encoding tricks similar to those described in Chapter 21 for the untyped $\lambda$-calculus:

$\tau_1 \times \tau_2 = \forall (r. (\tau_1 \rightarrow r \rightarrow r \rightarrow r \rightarrow r \rightarrow r))$

$\langle e_1, e_2 \rangle = \Lambda(r. \lambda(x: \tau_1 \rightarrow \tau_2 \rightarrow r. x(e_1)(e_2)))$

$e \cdot 1 = e[\tau_1] (\lambda(x: \tau_1. \lambda(y: \tau_2. x)))$

$e \cdot r = e[\tau_2] (\lambda(x: \tau_1. \lambda(y: \tau_2. y)))$

The static semantics given in Chapter 16 is derivable according to these definitions. Moreover, the following definitional equivalences are derivable in $\mathcal{L}_{\arrow \forall}$ from these definitions:

$\langle e_1, e_2 \rangle \cdot 1 \equiv e_1 : \tau_1$

and

$\langle e_1, e_2 \rangle \cdot r \equiv e_2 : \tau_2.$

The nullary sum, or void, type is definable in $\mathcal{L}_{\arrow \forall}$:

$\text{void} = \forall(r. r)$

$\text{abort}[\rho](e) = e[\rho]$

There is no definitional equivalence to be checked, there being no introductory rule for the void type.

Binary sums are also definable in $\mathcal{L}_{\arrow \forall}$:

$\tau_1 + \tau_2 = \forall(r. (\tau_1 \rightarrow r \rightarrow r \rightarrow r \rightarrow r))$

$1 \cdot e = \Lambda(r. \lambda(x: \tau_1 \rightarrow r. \lambda(y: \tau_2 \rightarrow r. x(e))))$

$r \cdot e = \Lambda(r. \lambda(x: \tau_1 \rightarrow r. \lambda(y: \tau_2 \rightarrow r. y(e))))$

$\text{case } e \{1 \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\} =

e[\rho] (\lambda(x_1: \tau_1. e_1))(\lambda(x_2: \tau_2. e_2))$
provided that the types make sense. It is easy to check that the following equivalences are derivable in $L\{\rightarrow\forall\}$:

$$\text{case } l \cdot \{1 \cdot x_1 \Rightarrow e_1 | r \cdot x_2 \Rightarrow e_2\} \equiv [e/x_1]e_1 : \rho$$

and

$$\text{case } r \cdot d_2 \{1 \cdot x_1 \Rightarrow e_1 | r \cdot x_2 \Rightarrow e_2\} \equiv [e/x_2]e_2 : \rho.$$ 

Thus the dynamic behavior specified in Chapter 17 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 $L\{\rightarrow\forall\}$. The key is the representation of the iterator, whose typing rule we recall here for reference:

$$\frac{e_0 : \text{nat}\quad e_1 : \tau\quad x : \tau\quad \vdash e_2 : \tau}{\text{natiter}(e_0; e_1; x; e_2) : \tau}.$$ 

Since the result type $\tau$ is arbitrary, this means that if we have an iterator, then it can be used to define a function of type

$$\text{nat} \rightarrow \forall(t. t \rightarrow (t \rightarrow t) \rightarrow t).$$ 

This function, when applied to an argument $n$, yields a polymorphic function that, for any result type, $t$, if given the initial result for $z$, and if given a function transforming the result for $x$ into the result for $s(x)$, 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 $L\{\rightarrow\forall\}$ by the following equations:

\[
\begin{align*}
\text{nat} &= \forall(t. t \rightarrow (t \rightarrow t) \rightarrow t) \\
z &= \Lambda(t. \lambda(z: t. \lambda(s: t \rightarrow t. z))) \\
s(e) &= \Lambda(t. \lambda(z: t. \lambda(s: t \rightarrow t. s(e[t](z)(s)))))) \\
\text{natiter}(e_0; e_1; x; e_2) &= e_0[\tau](e_1)(\lambda(x: \tau.e_2))
\end{align*}
\]

It is a straightforward exercise to check that the static and dynamic semantics given in Chapter 14 is derivable in $L\{\rightarrow\forall\}$ under these definitions.
23.3 Parametricity

A remarkable property of polymorphic typing is that it strongly constrains the behavior of an expression of that type. For example, if \( i \) is any expression of type \( \forall (t \cdot t \rightarrow t) \), then it must behave like the identity function in the following sense. For an arbitrary type \( \tau \) and an arbitrary expression \( e : \tau \), it must be that \( i[\tau](e) \equiv e \). The informal reason is that \( i \), being polymorphic, must, when applied to an arbitrary argument of arbitrary type, must return a result of that type. Since not even the type, much less the value, of the argument is known in advance, the function \( i \) has no choice but to return the argument as result if it is to achieve the specified typing. Similarly, if \( c \) is any expression of type \( \forall (t \cdot t \rightarrow t \rightarrow t) \), then for any type \( \tau \) and any \( e_1 : \tau \) and \( e_2 : \tau \), it must be that either \( c(e_1)(e_2) \equiv e_1 \) or \( c(e_1)(e_2) \equiv e_2 \).

A rigorous justification of these claims is deferred to Chapter 52. Meanwhile we content ourselves with a brief summary of the argument developed there. The crucial idea is that types may be interpreted as relations, and we may prove that every well-typed expression of \( \mathcal{L}\{\rightarrow \forall\} \) preserves any such relational interpretation. This is best explained by example. The upshot of Theorem 52.8 on page 482, specialized to the type \( i : \forall (t \cdot t \rightarrow t) \), is that for any type \( \tau \), any predicate \( P \) on expressions of type \( \tau \), and any \( e : \tau \), if \( P(e) \), then \( P(i(e)) \). Fix \( \tau \) and \( e : \tau \), and define \( i(x) \) to hold iff \( x \equiv e \). By Theorem 52.8 on page 482 we have that for any \( e' : \tau \), if \( e' \equiv e \), then \( i(e') \equiv e \). Noting that definitional equivalence is reflexive, it follows that \( i(e) \equiv e \). Similarly, if \( c : \forall (t \cdot t \rightarrow t \rightarrow t) \), then, fixing \( \tau \), \( e_1 : \tau \), and \( e_2 : \tau \), we may define \( c(e) \) to hold iff either \( e \equiv e_1 \) or \( e \equiv e_2 \). It follows from Theorem 52.8 on page 482 that either \( c(e_1)(e_2) \equiv e_1 \) or \( c(e_1)(e_2) \equiv e_2 \).

The important point here is that the properties of \( i \) and \( c \) are derived without knowing anything about these expressions themselves beyond their
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 $\mathcal{L}\{\rightarrow\forall\}$. 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 $\mathcal{L}\{\rightarrow\forall\}$ may be traced to the ability to instantiate a polymorphic type with another polymorphic type. For example, if we let $\tau$ be the type $\forall(t.t \rightarrow t)$, and, assuming that $e : \tau$, we may apply $e$ to its own type, obtaining the expression $e[\tau]$ of type $\tau \rightarrow \tau$. Written out in full, this is the type

$$\forall(t.t \rightarrow t) \rightarrow \forall(t.t \rightarrow t),$$

which is larger (both textually, and when measured by the number of occurrences of quantified types) than the type of $e$ itself. In fact, this type is large enough that we can go ahead and apply $e[\tau]$ to $e$ again, obtaining the expression $e[\tau](e)$, which is again of type $\tau$ — the very type of $e$!

This property of $\mathcal{L}\{\rightarrow\forall\}$ is called impredicativity\(^1\); the language $\mathcal{L}\{\rightarrow\forall\}$ is said to permit impredicative (type) quantification. The distinguishing char-

\(^1\)pronounced im-PRED-ic-a-tiv-it-y
23.4 Restricted Forms of Polymorphism

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 \( L\{\rightarrow\forall\} \), and is correspondingly the prime source of complexity when reasoning about it (for example, in the proof that all expressions of \( L\{\rightarrow\forall\} \) terminate).

Contrast this with \( L\{\rightarrow\} \), in which the type of an application of a function is evidently smaller than the type of the function itself. For if \( e : \tau_1 \rightarrow \tau_2 \), 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 \( \forall(t.t \rightarrow t) \) may be instantiated with the type \( u \rightarrow u \) to obtain the type \( (u \rightarrow u) \rightarrow (u \rightarrow u) \). This type has more symbols in it than \( \tau \), but is smaller in that it has fewer quantifiers. The restriction to quantification only over unquantified types is called predicative\(^2\) 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 \( L\{\rightarrow\forall\} \) 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 \( L\{\rightarrow\forall\} \), because it plays an important role in the design of practical polymorphic languages.

The prenex fragment of \( L\{\rightarrow\forall\} \) is designated \( L^1\{\rightarrow\forall\} \), for reasons that will become clear in the next subsection. It is defined by stratifying types into two classes, the monotypes (or rank-0 types) and the polytypes (or rank-1

\(^2\)pronounced PRED-ic-a-tive
23.4 Restricted Forms of Polymorphism

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 \( \Delta \vdash \tau \text{ mono} \) and \( \Delta \vdash \tau \text{ poly} \), where \( \Delta \) is a finite set of hypotheses of the form \( t \text{ mono} \), where \( t \) is a type variable not otherwise declared in \( \Delta \). The rules for deriving these judgements are as follows:

\[
\begin{align*}
\Delta, t \text{ mono} & \vdash t \text{ mono} \\
\Delta \vdash \tau_1 \text{ mono} & \quad \Delta \vdash \tau_2 \text{ mono} \\
\Delta & \vdash \text{arr}(\tau_1; \tau_2) \text{ mono} \\
\Delta \vdash \tau \text{ mono} & \quad \Delta \vdash \tau \text{ poly} \\
\Delta, t \text{ mono} & \vdash \tau \text{ poly} \\
\Delta & \vdash \text{all}(t. \tau) \text{ poly}
\end{align*}
\]

Base types, such as \( \text{nat} \) (as a primitive), or other type constructors, such as sums and products, would be added to the language as monotypes.

The static semantics of \( \mathcal{L}^1\{\to, \forall\} \) is given by rules for deriving hypothetical judgements of the form \( \Delta \Gamma \vdash e : \sigma \), where \( \Delta \) consists of hypotheses of the form \( t \text{ mono} \), and \( \Gamma \) consists of hypotheses of the form \( x : \sigma \), where \( \Delta \vdash \sigma \text{ poly} \). The rules defining this judgement are as follows:

\[
\begin{align*}
\Delta \Gamma, x : \tau & \vdash x : \tau \\
\Delta \vdash \tau_1 \text{ mono} & \quad \Delta \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \\
\Delta \Gamma & \vdash \text{lam}[\tau_1](x.e_2) : \text{arr}(\tau_1; \tau_2) \\
\Delta \Gamma & \vdash \text{ap}(e_1; e_2) : \tau \\
\Delta, t \text{ mono} & \vdash e : \tau \\
\Delta \Gamma & \vdash \text{Lam}(t.e) : \text{all}(t. \tau) \\
\Delta \vdash \tau \text{ mono} & \quad \Delta \Gamma \vdash e : \text{all}(t. \tau') \\
\Delta \Gamma & \vdash \text{App}[\tau](e) : [\tau/t'] \tau'
\end{align*}
\]

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 \( \sigma \).
23.4 Restricted Forms of Polymorphism

The restriction on the domain of a \( \lambda \)-abstraction to be a monotype means that a fully general \texttt{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 \( \mathcal{L}\{\rightarrow\forall_p\} \) with a primitive \texttt{let} construct whose static semantics is as follows:

\[
\Delta \vdash \tau_1 \text{ poly} \quad \Delta \Gamma \vdash e_1 : \tau_1 \quad \Delta \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \\
\Delta \Gamma \vdash \texttt{let}_{\tau_1}(e_1; x.e_2) : \tau_2.
\]

(23.7)

For example, the expression

\[
\texttt{let } I : \forall(t. t \rightarrow t) \text{ be } \Lambda(t. \lambda(x : t. x)) \text{ in } I[\tau \rightarrow \tau](I[\tau])
\]

has type \( \tau \rightarrow \tau \) for any polytype \( \tau \).

23.4.3 Rank-Restricted Fragments

The binary distinction between monomorphic and polymorphic types in \( \mathcal{L}^1\{\rightarrow\forall\} \) 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 \( \sigma \rightarrow \forall(t. \tau) \) and the type \( \forall(t. \sigma \rightarrow \tau) \) (where \( t \notin \sigma \)). This motivates the definition of a hierarchy of fragments of \( \mathcal{L}\{\rightarrow\forall\} \) that subsumes the prenex fragment as a special case.

We will define a judgement of the form \( \tau \text{ type } [k] \), where \( k \geq 0 \), to mean that \( \tau \) is a type of rank \( k \). Informally, types of rank 0 have no quantification, and types of rank \( k + 1 \) may involve quantification, but the domains of function types are restricted to be of rank \( k \). Thus, in the terminology of Section 23.4.2 on page 208, 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 \( \Delta \vdash \tau \text{ type } [k] \), where \( \Delta \) is a finite set of hypotheses of the form \( t_i \text{ type } [k_i] \) for some pairwise distinct set of type variables \( t_i \). The rules defining these judgements are as follows:

\[
\Delta, t \text{ type } [k] \vdash t \text{ type } [k] \quad (23.8a)
\]

\[
\Delta \vdash \tau_1 \text{ type } [0] \quad \Delta \vdash \tau_2 \text{ type } [0] \\
\Delta \vdash \texttt{arr}(\tau_1; \tau_2) \text{ type } [0] \quad (23.8b)
\]
23.5 Exercises

\[
\begin{align*}
\Delta \vdash \tau_1 &\enspace \text{type} \ [k] \quad \Delta \vdash \tau_2 &\enspace \text{type} \ [k+1] \\
\Delta \vdash \text{arr} (\tau_1; \tau_2) &\enspace \text{type} \ [k+1] \\
\Delta \vdash \tau &\enspace \text{type} \ [k] \\
\Delta \vdash \tau &\enspace \text{type} \ [k+1] \\
\Delta, t &\enspace \text{type} \ [k] \quad \Delta \vdash \tau &\enspace \text{type} \ [k+1] \\
\Delta \vdash \text{all} (t. \tau) &\enspace \text{type} \ [k+1]
\end{align*}
\] (23.8c)

With these restrictions in mind, it is a good exercise to define the static semantics of \( L^k \{ \rightarrow \forall \} \), the restriction of \( L \{ \rightarrow \forall \} \) 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 \( \tau \) type \([k]\). For example, the rank-limited rules for \( \lambda \)-abstractions is phrased as follows:

\[
\begin{align*}
\Delta \vdash \tau_1 &\enspace \text{type} \ [0] \\
\Delta, x : \tau_1 &\enspace \text{type} \ [0] \quad e_2 &\enspace \tau_2 \ [0] \\
\Delta \Gamma &\vdash \lambda m[\tau_1] (x.e_2) : \text{arr} (\tau_1; \tau_2) \ [0] \\
\Delta \vdash \tau_1 &\enspace \text{type} \ [k] \\
\Delta, x : \tau_1 &\enspace \text{type} \ [k] \quad e_2 &\enspace \tau_2 \ [k+1] \\
\Delta \Gamma &\vdash \lambda m[\tau_1] (x.e_2) : \text{arr} (\tau_1; \tau_2) \ [k+1]
\end{align*}
\] (23.9a)

The remaining rules follow a similar pattern.

The rank-limited languages \( L^k \{ \rightarrow \forall \} \) clarifies the requirement for a primitive \( \text{let} \) construct in \( L^1 \{ \rightarrow \forall \} \). The prenex fragment of \( L \{ \rightarrow \forall \} \) corresponds to the rank-one fragment \( L^1 \{ \rightarrow \forall \} \). The \text{let} construct for rank-one types is definable in \( L^2 \{ \rightarrow \forall \} \) from \( \lambda \)-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 \( L \{ \rightarrow \forall \} \) by exploiting the definability of iteration and binary products.

2. Investigate the representation of eager products and sums in eager and lazy variants of \( L \{ \rightarrow \forall \} \).

3. Show how to write an interpreter for \( L \{ \text{nat} \rightarrow \} \) in \( L \{ \rightarrow \forall \} \).
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 $L\{\rightarrow \forall \exists\}$ is the extension of $L\{\rightarrow \forall\}$ with the following constructs:

\[
\begin{array}{lll}
\text{Category} & \text{Item} & \text{Abstract} & \text{Concrete} \\
\hline
\text{Types} & \tau & := & \text{some}(t.\tau) & \exists(t.\tau) \\
\text{Expr} & e & := & \text{pack}[t.\tau][\rho](e) & \text{pack }\rho\text{ with }e\text{ as }\exists(t.\tau) \\
& & | & \text{open}[t.\tau][\rho](e_1; x. e_2) & \text{open }e_1\text{ as }t\text{ with }x:\tau\text{ in }e_2
\end{array}
\]

The introductory form for the existential type $\sigma = \exists(t.\tau)$ is a package of the form $\text{pack }\rho\text{ with }e\text{ as }\exists(t.\tau)$, where $\rho$ is a type and $e$ is an expression of type $[\rho/t]\tau$. The type $\rho$ is called the representation type of the package, and the expression $e$ is called the implementation of the package. The eliminatory form for existentials is the expression $\text{open }e_1\text{ as }t\text{ with }x:\tau\text{ in }e_2$, which opens the package $e_1$ for use within the client $e_2$ by binding its representation type to $t$ and its implementation to $x$ for use within $e_2$. Crucially, the typing rules ensure that the client is type-correct independently of the actual representation type used by the implementor, so that it may be varied without affecting the type correctness of the client.

The abstract syntax of the open construct specifies that the type variable, $t$, and the expression variable, $x$, are bound within the client. They may be renamed at will by $\alpha$-equivalence without affecting the meaning of the construct, provided, of course, that the names are chosen so as not to conflict with any others that may be in scope. In other words the type, $t$, may be thought of as a “new” type, one that is distinct from all other types, when it is introduced. This is sometimes called generativity of abstract types: the use of an abstract type by a client “generates” a “new” type within that client. This behavior is simply a consequence of identifying terms up to $\alpha$-equivalence, and is not particularly tied to data abstraction.

24.1.1 Static Semantics

The static semantics 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:

\[
\frac{\Delta, l \vdash \tau \text{ type}}{\Delta \vdash \text{some}(t.\tau) \text{ type}} \quad (24.1a)
\]

18:42
24.1 Existential Types 215

\[
\Delta \vdash \rho \text{ type} \quad \Delta, t \vdash \tau \text{ type} \quad \Delta \Gamma \vdash e : [\rho/t]\tau \\
\Delta \Gamma \vdash \text{pack}[t.\tau][\rho](e) : \text{some}(t.\tau)
\]  
(24.1b)

\[
\Delta \Gamma \vdash e_1 : \text{some}(t.\tau) \quad \Delta, t \vdash \tau_2, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2 \text{ type} \\
\Delta \Gamma \vdash \text{open}[t.\tau][\tau_2](e_1; t, x. e_2) : \tau_2
\]  
(24.1c)

Rule (24.1c) is complex, so study it carefully! There are two important things to notice:

1. The type of the client, \(\tau_2\), must not involve the abstract type \(t\). This restriction prevents the client from attempting to export a value of the abstract type outside of the scope of its definition.

2. The body of the client, \(e_2\), is type checked without knowledge of the representation type, \(t\). The client is, in effect, polymorphic in the type variable \(t\).

**Lemma 24.1** (Regularity). Suppose that \(\Delta \Gamma \vdash e : \tau\). If \(\Delta \vdash \tau_i \text{ type for each } x_i : \tau_i \text{ in } \Gamma\), then \(\Delta \vdash \tau \text{ type}\).

*Proof.* By induction on Rules (24.1).

24.1.2 Dynamic Semantics

The dynamic semantics of existential types is specified as follows:

\[
\{e \text{ val}\} \\
\text{pack}[t.\tau][\rho](e) \text{ val}
\]  
(24.2a)

\[
\left\{ \begin{array}{l}
e \mapsto e' \\
\text{pack}[t.\tau][\rho](e) \mapsto \text{pack}[t.\tau][\rho](e')
\end{array} \right\}
\]  
(24.2b)

\[
e_1 \mapsto e_1'
\]  
(24.2c)

\[
\text{open}[t.\tau][\tau_2](e_1; t, x. e_2) \mapsto \text{open}[t.\tau][\tau_2](e_1'; t, x. e_2)
\]  
(24.2d)

These rules endow \(L\{\rightarrow \forall \exists\}\) with a lazy semantics for packages. More importantly, these rules specify that there are no abstract types at run time! The representation type is exposed to the client by substitution when the package is opened. In other words, 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 $\mathcal{L}\{\rightarrow\forall\}$ 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. \qed

**Lemma 24.3** (Canonical Forms). If $e : \text{some}(1.\tau)$ and $e \text{ val}$, then $e = \text{pack}[1.\tau][\rho](e')$ for some type $\rho$ and some $e' \text{ val}$ such that $e' : [\rho/1]\tau$.

*Proof.* By rule induction on the static semantics, making use of the definition of closed values. \qed

**Theorem 24.4** (Progress). If $e : \tau$ then either $e \text{ val}$ or there exists $e'$ such that $e \mapsto e'$.

*Proof.* By rule induction on $e : \tau$, making use of the canonical forms lemma. \qed

### 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, $\tau$, 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, $\exists (t.\tau)$, which serves as the interface of the queue abstraction:

$$\exists (t.\langle \text{emp}: t, \text{ins}: \tau \times t \to t, \text{rem}: \tau \to \tau \times t \rangle).$$

The representation type, $t$, of queues is abstract — all that is specified about it is that it supports the operations $\text{emp}$, $\text{ins}$, and $\text{rem}$, 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, $e_i$, in which queues are represented as lists:

**pack list with** $\langle \text{emp} = \text{nil}, \text{ins} = e_i, \text{rem} = e_r \rangle$ as $\exists (t.\tau)$,

where

$$e_i : \tau \times \text{list} \to \text{list} = \lambda (x: \tau \times \text{list}. e'_i),$$

and

$$e_r : \text{list} \to \tau \times \text{list} = \lambda (x: \text{list}. e'_r).$$

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:

**pack list × list with** $\langle \text{emp} = \langle \text{nil}, \text{nil} \rangle, \text{ins} = e_i, \text{rem} = e_r \rangle$ as $\exists (t.\tau)$.

In this case $e_i$ has type

$$\tau \times (\text{list} \times \text{list}) \to (\text{list} \times \text{list}),$$

and $e_r$ has type

$$(\text{list} \times \text{list}) \to \tau \times (\text{list} \times \text{list}).$$
These operations “know” that queues are represented as values of type \texttt{list} \times \texttt{list}, and are implemented accordingly.

The important point is that the \textit{same} client type checks regardless of which implementation of queues we choose. This is because the representation type is hidden, or \textit{held abstract}, from the client during type checking. Consequently, it cannot rely on whether it is \texttt{list} or \texttt{list} \times \texttt{list} or some other type. That is, the client is \textit{independent} of the representation of the abstract type.

### 24.3 Definability of Existentials

It turns out that it is not necessary to extend $L\{\neg \forall\}$ 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 \textit{polymorphic} in the representation type. The typing rule for

$$\text{open } e \text{ as } t \text{ with } x : \tau \text{ in } e' : \tau', $$

where $e : \exists(t.\tau)$, specifies that $e' : \tau'$ under the assumptions $t$ type and $x : \tau$. In essence, the client is a polymorphic function of type

$$\forall(t.\tau \rightarrow \tau'),$$

where $t$ may occur in $\tau$ (the type of the operations), but not in $\tau'$ (the type of the result).

This suggests the following encoding of existential types:

$$\exists(t.\sigma) = \forall(t'.\forall(t.\sigma \rightarrow t') \rightarrow t')$$

$$\text{pack } \rho \text{ with } e \text{ as } \exists(t.\sigma) = \Lambda(t'.\lambda(x : \forall(t.\sigma \rightarrow t').\tau).\rho(e'))$$

$$\text{open } e \text{ as } t \text{ with } x : \sigma \text{ in } e' = e[\tau'](\Lambda(t.\lambda(x : \sigma.e')))$$

An existential is encoded as a polymorphic function taking the overall result type, $t'$, as argument, followed by a polymorphic function representing the client with result type $t'$, and yielding a value of type $t'$ as overall result. Consequently, the open construct simply packages the client as such a polymorphic function, instantiates the existential at the result type, $\tau$, and applies it to the polymorphic client. (The translation therefore depends on knowing the overall result type, $\tau$, of the open construct.) Finally, a package consisting of a representation type $\rho$ and an implementation $e$ is a...
polymorphic function that, when given the result type, \( t \), and the client, \( x \), instantiates \( x \) with \( \rho \) and passes to it the implementation \( e \).

It is then a straightforward exercise to show that this translation correctly reflects the static and dynamic semantics 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 facing 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 \( \forall (t.(\sigma \rightarrow t) \rightarrow t) \), where \( t \) does not occur free in \( \tau \) (but may, of course, occur in \( \sigma \)). Applying the parametricity property described informally in Chapter 23 (and developed rigorously in Chapter 52), 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 \( \sigma \) is the type

\[
\langle \text{emp:} t, \text{ins:} \tau \times t \rightarrow t, \text{rem:} t \rightarrow \tau \times t \rangle.
\]

Theorem 52.8 on page 482 ensures that if \( \rho \) and \( \rho' \) 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
\[
e = (\text{emp} = e_m, \text{ins} = e_i, \text{rem} = e_r)
\]
and
\[
e' = (\text{emp} = e'_m, \text{ins} = e'_i, \text{rem} = e'_r).
\]
For these implementations to respect \( R \) means that the following three conditions hold:

1. The empty queues are related: \( R(e_m, e'_m) \).
2. Inserting the same element on each of two related queues yields related queues: if \( d: \tau \) and \( R(q, q') \), then \( R(e_i(d)(q), e_i(d)(q')) \).
3. If two queues are related, their front elements are the same and their back elements are related: if \( R(q, q') \equiv (d, r), e_r(q) \equiv (d', r') \), then \( d \) is \( d' \) and \( R(r, r') \).

If such a relation \( R \) exists, then the implementations \( e \) and \( e' \) are said to be \textit{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 \textit{simulates} the other up to the relationship specified by \( R \).

To see how this works in practice, let us consider informally two implementations of the abstract type of queues specified above. For the reference implementation we choose \( \rho \) to be the type \texttt{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 \( \rho' \) to be the type \texttt{list} \times \texttt{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 \( \langle \text{cons}(d; b), f \rangle \), 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 \( \text{cons}(d; f') \), 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 \( \langle \text{nil}, \text{rev}(b) \rangle \), where \( \text{rev} \) 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 \texttt{list} and \texttt{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 \( \text{app}(b) (\text{rev}(f)) \), where \text{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 $\tau_1 \rightarrow \tau_2$ or $\text{list}$ 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 $\text{nat}$ 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 static semantics, and the dynamic phase implements the dynamic semantics. 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-
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 consisting of the constructors \( \text{nat} \) and \( \text{str} \)? This should be equivalent to \( \text{nat} \), 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 \( \text{Type} \). 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:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind</td>
<td>( \kappa )</td>
<td>Type</td>
<td>Type</td>
</tr>
<tr>
<td></td>
<td>Unit</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{Prod}(\kappa_1; \kappa_2) )</td>
<td>( \kappa_1 \times \kappa_2 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{Arr}(\kappa_1; \kappa_2) )</td>
<td>( \kappa_1 \rightarrow \kappa_2 )</td>
<td></td>
</tr>
</tbody>
</table>

The kinds consist of the kind of types, \( \text{Type} \), the unit kind, \( \text{Unit} \), and are closed under formation of product and function kinds.

The syntax of constructors is divided into two categories, the *neutral*
and the *canonical*, according to the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neutral</td>
<td>a</td>
<td>::= u</td>
<td>u</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>proj<a href="a">l</a> pr₁(a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>proj<a href="a">r</a> prᵣ(a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>app(a₁;c₂) a₁[c₂]</td>
</tr>
<tr>
<td>Canonical</td>
<td>c</td>
<td>::= atom(a) µ</td>
<td></td>
</tr>
</tbody>
</table>
|            |      |          | unit      *
|            |      |          | pair(c₁;c₂) ⟨c₁,c₂⟩ |
|            |      |          | lam(u.c)  λu.c |

The meta-variable *u* ranges over *constructor variables*.

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 *pr₁(*⟨c₁,c₂⟩*)*, which would be definitionally equivalent to *c₁*, 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 static semantics that ensures that only neutral constructors of kind *Type* may be treated as canonical. This requirement is imposed to limit the forms of canonical constructors of the other kinds. In particular, variables of function, product, or unit kind will turn out not to be canonical, but only neutral.

The static semantics of constructors and kinds is specified by the judgements

\[ \Delta \vdash a \uparrow \kappa \quad \text{neutral constructor formation} \]
\[ \Delta \vdash c \downarrow \kappa \quad \text{canonical constructor formation} \]

In each of these judgements \( \Delta \) is a finite set of hypotheses of the form

\[ u₁ \uparrow \kappa₁, \ldots, uₙ \uparrow \kappaₙ \]

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 parametric hypothetical judgements with parameters \( u₁, \ldots, uₙ \) that are determined by the forms of the hypotheses.

The rules for constructor formation are as follows:

\[ \bar{\Delta}, u \uparrow \kappa \vdash u \uparrow \kappa \quad (25.1a) \]
\[ \begin{align*}
\Delta \vdash a \uparrow \kappa_1 \times \kappa_2 \\
\Delta \vdash \text{pr}_1(a) \uparrow \kappa_1 
\end{align*} \] (25.1b)

\[ \begin{align*}
\Delta \vdash a \uparrow \kappa_1 \times \kappa_2 \\
\Delta \vdash \text{pr}_r(a) \uparrow \kappa_2 
\end{align*} \] (25.1c)

\[ \begin{align*}
\Delta \vdash a_1 \uparrow \kappa_2 \rightarrow \kappa \quad \Delta \vdash c_2 \downarrow \kappa_2 \\
\Delta \vdash a_1[c_2] \uparrow \kappa 
\end{align*} \] (25.1d)

\[ \begin{align*}
\Delta \vdash a \uparrow \text{Type} \\
\Delta \vdash c \downarrow \text{Type} 
\end{align*} \] (25.1e)

\[ \Delta \vdash \ast \downarrow 1 \] (25.1f)

\[ \begin{align*}
\Delta \vdash c_1 \downarrow \kappa_1 \quad \Delta \vdash c_2 \downarrow \kappa_2 \\
\Delta \vdash \langle c_1, c_2 \rangle \downarrow \kappa_1 \times \kappa_2 
\end{align*} \] (25.1g)

\[ \begin{align*}
\Delta, u \uparrow \kappa_1 \vdash c_2 \downarrow \kappa_2 \\
\Delta \vdash \lambda u . c_2 \downarrow \kappa_1 \rightarrow \kappa_2 
\end{align*} \] (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 $c = \ast$.

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 $\kappa = \kappa_1 \rightarrow \kappa_2$, 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 static semantics 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
static semantics of \( \mathcal{L} \) must be defined to have judgements of the following two forms:

\[
\Delta \vdash \tau \text{ type} \quad \text{type formation}
\]
\[
\Delta, \Gamma \vdash e : \tau \quad \text{expression formation}
\]

where, as before, \( \Gamma \) is a finite set of hypotheses of the form

\[
x_1 : \tau_1, \ldots, x_k : \tau_k
\]

for some \( k \geq 0 \) such that \( \Delta \vdash \tau_i \text{ type} \) for each \( 1 \leq i \leq k \).

As a general principle, every constructor of kind Type is a classifier:

\[
\frac{\Delta \vdash \tau \uparrow \text{Type}}{\Delta \vdash \tau \text{ type}} . \quad (25.2)
\]

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 \( \mathcal{L}\{\to, \forall\} \) from Chapter 23. In both cases we extend the universal quantifier \( \forall (t \cdot \tau) \) to admit quantification over an arbitrary kind, written \( \forall_{\kappa} u \cdot \tau \), 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:

\[
\Delta \vdash \to \uparrow \text{Type} \to \text{Type} \to \text{Type} \quad (25.3a)
\]
\[
\Delta \vdash \forall_{\kappa} \uparrow (\kappa \to \text{Type}) \to \text{Type} \quad (25.3b)
\]

We regard the classifier \( \tau_1 \to \tau_2 \) to be the application \( \to [\tau_1][\tau_2] \). Similarly, we regard the classifier \( \forall_{\kappa} u \cdot \tau \) to be the application \( \forall_{\kappa} [\lambda u \cdot \tau] \).

The predicative fragment excludes the constant specified by Rule (25.3b) in favor of a separate rule for the formation of universally quantified types:

\[
\Delta, u \uparrow \kappa \vdash \tau \text{ type} \quad \frac{}{\Delta \vdash \forall_{\kappa} u \cdot \tau \text{ type}} . \quad (25.4)
\]

The important point is that \( \forall_{\kappa} u \cdot \tau \) is a type (as classifier), but is not a constructor of kind type.
The significance of this distinction becomes apparent when we consider the introduction and elimination forms for the generalized quantifier, which are the same for both fragments:

\[ \Sigma, u \uparrow \kappa \Gamma \vdash e : \tau \]
\[ \Delta \Gamma \vdash \Lambda(u : \kappa.e) : \forall \kappa u. \tau \]  (25.5a)

\[ \Delta \Gamma \vdash e : \forall \kappa u. \tau \]
\[ \Delta \Gamma \vdash c \downarrow \kappa \]
\[ \Delta \Gamma \vdash e[c] : [c/u] \tau \]  (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 \( \kappa \). When \( \kappa \) 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 \( \kappa \) 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 \( \kappa \) 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 for arbitrary abt’s in Chapter 7. 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

\[ u \uparrow \text{Type} \rightarrow \text{Type} \vdash u[d] \uparrow \text{Type}, \]

where \( d \uparrow \text{Type} \). (It is not important what we choose for \( d \), so we leave it abstract.) Now if \( c \downarrow \text{Type} \rightarrow \text{Type} \), then by Lemma 25.1 on page 226 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

\[
[\lambda u'. c'/u][d] = [d/u']c'.
\]

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:

\[
\begin{align*}
[c/u : \kappa]a = a' & \quad \text{canonical into neutral yielding neutral} \\
[c/u : \kappa]a = c' \downarrow \kappa' & \quad \text{canonical into neutral yielding canonical and kind} \\
[c/u : \kappa]c' = c'' & \quad \text{canonical into canonical yielding canonical}
\end{align*}
\]

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.

\[
\begin{align*}
[c/u : \kappa]a' = a'' & \\
[c/u : \kappa]a'' = a''
\end{align*}
\quad (25.6a)
\]

\[
\begin{align*}
[c/u : \kappa]a' = c'' \downarrow \kappa'' & \\
[c/u : \kappa]a'' = c''
\end{align*}
\quad (25.6b)
\]
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.

\[
\frac{(u \neq u')} {[c/u : \kappa]u' = u'}
\]  

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.

\[
\frac{[c/u : \kappa]u = c \downarrow \kappa} {[c/u : \kappa]u = c \downarrow \kappa}
\]  

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.

\[
\frac{(u \neq u')} {[c/u : \kappa]u' = u'}
\]  

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.
Rule (25.8d) 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 $\kappa$, 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 $\kappa_2'$, 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

Control Effects
Chapter 27

Control Stacks

The technique of specifying the dynamic semantics as a transition system 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, $\mathcal{K}\{\text{nat} \rightarrow\}$, for the language $\mathcal{L}\{\text{nat} \rightarrow\}$. 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 $\mathcal{K}\{\text{nat} \rightarrow\}$ with the structural operational semantics of $\mathcal{L}\{\text{nat} \rightarrow\}$.

27.1 Machine Definition

A state, $s$, of $\mathcal{K}\{\text{nat} \rightarrow\}$ consists of a control stack, $k$, and a closed expression, $e$. States may take one of two forms:

1. An evaluation state of the form $k \triangleright e$ corresponds to the evaluation of a closed expression, $e$, relative to a control stack, $k$. 
2. A return state of the form $k \triangleleft e$, where $e$ 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:

\[
\begin{align*}
    \text{e stack} & \quad (27.1a) \\
    \text{f frame } k \text{ stack} & \quad (27.1b) \\
    k; f \text{ stack} & \quad (27.1b)
\end{align*}
\]

The definition of frame depends on the language we are evaluating. The frames of $K\{\text{nat} \to \}$ are inductively defined by the following rules:

\[
\begin{align*}
    s(-) \text{ frame} & \quad (27.2a) \\
    \text{ifz}(-; e_1; x.e_2) \text{ frame} & \quad (27.2b) \\
    \text{ap}(-; e_2) \text{ frame} & \quad (27.2c)
\end{align*}
\]

The frames correspond to rules with transition premises in the dynamic semantics of $L\{\text{nat} \to \}$. 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 $K\{\text{nat} \to \}$ is inductively defined by a set of inference rules. We begin with the rules for natural numbers.

\[
\begin{align*}
    k \triangleright z & \mapsto k \triangleleft z \quad (27.3a) \\
    k \triangleright s(e) & \mapsto k; s(-) \triangleright e \quad (27.3b) \\
    k; s(-) \triangleleft e & \mapsto k \triangleleft s(e) \quad (27.3c)
\end{align*}
\]

To evaluate $z$ we simply return it. To evaluate $s(e)$, we push a frame on the stack to record the pending successor, and evaluate $e$; when that returns with $e'$, we return $s(e')$ to the stack.
Next, we consider the rules for case analysis.

\[ k \triangleright \text{ifz}(e; e_1; x; e_2) \mapsto k; \text{ifz}(-; e_1; x; e_2) \triangleright e \]  
(27.4a)

\[ k; \text{ifz}(-; e_1; x; e_2) \triangleleft z \mapsto k \triangleright e_1 \]  
(27.4b)

\[ k; \text{ifz}(-; e_1; x; e_2) \triangleleft s(e) \mapsto k \triangleright [e/x]e_2 \]  
(27.4c)

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.

\[ k \triangleright \text{lam}[\tau](x; e) \mapsto k \triangleleft \text{lam}[\tau](x; e) \]  
(27.5a)

\[ k \triangleright \text{ap}(e_1; e_2) \mapsto k \triangleright \text{ap}(-; e_2) \triangleright e_1 \]  
(27.5b)

\[ k; \text{ap}(-; e_2) \triangleleft \text{lam}[\tau](x; e) \mapsto k \triangleright [e_2/x]e \]  
(27.5c)

\[ k \triangleright \text{fix}[\tau](x; e) \mapsto k \triangleright \text{fix}[\tau](x; e) \]  
(27.5d)

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 40 for more on evaluation of general recursion.)

The initial and final states of the K{nat \rightarrow} are defined by the following rules:

\[ e \triangleright e \text{ initial} \]  
(27.6a)

\[ e \text{ val} \]

\[ e \triangleleft e \text{ final} \]  
(27.6b)
27.2 Safety

To define and prove safety for $K\{\text{nat} \rightarrow \}$ requires that we introduce a new typing judgement, $k : \tau$, stating that the stack $k$ expects a value of type $\tau$. This judgement is inductively defined by the following rules:

\[\overline{e : \tau}\]  
\[
\left. \begin{array}{c}
k : \tau' \\
f : \tau \Rightarrow \tau'
\end{array} \right\} \overline{k ; f : \tau}
\]

(27.7a)

(27.7b)

This definition makes use of an auxiliary judgement, $f : \tau \Rightarrow \tau'$, stating that a frame $f$ transforms a value of type $\tau$ to a value of type $\tau'$.

\[\overline{s(\_): \text{nat} \Rightarrow \text{nat}}\]  
\[
\left. \begin{array}{c}
e_1 : \tau \\
x : \text{nat} \vdash e_2 : \tau
\end{array} \right\} \overline{\text{ifz}(\_ ; e_1 ; x . e_2) : \text{nat} \Rightarrow \tau}
\]

(27.8a)

(27.8b)

\[
\left. \begin{array}{c}
e_2 : \tau_2
\end{array} \right\} \overline{\text{ap}(\_ ; e_2) : \text{arr}(\tau_2 ; \tau) \Rightarrow \tau}
\]

(27.8c)

The two forms of $K\{\text{nat} \rightarrow \}$ state are well-formed provided that their stack and expression components match.

\[
\left. \begin{array}{c}
k : \tau \\
e : \tau
\end{array} \right\} \overline{k \triangleright e \text{ ok}}
\]

(27.9a)

\[
\left. \begin{array}{c}
k : \tau \\
e : \tau \text{ e val}
\end{array} \right\} \overline{k \triangleleft e \text{ ok}}
\]

(27.9b)

We leave the proof of safety of $K\{\text{nat} \rightarrow \}$ as an exercise.

**Theorem 27.1 (Safety).**

1. If $s \text{ ok and } s \mapsto s'$, then $s' \text{ ok}$.

2. If $s \text{ ok}$, then either $s \text{ final}$ or there exists $s'$ such that $s \mapsto s'$.
27.3 Correctness of the Control Machine

It is natural to ask whether $K\{\text{nat} \rightarrow \}$ correctly implements $L\{\text{nat} \rightarrow \}$. If we evaluate a given expression, $e$, using $K\{\text{nat} \rightarrow \}$, do we get the same result as would be given by $L\{\text{nat} \rightarrow \}$, and vice versa?

Answering this question decomposes into two conditions relating $K\{\text{nat} \rightarrow \}$ to $L\{\text{nat} \rightarrow \}$:

**Completeness** If $e \xrightarrow{\star} e'$, where $e' \text{ val}$, then $e \xrightarrow{\star} e'$.  

**Soundness** If $e \xrightarrow{\star} e' \text{ val}$, then $e \xrightarrow{\star} e'$.

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 $e \text{ val}$, then $e \xrightarrow{\star} e$.

2. If $e \xrightarrow{\star} e' \text{ val}$, then $e \xrightarrow{\star} e'$.

The first can be proved easily by induction on the structure of $e$. The second requires an inductive analysis of the derivation of $e \xrightarrow{\star} e'$, 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, $\text{ap}(e_1; e_2)$, then the first step of the machine is $k \xrightarrow{\star} _{} \text{ap}(e_1; e_2) \xrightarrow{\star} _{} \text{ap}(-; e_2) \xrightarrow{\star} _{} e_1$, and so we must consider evaluation of $e_1$ on a non-empty stack.

A natural generalization is to prove that if $e \xrightarrow{\star} e'$ and $k \xrightarrow{\star} _{} k' \text{ val}$, then $k \xrightarrow{\star} _{} k'$. Consider again the case $e = \text{ap}(e_1; e_2)$, $e' = \text{ap}(e_1'; e_2')$, with $e_1 \xrightarrow{\star} e_1'$. We are given that $k \xrightarrow{\star} _{} \text{ap}(e_1'; e_2) \xrightarrow{\star} _{} k' \text{ val}$, and we are to show that $k \xrightarrow{\star} _{} \text{ap}(e_1; e_2) \xrightarrow{\star} _{} k'$. It is easy to show that the first step of the former derivation is

$k \xrightarrow{\star} _{} \text{ap}(e_1'; e_2) \xrightarrow{\star} _{} k; \text{ap}(-; e_2) \xrightarrow{\star} _{} e_1$.  

We would like to apply induction to the derivation of $e_1 \xrightarrow{\star} e_1'$, but to do so we must have a $v_1$ such that $e_1' \xrightarrow{\star} v_1$, 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 semantics described in Chapter 12, which has the property that $e \downarrow e'$ iff $e \xrightarrow{\star} e'$ and $e'$ val.
Lemma 27.2. 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 12.2 on page 93 for $L\{\text{nat} \rightarrow\}$, 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 $e \triangleright e \mapsto^* e \triangleleft v$, because the intervening steps may involve alternations of evaluation and return states. Instead we regard each $K\{\text{nat} \rightarrow\}$ machine state as encoding an expression, and show that $K\{\text{nat} \rightarrow\}$ transitions are simulated by $L\{\text{nat} \rightarrow\}$ transitions under this encoding.

Specifically, we define a judgement, $s \leftrightarrow e$, stating that state $s$ “unravels to” expression $e$. It will turn out that for initial states, $s = e \triangleright e$, and final states, $s = e \triangleleft e$, we have $s \leftrightarrow e$. Then we show that if $s \mapsto^* s'$, where $s'$ final, $s \leftrightarrow e$, and $s' \leftrightarrow e'$, then $e'$ val and $e \mapsto^* e'$. For this it is enough to show the following two facts:

1. If $s \leftrightarrow e$ and $s$ final, then $e$ val.
2. If $s \mapsto^* s'$, $s \leftrightarrow e$, $s' \leftrightarrow e'$, 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 27.3. If $s \mapsto^* s'$, $s \leftrightarrow e$, and $s' \leftrightarrow e'$, then $e \mapsto^* e'$.

Corollary 27.4. $e \mapsto^* \overline{v}$ iff $e \triangleright e \mapsto^* e \triangleleft \overline{v}$.

The remainder of this section is devoted to the proofs of the soundness and completeness lemmas.

27.3.1 Completeness

Proof of Lemma 27.2. The proof is by induction on an evaluation semantics for $L\{\text{nat} \rightarrow\}$.

Consider the evaluation rule

$$
\frac{e_1 \Downarrow \lambda \tau_2 \ (x . e) \quad [e_2/x]e \Downarrow v}{\text{ap}(e_1; e_2) \Downarrow v} \quad (27.10)
$$

For an arbitrary control stack, $k$, we are to show that $k \triangleright \text{ap}(e_1; e_2) \mapsto^* k \triangleleft v$.

Applying both of the inductive hypotheses in succession, interleaved with
steps of the abstract machine, we obtain

\[
\begin{align*}
k \triangleright ap(e_1; e_2) & \mapsto k; ap(-; e_2) \triangleright e_1 \\
& \xrightarrow{\ast} k; ap(-; e_2) \triangleleft lam[t_2](x.e) \\
& \mapsto k \triangleright [e_2/x]e \\
& \xrightarrow{\ast} k \triangleleft v.
\end{align*}
\]

The other cases of the proof are handled similarly. \square

### 27.3.2 Soundness

The judgement \( s \leftarrow e' \), where \( s \) is either \( k \triangleright e \) or \( k \triangleleft e \), is defined in terms of the auxiliary judgement \( k \triangleright \triangleright e = e' \) by the following rules:

\[
\begin{align*}
k & \triangleright \triangleright e = e' \\
k & \triangleright e \leftarrow e'
\end{align*}
\]

(27.11a)

\[
\begin{align*}
k & \triangleright \triangleright e = e' \\
k & \triangleleft e \leftarrow e'
\end{align*}
\]

(27.11b)

In words, to unravel a state we wrap the stack around the expression. The latter relation is inductively defined by the following rules:

\[
k \triangleright \triangleright s(e) = e' \\
k \triangleright \triangleright ifz(e_1; e_2; x.e_3) = e'
\]

(27.12a)

\[
k \triangleright \triangleright ifz(-; e_2; x.e_3) \triangleright \triangleright e_1 = e' \\
k ; ap(e_1; e_2) = e \\
k ; ap(-; e_2) \triangleright \triangleright e_1 = e
\]

(27.12c)

(27.12d)

These judgements both define total functions.

**Lemma 27.5.** The judgement \( s \leftarrow e \) has mode \( (\forall, \exists!) \), and the judgement \( k \triangleright \triangleright e = e' \) 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 \( k \triangleright \triangleright e \) for the unique \( e' \) such that \( k \triangleright \triangleright e = e' \).

The following lemma is crucial. It states that unravelling preserves the transition relation.
Lemma 27.6. If $e \mapsto e'$, $k \triangleright e = d$, $k \triangleright e' = d'$, 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 $e = \text{ap}(e_1; e_2)$, $e' = \text{ap}(e'_1; e_2)$, and $e_1 \mapsto e'_1$. We have $k \triangleright e = d$ and $k \triangleright e' = d'$. It follows from Rules (27.12) that $k; \text{ap}(e_2) \triangleright e_1 = d$ and $k; \text{ap}(e_2) \triangleright e'_1 = d'$. So by induction $d \mapsto d'$, as desired.

For an example of a base case, suppose that $e = \text{ap}(\text{lam}[	au](x.e); e_2)$ and $e' = [e_2/x]e$ with $e \mapsto e'$ directly. Assume that $k \triangleright e = d$, 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 $k = k'; \text{ap}(\cdot; c_2)$. It follows from Rules (27.12) that $k' \triangleright \text{ap}(e; c_2) = d$ and $k' \triangleright \text{ap}(e'; c_2) = d'$. But by the SOS rules $\text{ap}(e; c_2) \mapsto \text{ap}(e'; c_2)$, 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 27.3 on page 242.

Proof of Lemma 27.3 on page 242. The proof is by case analysis on the transitions of $\mathcal{K}\{\text{nat} \rightarrow \}$. In each case after unravelling the transition will correspond to zero or one transitions of $\mathcal{L}\{\text{nat} \rightarrow \}$.

Suppose that $s = k \triangleright s(e)$ and $s' = k; s(\cdot) \triangleright e$. Note that $k \triangleright s(e) = e'$ iff $k; s(\cdot) \triangleright s(e) = e'$, from which the result follows immediately.

Suppose that $s = k; \text{ap}(\text{lam}[	au](x.e_1); \cdot) < e_2$ and $s' = k \triangleright [e_2/x]e_1$. Let $e'$ be such that $k; \text{ap}(\text{lam}[	au](x.e_1); \cdot) \triangleright e_2 = e'$ and let $e''$ be such that $k \triangleright [e_2/x]e_1 = e''$. Observe that $k \triangleright \text{ap}(\text{lam}[	au](x.e_1); e_2) = e'$. The result follows from Lemma 27.6.

\[\square\]

27.4 Exercises
Chapter 28

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. All too often explicit checks are omitted (by design or neglect), whereas exceptions cannot be ignored.

28.1 Failures

To begin with let us consider a simple control mechanism, which permits the evaluation of an expression to fail by passing control to the nearest enclosing handler, which is said to catch the failure. Failures are a simplified form of exception in which no value is associated with the failure. This allows us to concentrate on the control flow aspects, and to treat the associated value separately.

The following grammar describes an extension to \( \mathcal{L}\{\rightarrow\} \) to include failures:

\[
\begin{array}{llll}
\text{Category} & \text{Item} & \text{Abstract} & \text{Concrete} \\
\hline
\text{Expr} & e & ::= & \text{fail}[\tau] \quad \text{fail} \\
 & & & \text{catch}(e_1;e_2) \quad \text{try} \ e_1 \ \text{ow} \ e_2 \\
\end{array}
\]

The expression \( \text{fail}[\tau] \) aborts the current evaluation. The expression \( \text{catch}(e_1;e_2) \)
evaluates $e_1$. If it terminates normally, its value is returned; if it fails, its value is the value of $e_2$.

The static semantics of failures is quite straightforward:

$$\Gamma \vdash \text{fail}[\tau] : \tau$$

$$(28.1a)$$

$$\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau$$

$$\Gamma \vdash \text{catch}(e_1; e_2) : \tau$$

$$(28.1b)$$

Observe that a failure can have any type, because it never returns to the site of the failure. Both clauses of a handler must have the same type, to allow for either possible outcome of evaluation.

The dynamic semantics of failures uses a technique called stack unwinding. Evaluation of a \text{catch} installs a handler on the control stack. Evaluation of a \text{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.

This behavior is naturally specified using the abstract machine $\mathcal{K}\{\text{nat} \rightarrow\}$ from Chapter 27, because it makes the control stack explicit. We introduce a new form of state, $k \blacktriangleleft$, which passes a failure to the stack, $k$, in search of the nearest enclosing handler. A state of the form $\epsilon \blacktriangleleft$ is considered final, rather than stuck; it corresponds to an “uncaught failure” making its way to the top of the stack.

The set of frames is extended with the following additional rule:

$$e_2 \text{ exp}$$

$$\frac{\text{catch}(\_; e_2) \text{ frame}}{\text{catch}(\_; e_2)}$$

$$(28.2)$$

The transition rules of $\mathcal{K}\{\text{nat} \rightarrow\}$ are extended with the following additional rules:

$$k \triangleright \text{fail}[\tau] \mapsto k \blacktriangleleft$$

$$(28.3a)$$

$$k \triangleright \text{catch}(e_1; e_2) \mapsto k; \text{catch}(\_; e_2) \triangleright e_1$$

$$(28.3b)$$

$$k; \text{catch}(\_; e_2) \triangleleft v \mapsto k \triangleleft v$$

$$(28.3c)$$

$$k; \text{catch}(\_; e_2) \blacktriangleleft \mapsto k \triangleright e_2$$

$$(28.3d)$$

$$\left( f \neq \text{catch}(\_; e_2) \right)$$

$$k; f \blacktriangleleft \mapsto k \blacktriangleleft$$

$$(28.3e)$$
28.2 Exceptions

Evaluating \( \text{fail}[\tau] \) propagates a failure up the stack. Evaluating \( \text{catch}(e_1; e_2) \) 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 \( \mathcal{K}\{\text{nat} \rightarrow \} \), but we change the definition of final state to include these two forms:

\[
\frac{e \text{ val}}{e \leftarrow e \text{ final}} \quad (28.4a)
\]

\[
\frac{}{e \leftarrow \text{ final}} \quad (28.4b)
\]

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 to extend the definition of stack typing given in Chapter 27 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 28.1 (Safety).**

1. If \( s \text{ ok} \) and \( s \overset{s'}{\rightarrow} \), then \( s' \text{ ok} \).

2. If \( s \text{ ok} \), then either \( s \text{ final} \) or there exists \( s' \) such that \( s \overset{s'}{\rightarrow} \).

### 28.2 Exceptions

Let us now consider enhancing the simple failures mechanism of the preceding section with an exception mechanism that permits a value to be associated with the failure, which is then passed to the handler as part of the control transfer. The syntax of exceptions is given by the following grammar:

```
Category    Item       Abstract                     Concrete
Expr e ::= raise[\tau](e)  raise(e)               try e_1 ow x ⇒ e_2
          | handle(e_1; x. e_2)  try e_1 ow x ⇒ e_2
```

The argument to \( \text{raise} \) is evaluated to determine the value passed to the handler. The expression \( \text{handle}(e_1; x. e_2) \) 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 dynamic semantics of exceptions is a mild generalization of that of failures given in Section 28.1 on page 245. The failure state, \( k \triangleleft \), is extended to permit passing a value along with the failure, \( k \triangleleft e \), where \( e \text{ val} \). Stack frames include these two forms:

\[
\text{raise}[^\tau](\_) \text{ frame}
\] (28.5a)

\[
\text{handle}(\_;x.e_2) \text{ frame}
\] (28.5b)

The rules for evaluating exceptions are as follows:

\[
k \triangleright \text{raise}[^\tau](e) \mapsto k; \text{raise}[^\tau](\_) \triangleright e
\] (28.6a)

\[
k; \text{raise}[^\tau](\_) \triangleleft e \mapsto k \triangleleft e
\] (28.6b)

\[
k; \text{raise}[^\tau](\_ \triangleleft e) \mapsto k \triangleleft e
\] (28.6c)

\[
k \triangleright \text{handle}(e_1;x.e_2) \mapsto k; \text{handle}(\_;x.e_2) \triangleright e_1
\] (28.6d)

\[
k; \text{handle}(\_;x.e_2) \triangleleft e \mapsto k \triangleleft e
\] (28.6e)

\[
k; \text{handle}(\_;x.e_2) \triangleleft e \mapsto k \triangleright [e/x]e_2
\] (28.6f)

\[
f \neq \text{handle}(\_;x.e_2) \quad k; f \triangleleft e \mapsto k \triangleleft e
\] (28.6g)

The static semantics of exceptions generalizes that of failures.

\[
\Gamma \vdash e : \tau_{\text{exn}} \\
\Gamma \vdash \text{raise}[\tau](e) : \tau
\] (28.7a)

\[
\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{\text{exn}} \vdash e_2 : \tau \\
\Gamma \vdash \text{handle}(e_1;x.e_2) : \tau
\] (28.7b)

These rules are parameterized by the type of values associated with exceptions, \( \tau_{\text{exn}} \). But what should be the type \( \tau_{\text{exn}} \)?

The first thing to observe is that all exceptions should be of the same type, otherwise we cannot guarantee type safety. The reason is that a handler might be invoked by any raise expression occurring during the execution of the expression that it guards. If different exceptions could have
different associated values, the handler could not predict (statically) what type of value to expect, and hence could not dispatch on it without violating type safety.

The reason to associate data with an exception is to communicate to the handler some information about the use of the exceptional condition. But what should the type of this data be? A very naive suggestion might be to choose $\tau_{\text{exn}}$ to be the type $\text{str}$, so that, for example, one may write

$$\text{raise } "\text{Division by zero error.}"$$

to signal the obvious arithmetic fault. The trouble with this, of course, is that all information to be passed to the handler must be encoded as a string, and the handler must parse the string to recover that information!

Another all-too-familiar choice of $\tau_{\text{exn}}$ is the type $\text{nat}$. Exception conditions are encoded, by convention, as natural numbers.\footnote{In Unix these are called $\text{errno}$'s, for error numbers, with 0 being the number for “no error.”} This is obviously an impractical approach, since it requires that each system maintain a global assignment of numbers to error conditions, impeding or even precluding modular development. Moreover, the decoding of the error numbers is tedious and error prone. Surely there is a better way!

A more practical choice for $\tau_{\text{exn}}$ would be a distinguished labelled sum type of the form

$$\tau_{\text{exn}} = [\text{div:unit, fnf:string, ...}],$$

with one class for each exceptional condition and an associated data value of the type associated to that class in $\tau_{\text{exn}}$. This allows the handler to perform a simple symbolic case analysis on the class of the exception to recover the underlying data. For example, we might write

```haskell
try $e_1$ ow $x$ ⇒
  case $x$ {
    div ⟨⟩ ⇒ $e_{\text{div}}$
    | fnf $s$ ⇒ $e_{\text{fnf}}$
    | ... }
```

to recover from the exceptions specified in $\tau_{\text{exn}}$.

The chief difficulty with this approach is that, like error numbers, it requires a single global commitment to the type $\tau_{\text{exn}}$ that must be shared by all components of the program. This impedes separate development, and
requires all modules to be aware of all exceptions that may be raised anywhere within the program. The solution to this is to employ a dynamically extensible sum type for $\tau_{exn}$ that allows new classes to be generated from anywhere within the program in such a way that each component is assured to be allocated different classes from those generated elsewhere in the program.

Since extensible sums have application beyond serving as the type of exception values, we defer a detailed discussion to Chapter 36, which discusses them in isolation from exceptions.

### 28.3 Exercises
Chapter 29

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 first-class 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. First-class continuations never “expire”, and it is always sensible to reinstate a continuation without compromising safety. Thus first-class 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 first-class continuations useful? Fundamentally, they are representations of the control state of a computation at a given point in time. Using first-class 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.

29.1 Informal Overview

We will extend $\mathcal{L}\{\to\}$ with the type $\text{cont}(\tau)$ of continuations accepting values of type $\tau$. The introduction form for $\text{cont}(\tau)$ is $\text{letcc}[\tau](x.e)$, 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 \( \text{throw}[\tau](e_1; e_2) \), 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 \( \text{nat} \rightarrow \text{nat} \). 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 \( \mathcal{L}\{\text{nat} \rightarrow \} \), without short-cutting:

\[
\text{fix ms is} \\
\quad \lambda \ q : \text{nat} \rightarrow \text{nat}. \ \\
\quad \lambda \ n : \text{nat}. \ \\
\quad \text{case} \ n \ {\} \\
\quad \quad \quad z \Rightarrow s(z) \ \\
\quad \quad \mid s(n') \Rightarrow (q \ z) \times (ms \ (q \circ \text{succ}) \ n') \ \\
\}
\]

The recursive call composes \( q \) with the successor function to shift the sequence by one step.

Here is the version with short-cutting:

\[
\lambda \ q : \text{nat} \rightarrow \text{nat}. \ \\
\lambda \ n : \text{nat}. \ \\
\text{letcc ret : nat cont in} \ \\
\quad \text{let ms be} \ \\
\quad \quad \text{fix ms is} \ \\
\quad \quad \quad \lambda \ q : \text{nat} \rightarrow \text{nat}. \ \\
\quad \quad \quad \lambda \ n : \text{nat}. \ \\
\quad \quad \quad \text{case} \ n \ {\} \\
\quad \quad \quad \quad \quad z \Rightarrow s(z) \ \\
\quad \quad \quad \mid s(n') \Rightarrow \ \\
\quad \quad \quad \quad \quad \text{case} \ q \ z \ {\} \\
\quad \quad \quad \quad \quad \quad z \Rightarrow \text{throw} \ z \ \text{to ret} \ \\
\quad \quad \quad \quad \quad \mid s(n'') \Rightarrow (q \ z) \times (ms \ (q \circ \text{succ}) \ n') \ \\
\quad \quad \quad \} \ \\
\}
\]

\[ ms \ q \ n \]
The letcc binds the return point of the function to the variable ret for use within the main loop of the computation. If zero is encountered, control is thrown to ret, effecting an early return with the value zero.

Let’s look at another example: given a continuation k of type \( \tau \) \( \text{cont} \) and a function \( f \) of type \( \tau' \rightarrow \tau \), return a continuation \( k' \) of type \( \tau' \) \( \text{cont} \) with the following behavior: throwing a value \( v' \) of type \( \tau' \) to \( k' \) throws the value \( f(v') \) to \( k \). This is called composition of a function with a continuation. We wish to fill in the following template:

\[
\text{fun compose}(f: \tau' \rightarrow \tau, k: \tau \text{cont}) : \tau' \text{cont} = \ldots
\]

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 \( \text{throw } f(\ldots) \) 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

\[
\text{throw } f(\text{letcc } x: \tau' \text{cont in } \ldots) \text{ to } k
\]

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:

\[
\text{fun compose } (f: \tau' \rightarrow \tau, k: \tau \text{cont}) : \tau' \text{cont} = \\
\text{letcc } \text{ret: } \tau' \text{cont cont in } \\
\text{throw } (f(\text{letcc } r \text{ in } \text{throw } r \text{ to } \text{ret})) \text{ to } k
\]

The type of \( \text{ret} \) is that of a continuation-expecting continuation!

29.2 Semantics of Continuations

We extend the language of \( \mathcal{L}\{\rightarrow\} \) expressions with these additional forms:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau )</td>
<td>( \text{cont} (\tau) )</td>
<td>( \tau \text{ cont} )</td>
</tr>
<tr>
<td>Expr</td>
<td>( e )</td>
<td>( \text{letcc} [r] (x. e) )</td>
<td>( \text{letcc } x \text{ in } e )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{throw} [r] (e_1; e_2) )</td>
<td>( \text{throw } e_1 \text{ to } e_2 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{cont} (k) )</td>
<td></td>
</tr>
</tbody>
</table>

The expression \( \text{cont} (k) \) is a reified control stack; they arise during evaluation, but are not available as expressions to the programmer.

OCTOBER 16, 2009 DRAFT 18:42
The static semantics of continuations is defined by the following rules:

\[
\Gamma, x : \text{cont}(\tau) \vdash e : \tau \\
\Gamma \vdash \text{letcc}[\tau](x.e) : \tau
\]  
(29.1a)

\[
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \text{cont}(\tau_1) \\
\Gamma \vdash \text{throw}[\tau'](e_1;e_2) : \tau'
\]  
(29.1b)

The result type of a throw expression is arbitrary because it does not return to the point of the call.

The static semantics of continuation values is given by the following rule:

\[
k : \tau \\
\Gamma \vdash \text{cont}(k) : \text{cont}(\tau)
\]  
(29.2)

A continuation value \text{cont}(k) has type \text{cont}(\tau) exactly if it is a stack accepting values of type \tau.

To define the dynamic semantics, we extend \(K\{\text{nat} \rightarrow\}\) stacks with two new forms of frame:

\[
\begin{align*}
ev_2 & \quad \text{exp} \\
\text{throw}[\tau](\_;e_2) & \quad \text{frame}
\end{align*}
\]  
(29.3a)

\[
\begin{align*}
ev_1 & \quad \text{val} \\
\text{throw}[\tau](e_1;\_) & \quad \text{frame}
\end{align*}
\]  
(29.3b)

Every reified control stack is a value:

\[
k \quad \text{stack} \\
\text{cont}(k) & \quad \text{val}
\]  
(29.4)

The transition rules for the continuation constructs are as follows:

\[
k \triangleright \text{letcc}[\tau](x.e) \iff k \triangleright [\text{cont}(k)/x]e
\]  
(29.5a)

\[
k ; \text{throw}[\tau](v;\_) \triangleleft \text{cont}(k') \iff k' \triangleleft v
\]  
(29.5b)

\[
k ; \text{throw}[\tau](e_1;e_2) \triangleright k ; \text{throw}[\tau](\_;e_2) \triangleright e_1
\]  
(29.5c)

\[
ev_1 \quad \text{val} \\
k ; \text{throw}[\tau](\_;e_2) \triangleleft e_1 \iff k ; \text{throw}[\tau](e_1;\_) \triangleright e_2
\]  
(29.5d)

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 $L\{\to\}$ may be established by a simple extension to the safety proof for $K\{\text{nat}\to\}$ given in Chapter 27.

We need only add typing rules for the two new forms of frame, which are as follows:

\[
\frac{e_2 : \text{cont}(\tau)}{\text{throw}[\tau](\_; e_2) : \tau \Rightarrow \tau'} \quad (29.6a)
\]
\[
\frac{e_1 : \tau \quad e_1 \text{ val}}{\text{throw}[\tau](e_1; \_): \text{cont}(\tau) \Rightarrow \tau'} \quad (29.6b)
\]

The rest of the definitions remain as in Chapter 27.

**Lemma 29.1** (Canonical Forms). If $e : \text{cont}(\tau)$ and $e \text{ val}$, then $e = \text{cont}(k)$ for some $k$ such that $k : \tau$.

**Theorem 29.2** (Safety). 1. If $s \text{ ok}$ and $s \mapsto s'$, then $s' \text{ ok}$.

2. If $s \text{ ok}$, then either $s \text{ final}$ or there exists $s'$ such that $s \mapsto s'$.

### 29.3 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 sub-routine 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

\[ \text{state} \cong (\tau \times \text{state}) \text{cont}, \]

where \( \tau \) is the type of data exchanged by the routines. The solution to such an isomorphism is, of course, the recursive type

\[ \text{state} = \mu t. (\tau \times t) \text{cont}. \]

Thus a state, \( s \), encapsulates a pair consisting of a value of type \( \tau \) together with another state.

The routines pass control from one to the other by calling the function \( \text{resume} \) of type

\[ \tau \times \text{state} \rightarrow \tau \times \text{state}. \]

That is, given a datum, \( d \), and a state, \( s \), the application \( \text{resume}(\langle d, s \rangle) \) passes \( d \) and its own return address to the routine represented by the state \( s \). The function \( \text{resume} \) is defined by the following expression:

\[ \lambda (\langle x, s \rangle: \tau \times \text{state}). \text{letcc } k \text{ in } \text{throw}(x, \text{fold}(k)) \text{ to } \text{unfold}(s) \]

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 \((\tau \times \text{state}) \text{cont}\).

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 \( \text{corout} \) builds a coroutine from a data transformation routine; it has type

\[ (\tau \rightarrow \tau) \rightarrow (\tau \times \text{state}) \rightarrow \tau'. \]

The result type, \( \tau' \), 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 \( \text{corout} \) is defined by the following expression (with types omitted for concision):

\[ \lambda \text{next}. \text{fix } \text{loop } \text{is } \lambda \langle d, s \rangle. \text{loop}(\text{resume}(\langle \text{next}(d), s \rangle)). \]

Each time through the loop, the partner routine, \( s \), is resumed with the updated datum given by applying \( \text{next} \) to the current datum, \( d \).
Let \( \rho \) be the ultimate type of a computation consisting of two interacting coroutines that exchanges values of type \( \tau \) during their execution. The function \( \text{run} \), which has type

\[
\tau \to ((\rho \text{ cont} \to \tau \to \tau) \times (\rho \text{ cont} \to \tau \to \tau)) \to \rho,
\]
takes an initial value of type \( \tau \) and two routines, each of type

\[
\rho \text{ cont} \to \tau \to \tau,
\]
and builds a coroutine of type \( \rho \) from them. The first argument to each routine is the exit point, and the result is a data transformation operation. The definition of \( \text{run} \) begins as follows:

\[
\lambda \text{init}. \lambda (r_1, r_2). \text{letcc exit in let } r'_1 \text{ be } r_1(\text{exit}) \text{ in let } r'_2 \text{ be } r_2(\text{exit}) \text{ in } \ldots
\]

First, \( \text{run} \) 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 \( \text{exit} \). The implementation of \( \text{run} \) continues as follows:

\[
\text{corout}(r'_2)(\text{letcc } k \text{ in corout}(r'_1)((\text{init}, \text{fold}(k))))
\]

The routine \( r'_1 \) is called with the initial datum, \( \text{init} \), and the state \( \text{fold}(k) \), where \( k \) is the continuation corresponding to the call to \( r'_2 \). The first \( \text{resume} \) 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 \( \text{resume} \). Either may terminate the computation by throwing a result value, \( v \), of type \( \rho \) to the continuation \( \text{exit} \).

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 \textit{producer} routine and a \textit{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 \( \tau_i \text{ list} \) and \( \tau_o \text{ list} \), 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 \( \text{OK}(⟨i, o⟩) \) 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 \( \text{EMIT}(⟨v, ⟨i, o⟩⟩) \), where \( v \) is a value of type \( τ_i \text{opt} \), 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

\[
[\text{OK}: τ_i \text{list} \times τ_o \text{list}, \text{EMIT}: τ_i \text{opt} \times (τ_i \text{list} \times τ_o \text{list})].
\]

This type specifies the message protocol between the producer and the consumer described in the preceding paragraph.

The producer, \( \text{producer} \), is defined by the expression

\[
λ\text{exit}. λ\text{msg}. \text{case } \text{msg} \{ b_1 | b_2 | b_3 \},
\]

where the first branch, \( b_1 \), is

\[
\text{OK} \cdot ⟨\text{nil}, o⟩ \Rightarrow \text{EMIT} \cdot ⟨\text{null}, ⟨\text{nil}, o⟩⟩
\]

and the second branch, \( b_2 \), is

\[
\text{OK} \cdot ⟨\text{cons}(i; is), o⟩ \Rightarrow \text{EMIT} \cdot ⟨\text{just}(i), ⟨is, o⟩⟩,
\]

and the third branch, \( b_3 \), is

\[
\text{EMIT} \cdot _\_ \Rightarrow \text{error}.
\]

In words, if the input is exhausted, the producer emits the value \( \text{null} \), along with the current channel state. Otherwise, it emits \( \text{just}(i) \), where \( i \) is the first remaining input, and removes that element from the passed channel state. The producer cannot see an \( \text{EMIT} \) message, and signals an error if it should occur.

The consumer, \( \text{consumer} \), is defined by the expression

\[
λ\text{exit}. λ\text{msg}. \text{case } \text{msg} \{ b'_1 | b'_2 | b'_3 \},
\]

where the first branch, \( b'_1 \), is

\[
\text{EMIT} \cdot ⟨\text{null}, ⟨_, o⟩⟩ \Rightarrow \text{throw } o \to \text{exit},
\]
the second branch, \( b'_2 \), is

\[
\text{EMIT} \cdot \langle \text{just}(i), \langle i, os \rangle \rangle \Rightarrow \text{OK} \cdot \langle is, \text{cons}(f(i); os) \rangle,
\]

and the third branch, \( b'_3 \), is

\[
\text{OK} \cdot \_ \Rightarrow \text{error}.
\]

The consumer dispatches on the emitted datum. If it is absent, the output channel state is passed to \textit{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 \( \text{OK} \) is received, the consumer signals an error, as the producer never produces such a message.

The initial datum, \( \text{init} \), has the form \( \text{OK} \cdot \langle i, os \rangle \), where \( is \) and \( os \) are the initial input and output channel state, respectively. The computation is created by the expression

\[
\text{run}(\text{init})(\langle \text{producer}, \text{consumer} \rangle),
\]

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 \textit{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 \textit{threads}. A thread \textit{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 \textit{cooperative multi-threading}, since it is based on explicit yields, rather than implicit yields imposed by asynchronous events such as timer interrupts.

### 29.4 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

\[
\varepsilon;\text{throw}[\tau](-;k);\text{ap}(f;-)
\]

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 X

Types and Propositions
Chapter 30

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. Static semantics relates a program to the type it implements, and a dynamic semantics 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 constructive logic. In Chapter 31 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.

### 30.1 Constructive Semantics

Constructive logic is concerned with two judgements, $\phi$ prop, stating that $\phi$ expresses a proposition, and $\phi$ true, stating that $\phi$ is a true proposition. What distinguishes constructive from non-constructive logic is that a proposition is not conceived of as merely a truth value, but instead as a problem statement whose solution, if it has one, is given by a proof. A proposition is said to be true exactly when it has a proof, in keeping with ordinary mathematical practice. There is no other criterion of truth than the existence of a proof.

This principle has important, possibly surprising, consequences, the most important of which is that we cannot say, in general, that a proposition is either true or false. If for a proposition to be true means to have a proof of it, what does it mean for a proposition to be false? It means that we have a refutation of it, showing that it cannot be proved. That is, a proposition is false if we can show that the assumption that it is true (has a proof) contradicts known facts. In this sense constructive logic is a logic of positive, or affirmative, information — we must have explicit evidence in the form of a proof in order to affirm the truth or falsity of a proposition.

In light of this it should be clear that not every proposition is either true or false. For if $\phi$ expresses an unsolved problem, such as the famous $P \neq NP$ 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
30.2 Constructive Logic

Considered to be either true or false. For example, if \( \phi \) expresses an inequality between natural numbers, then \( \phi \) is decidable, because we can always work out, for given natural numbers \( m \) and \( n \), whether \( m \leq n \) or \( m \not\leq n \) — we can either prove or refute the given inequality. This argument does not extend to the real numbers. To get an idea of why not, consider the presentation of a real number by its decimal expansion. At any finite time we will have explored only a finite initial segment of the expansion, which is not enough to determine if it is, say, less than 1. For if we have determined the expansion to be 0.99...9, 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.

30.2 Constructive Logic

The judgements \( \phi \) prop and \( \phi \) true of constructive logic are rarely of interest by themselves, but rather in the context of a hypothetical judgement of the form

\[
\phi_1 \text{ true}, \ldots, \phi_n \text{ true} \vdash \phi \text{ true}.
\]

This judgement expresses that the proposition \( \phi \) is true (has a proof), \textit{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 categorical judgement \( \phi \) true.

The structural properties of the hypothetical judgement, when specialized to constructive logic, define what we mean by reasoning under hypotheses:

\[
\Gamma, \phi \text{ true} \vdash \phi \text{ true} \quad (30.1a)
\]

\[
\Gamma \vdash \phi \text{ true} \quad \Gamma, \phi \text{ true} \vdash \psi \text{ true} \quad (30.1b)
\]

\[
\Gamma \vdash \psi \text{ true} \quad (30.1c)
\]

\[
\Gamma, \phi \text{ true} \vdash \psi \text{ true} \quad (30.1d)
\]
The last two rules are implicit in that we regard $\Gamma$ as a *set* of hypotheses, so that two "copies" are as good as one, and the order of hypotheses does not matter.

### 30.2.1 Rules of Provability

The syntax of propositional logic is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prop</td>
<td>$\phi$</td>
<td>$\text{true}$</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{false}$</td>
<td>$\bot$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{and}(\phi_1;\phi_2)$</td>
<td>$\phi_1 \land \phi_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{or}(\phi_1;\phi_2)$</td>
<td>$\phi_1 \lor \phi_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{imp}(\phi_1;\phi_2)$</td>
<td>$\phi_1 \supset \phi_2$</td>
</tr>
</tbody>
</table>

The connectives of propositional logic (truth, falsehood, conjunction, disjunction, and implication) 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.

$$\Gamma \vdash \top \text{ true} \quad (30.2a)$$

*(no elimination rule)*

$$\Gamma \vdash \top \text{ true} \quad (30.2b)$$
30.2 Constructive Logic

Conjunction   Conjunction expresses the truth of both of its conjuncts.

\[
\Gamma \vdash \phi \text{ true} \quad \Gamma \vdash \psi \text{ true} \\
\hline
\Gamma \vdash \phi \land \psi \text{ true}
\] (30.3a)

\[
\Gamma \vdash \phi \land \psi \text{ true} \\
\hline
\Gamma \vdash \phi \text{ true}
\] (30.3b)

\[
\Gamma \vdash \phi \land \psi \text{ true} \\
\hline
\Gamma \vdash \psi \text{ true}
\] (30.3c)

Implication  Implication states the truth of a proposition under an assumption.

\[
\Gamma, \phi \text{ true} \vdash \psi \text{ true} \\
\hline
\Gamma \vdash \phi \vdash \psi \text{ true}
\] (30.4a)

\[
\Gamma \vdash \phi \vdash \psi \text{ true} \\
\hline
\Gamma \vdash \phi \text{ true} \\
\hline
\Gamma \vdash \psi \text{ true}
\] (30.4b)

Falsehood   Falsehood expresses the trivially false (refutable) proposition.

\[(no \ introduction \ rule)\] (30.5a)

\[
\Gamma \vdash \bot \text{ true} \\
\hline
\Gamma \vdash \phi \text{ true}
\] (30.5b)

Disjunction  Disjunction expresses the truth of either (or both) of two propositions.

\[
\Gamma \vdash \phi \text{ true} \\
\hline
\Gamma \vdash \phi \lor \psi \text{ true}
\] (30.6a)

\[
\Gamma \vdash \psi \text{ true} \\
\hline
\Gamma \vdash \phi \lor \psi \text{ true}
\] (30.6b)

\[
\Gamma \vdash \phi \lor \psi \text{ true} \\
\hline
\Gamma, \phi \text{ true} \vdash \theta \text{ true} \\
\hline
\Gamma, \psi \text{ true} \vdash \theta \text{ true} \\
\hline
\Gamma \vdash \theta \text{ true}
\] (30.6c)
Negation The negation, $\neg \phi$, of a proposition, $\phi$, may be defined as the implication $\phi \supset \bot$. This means that $\neg \phi$ true if $\phi$ true $\vdash \bot$ true, which is to say that the truth of $\phi$ is refutable in that we may derive a proof of falsehood from any purported proof of $\phi$. Because constructive truth is identified with the existence of a proof, the implied semantics of negation is rather strong. In particular, a problem, $\phi$, is open exactly when we can neither affirm nor refute it. This is in contrast to the classical conception of truth, which assigns a fixed truth value to each proposition, so that every proposition is either true or false.

30.2.2 Rules of Proof

The key to the propositions-as-types principle is to make explicit the forms of proof. The categorical judgement $\phi$ true, which states that $\phi$ has a proof, is replaced by the judgement $p : \phi$, stating that $p$ is a proof of $\phi$. (Sometimes $p$ is called a “proof term”, but we will simply call $p$ a “proof.”) The hypothetical 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 $\Gamma$ 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.

$$\Gamma \vdash \text{trueI} : \top \quad (30.7a)$$

$$\Gamma \vdash p : \phi \quad \Gamma \vdash q : \psi \quad \Gamma \vdash \text{andI}(p;q) : \phi \land \psi \quad (30.7b)$$

$$\Gamma \vdash p : \phi \land \psi \quad \Gamma \vdash \text{andE}[1](p) : \phi \quad (30.7c)$$

$$\Gamma \vdash p : \phi \land \psi \quad \Gamma \vdash \text{andE}[r](p) : \psi \quad (30.7d)$$

$$\Gamma, x : \phi \vdash p : \psi \quad \Gamma \vdash \text{impI}[\phi](x.p) : \phi \supset \psi \quad (30.7e)$$

$$\Gamma \vdash p : \phi \supset \psi \quad \Gamma \vdash q : \phi \quad \Gamma \vdash \text{impE}(p;q) : \psi \quad (30.7f)$$
30.3 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, $\phi$, to a type $\phi^*$:

<table>
<thead>
<tr>
<th>Proposition</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\top$</td>
<td>unit</td>
</tr>
<tr>
<td>$\bot$</td>
<td>void</td>
</tr>
<tr>
<td>$\phi \land \psi$</td>
<td>$\phi^* \times \psi^*$</td>
</tr>
<tr>
<td>$\phi \lor \psi$</td>
<td>$\phi^* \rightarrow \psi^*$</td>
</tr>
<tr>
<td>$\phi \lor \psi$</td>
<td>$\phi^* + \psi^*$</td>
</tr>
</tbody>
</table>

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 elim-
30.3 Propositions as Types

inatory forms of proofs and programs as well:

<table>
<thead>
<tr>
<th>Proof</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>trueI</td>
<td>triv</td>
</tr>
<tr>
<td>falseE<a href="p">φ</a></td>
<td>abort<a href="p%E2%88%97">φ∗</a></td>
</tr>
<tr>
<td>andI(p;q)</td>
<td>pair(p∗;q∗)</td>
</tr>
<tr>
<td>andE<a href="p">l</a></td>
<td>proj<a href="p%E2%88%97">l</a></td>
</tr>
<tr>
<td>andE<a href="p">r</a></td>
<td>proj<a href="p%E2%88%97">r</a></td>
</tr>
<tr>
<td>impI<a href="x.p">φ</a></td>
<td>lam<a href="x.p%E2%88%97">φ∗</a></td>
</tr>
<tr>
<td>impE(p;q)</td>
<td>ap(p∗;q∗)</td>
</tr>
<tr>
<td>orI[l]<a href="p">ψ</a></td>
<td>in[l]<a href="p%E2%88%97">ψ∗</a></td>
</tr>
<tr>
<td>orI[r]<a href="p">φ</a></td>
<td>in[r]<a href="p%E2%88%97">φ∗</a></td>
</tr>
<tr>
<td>orE<a href="p;x.q;y.r">φ;ψ</a></td>
<td>case(p∗;x.q∗;y.r∗)</td>
</tr>
</tbody>
</table>

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 30.1.

1. If φ prop, then φ∗ type

2. If Γ ⊢ p : φ, then Γ∗ ⊢ p∗ : φ∗.

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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>andE<a href="andI(p;q)">l</a></td>
<td>p</td>
</tr>
<tr>
<td>andE<a href="andI(p;q)">r</a></td>
<td>q</td>
</tr>
<tr>
<td>impE(impI<a href="x.q">φ</a>;p)</td>
<td>[p/x]q</td>
</tr>
<tr>
<td>orE<a href="orI%5Bl%5D%5B%CF%88%5D(p);x.q;y.r">φ;ψ</a></td>
<td>[p/x]q</td>
</tr>
<tr>
<td>orE<a href="orI%5Br%5D%5B%CF%86%5D(p);x.q;y.r">φ;ψ</a></td>
<td>[p/y]r</td>
</tr>
</tbody>
</table>

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 ⟨e₁,e₂⟩·1 ≡ e₁, 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.

### 30.4 Exercises
Chapter 31

Classical Logic

In Chapter 30 we saw that constructive logic is a logic of positive information in that the meaning of the judgement \( \phi \text{ true} \) is that there exists a proof of \( \phi \). A refutation of a proposition \( \phi \) consists of evidence for the hypothetical judgement \( \phi \text{ true} \vdash \bot \text{ true} \), asserting that the assumption of \( \phi \) leads to a contradiction. A proposition, \( \phi \), is said to be decidable iff either it, or its negation, is true. If truth is identified with possession of a proof, then not all propositions are decidable, for there are, and always will be, open problems for which we have neither a proof nor a refutation. That is, we cannot, for general \( \phi \), expect to have evidence for the judgement \( \phi \lor \neg \phi \text{ true} \), which is called the law of the excluded middle.

In contrast classical logic (the one we all learned in school) maintains a complete symmetry between truth and falsehood—that which is not true is false, and that which is not false is true. This amounts to the supposition that every proposition is decidable, from which it follows that classical truth does not imply possession of a proof, at least not by us finite beings. Instead, one may consider it to be “god’s view” of mathematics, in which the truth or falsity of every proposition is fully determined, rather than the “mortal’s view” that we are stuck with here on earth.

What is surprising is that the “absolutist” view of truth and falsehood inherent in classical logic is not, after all, at odds with a computational interpretation, provided that we are willing to accept a weaker interpretation of the computational content of proofs. Just as for constructive logic, evidence for \( \phi \) false in classical logic amounts to a proof that the assumption that \( \phi \) true leads to a contradiction. Rather than requiring that evidence for \( \phi \) true amount to a positive verification of \( \phi \), we instead settle for that the assumption of \( \phi \) false leads to a contradiction. If we do, in fact, have pos-
itive evidence for $\phi$ true, then obviously the assumption that $\phi$ false leads directly to a contradiction. The converse, however, holds only in limited cases (when $\phi$ is constructively decidable), which means that classical logic is, in general, weaker than constructive logic (that is, constructive logic is stronger than classical logic).

It follows that, classically, the law of the excluded middle holds, because it amounts to the assertion that $\phi$ true and $\phi$ false together entail a contradiction. The classical interpretation of the law is that “you cannot have it both ways”, which is rather different from its constructive interpretation, which says that “it must be one way or the other.” Open problems contradict the latter, but are entirely consistent with the former—an open problem is one for which we have neither a proof nor a refutation, not one for which we have both!

### 31.1 Classical Logic

The rules for the propositional connectives divide into two parts, those specifying its truth conditions and those specifying its falsity conditions. The rules for truth correspond to the introduction rules of constructive logic, and the rules for falsity correspond to the elimination rules. The symmetry between truth and falsity is expressed by the principle of indirect proof. To show that $\phi$ true it is enough to show that $\phi$ false entails a contradiction, and, conversely, to show that $\phi$ false it is enough to show that $\phi$ true leads to a contradiction. The second of these principles is constructively valid (indeed, one may regard it as the definition of falsity), but the former is the chief characteristic of classical logic, namely the principle of indirect proof.

#### Provability Rules

Classical logic is concerned with three basic judgement forms:

1. $\phi$ true, stating that proposition $\phi$ is true;
2. $\phi$ false, stating that proposition $\phi$ is false;
3. $\#$, stating a contradiction.

The rules of provability for classical logic are phrased in terms of 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. We write \( \Gamma \) for the collection of “truth” hypotheses, and \( \Delta \) for the collection of “false” hypotheses.

A contradiction arises whenever a proposition may be shown to be both true and false:

\[
\Delta \Gamma \vdash \phi \text{ false} \quad \Delta \Gamma \vdash \phi \text{ true} \\
\overline{\Delta \Gamma \vdash \#} \quad \text{(31.1a)}
\]

The hypothetical judgement is reflexive:

\[
\overline{\Delta, \phi \text{ false} \vdash \phi \text{ false}} \quad \text{(31.1b)}
\]

\[
\overline{\Delta, \phi \text{ true} \vdash \phi \text{ true}} \quad \text{(31.1c)}
\]

All propositions are either true or false:

\[
\overline{\Delta, \phi \text{ false} \vdash \#} \quad \text{(31.1d)}
\]

\[
\overline{\Delta, \phi \text{ true} \vdash \phi \text{ true}} \quad \text{(31.1c)}
\]

Truth is trivially true, and cannot be refuted.

\[
\overline{\Delta \Gamma \vdash \top \text{ true}} \quad \text{(31.1f)}
\]

Falsity is trivially false, and cannot be proved.

\[
\overline{\Delta \Gamma \vdash \bot \text{ false}} \quad \text{(31.1g)}
\]

A conjunction is true if both conjuncts are true, and is false if either conjunct is false.

\[
\Delta \Gamma \vdash \phi \text{ true} \quad \Delta \Gamma \vdash \psi \text{ true} \\
\overline{\Delta \Gamma \vdash \phi \wedge \psi \text{ true}} \quad \text{(31.1h)}
\]

\[
\overline{\Delta \Gamma \vdash \phi \text{ false}} \quad \text{(31.1i)}
\]

\[
\overline{\Delta \Gamma \vdash \psi \text{ false}} \quad \text{(31.1j)}
\]

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.

\[
\overline{\Delta, \phi \text{ true} \vdash \psi \text{ true}} \quad \text{(31.1k)}
\]
\[ \Delta \Gamma \vdash \phi \ \text{true} \quad \Delta \Gamma \vdash \psi \ \text{false} \]
\[ \Delta \Gamma \vdash \phi \supset \psi \ \text{false} \]

(31.1l)

A disjunction is true if either disjunct is true, and is false if both disjuncts are false.

\[ \Delta \Gamma \vdash \phi \ \text{true} \]
\[ \Delta \Gamma \vdash \phi \lor \psi \ \text{true} \]

(31.1m)

\[ \Delta \Gamma \vdash \psi \ \text{true} \]
\[ \Delta \Gamma \vdash \phi \lor \psi \ \text{true} \]

(31.1n)

\[ \Delta \Gamma \vdash \phi \ \text{false} \quad \Delta \Gamma \vdash \psi \ \text{false} \]
\[ \Delta \Gamma \vdash \phi \lor \psi \ \text{false} \]

(31.1o)

A negation is true if the negated proposition is false, and is false if it is true.

\[ \Delta \Gamma \vdash \phi \ \text{false} \]
\[ \Delta \Gamma \vdash \neg \phi \ \text{true} \]

(31.1p)

\[ \Delta \Gamma \vdash \phi \ \text{true} \]
\[ \Delta \Gamma \vdash \neg \phi \ \text{false} \]

(31.1q)

The following analogues of the elimination rules of constructive logic are derivable in classical logic:

\[ \Delta \Gamma \vdash \bot \ \text{true} \]
\[ \Delta \Gamma \vdash \phi \ \text{true} \]

(31.2a)

\[ \Delta \Gamma \vdash \phi \land \psi \ \text{true} \]
\[ \Delta \Gamma \vdash \phi \ \text{true} \]

(31.2b)

\[ \Delta \Gamma \vdash \phi \land \psi \ \text{true} \]
\[ \Delta \Gamma \vdash \psi \ \text{true} \]

(31.2c)

\[ \Delta \Gamma \vdash \phi \lor \psi \ \text{true} \]
\[ \Delta \Gamma, \phi \ \text{true} \vdash \gamma \ \text{true} \]
\[ \Delta \Gamma, \psi \ \text{true} \vdash \gamma \ \text{true} \]
\[ \Delta \Gamma \vdash \gamma \ \text{true} \]

(31.2d)

\[ \Delta \Gamma \vdash \phi \supset \psi \ \text{true} \]
\[ \Delta \Gamma \vdash \phi \ \text{true} \]
\[ \Delta \Gamma \vdash \psi \ \text{true} \]

(31.2e)

\[ \Delta \Gamma \vdash \neg \phi \ \text{true} \]
\[ \Delta \Gamma \vdash \phi \ \text{true} \]
\[ \Delta \Gamma \vdash \gamma \ \text{true} \]

(31.2f)

The proof that these are derivable is deferred to the next section, wherein we introduce syntax for proofs.
Proof Rules

The three provability judgement forms of classical logic may be re-formulated to give an explicit syntax for proofs, refutations, and contradictions:

1. $p : \phi$, stating that $p$ is a proof of $\phi$;
2. $k \div \phi$, stating that $k$ is a refutation of $\phi$;
3. $k \# p$, stating that $k$ and $p$ are contradictory.

The rules for formation of proofs are phrased in terms of hypothetical judgements of the form

$$\Delta, u \div \phi_1, \ldots, u_m \div \phi_m \vdash x_1 : \psi_1, \ldots, x_n : \psi_n \vdash J,$$

where $J$ is any of the three preceding basic judgements.

A contradiction arises whenever a proposition may be shown to be both true and false:

$$\Delta \Gamma \vdash k \div \phi \quad \Delta \Gamma \vdash p : \phi \quad \Delta \Gamma \vdash k \# p \tag{31.3a}$$

The syntax of a contradiction makes clear that it consists of a proof together with a refutation of the same proposition.

Reflexivity corresponds to the use of a hypothesis:

$$\Delta, u \div \phi \Gamma \vdash u \div \phi \tag{31.3b}$$

$$\Delta \Gamma, x : \phi \vdash x : \phi \tag{31.3c}$$

All propositions are either true or false:

$$\Delta, u \div \phi \Gamma \vdash k \# p$$

$$\Delta \Gamma \vdash \text{ccr}(u \div \phi ; k \# p) : \phi \tag{31.3d}$$

$$\Delta \Gamma, x : \phi \vdash k \# p$$

$$\Delta \Gamma \vdash \text{ccp}(x : \phi ; k \# p) \div \phi \tag{31.3e}$$

Truth is trivially true, and cannot be refuted:

$$\Delta \Gamma \vdash \langle \rangle : \top \tag{31.3f}$$
31.1 Classical Logic

Falsity is trivially false, and cannot be proved.

$$\Delta \Gamma \vdash \text{abort} \downarrow \bot$$  \hspace{1cm} (31.3g)

A conjunction is true if both conjuncts are true, and is false if either conjunct is false.

$$\Delta \Gamma \vdash p : \phi \quad \Delta \Gamma \vdash q : \psi$$  \hspace{1cm} \(\Delta \Gamma \vdash \langle p, q \rangle : \phi \land \psi\)  \hspace{1cm} (31.3h)

$$\Delta \Gamma \vdash k : \phi$$  \hspace{1cm} \(\Delta \Gamma \vdash \text{fst} ; k : \phi \land \psi\)  \hspace{1cm} (31.3i)

$$\Delta \Gamma \vdash k : \psi$$  \hspace{1cm} \(\Delta \Gamma \vdash \text{snd} ; k : \phi \land \psi\)  \hspace{1cm} (31.3j)

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.

$$\Delta \Gamma, x : \phi \vdash p : \psi$$  \hspace{1cm} \(\Delta \Gamma \vdash \lambda (x : \phi). p) : \phi \supset \psi\)  \hspace{1cm} (31.3k)

$$\Delta \Gamma \vdash p : \phi \quad \Delta \Gamma \vdash k : \psi$$  \hspace{1cm} \(\Delta \Gamma \vdash \text{app}(p) ; k : \phi \supset \psi\)  \hspace{1cm} (31.3l)

A disjunction is true if either disjunct is true, and is false if both disjuncts are false.

$$\Delta \Gamma \vdash p : \phi$$  \hspace{1cm} \(\Delta \Gamma \vdash \text{inl}(p) : \phi \lor \psi\)  \hspace{1cm} (31.3m)

$$\Delta \Gamma \vdash p : \psi$$  \hspace{1cm} \(\Delta \Gamma \vdash \text{inr}(p) : \phi \lor \psi\)  \hspace{1cm} (31.3n)

$$\Delta \Gamma \vdash k : \phi \quad \Delta \Gamma \vdash l : \psi$$  \hspace{1cm} \(\Delta \Gamma \vdash \text{case}(k; l) : \phi \lor \psi\)  \hspace{1cm} (31.3o)

A negation is true if the negated proposition is false, and is false if it is true.

$$\Delta \Gamma \vdash k : \phi$$  \hspace{1cm} \(\Delta \Gamma \vdash \text{not}(k) : \neg \phi\)  \hspace{1cm} (31.3p)

$$\Delta \Gamma \vdash p : \phi$$  \hspace{1cm} \(\Delta \Gamma \vdash \text{not}(p) : \neg \phi\)  \hspace{1cm} (31.3q)
31.2 Deriving Elimination Forms

One notable feature of classical logic is that there are only introductory forms, and no eliminatory forms. The eliminatory forms of proof in constructive logic, such as projection, case analysis, and application, arise as introductory forms of refutation in classical logic, whereas, by contrast, the introductory forms of constructive logic carry over directly to classical logic. While this brings out a pleasing symmetry in classical logic, it leads to a somewhat convoluted form of proof. For example, a proof of

\[(\phi \land (\psi \land \theta)) \supset (\theta \land \phi)\]

in classical logic has the form

\[\lambda(w: \phi \land (\psi \land \theta). ccr(u \div \theta \land \phi \cdot k \# w)),\]

where \(k\) is the refutation

\[fst; ccp(x: \phi. snd; ccp(y: \psi \land \theta.snd; ccp(z: \theta. u \# (z,x) \# y) \# w).)\]

This example makes clear that classical logic is biased towards indirect proof, which leads to a somewhat convoluted style of argument. For theorems that require indirect proof, there is no alternative, but the example above has a more succinct direct proof in constructive logic:

\[\lambda(w: \phi \land (\psi \land \theta). andI\text{(andE[r] (andE[r] (w)); andE[1] (w))}.)\]

By applying the proofs-as-programs correspondence given in Chapter 30, this may be re-written as the program

\[\lambda(w: \phi \times (\psi \times \theta). \langle w \cdot r \cdot r, w \cdot 1 \rangle).\]

Ideally, we would like to support both forms of proof, direct proof where applicable, and indirect proof where required. This may be achieved by showing that the elimination forms of constructive logic are derivable in classical logic. This may be achieved by making the following definitions:

\[
\begin{align*}
\text{falseE}[\phi](p) &= ccr(u \div \phi. \text{abort} \# p) \\
\text{andE}[1](p) &= ccr(u \div \phi. \text{fst}; u \# p) \\
\text{andE}[r](p) &= ccr(u \div \psi. \text{snd}; u \# p) \\
\text{impE}(p; q) &= ccr(u \div \psi. \text{app}(q); u \# p) \\
\text{orE}[\phi; \psi](p; x. q; y. r) &= ccr(u \div \gamma. \text{case}(ccp(x: \phi. u \# q); ccp(y: \psi. u \# r)) \# p)
\end{align*}
\]
It is straightforward to check that the expected elimination rules hold. For example, the rule
\[ \Delta \Gamma \vdash p : \phi \supset \psi \quad \Delta \Gamma \vdash q : \phi \]
\[ \Delta \Gamma \vdash \text{impE}(p; q) : \psi \]  
(31.4)
is derivable using the definition of \( \text{impE}(p; q) \) given above. By suppressing proof terms, we may derive the corresponding provability rule
\[ \Delta \Gamma \vdash \phi \supset \psi \true \quad \Delta \Gamma \vdash \false \]
\[ \Delta \Gamma \vdash \psi \true . \]  
(31.5)

31.3 Dynamics of Proofs

The dynamic semantics of classical logic may be described as a process of conflict resolution. The state of the abstract machine is a contradiction, \( k \# p \), between a refutation, \( k \), and a proof, \( p \), of the same proposition. Execution consists of “simplifying” the conflict based on the form of \( k \) and \( p \). This process is formalized by an inductive definition of a transition relation between contradictory states.

Here are the rules for each of the logical connectives, which all have the form of resolving a conflict between a proof and a refutation of a proposition formed with that connective.

\[
\begin{align*}
\text{fst}; k \# (p, q) & \mapsto k \# p & (31.6a) \\
\text{snd}; k \# (p, q) & \mapsto k \# q & (31.6b) \\
\text{case}(k; l) \# \text{inl}(p) & \mapsto k \# p & (31.6c) \\
\text{case}(k; l) \# \text{inr}(q) & \mapsto l \# q & (31.6d) \\
\text{app}(p); k \# \lambda(x: \phi. q) & \mapsto k \# [p/x]q & (31.6e) \\
\text{not}(p) \# \text{not}(k) & \mapsto k \# p & (31.6f)
\end{align*}
\]

The symmetry of the transition rule for negation is particularly elegant.

Here are the rules for the generic primitives relating truth and falsity.

\[
\begin{align*}
\text{ccp}(x: \phi. k \# p) \# q & \mapsto [q/x]k \# [q/x]p & (31.6g) \\
k \# \text{ccr}(u \div \phi. l \# p) & \mapsto [k/u]l \# [k/u]p & (31.6h)
\end{align*}
\]

These rules explain the terminology: “ccp” means “call with current proof”, and “ccr” means “call with current refutation”. 

18:42
Rules (31.6g) to (31.6h) overlap in that there are two possible transitions for a state of the form

\( \text{ccp}(x : \phi \cdot k \# p) \# \text{ccr}(u \div \phi \cdot l \# q). \)

This state may transition either to the state

\([r/x]k \# [r/x]p,\)

where \(r\) is \(\text{ccr}(u \div \phi \cdot l \# q)\), or to the state

\([m/u]l \# [m/u]q,\)

where \(m\) is \(\text{ccp}(x : \phi \cdot k \# p)\), and these are not equivalent.

There are two possible attitudes about this ambiguity. One is to simply accept that classical logic has a non-deterministic dynamic semantics, and leave it at that. But this means that it is difficult to predict the outcome of a computation, since it could be radically different in the case of the overlapping state just described. The alternative is to impose an arbitrary priority ordering among the two cases, either preferring the first transition to the second, or vice versa. Preferring the first corresponds, very roughly, to a “lazy” semantics for proofs, because we pass the unevaluated proof, \(r\), to the refutation on the left, which is thereby activated. Preferring the second corresponds to an “eager” semantics for proofs, in which we pass the unevaluated refutation, \(m\), to the proof, which is thereby activated. Dually, these choices correspond to an “eager” semantics for refutations in the first case, and a “lazy” one for the second. Take your pick.

How is computation to be started? The difficulty is that we need both a closed proof and a closed refutation of the same proposition, which is impossible since classical logic is consistent. The solution for an eager interpretation of proofs (and, correspondingly, a lazy interpretation of refutations) is simply to postulate an initial refutation, \(\text{halt}\), and to deem a state of the form \(\text{halt} \# p\) to be initial, and also final, provided that \(p\) is not a “ccr” instruction. The solution for a lazy interpretation of proofs (and an eager interpretation of refutations) is dual, taking \(k \# \text{halt}\) as initial, and also final, provided that \(k\) is not a “ccp” instruction.

### 31.4 Exercises
Part XI

Subtyping
Chapter 32

Subtyping

A subtype relation is a pre-order (reflexive and transitive relation) on types that validates the subsumption principle:

if \( \sigma \) is a subtype of \( \tau \), then a value of type \( \sigma \) may be provided whenever a value of type \( \tau \) is required.

The subsumption principle relaxes the strictures of a type system to permit values of one type to be treated as values of another.

Experience shows that the subsumption principle, 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.
32.1 Subsumption

A subtyping judgement has the form $\sigma \ll : \tau$, and states that $\sigma$ is a subtype of $\tau$. At a minimum we demand that the following structural rules of subtyping be admissible:

$$\tau \ll : \tau$$  \hspace{1cm} (32.1a)

$$\rho \ll : \sigma \quad \sigma \ll : \tau \quad \frac{}{\rho \ll : \tau}$$  \hspace{1cm} (32.1b)

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:

$$\Gamma \vdash e : \sigma \quad \sigma \ll : \tau \quad \frac{}{\Gamma \vdash e : \tau}$$  \hspace{1cm} (32.2)

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 $\sigma \ll : \tau$.

32.2 Varieties of Subtyping

In this section we will informally explore several different forms of subtyping for various extensions of $\mathcal{L}\{\rightarrow\}$. In Section 32.4 on page 294 we will examine some of these in more detail from the point of view of type safety.

32.2.1 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

$$\text{int} \ll : \text{rat} \ll : \text{real}$$

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.

### 32.2.2 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 dynamic semantics 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 static semantics of finite products (Rules (16.3)) ensures that the only operation we may perform on a value of type \(\tau\), other than to bind it to a variable, is to take the \(j\)th projection from it for some \(j \in J\) to obtain a
value of type $\tau_j$. Now suppose that $e$ is of type $\sigma$. If the projection $e \cdot j$ is to be well-formed, then $\sigma$ must be a finite product type $\prod_{i \in I} \sigma_i$ such that $j \in I$. Moreover, for this to be of type $\tau_j$, 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:

\[
\frac{J \subseteq I}{\prod_{i \in I} \tau_i <: \prod_{j \in J} \tau_j}.
\]

(32.3)

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 32.3 on page 290.

The argument for Rule (32.3) is based on a dynamic semantics 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 (32.3) may be stated in the form

\[
\frac{m \geq n}{\langle \tau_1, \ldots, \tau_m \rangle <: \langle \tau_1, \ldots, \tau_n \rangle}.
\]

(32.4)

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 $j$th 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 (32.3) takes the following form:

\[
\frac{m \geq n}{\langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle}.
\]

(32.5)

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 (32.5) 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.

### 32.2.3 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 static semantics of sums (Rules (17.3)) 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 $\sigma_i$ is equal to $\tau_i$. If the containment is strict, some cases cannot arise, but this does not disrupt safety. This leads to the following subtyping rule for finite sums:

$$I \subseteq J \quad \sum_{i \in I} \tau_i \prec \sum_{j \in J} \tau_j.$$  \hfill (32.6)

Note well the reversal of the containment as compared to Rule (32.3).

When $I$ and $J$ are initial segments of the natural numbers, we obtain the following special case of Rule (32.6):

$$m \leq n \quad [l_1 : \tau_1, \ldots, l_m : \tau_m] \prec [l_1 : \tau_1, \ldots, l_n : \tau_n].$$  \hfill (32.7)

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.
32.3 Variance

In addition to basic subtyping principles such as those considered in Section 32.2 on page 286, it is also important to consider the effect of subtyping on type constructors. A type constructor is said to be covariant 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.

32.3.1 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 \( \tau_j \), then it is sufficient to require that \( j \in I \) and \( \sigma_j <: \tau_j \). This is summarized by the following rule:

\[
(\forall i \in I) \sigma_i <: \tau_i \\
\prod_{i \in I} \sigma_i <: \prod_{i \in I} \tau_i
\]

(32.8)

It is implicit in this rule that the dynamic semantics 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 (32.8) reads as follows:

\[
\sigma_1 <: \tau_1 \ldots \sigma_n <: \tau_n \\
\langle \sigma_1, \ldots, \sigma_n \rangle <: \langle \tau_1, \ldots, \tau_n \rangle
\]

(32.9)

When specialized to symbolic labels, the covariance principle for finite products may be re-stated as follows:

\[
\sigma_1 <: \tau_1 \ldots \sigma_n <: \tau_n \\
\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle
\]

(32.10)

32.3.2 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:

\[
(\forall i \in I) \sigma_i <: \tau_i \\
\sum_{i \in I} \sigma_i <: \sum_{i \in I} \tau_i
\]

(32.11)
32.3 Variance

When specialized to symbolic labels as index sets, we obtain the following formulation of the covariance principle for sum types:

\[
\sigma_1 <: \tau_1 \quad \ldots \quad \sigma_n <: \tau_n \quad \Rightarrow \quad [l_1:\sigma_1,\ldots,l_n:\sigma_n] <: [l_1:\tau_1,\ldots,l_n:\tau_n].
\]

(32.12)

A case analysis on a value of the supertype is prepared, in the \(i\)th branch, to accept a value of type \(\tau_i\). By the premises of the rule, it is sufficient to provide a value of type \(\sigma_i\) instead.

32.3.3 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 \(e : \sigma \rightarrow \tau\). 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:

\[
\tau <: \tau' \quad \Rightarrow \quad \sigma \rightarrow \tau <: \sigma \rightarrow \tau'.
\]

(32.13)

Every function that delivers a value of type \(\tau\) must also deliver a value of type \(\tau'\), 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 \(e : \sigma \rightarrow \tau\). This means that \(e\) may be applied to any value of type \(\sigma\), and hence, by the subsumption principle, it may be applied to any value of any subtype, \(\sigma'\), of \(\sigma\). In either case it will deliver a value of type \(\tau\). Consequently, we may just as well think of \(e\) as having type \(\sigma' \rightarrow \tau\).

\[
\sigma' <: \sigma \\
\sigma \rightarrow \tau <: \sigma' \rightarrow \tau.
\]

(32.14)

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:

\[
\sigma' <: \sigma \quad \tau <: \tau' \\
\sigma \rightarrow \tau <: \sigma' \rightarrow \tau
\]

(32.15)

Beware of the reversal of the ordering in the domain!
32.3.4 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,

\[ \mu t. [\emptyset : \text{unit}, \text{binode} : (\text{data} : \text{nat}, \text{lft} : t, \text{rht} : t)] \]

and the type of “bare” binary trees, without labels on the nodes,

\[ \mu t. [\emptyset : \text{unit}, \text{binode} : (\text{lft} : t, \text{rht} : t)] \]

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:

\[ \mu t. [\emptyset : \text{unit}, \text{binode} : (\text{lft} : t, \text{rht} : t), \text{trinode} : (\text{lft} : t, \text{mid} : t, \text{rht} : t)] \]

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:

\[
\frac{t \text{ type } \vdash \sigma <: \tau}{\mu t. \sigma <: \mu t. \tau}'\quad (32.16)
\]

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 (32.16) validates the intuitively plausible subtyping between labeled binary tree and bare binary trees just described. To derive this reduces to checking the subtyping relationship

\[ (\text{data} : \text{nat}, \text{lft} : t, \text{rht} : t) <: (\text{lft} : t, \text{rht} : t) \]

generically in \( t \), which is evidently the case.
Unfortunately, Rule (32.16) also underwrites incorrect subtyping relationships, as well as some correct ones. As an example of what goes wrong, consider the recursive types

\[ \sigma = \mu t. \langle a : t \rightarrow \text{nat}, b : t \rightarrow \text{int} \rangle \]

and

\[ \tau = \mu t. \langle a : t \rightarrow \text{int}, b : t \rightarrow \text{int} \rangle. \]

We assume for the sake of the example that nat \( < \) int, so that by using Rule (32.16) we may derive \( \sigma <: \tau \), which we will show to be incorrect. Let \( e : \sigma \) be the expression

\[ \text{fold}(\langle a = \lambda (x : \sigma.4), b = \lambda (x : \sigma. q((\text{unfold}(x) \cdot a)(x)) \rangle), \]

where \( q : \text{nat} \rightarrow \text{nat} \) is the discrete square root function. Since \( \sigma <: \tau \), it follows that \( e : \tau \) as well, and hence

\[ \text{unfold}(e) : \langle a : \tau \rightarrow \text{int}, b : \tau \rightarrow \text{int} \rangle. \]

Now let \( e' : \tau \) be the expression

\[ \text{fold}(\langle a = \lambda (x : \tau.4), b = \lambda (x : \tau.0) \rangle). \]

(The important point about \( e' \) is that the a method returns a negative number; the b method is of no significance.) To finish the proof, observe that

\[ (\text{unfold}(e) \cdot b)(e') \mapsto q(-4), \]

which is a stuck state. We have derived a well-typed program that “gets stuck”, refuting type safety!

Rule (32.16) 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.
294 32.4 Safety for Subtyping

by examining the bodies of the recursive types. To do so we maintain a
finite set, \( \Psi \), 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:

\[
\frac{\Psi, s <: t \vdash \sigma <: \tau}{\Psi \vdash \mu \cdot s \cdot \sigma <: \mu \cdot t \cdot \tau} \tag{32.17}
\]

That is, to check whether \( \mu \cdot s \cdot \sigma <: \mu \cdot t \cdot \tau \), 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,

\[
\frac{\Psi, s <: t \vdash s <: t}{\Psi} \tag{32.18}
\]

Using reflexivity in conjunction with Rule (32.17), we may verify the sub-
typings 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.

32.4 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 underly-
ing value. This changes the proof of the preservation and progress theo-
rems, 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 lan-
guage with subtyping for product types. The subtyping relation is defined
by Rules (32.3) and (32.8). We assume that the static semantics includes
subsumption, Rule (32.2).

Lemma 32.1 (Structurality).

1. The tuple subtyping relation is reflexive and transitive.
2. The typing judgement \( \Gamma \vdash e : \tau \) is closed under weakening and substitution.
32.4 Safety for Subtyping

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 (16.3), augmented by Rule (32.2).

Lemma 32.2 (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 \( \sigma_i \) for \( i \in I \).

4. If \( \prod_{i \in I} \sigma_i <: \prod_{j \in J} \tau_j \), then \( I \subseteq J \) and \( \sigma_j <: \tau_j \) for each \( j \in J \).

Proof. By induction on the subtyping and typing rules, paying special attention to Rule (32.2).

Theorem 32.3 (Preservation). If \( e : \tau \) and \( e \mapsto e' \), then \( e' : \tau \).

Proof. By induction on Rules (16.4). For example, consider Rule (16.4d), so that \( e = \langle e_i \rangle_{i \in I} \cdot k, e' = e_k \). By Lemma 32.2 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 32.2 for each \( i \in I \) there exists \( \sigma_i \) such that \( e_i : \sigma_i \) and \( \prod_{i \in I} \sigma_i <: \prod_{j \in J} \tau_j \). By Lemma 32.2 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 remaining cases are similar.

Lemma 32.4 (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 \( I \subseteq I \), and \( e_j : \tau_j \) for each \( j \in J \).

Proof. By induction on Rules (16.3) augmented by Rule (32.2).

Theorem 32.5 (Progress). If \( e : \tau \), then either \( e \) val or there exists \( e' \) such that \( e \mapsto e' \).
32.4 Safety for Subtyping

Proof. By induction on Rules (16.3) augmented by Rule (32.2). The rule of subsumption is handled by appeal to the inductive hypothesis on the premise of the rule. Rule (16.4d) follows from Lemma 32.4 on the preceding page.

To account for recursive subtyping in addition to finite product subtyping, the following inversion lemma is required.

Lemma 32.6.

1. If \( \Psi, s : t \vdash \sigma' : \tau' \) and \( \Psi, \sigma : \tau \) then \( \Psi, [\sigma/s] \sigma' : [\tau/t] \tau' \).

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 \( \rho \) or \( \tau \). 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.
32.5 Exercises

32.5 Exercises
The expression \( \text{let } e_1 : \tau \text{ be } x \text{ in } e_2 \) is a form of abbreviation mechanism by which we may bind \( e_1 \) to the variable \( x \) for use within \( e_2 \). In the presence of function types this expression is definable as the application \( \lambda (x: \tau. e_2) \langle e_1 \rangle \), which accomplishes the same thing. It is natural to consider an analogous form of \( \text{let} \) expression which permits a type expression to be bound to a type variable within a specified scope. The expression \( \text{let } t \text{ be } \tau \text{ in } e \) binds \( t \) to \( \tau \) within \( e \), so that one may write expressions such as

\[
\text{let } t \text{ be } \text{nat} \times \text{nat} \text{ in } \lambda (x:t. s(x \cdot 1)).
\]

For this expression to be type-correct the type variable \( t \) must be synonymous with the type \( \text{nat} \times \text{nat} \), for otherwise the body of the \( \lambda \)-abstraction is not type correct.

Following the pattern of the expression-level \( \text{let} \), we might guess that \( \text{lettype} \) is an abbreviation for the polymorphic instantiation \( \Lambda (t.e) [\tau] \), which binds \( t \) to \( \tau \) within \( e \). This does, indeed, capture the dynamic semantics of type abbreviation, but it fails to validate the intended static semantics. The difficulty is that, according to this interpretation of \( \text{lettype} \), 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 synonymous with \( \tau \). Thus, in the above example, the expression \( s(x \cdot 1) \) fails to type check, unless the binding of \( t \) were exposed.

The proposed definition of \( \text{lettype} \) 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 let \( t \) be \( \tau \) in \( e \) would be taken as a primitive form of expression whose static semantics is given by the following rule:

\[
\Gamma \vdash \tau \in \[\tau/t\]e : \tau'
\]

(33.1)

This would address the problem of supporting type abbreviations, but it does so in a rather \textit{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 \textit{singleton kinds}, which not only account for type abbreviations but also play a crucial role in the design of module systems.

33.1 Informal Overview

The central organizing principle of type theory is \textit{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 \textit{nat}), and not on their specific form or value. But in the case of a type abbreviation of the form let \( t \) be \( \tau \) in \( e \), the principle of compositionality dictates that the only thing that \( e \) "knows" about the type variable \( t \) is its kind, namely \textit{Type}, and not its binding, namely \( \tau \).

This is accurately captured by the proposed representation of type abbreviation as the combination of type abstraction and type application, but, as we have just seen, this is not the intended meaning of the construct!

We could, as suggested in the introduction, abandon the core principles of type theory, and introduce type abbreviations as a primitive notion. But there is no need to do so. Instead we can simply note that what is needed is for the kind of \( t \) to capture its identity. This may be achieved through the notion of a \textit{singleton kind}. Informally, the kind \( \text{Eqv}(\tau) \) is the kind of types that are definitionally equivalent to \( \tau \). That is, up to definitional equality, this kind has only one inhabitant, namely \( \tau \). Consequently, if \( u :: \text{Eqv}(\tau) \) is a variable of singleton kind, then within its scope, the variable \( u \) is synonymous with \( \tau \). Thus we may represent let \( t \) be \( \tau \) in \( e \) by
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 \( \text{Eqv}(\tau) :<: \text{Type} \), 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 \( \text{Eqv}(\tau) \). 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 \( \text{Eqv}(\tau) \) 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, \( \Pi u : \kappa_1. \kappa_2 \) classifies functions that, when applied to a constructor \( c_1 :: \kappa_1 \), results in a constructor of kind \( [c_1/u]\kappa_2 \). The important point is that the kind of the result is sensitive to the argument, and not just to its kind.\(^1\) The dependent product kind, \( \Sigma u : \kappa_1. \kappa_2 \), classifies pairs \( \langle c_1, c_2 \rangle \) such that \( c_1 :: \kappa_1 \), as might be expected, and \( c_2 :: [c_1/u]\kappa_2 \), in which the kind of the second component is sensitive to the first component itself, and not just its kind.

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 \( \text{Eqv}(c :: \kappa) \), where \( \kappa \) 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.

---

\(^1\)As we shall see in the development, the propagation of information as sketched here is managed through the use of singleton kinds.
Part XII

Symbols
Chapter 34

Symbols

A symbol is an atomic datum with no internal structure. The only way to compute with an unknown symbol is to compare it for identity with one or more known symbols, and branching according to the outcome. We shall make use of symbols for several purposes, including fluid binding, assignable variables, tags for classification of data, and names of communication channels. The common characteristic is that symbols are used as indices into a family of operations. To ensure safety while maximizing flexibility, we associate a unique type a type with each symbol, without restriction on the type.

34.1 Statics

We study a small language, called $\mathcal{L}\{\text{sym}\}$, for computing with symbols. The syntax of $\mathcal{L}\{\text{sym}\}$ is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$ :: $\text{sym}(\tau)$</td>
<td>$\tau\text{ sym}$</td>
<td></td>
</tr>
<tr>
<td>Expr</td>
<td>$e$ :: $\text{new}<a href="a.e">\tau</a>$</td>
<td>$\text{new} a : \tau \text{ in } e$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{sym}[a]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>$</td>
<td>$\text{sym}[a]$</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>$</td>
<td>$\text{scase}<a href="e;a_0.e_0;r_1,%5Cldots,r_n">t.\tau</a>$</td>
</tr>
<tr>
<td>Rule</td>
<td>$r$ :: $\text{sym?}<a href="e">a</a>$</td>
<td>$\text{sym}[a] \Rightarrow e$</td>
<td></td>
</tr>
</tbody>
</table>

In the match expression $\text{scase}[t.\tau](e;a_0.e_0;r_1,\ldots,r_n)$ the symbol $a_0$ is bound within $e_0$, but the symbols $a_1,\ldots,a_n$ occurring within $r_1,\ldots,r_n$, respectively, are not bound.

The static semantics of $\mathcal{L}\{\text{sym}\}$ defines a judgements of the form $\Gamma \vdash \Sigma$, $e : \tau$, where $\Sigma$ is a symbol context consisting of a finite set of declarations of
the form

\[ a_1 : \tau_1, \ldots, a_n : \tau_n. \]

The symbol context, \( \Sigma \), associates a type to each of a finite set of pairwise distinct symbols.

The rules defining the static semantics of \( L\{\text{sym}\} \) are as follows:

\[
\begin{align*}
\Gamma \vdash \Sigma, \ a : \sigma & \quad \Rightarrow \quad \Gamma \vdash \Sigma, \ \text{new}[\sigma](a.e) : \tau \\
\Gamma \vdash \Sigma, \ a : \sigma & \quad \Rightarrow \quad \Gamma \vdash \Sigma, \ \text{sym}[a] : \text{sym}(\sigma) \\
\Gamma \vdash \Sigma, \ a : \sigma & \quad \Rightarrow \quad \Gamma \vdash \Sigma, \ \text{sym}?(a.e) : \sigma > \tau \\
\Gamma \vdash \Sigma, \ e : \text{sym}(\sigma) & \quad \Gamma \vdash \Sigma, \ a_0 : [\sigma/t] \tau \\
\Gamma \vdash \Sigma, \ r_1 : \sigma_1 > [\sigma_1/t] \tau & \quad \ldots \quad \Gamma \vdash \Sigma, \ r_n : \sigma_n > [\sigma_n/t] \tau \\
\Gamma \vdash \Sigma, \ \text{scase}[t.\tau](e; a_0.e_0; r_1, \ldots, r_n) : [\sigma/t] \tau
\end{align*}
\]

Rule (34.1a) gives the static semantics for the expression \( \text{new}[\tau](a.e) \), which allocates a fresh symbol, \( a \), of type \( \tau \) for use within the expression, \( e \). It is implicit that \( a \) is chosen to not already be declared in \( \Sigma \), ensuring that it is not otherwise in use. This assumption may always be met by suitably renaming the bound name, \( a \), of the \text{new} expression prior to applying the rule.

Rule (34.1b) is the introduction rule for the type \( \text{sym}(\sigma) \). It states that if \( a \) is a symbol with associated type \( \sigma \), then \( \text{sym}[a] \) is an expression of type \( \text{sym}(\sigma) \). The expression \( \text{scase}[t.\tau](e; a_0.e_0; r_1, \ldots, r_n) \) discriminates the value of \( e \) according to the rules \( r_1, \ldots, r_n \). If \( e \) evaluates to \( \text{sym}[a] \) for some \( 1 \leq i \leq n \), then the case expression evaluates to \( e_i \). Otherwise, if \( e \) evaluates to \( \text{sym}[a] \) such that \( a \) is not among the \( a_1, \ldots, a_n \), then the case expression evaluates to \( [a/a_0]e_0 \). (The non-matching symbol is passed to \( e_0 \) so that it can be used as the subject of further case analysis.)

The static semantics of this construct is given in terms of a specified type operator, \( t.\tau \), which serves to propagate the type of a matched symbol to the corresponding branch of the case analysis. At the outset we know that the value, \( \text{sym}[a] \), of \( e \) is a symbol with associated type \( \sigma \), and that each symbol \( a_1, \ldots, a_n \) has associated type \( \sigma_1, \ldots, \sigma_n \), respectively. The type of
the case analysis is \([\sigma/t]\tau\), which propagates the type of the matched symbol into the result of the case analysis. The type of the \(i\)th branch is, however, \([\sigma_i/t]\tau\). In the case that the symbol \(a\) is the symbol \(a_i\), we acquire the information that \(\sigma\) is \(\sigma_i\), and hence we satisfy the requirement that the overall type be \([\sigma/t]\tau\). If, however, the symbol \(a\) does not match any branch, then the result of the case analysis is \(e_0\), which has type \([\sigma/t]\tau\), as required.

### 34.2 Scoped Dynamics

Were it not for the new construct, the dynamics of \(\mathcal{L}\{\text{sym}\}\) would be given by a transition judgement of the form \(e \xrightarrow{\Sigma} e'\), where \(\Sigma\) declares the symbols that are active at the point of evaluation. Informally, the new construct introduces a new, or fresh, symbol, at the point at which it is executed. But there are two distinct interpretations of what this means.

The scoped, or stack-like, interpretation specifies that \(\text{new} \sigma \cdot (a.e)\) generates a new symbol, \(a\), for use during the evaluation of \(e\). After evaluation of \(e\) completes, the symbol is discarded, since its scope is confined to that expression. For this to make sense, however, we must ensure that the value of \(e\) does not depend on \(a\), otherwise the returned value will escape its scope.

The scoped dynamics of \(\mathcal{L}\{\text{sym}\}\) comprises the judgement \(e \text{ val}\Sigma\), stating that \(e\) is a value relative to symbols \(\Sigma\), and the judgement \(e \xrightarrow{\Sigma} e'\), stating that \(e\) transitions to \(e'\) in the presence of the symbols \(\Sigma\). These judgements are defined by the following rules:

\[
\frac{\text{sym}[a] \text{ val}_{\Sigma, a \cdot \sigma}}{e \xrightarrow{\Sigma, a \cdot \sigma} e'} \tag{34.2a}
\]

\[
\frac{\text{sym}[a] \text{ val}_{\Sigma, a \cdot \sigma} \quad e \xrightarrow{\Sigma, a \cdot \sigma} e'}{\text{new} \sigma \cdot (a.e) \xrightarrow{\Sigma, \text{new} \sigma \cdot (a.e')} e' \quad \text{new} \sigma \cdot (a.e) \text{ val}_{\Sigma} \quad a \notin e \quad \text{new} \sigma \cdot (a.e) \xrightarrow{\Sigma, e} e'} \tag{34.2b}
\]

\[
\frac{e \xrightarrow{\Sigma, e'} s\text{case}[t \cdot \tau](e; a_0 \cdot e_0; r_1, \ldots, r_n)}{s\text{case}[t \cdot \tau](e'; a_0 \cdot e_0; r_1, \ldots, r_n) \xrightarrow{\Sigma, e' \cdot e_0}} \tag{34.2d}
\]

\[
\frac{s\text{case}[t \cdot \tau]([a/a_0] e_0)}{s\text{case}[t \cdot \tau]([a/a_0] e_0) \xrightarrow{\Sigma, [a/a_0] e_0}} \tag{34.2e}
\]
308 34.3 Unscoped Dynamics

\[
\begin{align*}
  a &= a_1 \\
  \text{scase}[t.\tau](\text{sym}[a];a_0.e_0;\text{sym?}[a_1](e_1),\ldots) &\rightsquigarrow a_1 \\
  a \neq a_1 \\
  \text{scase}[t.\tau](\text{sym}[a];a_0.e_0;\text{sym?}[a_1](e_1),\text{sym?}[a_2](e_2),\ldots) &\rightsquigarrow \text{scase}[t.\tau](\text{sym}[a];a_0.e_0;\text{sym?}[a_2](e_2),\ldots)
\end{align*}
\]  

The second premise of Rule (34.2c) imposes the requirement that the returned value from the new is independent of the symbol \( a \).

Since the static semantics of \( \mathcal{L}\{\text{sym}\} \) does not ensure that a symbol does not escape its scope, a well-typed expression may be “stuck,” in violation of the progress theorem. One way to handle this is to treat such a situation as a checked error and to weaken the progress theorem accordingly (as described in Chapter 11). Another is to consider a type system that ensures that symbols cannot escape their scope during execution. (See, for example, Chapter 37 for one such approach.)

34.3 Unscoped Dynamics

The unscoped, or heap-like, interpretation specifies that new[\( \sigma \)](a.e) generates a new symbol that may be used within \( e \), without imposing the requirement that the scope of \( a \) be limited to \( e \). This means that the symbol must continue to be available even after \( e \) returns a value, leading to a somewhat different semantics consisting of judgements of the form \( e @ \Sigma \mapsto e' @ \Sigma' \). Such a judgement states that evaluation of \( e \) relative to active symbols \( \Sigma \) results in the expression \( e' \) in an extension \( \Sigma' \) of \( \Sigma \). New symbols come into existence during execution, but old symbols are never thrown away, nor are their associated types ever altered.

The unscoped dynamics of \( \mathcal{L}\{\text{sym}\} \) records the active set of symbols using a transition judgement of the form \( e @ \Sigma \mapsto e' @ \Sigma' \) to ensure that symbols persist beyond their scope of declaration.

\[
\begin{align*}
  \text{sym}[a] \text{ val}_{\Sigma,a:\sigma} & \quad (34.3a) \\
  a \notin \text{ dom}(\Sigma) & \quad (34.3b)
\end{align*}
\]
34.4 Safety

Since the scoped dynamics of \( \mathcal{L}\{\text{sym}\} \) is not type safe (absent the modifications discussed earlier), we will concentrate here on the safety of the unscoped dynamics of \( \mathcal{L}\{\text{sym}\} \).

**Theorem 34.1** (Preservation). Suppose that \( e \in \Sigma \mapsto e' \in \Sigma' \). Then \( \Sigma' \supseteq \Sigma \) and \( \Sigma' \vdash e' : \tau \).

**Proof.** By rule induction on Rules (34.3). The most interesting case arises when \( e = \text{sym}[a] \) and \( a = a_i \) for some rule \( \text{sym}[a_i](e_i) \). By inversion of typing we know that \( \Sigma \vdash e_i : [\sigma_i/t] \tau \). We are to show that \( \Sigma \vdash e_i : [\sigma/t] \tau \). Noting that if \( a = a_i \), then by unicity of typing, \( \sigma_i = \sigma \), the result follows immediately.

**Lemma 34.2** (Canonical Forms). Suppose that \( \Sigma \vdash e : \sigma \text{ sym and } e \text{ val}_{\mathcal{L}} \). Then \( e = \text{sym}[a] \) for some \( a \) such that \( \Sigma = \Sigma', a : \sigma \).

**Proof.** By rule induction on Rules (34.1), taking account of the definition of values.

The chief difference compared to Rules (34.2) is that evaluation of subexpressions can extend \( \Sigma \), and that the body of a new can be exited without restriction since the new symbol persists beyond its scope.
Theorem 34.3 (Progress). Suppose that $\Sigma \vdash e : \tau$. Then either $e \text{ val}_{\Sigma}$, or $e @ \Sigma \rightarrow e' @ \Sigma'$ for some $\Sigma'$ and $e'$.

Proof. By rule induction on Rules (34.1). For a case analysis of the form $\text{scase}[f, \tau](e_{a_0} e_{r_1} \ldots e_{r_n})$, where $e \text{ val}_{\Sigma}$, we have by Lemma 34.2 on the previous page that $e = \text{sym}[a]$ for some symbol $a$ of type $\sigma$. Then either $a = a_i$ for some rule $\text{sym}(a_i)(e_i)$, in which case we progress to $e_i$, or, if no rule matches, we progress to $[a/a_0]e_0$. \hfill \Box

34.5 Exercises
Chapter 35

Fluid Binding

Recall from Chapter 13 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.

35.1 Statics
The language $L\{\text{fluid}\}$ extends the language $L\{\text{sym}\}$ defined in Chapter 34 with the following additional constructs:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$\text{put}<a href="e_1;e_2">a</a>$</td>
<td>$\text{put } a \text{ is } e_1 \text{ in } e_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{get}[a]$</td>
<td>$a$</td>
</tr>
</tbody>
</table>

As in Chapter 34, the variable $a$ ranges over some fixed set of symbols. The expression $\text{get } a$ evaluates to the value of the current binding of $a$, if it has one, and is stuck otherwise. The expression $\text{put } a \text{ is } e_1 \text{ in } e_2$ binds the symbol $a$ to the value $e_1$ for the duration of the evaluation of $e_2$, at which point the binding of $a$ reverts to what it was prior to the execution. The symbol $a$ is not bound by the $\text{put}$ expression, but is instead a parameter of it.

The static semantics of $L\{\text{fluid}\}$ is defined by judgements of the form

$$ \Gamma \vdash \Sigma e : \tau, $$

where $\Sigma$ 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 $\Gamma$ 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 static semantics of $L\{\text{fluid}\}$ extends that of $L\{\text{sym}\}$ (see Chapter 34) with the following two rules:

\[
\frac{\Sigma \vdash a : \tau}{\Gamma \vdash \Sigma \text{get}[a] : \tau} \tag{35.1a}
\]

\[
\frac{\Sigma \vdash a : \tau_1 \quad \Gamma \vdash \Sigma e_1 : \tau_1 \quad \Gamma \vdash \Sigma e_2 : \tau_2}{\Gamma \vdash \Sigma \text{put}[a](e_1;e_2) : \tau_2} \tag{35.1b}
\]

Rule (35.1b) specifies that the symbol $a$ is a parameter of the expression that must be declared in $\Sigma$.

### 35.2 Dynamics

The dynamics of $L\{\text{fluid}\}$ is defined by maintaining 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 $e \overset{\Sigma}{\mu} e'$, where $\Sigma$ is as in the static semantics, and $\mu$ is a finite function mapping some subset of the symbols declared in $\Sigma$ to values of appropriate type. If $\mu$ is defined for some symbol $a$, then it has the form $\mu' \otimes \langle a : e \rangle$ for some $\mu'$ and value $e$. If, on the other hand, $\mu$ is undefined for some symbol $a$, we may regard it as
having the form $\mu' \otimes \langle a : \_ \rangle$. We will write $\langle a : \_ \rangle$ to stand ambiguously for either $\langle a : \_ \rangle$ or $\langle a : e \rangle$ for some expression $e$.

The dynamic semantics of $L\{\text{fluid}\}$ is given by the following rules:

$$
\begin{align*}
& e \text{ val} \\
& \text{get}[a] \xrightarrow{\Sigma, a : \tau} e \\
& \text{put}[a] (e_1; e_2) \xrightarrow{\Sigma} \text{put}[a] (e_1' ; e_2) \\
& \text{put}[a] (e_1; e_2) \xrightarrow{\Sigma, a : \tau} \text{put}[a] (e_1 ; e_2') \\
& \text{put}[a] (e_1; e_2) \xrightarrow{\Sigma} \text{put}[a] (e_1 ; e_2') \\
& \text{put}[a] (e_1; e_2) \xrightarrow{\Sigma} e_2 \\
\end{align*}
$$

(35.2a) (35.2b) (35.2c) (35.2d)

Rule (35.2a) specifies that $\text{get}[a]$ evaluates to the current binding of $a$, if any. Rule (35.2b) specifies that the binding for the symbol $a$ is to be evaluated before the binding is created. Rule (35.2c) 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 (35.2d) eliminates the fluid binding for $a$ once evaluation of the extent of the binding has completed.

According to the dynamic semantics defined by Rules (35.2), there is no transition of the form $\text{get}[a] \xrightarrow{\Sigma} e$ (for any $e$) if $a \notin \text{dom}(\Sigma)$. Since such an expression is considered well-formed in the static semantics, the dynamic semantics must explicitly check for unbound symbols. This is expressed by the judgement $e \text{ unbound}_\Sigma$, which is inductively defined by the following rules:

$$
\begin{align*}
& a \notin \text{dom}(\Sigma) \\
& \text{get}[a] \text{ unbound}_\Sigma \\
& e_1 \text{ unbound}_\Sigma \\
& \text{put}[a] (e_1; e_2) \text{ unbound}_\Sigma \\
\end{align*}
$$

(35.3a) (35.3b)

\footnote{In the presence of other language constructs, stuck states would have to be propagated through the evaluated arguments of a compound expression as described in Chapter 11.}
35.3 Type Safety

Define the auxiliary judgement $\mu : \Sigma$ by the following rules:

\[
\begin{align*}
\emptyset & : \emptyset \\
\frac{\Gamma \vdash e : \tau}{\mu \otimes \langle a : e \rangle : \Sigma, a : \tau} & (35.4b) \\
\frac{\mu : \Sigma}{\mu \otimes \langle a : \bullet \rangle : \Sigma, a : \tau} & (35.4c)
\end{align*}
\]

These rules specify that if a symbol is bound to a value, then that value must be of the type associated to the symbol by $\Sigma$. No demand is made in the case that the symbol is unbound (equivalently, bound to a "black hole").

**Theorem 35.1** (Preservation). If $e \frac{\Sigma}{\mu} e'$, where $\mu : \Sigma$ and $\Gamma \vdash e : \tau$, then $\Gamma \vdash e' : \tau$.

*Proof.* By rule induction on Rules (35.2). Rule (35.2a) is handled by the definition of $\mu : \Sigma$. Rule (35.2b) follows immediately by induction. Rule (35.2d) is handled by inversion of Rules (35.1). Finally, Rule (35.2c) is handled by inversion of Rules (35.1) and induction. 

**Theorem 35.2** (Progress). If $\Gamma \vdash e : \tau$ and $\mu : \Sigma$, then either $e \text{ val}_\mu$, or $e \text{ unbound}_\mu$, or there exists $e'$ such that $e \frac{\Sigma}{\mu} e'$.

*Proof.* By induction on Rules (35.1). For Rule (35.1a), we have $\Sigma \vdash a : \tau$ from the premise of the rule, and hence, since $\mu : \Sigma$, we have either $\mu(a) = \bullet$ (unbound) or $\mu(a) = e$ for some $e$ such that $\Gamma \vdash e : \tau$. In the former case we have $e \text{ unbound}_\mu$, and in the latter we have $\text{get}[a] \frac{\Sigma}{\mu} e$. 
For Rule (35.1b), we have by induction that either $e_1$ val or $e_1 \text{ unbound}_\mu$, or $e_1 \frac{\Sigma}{\mu} e_1'$. In the latter two cases we may apply Rule (35.2b) or Rule (35.3b), respectively. If $e_1$ val, we apply induction to obtain that either $e_2$ val, in which case Rule (35.2d) applies; $e_2$ unbound $\mu$, in which case Rule (35.3b) applies; or $e_2 \frac{\Sigma}{\mu} e_2'$, in which case Rule (35.2c) applies.

35.4 Subtleties of Fluid Binding

Fluid binding in the context of a first-order language is easy to understand. If the expression put $a$ is $e_1$ in $e_2$ has a type such as 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 put $a$ is $e_1$ in $e_2$ is a function type, so that the returned value is a $\lambda$-abstraction? In that case the body of the $\lambda$ 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

$$\text{put } a \text{ is } 17 \text{ in } \lambda(x:\text{nat}.x + \text{get } a), \quad (35.5)$$

which has type 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 17, and returns the function $\lambda(x:\text{nat}.x + \text{get } a)$. 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 (35.5) to an argument will incur an error because the symbol $a$ is not bound.

Contrast this with the similar expression

$$\text{let } y \text{ be } 17 \text{ in } \lambda(x:\text{nat}.x + y), \quad (35.6)$$

in which we have replaced the fluid-bound symbol, $a$, by a statically bound variable, $y$. This expression evaluates to $\lambda(x:\text{nat}.x + 17)$, which adds 17 to its argument when applied. There is never any possibility of an unbound identifier arising at execution time, precisely because the identification of...
Subtleties of Fluid Binding

It is not possible to say that either of these behaviors is “right” or “wrong,” but experience has shown that providing only one or the other of these behaviors is a mistake. 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. The main use of fluid binding is to avoid having to pass “extra” parameters 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-specify it at each call site.

For example, let $e$ stand for the value of expression (35.5), a $\lambda$-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

$$\text{put } a \text{ is } 7 \text{ in } (e(9))$$

evaluates to $15$, and the expression

$$\text{put } a \text{ is } 8 \text{ in } (e(9))$$

evaluates to $17$. Writing just $e(9)$, without a surrounding binding for $a$, results in a run-time error attempting to retrieve the binding of the unbound symbol $a$.

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

$$e'(7)(9)$$

and

$$e'(8)(9),$$

respectively, where $e'$ is the $\lambda$-abstraction

$$\lambda(a:\text{nat}. \lambda(x:\text{nat}. x + a)).$$

Using additional arguments can be slightly inconvenient, though, when several call sites have the same binding for $a$. Using fluid binding we may write

$$\text{put } a \text{ is } 7 \text{ in } \langle e(8), e(9) \rangle,$$
whereas using an additional argument we must write

\[ \langle e'(7)(8), e'(7)(9) \rangle. \]

However, this sort of redundancy can be mitigated by simply factoring out the common part, writing

\[ \text{let } f \text{ be } e'(7) \text{ in } \langle f(8), f(9) \rangle. \]

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 \(\text{nat} \rightarrow_a \text{nat}\), reflecting the demand that a binding for \(a\) be provided at the call site. A type system of this sort is developed in Chapter 49.

### 35.5 Dynamic Fluids

Fluid-bound identifiers in \(\mathcal{L}\{\text{fluid}\}\) are static in that the get and put operations specify the symbol on which to act. We can, if we wish, code up an indexing scheme whereby the choice of symbol on which to act is determined dynamically, rather than statically. For example, if we wish to dynamically determine whether to act on symbols \(a_1\) or \(a_2\), we may carry around a boolean condition that, if true, means to act on \(a_1\), and, if false, means to act on \(a_2\). This obviously generalizes to any finite selection of symbols, but requires a bit of effort to arrange an indexing scheme for the symbols at hand.

A more direct approach is to introduce a type \(\tau_{\text{fluid}}\) of dynamic fluids whose values are symbols determined at execution time. Associated to this type are get and put operations that act on a dynamically determined symbol. Interestingly, no new primitives are required to support dynamic fluids. Specifically, we may define the type of dynamic fluids and its associated operations as follows:

\[
\begin{align*}
\tau_{\text{fluid}} &= \tau_{\text{sym}} \\
\text{getf} e &= \text{scase } e \{ \varepsilon \} \text{ ow } a \text{. (get } a) \\
\text{putf} e \text{ is } e_1 \text{ in } e_2 &= \text{scase } e \{ \varepsilon \} \text{ ow } a \text{. (put } a \text{ is } e_1 \text{ in } e_2). 
\end{align*}
\]
The `case` operations serve to introduce a name for the symbol resulting from evaluation of the expression \( e \) so that we may make use of the static `put` and `get` operations of \( \mathcal{L}\{\text{fluid}\} \).

New dynamic fluids may be allocated by writing

\[
\text{newfl} \ x: \tau_{\text{fluid}} = e_1 \ in \ e_2,
\]

which stands for the expression

\[
\text{new} \ a: \tau \ in \ (\text{put} \ a \ is \ e_1 \ in \ (\text{let} \ x \ be \ sym[a] \ in \ e_2)).
\]

This expression allocates a new symbol, initializes its binding to \( e_1 \), and makes the new symbol available within \( e_2 \) by binding it to the variable \( x \) of type \( \tau_{\text{fluid}} \).

### 35.6 Exercises

1. Formalize *deep binding* and *shallow binding* using the stack machine of Chapter 27.
Chapter 36

Dynamic Classification

Sum types may be used to classify data values by labelling them with a class identifier that determines the type of the associated data item. For example, a sum type of the form $\sum \langle i_0 : \tau_0, \ldots, i_{n-1} : \tau_{n-1} \rangle$ consists of $n$ distinct classes of data, with the $i$th class labelling a value of type $\tau_i$. A value of this type is introduced by the expression $i \cdot e_i$, where $0 \leq i < n$ and $e_i : \tau_i$, and is eliminated by an $n$-ary case analysis binding the variable $x_i$ to the value of type $\tau_i$ labelled with class $i$.

Sum types are useful in situations where the type of a data item can only be determined at execution time, for example when processing input from an external data source. For example, a data stream from a sensor might consist of several different types of data according to the form of a stimulus. To ensure safe processing the items in the stream are labeled with a class that determines the type of the underlying datum. The items are processed by performing a case analysis on the class, and passing the underlying datum to a handler for items of that class.

A difficulty with using sums for this purpose, however, is that the developer must specify in advance the classes of data that are to be considered. That is, sums support static classification of data based on a fixed collection of classes. While this works well in the vast majority of cases, there are situations where static classification is inadequate, and dynamic classification is required. For example, we may wish to classify data in order to keep it secret from an intermediary in a computation. By creating a fresh class at execution time, two parties engaging in a communication can arrange that they, and only they, are able to compute with a given datum; all others must merely handle it passively without examining its structure or value.
One example of this sort of interaction arises when programming with exceptions, as described in Chapter 28. One may consider the value associated with an exception to be a secret that is shared between the program component that raises the exception and the program component that handles it. No other intervening handler may intercept the exception value; only the designated handler is permitted to process it. This behavior may be readily modelled using dynamic classification. Exception values are dynamically classified, with the class of the value known only to the raiser and to the intended handler, and to no others.

One may wonder why dynamic, as opposed to static, classification is appropriate for exception values. To do otherwise—that is, to use static classification—would require a global commitment to the possible forms of exception value that may be used in a program. This creates problems for modularity, since any such global commitment must be made for the whole program, rather than for each of its components separately. Dynamic classification ensures that when any two components are integrated, the classes they introduce are disjoint from one another, avoiding integration problems while permitting separate development.

### 36.1 Statics

The language clsfd uses (dynamically generated) symbols (Chapter 34) as class identifiers. The syntax of clsfd extends that of \( \mathcal{L}\{\text{sym}\} \) with the following additional constructs:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type ( \tau )</td>
<td>:=</td>
<td>clsfd</td>
<td>clsfd</td>
</tr>
<tr>
<td>Expr ( e )</td>
<td>::=</td>
<td>( \text{in}<a href="e">a</a> ) ( \mid \text{ccase}(e; e_0; r_1, \ldots, r_n) )</td>
<td>( \text{in}<a href="e">a</a> ) ( \mid \text{ccase}(e; e_0; r_1, \ldots, r_n) )</td>
</tr>
<tr>
<td>Rule ( r )</td>
<td>::=</td>
<td>( \text{in?}<a href="x.e">a</a> )</td>
<td>( \text{in}<a href="x">a</a> \Rightarrow e )</td>
</tr>
</tbody>
</table>

The expression \( \text{in}[a](e) \) classifies the value of the expression \( e \) by labelling it with the symbol \( a \). The expression \( \text{ccase} e \{ r_1 \mid \ldots \mid r_n \} e_0 \) analyzes the class of \( e \) using the rules \( r_1, \ldots, r_n \). Rule \( r_i \) has the form \( \text{in}[a_i](x_i) \Rightarrow e_i \), consisting of a symbol, \( a_i \), representing a candidate class of the analyzed value; a variable, \( x_i \), representing the associated data value for a value of that class; and an expression, \( e_i \), to be evaluated in the case that the analyzed expression is labelled with class \( a_i \). If the class of the analyzed value does not match any of the rules, the default expression, \( e_0 \), is evaluated instead. A default case is required, since no static type system can, in general,
circumscribe the set of possible classes of a classified value, and hence pattern matches on classified values cannot be guaranteed to be exhaustive.

The static semantics of \texttt{clsfd} extends that of \texttt{L\{\textbf{sym}\}} with the following additional rules:

\[
\Sigma \vdash a : \tau \quad \Gamma \vdash e : \tau \\
\Gamma \vdash \text{\texttt{in}}[a](e) : \texttt{clsfd}
\]

\[
\Gamma \vdash e : \texttt{clsfd} \quad \Gamma \vdash e_0 : \tau \quad \Gamma \vdash r_1 : \sigma_1 > \tau \quad \ldots \quad \Gamma \vdash r_n : \sigma_n > \tau \\
\Gamma \vdash \text{\texttt{ccase}}(e; e_0; r_1, \ldots, r_n) : \tau
\]

\[
\Sigma \vdash a : \sigma \quad \Gamma, x : \sigma \vdash e : \tau \\
\Gamma \vdash \text{\texttt{in}}?[a](x.e) : \sigma > \tau
\]

### 36.2 Dynamics

The dynamics of \texttt{clsfd} extends that of \texttt{L\{\textbf{sym}\}} (see Chapter 34) to give meaning to the classification constructs. We will assume here an unscoped dynamics for symbols, since this best reflects the intended usage of dynamic classification.

\[
\begin{array}{c}
e \text{val}_{\Sigma,a: \tau} \\
\text{\texttt{in}}[a](e) \text{ val}_{\Sigma,a: \tau}
\end{array}
\]

\[
\begin{array}{c}
e \otimes \Sigma \mapsto e' \otimes \Sigma' \\
\text{\texttt{in}}[a](e) \otimes \Sigma \mapsto \text{\texttt{in}}[a](e') \otimes \Sigma'
\end{array}
\]

\[
\begin{array}{c}
e \otimes \Sigma \mapsto e' \otimes \Sigma' \\
\text{\texttt{ccase}}(e; e_0; r_1, \ldots, r_n) \otimes \Sigma \mapsto \text{\texttt{ccase}}(e'; e_0; r_1, \ldots, r_n) \otimes \Sigma'
\end{array}
\]

\[
\begin{array}{c}
\text{\texttt{in}}[a](e) \text{ val}_{\Sigma} \\
\text{\texttt{ccase}}(\text{\texttt{in}}[a](e); e_0; e) \otimes \Sigma \mapsto e_0 \otimes \Sigma
\end{array}
\]

\[
\begin{array}{c}
\text{\texttt{in}}[a](e) \text{ val}_{\Sigma} \\
a = a_1 \\
\text{\texttt{ccase}}(\text{\texttt{in}}[a](e); e_0; \text{\texttt{in}}?[a_1](x_1.e_1), \ldots) \otimes \Sigma \\
\mapsto [e/x_1]e_1 \otimes \Sigma
\end{array}
\]

\[
\begin{array}{c}
\text{\texttt{in}}[a](e) \text{ val}_{\Sigma} \\
a \neq a_1 \\
n > 0 \\
\text{\texttt{ccase}}(\text{\texttt{in}}[a](e); e_0; \text{\texttt{in}}?[a_1](x_1.e_1), \text{\texttt{in}}?[a_2](x_2.e_2), \ldots) \otimes \Sigma \\
\mapsto \text{\texttt{ccase}}(\text{\texttt{in}}[a](e); e_0; \text{\texttt{in}}?[a_2](x_2.e_2), \ldots) \otimes \Sigma
\end{array}
\]
36.3 Defining Classification

Dynamic classification is definable in a language with symbols, products, and existentials. Specifically, the type \textit{clsfd} may be considered to stand for the existential type

$$\exists (t.t\text{sym} \times t).$$

The classified value in\{a\}(e) is defined to be the package

$$\text{pack } \tau \text{ with } (\text{sym}[a],e) \text{ as } \exists (t.t\text{sym} \times t),$$

where \(a\) is a symbol of type \(\tau\). Now suppose that the class case expression

\[\text{ccase } e \{ r_1 | \ldots | r_n \} \text{ ow } e'\]

has type \(\rho\), where \(r_i\) is the rule in\{a\}(\(x_i\) \(\Rightarrow e_i : \tau_i > \rho\)). This expression is defined to be

$$\text{open } e \text{ as } t \text{ with } (x,y):t \text{sym} \times t \text{ in } (e_{\text{body}}(y)),$$

where \(e_{\text{body}}\) is an expression to be defined shortly. Case analysis proceeds by opening the package, \(e\), representing the classified value, and decomposing it into a type, \(t\), a symbol, \(x\), of type \(t\text{sym}\), and an underlying value, \(y\), of type \(t\). The body of the \textit{open} analyzes the class \(x\), yielding a function of type \(t \rightarrow \rho\), which is then applied to \(y\) to pass the underlying value to the appropriate branch.

The core of the case analysis, namely the expression \(e_{\text{body}}\), analyzes the encapsulated class, \(x\), of the package. The case analysis is parameterized by the type abstractor \(u.u \rightarrow \rho\), where \(u\) is not free in \(\rho\). The overall type of the case is \([t/u]u \rightarrow \rho = t \rightarrow \rho\), which ensures that the application to \(y\) to the classified value is well-typed. Each branch of the case analysis has type \(\tau_i \rightarrow \rho\). Putting it all together, the expression \(e_{\text{body}}\) is defined to be the expression

\[\text{scase } x \{ r'_1 | \ldots | r'_n \} \text{ ow } \ldots \lambda (\ldots.t.e_0),\]

where for each \(1 \leq i \leq n\), the rule \(r'_i : \tau_i > (\tau_i \rightarrow \rho)\) is defined to be

\[\text{cls } [a_i] \Rightarrow \lambda (x_i : \tau_i.e_i).\]
One may check that the static and dynamic semantics of clsfd are derivable according to these definitions.

36.4 Exercises

1. Derive the Standard ML exception mechanism from the machinery developed here.
Part XIII

Storage Effects
Chapter 37

Reynolds’s IA

Reynolds’s Idealized Algol, or IA, augments the expression types with a command type and higher-order function types to form an elegant block-structured programming language reminiscent of the classic language Algol. Like its progenitor, IA features a rich higher-order recursive function mechanism on top of a simple imperative language of commands. Commands are executed for their effect on the contents of the assignable variables, or just assignables, that are active within that command. Assignables are introduced by declaring them, and their contents are retrieved and altered by operations to get and set their contents, respectively. Commands may be sequenced one after the other, and there is a null command to end such a sequence. Repetition is achieved by a combination of sequencing and recursion.

The language IA imposes a modal distinction between pure expressions and impure commands. The meaning of a pure expression is independent of the assignables that may in scope at that point. This ensures that there are no constraints on the order of evaluation of the components of a compound expression. The meaning of a command, however, depends on and effects the contents of the assignables, requiring a strictly specified evaluation order to ensure that the behavior of a command is predictable (that is, execution is deterministic).

The language IA is carefully designed to adhere to the stack discipline for allocating assignables. This means that an assignable may be deallocated on exit from the scope of its declaration, and hence that assignables may be allocated on a stack. This avoids the need for any form of automatic storage management beyond the conventional run-time stack.\(^1\) Surprisingly, this

\(^1\)A run-time stack is, of course, a simple form of automatic storage management, but it is
property is maintained even in the presence of a type of references (abstract pointers) to assignable variables.

### 37.1 Commands

The syntax of $\mathcal{L}\{\text{nat cmd} \to \}$ distinguishes pure expressions from impure commands. The expressions include those of $\mathcal{L}\{\text{nat} \to \}$ (as described in Chapter 15), augmented with one additional construct, and the commands are those of a simple imperative programming language based on assignment. The language maintains a sharp distinction between mathematical variables, or just variables, and assignable variables, or just assignables. Variables are introduced by $\lambda$-abstraction, and are given meaning by substitution. Assignables are introduced by a scoped 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 37.4 on page 336 for the general case.)

The syntax of $\mathcal{L}\{\text{nat cmd} \to \}$ is given by the following grammar, from which we have omitted repetition of the expression syntax of $\mathcal{L}\{\text{nat} \to \}$ for the sake of brevity.

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>cmd</td>
<td>cmd</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>cmd($m$)</td>
<td>cmd($m$)</td>
</tr>
<tr>
<td>Cmd</td>
<td>$m$</td>
<td>ret</td>
<td>ret</td>
</tr>
<tr>
<td></td>
<td></td>
<td>seq($e$; $x$. $m$)</td>
<td>$x \leftarrow e$; $m$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dcl($e$; $a$. $m$)</td>
<td>dcl $a := e$ in $m$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>get[$a$]</td>
<td>$a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set<a href="$e$">$a$</a></td>
<td>$a := e$</td>
</tr>
</tbody>
</table>

The expression cmd($m$) consists of the unevaluated command, $m$, thought of as a value of type cmd. The command, ret$e$, returns the value of the expression $e$ without having any effect on the assignables. The command

---

popular to draw a distinction between it and a more general form, called garbage collection. The difference lies largely in the eyes of the beholder.
37.1 Commands

seq(e; x. m) evaluates e to an encapsulated command, which is then executed and its returned value is substituted for x prior to executing m. The command 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 get[a] returns the current contents of the assignable, a, and the command set[a](e) changes the contents of the assignable a to the value of e, and returns that value.

37.1.1 Statics

The static semantics of $L\{\text{nat cmd} \rightarrow \}$ consists of two forms of judgement:

1. Expression typing: $\Gamma \vdash_{\Sigma} e : \tau$.
2. Command formation: $\Gamma \vdash_{\Sigma} m \text{ ok}$.

The context, $\Gamma$, specifies the types of variables, as usual, and the state, $\Sigma$, consists of a finite set of assignables. These judgements are inductively defined by the following rules:

$$\Gamma \vdash_{\Sigma} m \text{ ok} \quad \Rightarrow \quad \Gamma \vdash_{\Sigma} \text{cmd}(m) : \text{cmd} \quad \text{(37.1a)}$$

$$\Gamma \vdash_{\Sigma} e : \text{nat} \quad \Rightarrow \quad \Gamma \vdash_{\Sigma} \text{ret}(e) \text{ ok} \quad \text{(37.1b)}$$

$$\Gamma \vdash_{\Sigma} e : \text{cmd} \quad \Gamma, x : \text{nat} \vdash_{\Sigma} m \text{ ok} \quad \Rightarrow \quad \Gamma \vdash_{\Sigma} \text{seq}(e; x. m) \text{ ok} \quad \text{(37.1c)}$$

$$\Gamma \vdash_{\Sigma} e : \text{nat} \quad \Gamma \vdash_{\Sigma, a} m \text{ ok} \quad \Rightarrow \quad \Gamma \vdash_{\Sigma, a} \text{dcl}(e; a. m) \text{ ok} \quad \text{(37.1d)}$$

$$\Gamma \vdash_{\Sigma, a} e : \text{nat} \quad \Rightarrow \quad \Gamma \vdash_{\Sigma, a} \text{set}[a](e) \text{ ok} \quad \text{(37.1e)}$$

$$\Gamma \vdash_{\Sigma, a} \text{get}[a] \text{ ok} \quad \text{(37.1f)}$$

Rule (37.1a) is the introductory rule for the type cmd, and Rule (37.1c) is the corresponding eliminatory form. Rule (37.1d) introduces a new assignable for use within a specified command. The name, $a$, of the assignable is bound by the declaration, and hence may be renamed to satisfy the implicit constraint that it not already be present in $\Sigma$. Rule (37.1f) states that
the command to retrieve the contents of an assignable, \(a\), returns a natural number. Rule (37.1e) that we may assign a natural number to an assignable, and that the returned value, which will be the assigned value, is of the same type.

37.1.2 Dynamics

The dynamic semantics of \(L\{\text{nat cmd} \rightarrow\}\) is defined in terms of a memory, \(\mu\), a finite function assigning a numeral to each of a finite set of assignables.

The dynamics of expressions consists of these two judgement forms:

1. \(e \text{ val}_\Sigma\), stating that \(e\) is a value relative to \(\Sigma\).
2. \(e \overset{\Sigma}{\rightarrow} e'\), 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 dynamic semantics of \(L\{\text{nat} \rightarrow\}\) (see Chapter 15). It is important, however, that the successor operation be given an \textit{eager}, rather than \textit{lazy}, semantics so that a closed value of type \textsf{nat} is a numeral (finite composition of successors starting with zero).

\[
\text{cmd}(m) \text{ val}_\Sigma \quad (37.2a)
\]

Rule (37.2a) states that an encapsulated command is a value.

The dynamics of commands consists of these two judgement forms:

1. \(m \oplus_\mu \text{ final}_\Sigma\) stating that the state \(m \oplus_\mu\) is fully executed.
2. \(m \oplus_\mu \overset{\Sigma}{\rightarrow} m' \oplus_\mu'\) stating that the state \(m \oplus_\mu\) steps to the state \(m \oplus_\mu, \) relative to the set of assignables, \(\Sigma\).

These judgements are inductively defined by the following rules:

\[
\text{ret}(e) \oplus_\mu \text{ final}_\Sigma \quad (37.3a)
\]

\[
\text{seq}(e; x. m) \oplus_\mu \overset{\Sigma}{\rightarrow} \text{seq}(e'; x. m) \oplus_\mu \quad (37.3b)
\]

\[
\text{cmd}(m_1) \oplus_\mu \overset{\Sigma}{\rightarrow} \text{cmd}(m'_1) \oplus_\mu' \quad (37.3c)
\]
37.1 Commands

\[
e \text{val}_\Sigma \quad \text{seq}(\text{cmd}(\text{ret}(e)); x.m) @ \mu \xrightarrow{\Sigma} [e/x]m @ \mu
\]

(37.3d)

\[
\text{get}[a] @ \mu \otimes \langle a:e \rangle \xrightarrow{\Sigma^a} \text{ret}(e) @ \mu \otimes \langle a:e \rangle
\]

(37.3e)

\[
e \xrightarrow{\Sigma} e'
\]

\[
\text{set}[a](e) @ \mu \xrightarrow{\Sigma} \text{set}[a](e') @ \mu
\]

(37.3f)

\[
\text{set}[a](e) @ \mu \otimes \langle a: \_ \rangle \xrightarrow{\Sigma} \text{ret}(e) @ \mu \otimes \langle a:e \rangle
\]

(37.3g)

\[
e \xrightarrow{\Sigma} e'
\]

\[
\text{dcl}(e;a.m) @ \mu \xrightarrow{\Sigma}, \text{dcl}(e';a.m) @ \mu
\]

(37.3h)

\[
e \text{val}_\Sigma \quad m @ \mu \otimes \langle a:e \rangle \xrightarrow{\Sigma^a} m' @ \mu' \otimes \langle a:e' \rangle
\]

(37.3i)

\[
\text{dcl}(e;a.m) @ \mu \xrightarrow{\Sigma}, \text{dcl}(e';a.m') @ \mu'
\]

(37.3j)

\[
e \text{val}_\Sigma \quad e' \text{val}^a_{\Sigma,a} \quad \text{dcl}(e;a.\text{ret}(e')) @ \mu \xrightarrow{\Sigma} \text{ret}(e') @ \mu
\]

Rule (37.3a) specifies that a \text{ret} command returns the value of its argument. Rules (37.3b) to (37.3d) specify the semantics 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 (37.3e) to (37.3g) define the semantics of retrieval and assignment to an assignable variable. Rules (37.3h) to (37.3j) for declaring an assignable define the concept of block structure. A declaration is executed by extending the memory with a fresh assignable with initial value specified in the declaration, and executing the body to completion. Execution of the body may alter the contents of any assignable variable, including the declared one. Once the body has been executed, its return value is used as the return value of the declaration.

37.1.3 Safety

The auxiliary judgement \(\mu : \Sigma\) is defined by these two conditions:

1. \(\text{dom}(\mu) = \Sigma\);
2. for each $a \in \Sigma$, there is an $e$ such that $e \text{val}_\Sigma$ and $\vdash e : \text{nat}$.

This judgement states that the memory provides bindings for each of the assignables in $\Sigma$, and that each binding is a closed value of type nat (since assignables can only contain natural numbers).

A state is well-formed, written $m @ \mu \text{ok}_\Sigma$, iff $\mu : \Sigma$ and $\vdash m : \text{ok}_\Sigma$.

**Theorem 37.1** (Preservation).

1. If $e \overset{\Sigma}{\rightarrow} e'$ and $\vdash e : \tau$, then $\vdash e' : \tau$.

2. If $m @ \mu \text{ok}_\Sigma$ and $m @ \mu \overset{\Sigma}{\rightarrow} m' @ \mu'$, then $m' @ \mu' \text{ok}_\Sigma$.

**Proof.** By induction on Rules (37.2) and (37.3). Consider Rule (37.3i). Assume that $\text{dc1}(e; a.m) @ \mu \text{ok}_\Sigma$. By inversion of typing we have $\vdash e : \text{nat}$, $\vdash m : \Sigma$, and $\mu : \Sigma$. Since $e \text{val}_\Sigma$, it follows that $\mu \otimes (a : e) : \Sigma, a$, and hence $m @ \mu \otimes (a : e) \text{ok}_\Sigma$. By induction we have $m' @ \mu' \otimes (a : e') \text{ok}_\Sigma$, from which it follows that $\text{dc1}(e'; a.m') @ \mu' \text{ok}_\Sigma$, as required.

Consider Rule (37.3i). Assume that $\text{dc1}(e; a.\text{ret}(e')) @ \mu \text{ok}_\Sigma$. We are to show that $\text{ret}(e) @ \mu \text{ok}_\Sigma$. By inversion we know that $\vdash e : \text{nat}$, $\vdash e : \text{nat}$, and hence that $\vdash e' : \text{nat}$. It suffices to show that $\vdash e' : \text{nat}$. But since $e' \text{val}_\Sigma$, the eager semantics of the successor operation ensures that $e'$ must be a numeral, and hence $\vdash e' : \text{nat}$, as required. \qed

**Theorem 37.2** (Progress).

1. If $\vdash e : \tau$, then either $e \text{val}_\Sigma$, or there exists $e'$ such that $e \overset{\Sigma}{\rightarrow} e'$.

2. If $m @ \mu \text{ok}_\Sigma$, then either $m \text{final}_\Sigma$, or $m @ \mu \overset{\Sigma}{\rightarrow} m' @ \mu'$ for some $\mu'$ and $m'$.

**Proof.** By induction on Rules (37.1). Consider Rule (37.1d). By the first inductive hypothesis we have either $e \overset{\Sigma}{\rightarrow} e'$ or $e \text{val}_\Sigma$. In the former case Rule (37.3h) applies. In the latter, we have by the second inductive hypothesis either $m \text{final}_\Sigma$ or $m @ \mu \otimes (a : e) \overset{\Sigma}{\rightarrow} m' @ \mu \otimes (a : e')$. In the former case we apply Rule (37.3i), and in the latter, Rule (37.3j). \qed
37.2 Some Programming Idioms

The language $L\{\text{nat cmd} \rightarrow \}$ 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 $L\{\text{nat cmd} \rightarrow \}$.

We begin by defining the binary sequential composition of commands, written $\{x \leftarrow m_1 ; m_2\}$, to be the command $\text{seq}(\text{cmd}(m_1); x . m_2)$. This generalizes to the $n$-ary sequential composition by defining

$$\{x_1 \leftarrow m_1 ; \ldots; x_{n-1} \leftarrow m_{n-1} ; m_n\},$$

to stand for the iterated form

$$\{x_1 \leftarrow m_1 ; \ldots; \{x_{n-1} \leftarrow m_{n-1} ; \{x_n \leftarrow m_n ; \text{ret } x_n\}\}\}. $$

A related idiom, written $\text{run } e$, which executes an encapsulated command and returns its result, is defined to be the command $\text{seq}(e; x . \text{ret } x)$. The conditional command, $\text{ifz } (m) \ m_1 \ \text{else } m_2$, executes either $m_1$ or $m_2$ according to whether the result of executing $m$ is zero or not:

$$\text{seq}(\text{cmd}(m); x . \text{seq}(\text{ifz}(x; \text{cmd}(m_1); \text{cmd}(m_2)); y . \text{ret } (y))).$$

The returned value of the conditional is the value returned by the selected command.

The loop command, $\text{whilenz } (m_1) \ m_2$, repeatedly executes the command $m_2$ while the command $m_1$ yields a non-zero number. It is defined as follows:

$$\text{run fix loop } : \text{cmd is cmd(\text{ifz}(m_1) \ \text{ret } z \ \text{else do } \{x \leftarrow m_2 ; \text{run } \text{loop}\})}. $$

This command 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 $\tau \rightarrow \text{cmd}$ that takes an argument of some type $\tau$, 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 $\text{call } e_1(e_2)$ is defined to be the command $\text{run } (e_1(e_2))$, which immediately runs the result of applying $e_1$ to $e_2$. 
37.3 References to Assignables

The commands `get(a)` and `set(a; e)` make explicit the assignable on which they operate. This has the advantage that if \( a \neq b \), then an assignment to \( a \) cannot possibly affect the contents of \( b \), nor the other way around. In some situations it can be useful to defer the choice of target for an assignment to execution time. For example, one may wish to pass to a procedure an assignable determined by the caller with the intention of having the body of the procedure assign some result to it. The procedure cannot itself use a statement of the form `set(a; e)`, precisely because it does not know what assignable to use as the target of the assignment. What is required is to pass the procedure the capability to assign to an assignable of the caller’s choice, even if the target is not accessible to the called procedure.

One way to achieve this is to enrich the language with a type of references to assignables, which may be thought of as abstract “pointers” to assignables. Reference types provide operations to get and set the contents of an assignable based on a reference to it. Surprisingly, adding references does not disrupt the block structure of the language, nor does it compromise safety. The syntax this extension is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau )</td>
<td>ref</td>
<td>ref</td>
</tr>
<tr>
<td>Expr</td>
<td>( e )</td>
<td>ref[a]</td>
<td>ref[a]</td>
</tr>
<tr>
<td>Cmd</td>
<td>( m )</td>
<td>get(e)</td>
<td>!e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set(e_1; e_2)</td>
<td>( e_1 := e_2 )</td>
</tr>
</tbody>
</table>

The `set` and `get` primitives take an argument of type `ref` that determines the target assignable.

The static semantics of this extension is given by the following rules:

\[
\Gamma \vdash_{\Sigma} \ref[a] : \ref \quad (37.4a)
\]

\[
\Delta \vdash_{\Sigma} e : \ref \quad \Gamma \vdash_{\Sigma} \text{get}(e) \text{ ok} \quad (37.4b)
\]

\[
\Delta \vdash_{\Sigma} e_1 : \ref \quad \Gamma \vdash_{\Sigma} e_2 : \text{nat} \quad \Gamma \vdash_{\Sigma} \text{set}(e_1; e_2) \text{ ok} \quad (37.4c)
\]

Rule (37.4a) specifies that the name of any active assignable is an expression of type \( \text{ref} \).
The dynamic semantics determines the underlying assignable, then performs the corresponding operation on it.

\[
\text{ref}[a] \triangleright \text{val}_{\Sigma,a}
\]  

(37.5a)

\[
e \xrightarrow{\Sigma} e' \\
\text{get}(e) @ \mu \xrightarrow{\Sigma} \text{get}(e') @ \mu
\]  

(37.5b)

\[
\text{get(ref}[a]) @ \mu \xrightarrow{\Sigma} \text{get}[a] @ \mu
\]  

(37.5c)

\[
e_1 \xrightarrow{\Sigma} e'_1 \\
\text{set}(e_1; e_2) @ \mu \xrightarrow{\Sigma} \text{set}(e'_1; e_2) @ \mu
\]  

(37.5d)

\[
\text{set(ref}[a]; e) @ \mu \xrightarrow{\Sigma} \text{set}[a](e) @ \mu
\]  

(37.5e)

The inclusion of the type \text{ref} does not disrupt the block structure of the language. The essential point is that commands cannot (directly or indirectly) return values that depend on locally declared assignables—in particular, a reference to an assignable cannot be exported outside of the scope of its declaration.

It is a good exercise to check the safety of the extension of \( \mathcal{L}\{\text{nat cmd} \to \} \) with reference types.

The addition of reference types introduces the problem of aliasing. While it is obvious that if \( a \) and \( b \) are different assignables, then an assignment to \( a \) cannot influence the contents of \( b \), the analogous situation with references is not so easily handled. For if \( x \) and \( y \) are distinct variables of type \text{ref}, then an assignment to \( x \) may well affect the contents of \( y \), precisely because \( x \) and \( y \) could turn out to be a reference to the same assignable! Reasoning about programs that involve references is error-prone, because we must consider all possible pairwise aliasing relationships of the variables involved to convince ourselves of its correctness in all circumstances. Many programming languages fail to distinguish between assignables and variables, treating all variables as references to assignables, with the consequence that the possibilities for aliasing increase quadratically in the number of variables in scope at a given point.
37.4 Typed Commands and Assignables

So far we have restricted the return value of a command, and the contents of an assignable, to be of type nat. Can this restriction be relaxed? The main complication is that we wish to ensure that the dynamics given in Section 37.1.2 on page 330 is not disrupted. In particular, the language should remain *stack-implementable* in the sense that an assignable can be deallocated from the memory upon exiting the scope of its declaration. The proof of Theorem 37.1 on page 332 relies on the fact that the returned value of a command can only be a fully evaluated natural number.

To see what can go wrong, suppose that we allow commands to return a value of arbitrary type. Absent any restrictions on the type \( \tau \), the resulting language is not type safe. For example, if we may return values of procedure type, then we may violate safety as follows:

\[
\text{dcl } a := z \text{ in ret } (\lambda (x : \text{nat} . \text{cmd } a := x)).
\]

This command, when executed, allocates a new assignable, \( a \), and returns a procedure that, when called, assigns its argument to \( a \). But this makes no sense, because the assignable, \( a \), is deallocated when the body of the declaration returns, but the returned value still refers to it! If the returned procedure is called, execution will get stuck in the attempt to assign to \( a \).

A similar example shows that admitting assignables of arbitrary type is also unsound:

\[
\text{dcl } a := z \text{ in } b := \lambda (x : \text{nat} . \text{cmd } a := x).
\]

Here we assume that \( b \) is an assignable of procedure type. We assign to it a procedure that assigns to 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 \( a \)!

Recall that the critical step in the proof of safety given in Section 37.1.3 on page 331 is to ensure the following *safety condition*:

\[
\text{if } \vdash_{\Sigma,a} e : \tau \text{ and } e \text{ val}_{\Sigma,a}, \text{ then } \vdash_{\Sigma} e : \tau. \tag{37.6}
\]

When \( \tau = \text{nat} \), this step is ensured, because \( e \) must be a numeral. If, on the other hand, \( \tau \) is a procedure type, then \( e \) may contain uses of the locally declared assignable, \( a \), and, indeed, the above counterexamples exploit exactly this possibility.
We say that a type, $\tau$, is mobile, written $\tau$ mobile, if the safety condition (37.6) is valid. The proof of safety shows that $\texttt{nat}$ is mobile. Obviously, reference types are not mobile. The counterexamples show that procedure types are not mobile, and simple variations of these examples show that command types are not mobile either. What about function types that are not procedure types, such as $\texttt{nat} \rightarrow \texttt{nat}$? One may think so, because assignables may only be used in commands, and not within pure expressions, and hence a pure function cannot depend on an assignable. While this is, in fact, the case, the safety condition (37.6) need not be satisfied for such a type. For example, consider the following value of type $\texttt{nat} \rightarrow \texttt{nat}$:

$$\lambda(x: \texttt{nat}. \lambda(_: \texttt{cmd}. z)(\texttt{cmd}(\texttt{get}[a])))$$

Although the assignable $a$ is not actually needed to compute the result, it nevertheless occurs in the value, in violation of the safety condition.

### 37.5 Exercises
Chapter 38

Mutable Cells

Data types constructed from sums, products, and recursive types classify immutable data in that a value of such a type, once constructed, cannot be changed. For example, the type of lists of natural numbers, which may be defined to be the recursive type \( \mu t. \text{unit} + \text{nat} \times t \), consists of finite sequences of natural numbers represented using sums to distinguish empty from nonempty lists, and using recursive folds to mediate the recursion. A value, \( l \), of this type is finite sequence whose elements are fixed for all time. There is no possibility to “remove” or “change” an element of \( l \) itself, but we may, of course, compute with \( l \) to produce a separate list, \( l' \), that is computed from \( l \) by, say, deleting all occurrences of zero from \( l \), or by appending another list to it. Using \( l \) in this manner does not alter or destroy it, so that it can, of course, be used in further computations. For this reason, immutable data structures, such as lists, are said to be persistent, because they permit the original data object to be used even after an operation has been applied to it.

This behavior is in sharp contrast to conventional textbook treatments of data structures such as lists and trees, which are invariably defined by destructive operations that modify, or mutate, the data structure “in place”. Inserting an element into a binary tree changes the tree itself to include the new element; the original tree is lost in the process, and all references to it reflect the change. Such data structures are said to be ephemeral, in that changes to them destroy the original. In some cases ephemeral data structures are essential to the task at hand; in other cases a persistent representation would do just as well, or even better. For example, a data structure modeling a shared database accessed by many users simultaneously is naturally ephemeral in that the changes made by one user are to be immedi-
ately propagated to the computations made by another. On the other hand, data structures used internally to a body of code, such as a search tree, need no such capability, and may often be usefully represented persistently.

A natural way to support both persistent and ephemeral data structures is to introduce the type \( \tau \text{ref} \) of references to mutable cells holding a value of type \( \tau \). A value of this type is the name of, or a reference to, a mutable cell whose contents may change without changing its identity. Since references are values, they may be passed as arguments to or returned as values from functions, and may appear as components of a data structure. This means that alterations to the contents of a mutable cell may be made at one or more sites far removed from the site at which it was created. This is both a boon and a bane. On the one hand this sort of “action at a distance” can be a very useful programming device, but on the other it is for this very reason that it is difficult to ensure correctness of programs that use mutable storage. In a fully expressive language one has the opportunity, but not the obligation, to use mutation; you pay your money and you take your chances. Many less expressive languages offer nothing but mutable data structures, needlessly emphasizing ephemeral over persistent data structures.

By combining reference types with other type constructors we may represent a rich variety of data structures. For example, the type \( \text{nat} \text{ref} \times \text{nat} \text{ref} \) consists of a pair of references to mutable natural numbers, whereas the type \( (\text{nat} \times \text{nat}) \text{ref} \) consists of a reference to a mutable cell containing immutable pairs of natural numbers. To take another example, the type \( \text{nat} \text{ref} \rightarrow \text{nat} \) consists of functions that take a mutable cell as argument and return a natural number as result. The contents of the argument cell may be accessed or altered by the function itself, the caller, or both.

In this chapter we consider two ways to incorporate reference types in a programming language that differ in whether the reliance on mutable storage is made explicit in the type. In the modal, or monadic, approach operations on mutable cells are (impure) commands, rather than (pure) expressions. Unevaluated commands may be packaged up as values that may be passed as arguments, returned as results, or occur in data structures. Consequently, no restrictions on evaluation order need be imposed; the modal approach is equally compatible with either by-name or by-value application, and with eager or lazy data structures. In the integral, or non-modal, formulation operations on mutable cells are forms of expression that may appear anywhere in a program. To ensure that mutation effects occur in a predictable and controllable manner we impose a strict, call-by-value evaluation order for all constructs of the language.
38.1 Modal Formulation

A mutable cell is a persistent assignable variable—one whose validity extends beyond the scope in which it is declared. Equivalently, we eschew the scoping of assignable variables in favor of a single global scope encompassing the declarations of all active mutable cells ever allocated in a program. This ensures that a cell may be embedded in a data structure, or stored in another mutable cell, without concern for exceeding the scope of its validity.

The modal formulation of mutable cells, called $\mathcal{L}\{\text{ref cmd}\}$, is a simple modification of the modal formulation of assignable variables given in Chapter 37. We simply decree that all types are mobile, so that a value of any type may be stored in a variable, or returned as the result of executing a command. This, of course, ruins the stack implementability of the language, since now references to assignable variables may escape the scope of their declaration. In compensation we give a dynamic semantics in which assignable variables are heap allocated, rather than stack allocated. This means that the “scope” of an assignable variable is global, rather than confined to a local declaration, allowing references to it to be used freely anywhere in the program.

38.1.1 Syntax

The syntax of $\mathcal{L}\{\text{ref cmd}\}$ is derived from that of $\mathcal{L}\{\text{nat cmd $\rightarrow$}\}$, with a few modifications and simplifications arising from eliminating the mobility restrictions on the types of variables and commands. Most significantly, since assignable variables are to be dynamically allocated on a global heap, it is no longer sensible to track their scope of validity in the static semantics. Accordingly we consider only the type of dynamically determined references described in Section 37.3 on page 334, and eliminate the primitives for getting and setting statically determined assignable variables. Since there is no longer any need to track the scope of an assignable variable, we replace the declaration primitive by a command to allocate a reference to a new variable whose result is that reference. (This is sensible because a command may now return a value of any type.)
The syntax of $\mathcal{L}\{\text{ref cmd}\}$ is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$\text{cmd}(\tau)$</td>
<td>$\tau\text{cmd}$</td>
</tr>
<tr>
<td></td>
<td>$\tau\text{ref}$</td>
<td>$\text{ref}(\tau)$</td>
<td>$\tau\text{ref}$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$\text{cmd}(m)$</td>
<td>$\text{cmd},m$</td>
</tr>
<tr>
<td></td>
<td>$\text{ref}[a]$</td>
<td>$\text{ref}[a]$</td>
<td>$\text{ref}[a]$</td>
</tr>
<tr>
<td>Comm</td>
<td>$m$</td>
<td>$\text{ret}(e)$</td>
<td>$\text{ret},e$</td>
</tr>
<tr>
<td></td>
<td>$x \leftarrow e,;,m$</td>
<td>$\text{seq}(e,;,x,.,m)$</td>
<td>$x \leftarrow e,;,m$</td>
</tr>
<tr>
<td></td>
<td>$\text{new}<a href="e">\tau</a>$</td>
<td>$\text{new},e$</td>
<td>$\text{new},e$</td>
</tr>
<tr>
<td></td>
<td>$\text{get}(e)$</td>
<td>$\text{get},e$</td>
<td>$\text{get},e$</td>
</tr>
<tr>
<td></td>
<td>$e_1 := e_2$</td>
<td>$\text{set}(e_1;,e_2)$</td>
<td>$e_1 := e_2$</td>
</tr>
</tbody>
</table>

The type $\tau\text{ref}$ is the type of references to heap-allocated cells. The operations $\text{get}\,e$ and $e_1 := e_2$ retrieve and assign, respectively, the contents of a heap-allocated cell, given a dynamically determined reference to it. The apparatus of commands remains as in Section 37.4 on page 336, but with all restrictions on mobility lifted (that is, by regarding all types to be mobile).

### 38.1.2 Statics

The static semantics of $\mathcal{L}\{\text{ref cmd}\}$ is defined similarly to Chapter 37, with the following rules for reference types replacing those for variable types:

\[
\Gamma \vdash_{\Sigma,a:\tau} \text{ref}[a] : \text{ref}(\tau) \quad (38.1a)
\]

\[
\Gamma \vdash_{\Sigma} e : \tau \\
\Gamma \vdash_{\Sigma} \text{new}[\tau](e) \sim \text{ref}(\tau) \quad (38.1b)
\]

\[
\Gamma \vdash_{\Sigma} e : \text{ref}(\tau) \\
\Gamma \vdash_{\Sigma} \text{get}(e) \sim \tau \quad (38.1c)
\]

\[
\Gamma \vdash_{\Sigma} e_1 : \text{ref}(\tau) \\
\Gamma \vdash_{\Sigma} e_2 : \tau \\
\Gamma \vdash_{\Sigma} \text{set}(e_1;\,e_2) \sim \tau \quad (38.1d)
\]

The only role of $\Sigma$ in the static semantics is to determine the type of a reference, $\text{ref}[a]$. Moreover, $\Sigma$ never changes in the static semantics, but rather is determined by the dynamic semantics as new cells are allocated by executing the command $\text{new}[\tau](e)$. The context $\Sigma$ is taken to be empty in the initial state, ensuring that references only arise via this mechanism, and cannot appear in a source program.
38.1 Modal Formulation

38.1.3 Dynamics

The dynamics of $L\{\text{nat cmd ref} \rightarrow\}$ consists of the following judgements:

\[ e \text{ val}_\Sigma \]  
\[ e \overset{\Sigma}{\rightarrow} e' \]  
\[ m @ \mu : \Sigma \mapsto m' @ \mu' : \Sigma' \]

where $e$ is a value in context $\Sigma$, $e$ steps to $e'$ in context $\Sigma$, and $m$ in $\mu : \Sigma$ steps to $m'$ in $\mu' : \Sigma'$.

The declarations, $\Sigma$, specify the types of active references, and the memory, $\mu$, is a finite mapping that provides bindings for them. A significant difference compared to the dynamics of $L\{\text{nat cmd} \rightarrow\}$ given in Chapter 37 is that the execution of a command can both extend the domain of the memory as well as alter its contents. It is in this sense that cells are heap-allocated, whereas assignable variables are stack-allocated.

A representative selection of rules defining the dynamics of $L\{\text{ref cmd} \rightarrow\}$ is as follows:

\[ e \text{ val}_\Sigma \]
\[ \text{ret}(e) @ \mu : \Sigma \text{ final} \] (38.2a)

\[ \text{ref}[a] \text{ val}_{\Sigma,a:\tau} \] (38.2b)

\[ e \overset{\Sigma}{\rightarrow} e' \]
\[ \text{new}[\tau](e) @ \mu : \Sigma \mapsto \text{new}[\tau](e') @ \mu : \Sigma \] (38.2c)

\[ e \text{ val}_\Sigma \quad a \notin \text{dom}(\Sigma) \]
\[ \text{new}[\tau](e) @ \mu : \Sigma \mapsto \text{ref}[a] @ \mu \otimes (a:e) : \Sigma, a : \tau \] (38.2d)

\[ e \overset{\Sigma}{\rightarrow} e' \]
\[ \text{get}(e) @ \mu : \Sigma \mapsto \text{get}(e') @ \mu : \Sigma \] (38.2e)

\[ \text{get}(\text{ref}[a]) @ \mu \otimes (a:e) : \Sigma, a : \tau \mapsto e @ \mu \otimes (a:e) : \Sigma, a : \tau \] (38.2f)

\[ e_1 \overset{\Sigma}{\rightarrow} e'_1 \]
\[ \text{set}(e_1;e_2) @ \mu : \Sigma \mapsto \text{set}(e'_1;e_2) @ \mu : \Sigma \] (38.2g)

\[ e_1 \text{ val}_\Sigma \quad e_2 \overset{\Sigma}{\rightarrow} e'_2 \]
\[ \text{set}(e_1;e_2) @ \mu : \Sigma \mapsto \text{set}(e_1;e'_2) @ \mu : \Sigma \] (38.2h)

\[ e \text{ val}_\Sigma \]
\[ \text{set}(\text{ref}[a];e) @ \mu \otimes (a:e) : \Sigma, a : \tau \mapsto e @ \mu \otimes (a:e) : \Sigma, a : \tau \] (38.2i)
Rule (38.2b) states that a reference is a form of value. Rules (38.2c) and (38.2d) state that a reference is created by choosing a fresh name and binding it to an initial value. The remaining rules define the semantics of the get and set operations on references.

Execution of commands can only increase the set of active references; mutable cells are ever deallocated.

Lemma 38.1 (Monotonicity). Suppose that \( \text{dom}(\mu) = \text{dom}(\Sigma) \). If \( m \circ \mu : \Sigma \mapsto m' \circ \mu' : \Sigma' \), then \( \Sigma' \supseteq \Sigma \) and \( \text{dom}(\mu') = \text{dom}(\Sigma') \).

38.2 Integral Formulation

An alternative to the modal formulation of mutation is simply to add reference types to PCF so that any expression may have an effect as well as a value. This has the virtue of minimizing the syntactic overhead of distinguishing pure (effect-free) expressions from impure (effect-ful) commands, and the vice of severely weakening the meanings of typing assertions. In the modal formulation the type \( \text{unit} \rightarrow \text{unit} \) is “boring” in that it contains only the identity function and the divergent function, whereas in the integral formulation the same type is “interesting” because it contains many more functions that, when called, may refer to and alter the contents of reference cells. In the integral setting the type of an expression says less about its behavior than in the modal setting, precisely because it does not reveal whether the expression has effects when evaluated. While this may sound like a disadvantage, it can also be seen as an advantage. For if a context demands an expression of a type \( \tau \), we have the freedom in the integral case to provide any expression, including one with effects, whereas in the modal case the expressions of a type must always be pure. So, for example, if we wish to include effects that collect profiling information, we may easily do this in the integral setting, but must restructure the program in the modal setting to permit a command type \( \tau \) to be passed where previously an expression of this type was required. This can do violence to the structure of a program.

Just as the modal formulation of references relies on the elimination form for the type \( \text{cmd} \tau \) to sequence the order of effects in a command, so too must the integral formulation provide some means of sequencing effects in an expression. This can be achieved in several different ways, so long as the by-value \( \text{let} \) construct described in Chapter 10 is definable. For then we
may write \texttt{let} \(x\) be \(e_1\) in \(e_2\) to ensure that \(e_1\) is evaluated with full effect before \(e_2\) is evaluated at all. One way to achieve this is to include the by-value \texttt{let} construct as primitive. Another is to impose the call-by-value evaluation order for function applications, so that the by-value \texttt{let} is definable. If functions are evaluated by-name, and all data structures are evaluated lazily, then the sequentializing \texttt{let} is not definable, with crippling results. It is for this reason that the integral approach is only ever considered in the context of a strict, rather than lazy, programming language.

### 38.2.1 Statics

The statics of the integral formulation of references may be obtained by collapsing the mode distinction between expressions and commands, treating the operations on references as forms of expression.

\[
\Gamma \vdash_{\Sigma, \tau} \text{ref} \langle a \rangle : \text{ref} (\tau)
\]

(38.3a)

\[
\Gamma \vdash_{\Sigma} e : \tau
\]

(38.3b)

\[
\Gamma \vdash_{\Sigma} \text{new} [\tau] (e) : \text{ref} (\tau)
\]

(38.3c)

\[
\Gamma \vdash_{\Sigma} e : \text{ref} (\tau)
\]

\[
\Gamma \vdash_{\Sigma} \text{get} (e) : \tau
\]

(38.3d)

\[
\Gamma \vdash_{\Sigma} e_1 : \text{ref} (\tau) \quad \Gamma \vdash_{\Sigma} e_2 : \tau
\]

\[
\Gamma \vdash_{\Sigma} \text{set} (e_1; e_2) : \tau
\]

The remaining rules are the obvious adaptations of those given in Chapter 15, augmented by the assignment, \(\Sigma\), of types to references.

### 38.2.2 Dynamics

The dynamic semantics of the integral formulation of references consists of transition judgements of the form

\[ e \odot \mu : \Sigma \leftrightarrow e' \odot \mu' : \Sigma' . \]

This judgement states that each step of evaluation of an expression relative to a memory may alter or extend the memory.

The rules defining the dynamics of references are as follows:

\[
\text{ref} \langle a \rangle \text{ val}_{\Sigma, \tau}
\]

(38.4a)
The only difference compared to Rules (38.2) is that evaluation of sub-expressions can extend or alter the memory.

### 38.3 Safety

The proof of safety for a language with reference types must account for the types of the values stored in each active reference cell. For example, according to Rules (38.1) the command `get(ref[a])` has type `τ`, provided that `Σ` assigns the type `τ` to the reference `a`. Since this command returns the value that is assigned to location `a` in the memory, `μ`, it is essential for type preservation that this value be of type `τ`. This leads to the following tentative definition of the judgement `e @ μ : Σ ok`:

1. `μ : Σ`, and
2. `⊢Σ e : τ`.

...
The first condition specifies that the memory must conform to the type assumptions, $\Sigma$. The second states that $e$ must be well-typed relative to these assumptions.

But how are we to define the judgement $\mu : \Sigma$? It is tempting to simply require that if $\Sigma \vdash a : \sigma$, then there exists $v$ such that $\mu(a) = v$, $v \text{ val}$, and $v : \sigma$. That is, every active location must contain a value that is of the type specified by $\Sigma$. This almost works, except that it overlooks the possibility that $v$ may involve active references $b$ whose types are determined by $\Sigma$ itself. For example, if $\Sigma \vdash a : \text{nat ref}$, then $\mu$ must be of the form

$$\mu' \otimes \langle \langle b : \pi \rangle : \otimes \rangle \langle \langle \text{ref}[b] \rangle : \rangle$$

In this case we may consider the reference $b$ to “precede” $a$ in that the contents of $a$ refers to $b$, and $b$ refers to no other references. One might even suspect that this is representative of the general case, but this is not so—it is entirely possible to have circular dependencies among references. In particular, the contents of a cell may contain a reference to the cell itself! For example, consider a memory, $\mu$, of the form $\mu' \otimes \langle \langle a : v \rangle \rangle$, where $v$ is the $\lambda$-abstraction

$$\lambda(x : \text{nat}. \text{if} z \{ x \Rightarrow 1 \mid s(x') \Rightarrow x * (!a)(x') \}).$$

This function implements self-reference by indirecing through the cell $a$, which contains the function itself!

These considerations imply that the contents of each cell in $\mu$ must be a value of the type assigned to that cell in $\Sigma$, relative to the entire set of typing assumptions $\Sigma$. This allows for cyclic dependencies in which, for example, the contents of each location may well depend on the type of every location, including itself. This leads to the following definition of a well-formed state:

$$\frac{\Gamma \Sigma \mu : \Sigma \quad \Gamma \Sigma e : \tau}{e \otimes \mu : \Sigma \text{ ok}} (38.5)$$

where the first premise means that for every $a$, if $\Sigma \vdash a : \sigma$, then $\mu(a) = v$ such that $\Sigma \vdash v : \sigma$.

We will consider here the safety of the integral formulation, leaving the modal formulation as an exercise.

**Theorem 38.2** (Preservation). If $e \otimes \mu : \Sigma \text{ ok}$ and $e \otimes \mu : \Sigma \rightarrow e' \otimes \mu' : \Sigma'$, then $e' \otimes \mu' : \Sigma' \text{ ok}$. 

OCTOBER 16, 2009 DRAFT 18:42
Proof. Consider the transition

\[ \text{new}[\tau](e) @ \mu : \Sigma \mapsto \text{ref}[a] @ \mu \otimes \langle a : e \rangle : \Sigma, a : \tau \]

where \( a \notin \text{dom}(\Sigma) \). By inversion of typing \( \vdash_\Sigma e : \tau \). To complete the proof, note that \( \vdash_{\Sigma', \tau} \text{ref}[a] : \text{ref}(\tau) \) and \( \Sigma, a : \tau \vdash \mu \otimes \langle a : e \rangle : \Sigma, a : \tau \). \( \square \)

Theorem 38.3 (Progress). If \( e @ \mu : \Sigma \text{ ok} \) then either \( e @ \mu : \Sigma \text{ final} \) or \( e @ \mu \mapsto e' @ \mu' : \Sigma' \) for some \( \Sigma', \mu', \) and \( e' \).

Proof. For example, suppose that \( \vdash_\Sigma \text{get}(e) : \tau \), where \( \vdash_\Sigma e : \tau \). By induction and the definition of final states, either \( e \text{ val} \) or there exists \( \mu' \) and \( e' \) such that \( e @ \mu : \Sigma \mapsto e' @ \mu' : \Sigma' \). In the latter case we have

\[ \text{get}(e) @ \mu : \Sigma \mapsto \text{get}(e') @ \mu' : \Sigma'. \]

In the former it follows that \( e = \text{ref}[a] \) for some \( a \) such that \( \Sigma = \Sigma', a : \tau \). Since \( \vdash_\Sigma \mu : \Sigma \), it follows that \( \mu = \mu' \otimes \langle a : e' \rangle \) for some \( \mu' \) and \( e' \) such that \( \vdash_\Sigma e' : \tau \). But then we have \( \text{get}(e) @ \mu : \Sigma \mapsto e' @ \mu : \Sigma \). \( \square \)

### 38.4 Integral versus Modal Formulation

The modal and integral formulations of references have complementary strengths and weaknesses. The chief virtue of the modal formulation is that the use of state is confined to commands, leaving the semantics of expressions alone. One consequence is that typing judgements for expressions retain their force even in the presence of references, so that the type \( \text{unit} \to \text{unit} \) remains “boring”, and the type \( \text{nat} \to \text{nat} \) consists solely of partial functions on the natural numbers. By contrast the integral formulation enjoys none of these properties. Any expression may have an effect on memory, and the semantics of typing assertions is therefore significantly altered. In particular, the type \( \text{unit} \to \text{unit} \) is “interesting”, and the type \( \text{nat} \to \text{nat} \) contains procedures that in no way represent partial functions such as the procedure that, when called for the \( i \)th time, adds \( i \) to its argument.

While the modal separation of pure from impure expressions may seem like an unalloyed benefit, it is important to recognize that the situation is not nearly so simple. The modal approach impedes the use of mutable storage to implement purely functional behavior. For example, a self-adjusting
Integral versus Modal Formulation

A 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. The use of mutation is an example of a benign effect, a use of mutation that is not semantically visible to the client of an abstraction, but allows for more efficient execution time. In the modal formulation any use of a storage effect confines the programmer to the command sub-language, with no possibility of escape. That is, there is no way to restore the purity of an impure computation.

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 mutable cells for collecting profiling information and to insert code to update these cells during execution. 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 mutable cell.

In the integral formulation we may implement the factorial function using backpatching as follows:

\[
\begin{align*}
\text{let } r & \text{ be new}(\lambda n:nat. n) \text{ in} \\
\text{let } f & \text{ be } \lambda n:nat. \text{ifz}(n, 1, n'.n * (\text{get } r)(n')) \text{ in} \\
\text{let } & \text{ be set } r := f \text{ in } f
\end{align*}
\]

This expression returns a function of type \(nat \rightarrow nat\) that is obtained by (a) allocating a reference cell initialized arbitrarily with a function of this type, (b) defining a \(\lambda\)-abstraction in which each “recursive call” consists of retrieving and applying the function stored in that cell, (c) assigning this function to the cell, and (d) returning that function. The result is a value of function type that uses a reference cell “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.

\[
\begin{align*}
\text{do } & \{ \\
& r \leftarrow \text{return (new } (\lambda n:nat. \text{comp(return } (n)))) \\
& ; f \leftarrow \text{return } (\lambda n:nat. \ldots) \\
& ; \_ \leftarrow \text{set } r := f \\
& ; \text{return } f
\}
\end{align*}
\]

where the elided \(\lambda\)-abstraction is given as follows:
\[ \lambda(n:\text{nat}. \\
\quad \text{if} \, z(n, \\
\quad \quad \text{comp} \,(\text{return}(\text{1})), \\
\quad \quad n'.\text{comp}(
\quad \quad \text{do} \\
\quad \quad \quad \quad f' \leftarrow \text{get} \, r \\
\quad \quad \quad \quad ; \, \text{return} \, (n*f'(n')) \\
\quad \quad \quad \text{)}} 
\]  

Each branch of the conditional test returns a suspended command. In the case that the argument is zero, the command simply returns the value 1. Otherwise, it fetches the contents of the associated reference cell, applies this to the predecessor, and returns the result of the appropriate calculation.

The modal implementation of factorial is a command (not an expression) of type \( \text{nat} \rightarrow (\text{nat cmd}) \), 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 reference cell used to implement backpatching.

2. The body of the factorial function is impure, because it accesses the reference cell 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

\[
\text{do} \{ \\
\quad f \leftarrow \text{fact} \\
\quad ; \, \text{x} \leftarrow \text{let} \, \text{comp} \,(x:\text{nat}) \, \text{be} \, f(n) \, \text{in} \, \text{return} \, x \\
\quad ; \, \text{return} \, x \\
\}.
\]

Here \textit{fact} stands for the command implementing factorial given above. This is bound to a variable, \( f \), which is then applied to yield an encapsulated command that, when activated, computes the desired result. This result is returned to the caller, which must itself be a command, and not an expression, propagating the reliance on effects from the callee to the caller.

### 38.5 Exercises
Part XIV

Laziness
Chapter 39

Eagerness and Laziness

A fundamental distinction between eager, or strict, and lazy, or non-strict, evaluation arises in the dynamic semantics of function, product, sum, and recursive types. This distinction is of particular importance in the context of $\mathcal{L}\{\mu\}$, which permits the formation of divergent expressions. So far in this text (and in practice) the choice between eager and lazy evaluation is regarded as a matter of language design, but we will argue in this chapter that it is better viewed as a type distinction.

39.1 Eager and Lazy Dynamics

According to the methodology outlined in Chapter 11, language features are identified with types. The constructs of the language arise as the introductory and eliminatory forms associated with a type. The static semantics specifies how these may be combined with each other and with other language constructs in a well-formed program. The dynamic semantics specifies how these constructs are to be executed, subject to the requirement of type safety. Safety is assured by the conservation principle, which states that the introduction forms are the values of the type, and the elimination forms are inverse to the introduction forms.

Within these broad guidelines there is often considerable leeway in the choice of dynamic semantics for a language construct. For example, consider the dynamic semantics of function types given in Chapter 13. There we specified that $\lambda$-abstractions are values, and that applications are eval-
uated according to the following rules:

\[
\frac{e_1 \rightarrow e'_1}{e_1(e_2) \rightarrow e'_1(e_2)} \tag{39.1a}
\]

\[
\frac{e_1 \text{ val} \quad e_2 \rightarrow e'_2}{e_1(e_2) \rightarrow e_1(e'_2)} \tag{39.1b}
\]

\[
\frac{e_2 \text{ val}}{\lambda(x: \tau. e)(e_2) \rightarrow [e_2/x]e} \tag{39.1c}
\]

The first of these states that to evaluate an application \(e_1(e_2)\) we must first of all evaluate \(e_1\) to determine what function is being applied. The third of these states that application is inverse to abstraction, but is subject to the requirement that the argument be a value. For this to be tenable, we must also include the second rule, which states that to apply a function, we must first evaluate its argument. This is called the call-by-value, or strict, or eager, evaluation order for functions.

Regarding a \(\lambda\)-abstraction as a value is inevitable so long as we retain the principle that only closed expressions (complete programs) can be executed. Similarly, it is natural to demand that the function part of an application be evaluated before the function can be called. On the other hand it is somewhat arbitrary to insist that the argument be evaluated before the call, since nothing seems to oblige us to do so. This suggests an alternative evaluation order, called call-by-name,\(^1\) or lazy, which states that arguments are to be passed unevaluated to functions. Consequently, function parameters stand for computations, not values, since the argument is passed in unevaluated form. The following rules define the call-by-name evaluation order:

\[
\frac{e_1 \rightarrow e'_1}{e_1(e_2) \rightarrow e'_1(e_2)} \tag{39.2a}
\]

\[
\frac{\lambda(x: \tau. e)(e_2) \rightarrow [e_2/x]e}{e_2 \text{ val}} \tag{39.2b}
\]

We omit the requirement that the argument to an application be a value.

This example illustrates some general principles governing the dynamic semantics of a language.

1. The conservation principle states that a type is defined by its introductory forms, and that the eliminatory forms invert the introductory forms. This has several implications:

\(^1\)For obscure historical reasons.
(a) The instruction steps of the dynamic semantics state that the eliminatory forms are post-inverse to the introductory forms.

(b) The principal argument of an elimination form must be evaluated to determine which introductory form is provided in that position before execution of an instruction step is possible.

(c) The values of a type consist only of closed terms of outermost introductory form.

2. Some evaluation order decisions are left undetermined by this principle.

(a) Whether or not to evaluate the non-principal arguments of an eliminatory form.

(b) Whether or not to evaluate the subexpressions of a value.

Let us apply these principles to the product type. First, the sole argument to the elimination forms is, of course, principal, and hence must be evaluated. Second, if the argument is a value, it must be a pair (the only introductory form), and the projections extract the appropriate component of the pair.

\[
\begin{align*}
\langle e_1, e_2 \rangle & \text{ val} \\
\langle e_1, e_2 \rangle \cdot 1 & \mapsto e_1
\end{align*}
\]

\[
\begin{align*}
\langle e_1, e_2 \rangle & \text{ val} \\
\langle e_1, e_2 \rangle \cdot x & \mapsto e_1
\end{align*}
\]

\[
\begin{align*}
e & \mapsto e' \\
e \cdot 1 & \mapsto e' \cdot 1
\end{align*}
\]

\[
\begin{align*}
e & \mapsto e' \\
e \cdot x & \mapsto e' \cdot x
\end{align*}
\]

Since there is only one introductory form for the product type, a value of product type must be a pair. But this leaves open whether the components of a pair value must themselves be values or not. The eager (or strict) semantics evaluates the components of a pair before deeming it to be a value: specified by the following additional rules:

\[
\begin{align*}
e_1 & \text{ val} \\
e_2 & \text{ val} \\
\langle e_1, e_2 \rangle & \text{ val} \\
\end{align*}
\]

\[
\begin{align*}
e_1 & \mapsto e'_1 \\
\langle e_1, e_2 \rangle & \mapsto \langle e'_1, e_2 \rangle
\end{align*}
\]
\[
\frac{e_1 \text{ val } \quad e_2 \mapsto e'_2}{\langle e_1, e_2 \rangle \mapsto \langle e_1, e'_2 \rangle}
\] (39.9)

The lazy (or non-strict) semantics, on the other hand, deems any pair to be a value, regardless of whether its components are values:

\[
\frac{}{\langle e_1, e_2 \rangle \text{ val}}
\] (39.10)

There are similar alternatives for sum and recursive types, differing according to whether or not the argument of an injection, or to the introductory half of an isomorphism, is evaluated. There is no choice, however, regarding evaluation of the branches of a case analysis, since each branch binds a variable to the injected value for each case. Incidentally, this explains the apparent restriction on the evaluation of the conditional expression, \(\text{if } e \text{ then } e_1 \text{ else } e_2\), arising from the definition of \(\text{bool}\) to be the sum type \(\text{unit} + \text{unit}\) as described in Chapter 17 — the “then” and the “else” branches lie within the scope of an (implicit) bound variable, and hence are not eligible for evaluation!

### 39.2 Eager and Lazy Types

Rather than specify a blanket policy for the eagerness or laziness of the various language constructs, it is more expressive to put this decision into the hands of the programmer by a type distinction. That is, we can distinguish types of by-value and by-name functions, and of eager and lazy versions of products, sums, and recursive types.

We may give eager and lazy variants of product, sum, function, and recursive types according to the following chart:

<table>
<thead>
<tr>
<th>Type</th>
<th>Eager</th>
<th>Lazy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit</td>
<td>(1)</td>
<td>(\top)</td>
</tr>
<tr>
<td>Product</td>
<td>(\tau_1 \otimes \tau_2)</td>
<td>(\tau_1 \times \tau_2)</td>
</tr>
<tr>
<td>Void</td>
<td>(\bot)</td>
<td>(0)</td>
</tr>
<tr>
<td>Sum</td>
<td>(\tau_1 + \tau_2)</td>
<td>(\tau_1 \oplus \tau_2)</td>
</tr>
<tr>
<td>Function</td>
<td>(\tau_1 \circ \tau_2)</td>
<td>(\tau_1 \rightarrow \tau_2)</td>
</tr>
</tbody>
</table>

We leave it to the reader to formulate the static and dynamic semantics of these constructs using the following grammar of introduction and elimination.
39.3 Self-Reference

We have seen in Chapter 15 that we may use general recursion at the expression level to define recursive functions. In the presence of laziness we may also define other forms of self-referential expression. For example, consider the so-called lazy natural numbers, which are defined by the recursive type

\[ \text{lnat} = \mu t. \top \oplus t. \]

The successor operation for the lazy natural numbers is defined by the equation \( \text{lsucc}(e) = \text{fold}(\text{rht}(e)) \). Using general recursion we may form the lazy natural number

\[ \omega = \text{fix} x:\text{lnat is lsucc}(x), \]

which consists of an infinite stack of successors!

Of course, one could argue (correctly) that \( \omega \) is not a natural number at all, and hence should not be regarded as one. So long as we can distinguish the type \( \text{lnat} \) from the type \( \text{nat} \), there is no difficulty—\( \omega \) is the infinite lazy natural number, but it is not an eager natural number. But if the distinction is not available, then serious difficulties arise. For example, lazy languages provide only lazy product and sum types, and hence are only capable of defining the lazy natural numbers as a recursive types. In such languages \( \omega \) is said to be a “natural number”, but only for a non-standard use of the term; the true natural numbers are simply unavailable.

It is a significant weakness of lazy languages is that they provide only a paucity of types. One might expect that, dually, eager languages are similarly disadvantaged in providing only eager, but not lazy types. However,
in the presence of function types (the common case), we may encode the lazy types as instances of the corresponding eager types, as we describe in the next section.

39.4 Suspension Type

The essence of lazy evaluation is the suspension of evaluation of certain expressions. For example, the lazy product type suspends evaluation of the components of a pair until they are needed, and the lazy sum type suspends evaluation of the injected value until it is required. To encode lazy types as eager types, then, requires only that we have a type whose values are unevaluated computations of a specified type. Such unevaluated computations are called suspensions, or thunks.\(^2\) Moreover, since general recursion requires laziness in order to be useful, it makes sense to confine general recursion to suspension types. To model this we consider self-referential unevaluated computations as values of suspension type.

The abstract syntax of suspensions is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>(\tau)</td>
<td>(\text{susp}(\tau))</td>
<td>(\tau) susp</td>
</tr>
<tr>
<td>Expr</td>
<td>(e)</td>
<td>(\text{susp}<a href="x.e">\tau</a>)</td>
<td>susp (x: \tau) is (e)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{force}(e))</td>
<td>force((e))</td>
</tr>
</tbody>
</table>

The introduction form binds a variable that stands for the suspension itself. The elimination form evaluates \(e_1\) to a suspension, then evaluates that suspension, binding its value to \(x\) for use within \(e_2\). As a notational convenience, we sometimes write \(\text{susp}(e)\) for \(\text{susp}[\tau](x.e)\), where \(x\) is chosen so as not to occur free in \(e\).

The static semantics of suspensions is given by the following typing rules:

\[
\Gamma, x : \text{susp}(\tau) \vdash e : \tau \\
\Gamma \vdash \text{susp}[\tau](x.e) : \text{susp}(\tau)
\]

\[
\Gamma \vdash e : \text{susp}(\tau) \\
\Gamma \vdash \text{force}(e) : \tau
\]

In Rule (39.11a) the variable \(x\), which refers to the suspension itself, is assumed to have type \(\text{susp}(\tau)\) while checking that the suspended computation, \(e\), has type \(\tau\).

\(^2\)The etymology of this term is uncertain, but its usage persists.
The dynamic semantics of suspensions is given by the following rules:

\[
\begin{align*}
\text{susp}[\tau](x.e) & \rightarrow \text{val} & (39.12a) \\
e \rightarrow e' & \quad \text{force}(e) \rightarrow \text{force}(e') & (39.12b) \\
\text{force(susp}[\tau](x.e)) & \rightarrow [\text{susp}[\tau](x.e)/x]e & (39.12c)
\end{align*}
\]

Rule (39.12c) implements recursive self-reference by replacing \(x\) by the suspension itself before substituting it into the body of the \text{let}.

It is straightforward to formulate and prove type safety for self-referential suspensions. We leave the proof as an exercise for the reader.

**Theorem 39.1** (Safety). If \(e : \tau\), then either \(e \rightarrow \text{val}\) or there exists \(e' : \tau\) such that \(e \rightarrow e'\).

We may use suspensions to encode the lazy type constructors as instances of the corresponding eager type constructors as follows:

\[
\begin{align*}
\top & = 1 & (39.13a) \\
\langle \rangle & = \bullet & (39.13b)
\end{align*}
\]

\[
\begin{align*}
\tau_1 \times \tau_2 & = \tau_1 \text{susp} \otimes \tau_2 \text{susp} & (39.14a) \\
\langle e_1, e_2 \rangle & = \text{susp}(e_1) \otimes \text{susp}(e_2) & (39.14b) \\
e \cdot l & = \text{let } x \otimes \_ \text{ be } e \text{ in force}(x) & (39.14c) \\
e \cdot r & = \text{let } \_ \otimes y \text{ be } e \text{ in force}(y) & (39.14d)
\end{align*}
\]

\[
\begin{align*}
0 & = \bot & (39.15a) \\
\text{abort}_\tau(e) & = \text{abort}_\tau e & (39.15b)
\end{align*}
\]

\[
\begin{align*}
\tau_1 \oplus \tau_2 & = \tau_1 \text{susp} + \tau_2 \text{susp} & (39.16a) \\
\text{lft}(e) & = l \cdot \text{susp}(e) & (39.16b) \\
\text{rht}(e) & = r \cdot \text{susp}(e) & (39.16c)
\end{align*}
\]
choose \( e \{ \text{lf}(x_1) \Rightarrow e_1 \mid \text{rht}(x_2) \Rightarrow e_2 \} \)

\[
= \text{case } e \{ 1 \cdot y_1 \Rightarrow [\text{force}(y_1)/x_1]e_1 \mid x \cdot y_2 \Rightarrow [\text{force}(y_2)/x_2]e_2 \}
\]

\[\text{(39.16d)}\]

\[
\tau_1 \rightarrow \tau_2 = \tau_1 \text{susp} \circ \rightarrow \tau_2 \quad \text{(39.17a)}
\]

\[
\lambda(x:\tau_1.e_2) = \lambda^0(x:\tau_1 \text{susp.}[\text{force}(x)/x]e_2) \quad \text{(39.17b)}
\]

\[
e_1(e_2) = \text{ap}^0(e_1; \text{susp}(e_2)) \quad \text{(39.17c)}
\]

In the case of lazy case analysis and call-by-name functions we replace occurrences of the bound variable, \( x \), with \( \text{force}(x) \) to recover the value of the suspension bound to \( x \) whenever it is required. Note that \( x \) may occur in a lazy context, in which case \( \text{force}(x) \) is delayed. In particular, expressions of the form \( \text{susp}(\text{force}(x)) \) may be safely replaced by \( x \), since forcing the former computation simply forces \( x \).

### 39.5 Exercises
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 semantics described in Chapter 39 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.

A third aspect of laziness is the ability to form recursive values, including as a special case recursive functions. Using general recursion we can create self-referential expressions, but these are only useful if the self-referential expression can be evaluated without needing its own values. Function abstractions provide one such mechanism, but so do lazy data constructors.

These aspects of laziness are often consolidated into a programming language with call-by-need function evaluation, lazy data structures, and unrestricted uses of recursion. Such languages are called lazy languages, because they impose the lazy evaluation strategy throughout. These are to be contrasted with strict languages, which impose an eager evaluation strategy throughout. This leads to a sense of opposition between two incompatible
points of view, but, as we discussed in Chapter 39, experience has shown that this apparent conflict is neither necessary nor desirable. Rather than accept these as consequences of language design, it is preferable to put the distinction in the hands of the programmer by introducing a type of suspended computations whose evaluation is memoized so that they are only ever evaluated once. The ambient evaluation strategy remains eager, but we now have a value representing an unevaluated expression. Moreover, we may confine self-reference to suspensions to avoid the pathologies of laziness while permitting self-referential data structures to be programmed.

40.1 Need Dynamics

The distinguishing feature of call-by-need, as compared to call-by-name, is that it ensures that the binding of a variable is evaluated at most once, when it is needed, and never again. This is achieved by mutation of a data structure recording the bindings of all active variables. When a variable is first used, its binding is evaluated and replaced by the value so determined so that subsequent accesses return that value immediately.

The call-by-need dynamic semantics of $L\{\mathsf{nat} \rightarrow \}$ is given by a transition system whose states have the form $e @ \mu$, where $\mu$ is a finite function mapping variables to expressions (not necessarily values!), and $e$ is an expression whose free variables lie within the domain of $\mu$. (We use the same notation for finite functions as in Chapter 38.)

The rules defining the call-by-need dynamic semantics of $L\{\mathsf{nat} \rightarrow \}$ are as follows:

\[
\begin{align*}
\overline{z} &\vdash \text{val} & (40.1a) \\
\overline{s(x)} &\vdash \text{val} & (40.1b) \\
\overline{\lambda x.e} &\vdash \text{val} & (40.1c) \\
\overline{e@\overline{\emptyset}} &\vdash \text{initial} & (40.1d) \\
\overline{e} &\vdash \text{val} & (40.1e) \\
\overline{e} &\vdash \mu \vdash \text{final} & (40.1f)
\end{align*}
\]
40.1 Need Dynamics

\[ e \otimes \mu \otimes \langle x: \bullet \rangle \rightarrow e' \otimes \mu' \otimes \langle x: \bullet \rangle \]  
\[ x \otimes \mu \otimes \langle x: e \rangle \rightarrow x \otimes \mu' \otimes \langle x: e' \rangle \]  

\[ s(e) \otimes \mu \rightarrow s(x) \otimes \mu \otimes \langle x: e \rangle \]  

\[ e \otimes \mu \rightarrow e' \otimes \mu' \]  
\[ \text{ifz}(e; e_0; x. e_1) \otimes \mu \rightarrow \text{ifz}(e'; e_0; x. e_1) \otimes \mu' \]  

\[ \text{ifz}(x; e_0; x. e_1) \otimes \mu \rightarrow e_0 \otimes \mu \]  

\[ \text{ifz}(s(x); e_0; x. e_1) \otimes \mu \rightarrow e_1 \otimes \mu \]  

\[ e_1 \otimes \mu \rightarrow e'_1 \otimes \mu' \]  
\[ e_1(e_2) \otimes \mu \rightarrow e'_1(e_2) \otimes \mu' \]  

\[ x \not\in \text{dom}(\mu) \]  
\[ \lambda(x: \tau. e)(e_2) \otimes \mu \rightarrow e \otimes \mu \otimes \langle x: e_2 \rangle \]  

\[ x \not\in \text{dom}(\mu) \]  
\[ \text{fix}[\tau](x. e) \otimes \mu \rightarrow x \otimes \mu \otimes \langle x: e \rangle \]  

Rules (40.1a) through (40.1c) specify that \( z \) is a value, any expression of the form \( s(x) \), where \( x \) is a variable, is a value, and any \( \lambda \)-abstraction, possibly containing free variables, is a value. Importantly, variables themselves are not values, since they may be bound by the memory to an unevaluated expression.

Rule (40.1d) specifies that an initial state consists of a binding for a closed expression, \( e \), in memory, together with a demand for its binding. Rule (40.1e) specifies that a final state has the form \( e \otimes \mu \), where \( e \) is a value.

Rule (40.1h) specifies that evaluation of \( s(e) \) yields the value \( s(x) \), where \( x \) is bound in the memory to \( e \) in unevaluated form. This reflects a lazy semantics for the successor, in which the predecessor is not evaluated until it is required by a conditional branch. Rule (40.1k), which governs a conditional branch on a successor, makes use of \( \alpha \)-equivalence to choose the bound variable, \( x \), for the predecessor to be the variable to which the predecessor was already bound by the successor operation. Evaluation of the successor branch of the conditional may make a demand on \( x \), which would then cause the predecessor to be evaluated, as discussed above.
Rule (40.1l) specifies that the value of the function position of an application must be determined before the application can be executed. Rule (40.1m) specifies that to evaluate an application of a $\lambda$-abstraction we create a fresh binding of its parameter to its unevaluated argument, and continue by evaluating its body. The freshness condition may always be met by implicitly renaming the bound variable of the $\lambda$-abstraction to be a variable not otherwise bound in the memory. Thus, each call results in a fresh binding of the parameter to the argument at the call.

The rules for variables are crucial, since they implement memoization. Rule (40.1f) governs a variable whose binding is a value, which is returned as the value of that variable. Rule (40.1g) specifies that if the binding of a variable is required and that binding is not yet a value, then its value must be determined before further progress can be made. This is achieved by switching the “focus” of evaluation to the binding, while at the same time replacing the binding by a black hole, which represents the absence of a value for that variable (since it has not yet been determined). Evaluation of a variable whose binding is a black hole is “stuck”, since it indicates a circular dependency of the value of a variable on the variable itself.

Rule (40.1n) implements general recursion. Recall from Chapter 15 that the expression $\text{fix}[\tau](x.e)$ stands for the solution of the recursion equation $x = e$, where $x$ may occur within $e$. Rule (40.1n) obtains the solution directly by equating $x$ to $e$ in the memory, and returning $x$. The role of the black hole becomes evident when evaluating an expression such as $\text{fix}[x: \tau]_\tau x$. Evaluation of this expression binds the variable $x$ to itself in the memory, and then returns $x$, creating a demand for its binding. Applying Rule (40.1g), we see that this immediately leads to a stuck state in which we require the value of $x$ in a memory in which it is bound to the black hole. This captures the inherent circularity in the purported definition of $x$, and amounts to catching a potential infinite loop before it happens. Observe that, by contrast, an expression such as $\text{fix }f:\sigma \to \tau \text{ is } \lambda(x: \sigma. e)$ does not get stuck, because the occurrence of the recursively defined variable, $f$, lies within the $\lambda$-expression. Evaluation of a $\lambda$-abstraction, being a value, creates no demand for $f$, so the black hole is not encountered. Rule (40.1g) backpatches the binding of $f$ to be the $\lambda$-abstraction itself, so that subsequent uses of $f$ evaluate to it, as would be expected. Thus recursion is automatically implemented by the backpatching technique described in Chapter 38.
40.2 Safety

The type safety of the by-need semantics for lazy $L\{\text{nat} \rightarrow \}$ is proved using methods similar to those developed in Chapter 38 for references. To do so we define the judgement $e \otimes \mu \text{ ok}$ to hold iff there exists a set of typing assumptions $\Gamma$ governing the variables in the domain of the memory, $\mu$, such that

1. if $\Gamma = \Gamma', x : \tau_x$ and $\mu(x) = e \neq \bullet$, then $\Gamma \vdash e : \tau_x$.

2. there exists a type $\tau$ such that $\Gamma \vdash e : \tau$.

As a notational convenience, we will sometimes write $\mu : \Gamma \vdash e : \tau$ for the conjunction of these two conditions.

**Theorem 40.1** (Preservation). If $e \otimes \mu \mapsto e' \otimes \mu'$ and $e \otimes \mu \text{ ok}$, then $e' \otimes \mu' \text{ ok}$.

**Proof.** The proof is by rule induction on Rules (40.1). For the induction we prove the stronger result that if $\mu : \Gamma$ and $\Gamma \vdash e : \tau$, then there exists $\Gamma'$ such that $\mu' : \Gamma \Gamma' \vdash e' : \tau$. We will consider two illustrative cases of the proof.

Consider Rule (40.11l), for which $e = e_1(e_2)$. Suppose that $\mu : \Gamma$ and $\Gamma \vdash e : \tau$. Then by inversion of typing $\Gamma \vdash e_1 : \tau_2 \rightarrow \tau$ for some type $\tau_2$ such that $\Gamma \vdash e_2 : \tau_2$. So by induction there exists $\Gamma'$ such that $\mu' : \Gamma \Gamma' \vdash e'_1 : \tau_2 \rightarrow \tau$. By weakening $\Gamma \Gamma' \vdash e_2 : \tau_2$, and hence $\mu' : \Gamma \Gamma' \vdash e'_1(e_2) : \tau$. We have only to notice that $e' = e'_1(e_2)$ to complete this case.

Consider Rule (40.11g), for which we have $e = e' = x$, $\mu = \mu_0 \otimes \langle x : e_0 \rangle$, and $\mu' = \mu_0 \otimes \langle x : e'_0 \rangle$, where $e_0 \otimes \mu_0 \otimes \langle x : \bullet \rangle \mapsto e'_0 \otimes \mu'_0 \otimes \langle x : \bullet \rangle$. Assume that $\mu : \Gamma \vdash e : \tau$; we are to show that there exists $\Gamma'$ such that $\mu' : \Gamma \Gamma' \vdash e'_0 : \tau$. Since $\mu : \Gamma$ and $e$ is the variable $x$, we have that $\Gamma = \Gamma''$, $x : \tau$ and $\Gamma \vdash e_0 : \tau$. Therefore $\mu_0 \otimes \langle x : \bullet \rangle : \Gamma$, so by induction there exists $\Gamma'$ such that $\mu'_0 \otimes \langle x : \bullet \rangle : \Gamma \Gamma' \vdash e'_0 : \tau$. But then $\mu'_0 \otimes \langle x : e'_0 \rangle : \Gamma \Gamma' \vdash x : \tau$, as required.

The progress theorem must be stated so as to account for accessing a variable that is bound to a black hole, which is tantamount to a detectable form of looping. Since the type system does not rule this out, we define the judgement $e \otimes \mu \text{ loops}$ by the following rules:

$$
\text{x \otimes \mu \otimes \langle x : \bullet \rangle \text{ loops \hspace{1cm} (40.2a)}}
$$

$$
e \otimes \mu \otimes \langle x : \bullet \rangle \text{ loops}
$$

$$
x \otimes \mu \otimes \langle x : e \rangle \text{ loops \hspace{1cm} (40.2b)}
$$
\[
\frac{e @ \mu \text{ loops}}{\text{ifz}(e; e_0; x. e_1) @ \mu \text{ loops}} \quad (40.2c)
\]
\[
\frac{e_1 @ \mu \text{ loops}}{\text{ap}(e_1; e_2) @ \mu \text{ loops}} \quad (40.2d)
\]

In general looping is propagated through the principal argument of every eliminatory construct, since this argument position must always be evaluated in any transition sequence involving it.

The progress theorem is weakened to account for detectable looping.

**Theorem 40.2** (Progress). If \( e @ \mu \text{ ok} \), then either \( e @ \mu \text{ final} \), or \( e @ \mu \text{ loops} \), or there exists \( \mu' \) and \( e' \) such that \( e @ \mu \rightarrow e' @ \mu' \).

**Proof.** We prove by rule induction on the static semantics that if \( \mu : \Gamma \vdash e : \tau \), then either \( e \text{ val} \), or \( e @ \mu \text{ loops} \), or \( e @ \mu \rightarrow e' @ \mu' \) for some \( \mu' \) and \( e' \). The proof is by lexicographic induction on the measure \((m,n)\), where \( n \geq 0 \) is the size of \( e \) and \( m \geq 0 \) is the sum of the sizes of the non-black-hole bindings of each variable in the domain of \( \mu \). This means that we may appeal to the inductive hypothesis for sub-expressions of \( e \), since they have smaller size, provided that the size of the memory remains fixed. Since the size of \( \mu \otimes \langle x : \bullet \rangle \) is strictly smaller than the size of \( \mu \otimes \langle x : e_x \rangle \) for any expression \( e_x \), we may also appeal to the inductive hypothesis for expressions larger than \( e \), provided we do so relative to a smaller memory.

As an example of the former case, consider the case of Rule (15.1f), for which \( e = \text{ap}(e_1; e_2) \), where \( \mu : \Gamma \vdash e_1 : \text{arr}(\tau_2; \tau) \) and \( \mu : \Gamma \vdash e_2 : \tau_2 \). By the induction hypothesis applied to \( e_1 \), we have that either \( e_1 \text{ val} \) or \( e_1 @ \mu \text{ loops} \) or \( e_1 @ \mu \rightarrow e_1' @ \mu' \).

In the first case it may be shown that \( e_1 = 1\text{am}[\tau_2](x.e) \), and hence that \( \text{ap}(e_1; e_2) @ \mu \rightarrow e @ \mu' \otimes \langle x : e_2 \rangle \) by Rule (40.1m), where \( x \) is chosen by \( \alpha \)-equivalence to lie outside of the domain of \( \mu' \). In the second case we have by Rule (40.2d) that \( \text{ap}(e_1; e_2) @ \mu \text{ loops} \). In the third case we have by Rule (40.1l) that \( \text{ap}(e_1; e_2) @ \mu \rightarrow \text{ap}(e_1'; e_2) @ \mu' \).

Now consider Rule (15.1a), for which we have \( \Gamma \vdash x : \tau \) with \( \Gamma = \Gamma', x : \tau \). For any \( \mu \) such that \( \mu : \Gamma \), we have that \( \mu = \mu_0 \otimes \langle x : e_0 \rangle \) with \( \mu_0 \otimes \langle x : \bullet \rangle : \Gamma \vdash e_0 : \tau \). Since the memory \( \mu_0 \otimes \langle x : \bullet \rangle \) is smaller than the memory \( \mu \), we have by induction that either \( e_0 \text{ val} \) or \( e_0 @ \mu_0 \otimes \langle x : \bullet \rangle \text{ loops} \), or \( e_0 @ \mu_0 \otimes \langle x : \bullet \rangle \rightarrow e_0' @ \mu_0' \otimes \langle x : \bullet \rangle \).

If \( e_0 \text{ val} \), then \( x @ \mu_0 \otimes \langle x : e_0 \rangle \rightarrow e_0 @ \mu_0 \otimes \langle x : e_0 \rangle \) by Rule (40.1f). If \( e_0 @ \mu_0 \otimes \langle x : \bullet \rangle \text{ loops} \), then \( x @ \mu_0 \otimes \langle x : e_0 \rangle \) loops by Rule (40.2b). Finally, if
A pair is considered a value only if its arguments are variables (Rule (40.3a)), which are introduced when the pair is created (Rule (40.3b)). The first and second projections evaluate to one or the other variable in the pair, inducing a demand for the value of that component. This ensures that another occurrence of the same projection of the same pair will yield the same value without having to recompute it.

We may similarly devise a need semantics for sum types and recursive types, following a very similar pattern. The semantics for the type nat given in Section 40.1 on page 362 is an example of the need semantics for
a particular recursive sum type. This example may readily be extended to cover the general case. In particular the need dynamic of sum type is given by the following rules:

\[
\begin{align*}
\text{in[l][τ]}(x) & \text{ val} \quad (40.4a) \\
\text{in[r][τ]}(x) & \text{ val} \quad (40.4b) \\
\text{in[l][τ]}(e) \otimes \mu & \mapsto \text{in[l][τ]}(x) \otimes \mu \otimes \langle x : e \rangle \quad (40.4c) \\
\text{in[r][τ]}(e) \otimes \mu & \mapsto \text{in[r][τ]}(x) \otimes \mu \otimes \langle x : e \rangle \quad (40.4d) \\
\text{e} \otimes \mu & \mapsto \text{e}' \otimes \mu' \quad (40.4e) \\
\text{case(e; x₁.e₁; x₂.e₂)} \otimes \mu & \mapsto \text{case(e; x₁.e₁; x₂.e₂)} \otimes \mu' \quad (40.4f) \\
\text{e} \otimes \mu \text{ loops} & \quad (40.4g) \\
\text{case(in[l][τ]}(x₁); x₁.e₁; x₂.e₂) \otimes \mu & \mapsto e₁ \otimes \mu \\
\text{case(in[r][τ]}(x₂); x₁.e₁; x₂.e₂) \otimes \mu & \mapsto e₂ \otimes \mu \\
\end{align*}
\]

The need dynamics of recursive types follows a very similar pattern:

\[
\begin{align*}
\text{fold[t.τ]}(x) & \text{ val} \quad (40.5a) \\
\text{fold[t.τ]}(e) \otimes \mu & \mapsto \text{fold[t.τ]}(x) \otimes \mu \otimes \langle x : e \rangle \quad (40.5b) \\
\text{e} \otimes \mu & \mapsto \text{e}' \otimes \mu' \quad (40.5c) \\
\text{unfold(e)} \otimes \mu & \mapsto \text{unfold(e')} \otimes \mu' \quad (40.5d) \\
\text{e} \otimes \mu \text{ loops} & \quad (40.5e) \\
\text{unfold(fold[t.τ]}(x)) \otimes \mu & \mapsto x \otimes \mu
\end{align*}
\]
40.4 Suspensions By Need

The similarities among the need dynamics for products, sums, and recursive types may be consolidated by considering a need dynamics for the suspension type described in Section 39.4 on page 358.

\[
\begin{align*}
&\text{\(x\ \text{val}\)} & (40.6a) \\
&\text{\(\text{ susp}(\tau)(x.e) \otimes \mu \rightarrow x \otimes \mu \otimes \langle x:e \rangle\)} & (40.6b) \\
&\text{\(e \otimes \mu \rightarrow e' \otimes \mu'\)} & (40.6c) \\
&\text{\(\text{ force}(e) \otimes \mu \rightarrow \text{ force}(e') \otimes \mu'\)} & (40.6d) \\
&\text{\(e\ \text{val}\)} & (40.6e)
\end{align*}
\]

The main difference compared to the by-need dynamics for function, product, sum, and recursive types is that variables are now considered to be values. Instead, there is a construct for forcing evaluation that implements the by-need semantics.

The safety of the need dynamics for suspensions is proved by means very similar to that developed in Section 40.2 on page 365, with the modification that the type of a memory must account for explicit suspension types. Specifically, we define the judgement \(e \otimes \mu \ \text{ok}\) to hold iff there exists a set of typing assumptions \(\Gamma\) governing the variables in the memory, \(\mu\), such that

1. if \(\Gamma = \Gamma', x : \tau_x \text{ susp} \) and \(\mu(x) = e \neq \bullet\), then \(\Gamma \vdash e : \tau_x\).

2. there exists a type \(\tau\) such that \(\Gamma \vdash e : \tau\).

These conditions specify that whereas a variable representing a suspension has type \(\tau \text{ susp}\), the expression bound to it is to have type \(\tau\). The canonical forms lemma must be altered to state that a value of suspension type is a variable in the memory, which is enough to ensure progress.

40.5 Exercises
Part XV

Parallelism
Chapter 41

Speculation

The semantics of call-by-need given in Chapter 40 suggests opportunities for speculative evaluation. Evaluation of a delayed binding is initiated as soon as the binding is created, executing simultaneously with the evaluation of the body. Should the variable ever be needed, evaluation of the body synchronizes with the concurrent evaluation of the binding, and proceeds only once the value is available. This form of execution is called speculative because it is not certain at the outset whether the value will be needed, and hence the work expended on it may be wasted. However, if we have available more computing resources than are needed, it does little harm to evaluate expressions speculatively, and it may do some good in the case that the value is eventually needed. If computing resources are scarce, however, then speculation can hinder performance since it introduces contention that would not otherwise be present. In Chapter 42 we will explore work-efficient parallelism, which never performs more work than is strictly necessary in a computation.

This chapter is in need of substantial revision.

41.1 Speculative Evaluation

An interesting variant of the call-by-need semantics is obtained by relaxing the restriction that the bindings of variables be evaluated only once they are needed. Instead, we may permit a step of execution of the binding of any variable to occur at any time. Specifically, we replace the second variable
rule given in Section 40.1 on page 362 by the following general rule:

\[
\begin{align*}
    e @ \mu \otimes \langle y : \bullet \rangle & \mapsto e' @ \mu' \otimes \langle y : \bullet \rangle \\
    e_0 @ \mu \otimes \langle y : e' \rangle & \mapsto e_0 @ \mu' \otimes \langle y : e' \rangle
\end{align*}
\] (41.1)

This rule permits any variable binding to be chosen at any time as the focus
of attention for the next evaluation step. The first variable rule remains as-
is, so that, as before, a variable may be evaluated only after the value of its
binding has been determined.

This semantics is said to be non-deterministic because the transition re-
lation is no longer a partial function on states. That is, for a given state
\( e @ \mu \), there may be many different states \( e' @ \mu' \) such that \( e @ \mu \mapsto e' @ \mu' \),
precisely because the foregoing rule permits us to shift attention to any lo-
cation in memory at any time. The rules abstract away from the specifics of
how such “context switches” might be scheduled, permitting them to oc-
cur at any time so as to be consistent with any scheduling strategy. In this
sense non-determinism models parallel execution by permitting the indi-
vidual steps of a complete computation to be interleaved in an arbitrary
manner.

The non-deterministic semantics is said to be speculative, because it per-
mits evaluation of any suspended expression at any time, without regard
to whether its value is needed to determine the overall result of the compu-
tation. In this sense it is contrary to the spirit of call-by-need, since it may
perform work that is not strictly necessary. The benefit of speculation is
that it leads to a form of parallel computation, called speculative parallelism,
which seeks to exploit computing resources that would otherwise be left
idle. Ideally one should only use processors to compute results that are
needed, but in some situations it is difficult to make full use of available
resources without resorting to speculation.

41.2 Speculative Parallelism

The non-deterministic semantics given in Section 41.1 on the preceding
page captures the idea of speculative execution, but addresses parallelism
only indirectly, by avoiding specification of when the focus of evaluation
may shift from one suspended expression to another. The semantics is
specified from the point of view of an omniscient observer who sequen-
tializes the parallel execution into a sequence of atomic steps. No particu-
lar sequentialization is enforced; rather, all possible sequentializations are
derivable from the rules.
A more accurate model is one that makes explicit the parallel speculative evaluation of some number of suspended computations. We model this using a judgement of the form $\mu \rightarrow \mu'$, which specifies the simultaneous execution of a computation step on each of $k > 0$ suspended computations.

$$
\begin{align*}
&\left\{ e_i \otimes \mu \otimes \langle x_1 : \bullet \rangle \otimes \cdots \otimes \langle x_k : \bullet \rangle \right\} \\
&\quad \rightarrow \\
&\left\{ e'_i \otimes \mu \otimes \langle x_1 : \bullet \rangle \otimes \cdots \otimes \langle x_k : \bullet \rangle \otimes \mu_i \right\} \\
&\quad \rightarrow \\
&\left\{ \mu \otimes \langle x_1 : e'_1 \rangle \otimes \cdots \otimes \langle x_k : e'_k \rangle \otimes \mu_1 \otimes \cdots \otimes \mu_k \right\}
\end{align*}
$$

This rule may be seen as a generalization of Rule (40.1g), except that it applies independently of whether there is a demand for any of the variables involved. The transition consists of choosing $k > 0$ suspended computations on which to make progress, and simultaneously taking a step on each, and restoring the results to the memory. The choice of $k$ is left unspecified, but is fixed for all inferences; in practice it would be the number of available processors.

The speculative parallel semantics of $\mathcal{L}\{\text{nat} \rightarrow\}$ is defined by replacing Rule (40.1g) by the following rule:

$$
\frac{\mu \rightarrow \mu'}{e \otimes \mu \rightarrow e \otimes \mu'}
$$

This rules specifies that, at any moment, we may make progress by executing a step of evaluation on some number of suspended computations. Since Rule (40.1g) has been omitted, this rule must be applied sufficiently often to ensure that the binding of any required variable is fully evaluated before its value is required. The goal of speculative execution is to ensure that this is always the case, but in practice a computation must sometimes be suspended to await completion of evaluation of the binding of some variable.

There is a technical complication with Rule (41.2), however, that lies at the heart of any parallel programming language. When executing computations in parallel, it is possible that two or more of them choose the same variable to represent a new suspended computation. Formally, this occurs when the domain of $\mu_i$ intersects the domain of $\mu_j$ for some $i \neq j$ in the premise of Rule (41.2). In practice this corresponds to two threads.
attempting to allocate memory at the same time: some synchronization is required to resolve the contention. In a formal model we may leave abstract the means of achieving this, and simply demand as a side condition that the memories $\mu_1, \ldots, \mu_k$ have disjoint domains. This may always be achieved by choosing variable names independently for each thread. In an implementation some method is required to support memory allocation in parallel, using one of several synchronization methods.

41.3 Exercises
Chapter 42

Work-Efficient Parallelism

In this chapter we study the concept of work-efficient parallelism, which exploits opportunities for parallelism without increasing the workload compared to a sequential execution. This is in contrast to speculative parallelism (see Chapter 41), which exposes parallelism, but potentially at the cost of doing more work than would be done in the sequential case. In a speculative semantics we may evaluate suspended computations even though their value is never required for the ultimate result. The work expended in computing the value of the suspension is wasted; it keeps the processor warm, but could just as well have been omitted. In contrast work-efficient parallelism never wastes effort; it only performs computations whose results are required for the final outcome.

To make these ideas precise we make use of a cost semantics, which determines not only the value of an expression, but a measure of the cost of evaluating it. The costs are chosen so as to expose both opportunities for and obstructions to parallelism. If one computation depends on the result of another, then there is a sequential dependency between them that precludes their execution in parallel. If, on the other hand, two computations are independent of one another, then they can be executed in parallel. Functional languages without state provide ample opportunities for parallelism, and will be the focus of our work in this chapter.

42.1 Nested Parallelism

We begin with a very simple parallel language, \( \mathcal{L}\{\text{and}\} \), whose sole source of parallelism arises from the evaluation of two variable bindings simultaneously. This is modelled by a construct of the form \( \text{let } x_1 = e_1 \text{ and } x_2 = e_2 \text{ in } e \),
378

42.1 Nested Parallelism

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 \textit{parallel pair} construct might be defined as the expression

\[
\text{let } x_1 = e_1 \text{ and } x_2 = e_2 \text{ in } (x_1, x_2),
\]

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

\[
\text{let}(e_1; e_2; x_1.x_2.e),
\]

which makes clear that the variables \( x_1 \) and \( x_2 \) are bound \textit{only} within \( e \), and not within their bindings. This ensures that evaluation of \( e_1 \) is independent of evaluation of \( e_2 \), and \textit{vice versa}. The typing rule for an expression of this form is given as follows:

\[
\frac{
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e : \tau
}{
\Gamma \vdash \text{let}(e_1; e_2; x_1.x_2.e) : \tau}
\]  

(42.1)

Although we emphasize the case of binary parallelism, it should be clear that this construct easily generalizes to \( n \)-way parallelism for any \textit{static} value of \( n \). One may also define an \( n \)-way parallel \texttt{let} construct from the binary parallel \texttt{let} by cascading binary splits. (For a treatment of \( n \)-way parallelism for a \textit{dynamic} value of \( n \), see Section 42.3 on page 384.)

We will give both a \textit{sequential} and a \textit{parallel} dynamic semantics of the parallel \texttt{let} construct. The definition of the sequential dynamics as a transition judgement of the form \( e \rightarrow_{\text{seq}} e' \) is entirely straightforward:

\[
\frac{
e_1 \mapsto e'_1}{\text{let}(e_1; e_2; x_1.x_2.e) \rightarrow_{\text{seq}} \text{let}(e'_1; e_2; x_1.x_2.e)}
\]

(42.2a)

\[
\frac{
e_1 \text{ val } e_2 \mapsto e'_2}{\text{let}(e_1; e_2; x_1.x_2.e) \rightarrow_{\text{seq}} \text{let}(e_1; e'_2; x_1.x_2.e)}
\]

(42.2b)

\[
\frac{
e_1 \text{ val } e_2 \text{ val}}{
\text{let}(e_1; e_2; x_1.x_2.e) \rightarrow_{\text{seq}} \left[e_1, e_2, x_1, x_2\right]e'}
\]

(42.2c)
42.1 Nested Parallelism

The parallel dynamics is given by a transition judgement of the form $e \mapsto_{\text{par}} e'$, defined as follows:

\[
\begin{align*}
\frac{e \mapsto_{\text{par}} e_1'}{\text{let}(e; e_1'; e_2) \mapsto_{\text{par}} \text{let}(e_1'; e_2; x_1 \cdot x_2 \cdot e)} \tag{42.3a} \\
\frac{e \mapsto_{\text{par}} e_2'}{\text{let}(e_1; e_2; x_1 \cdot x_2 \cdot e) \mapsto_{\text{par}} \text{let}(e_1; e_2'; x_1 \cdot x_2 \cdot e)} \tag{42.3b} \\
\frac{e \mapsto_{\text{par}} e_2'}{\text{let}(e_1; e_2; x_1 \cdot x_2 \cdot e) \mapsto_{\text{par}} \text{let}(e_1; e_2'; x_1 \cdot x_2 \cdot e)} \tag{42.3c} \\
\frac{e \mapsto_{\text{par}} e_2'}{\text{let}(e_1; e_2; x_1 \cdot x_2 \cdot e) \mapsto_{\text{par}} [e_1, e_2/x_1, x_2]e} \tag{42.3d}
\end{align*}
\]

The parallel semantics 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. (We will return to this point in Section 42.4 on page 386.)

An important advantage of the present approach is captured by the implicit parallelism theorem, which states that the sequential and the parallel semantics 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. Put in other terms, $L\{\}$ and $L\{\text{conc}\}$ exhibits deterministic parallelism, which does not effect the correctness of programs, in contrast to the language $L\{\text{conc}\}$ (to be considered in Chapter 43), which exhibits non-deterministic parallelism, or concurrency.

**Lemma 42.1.** If $\text{let}(e_1; e_2; x_1 \cdot x_2 \cdot e) \mapsto_{\text{par}}^* v$ with $v \text{ val}$, then there exists $v_1 \text{ val}$ and $v_2 \text{ val}$ such that $e_1 \mapsto_{\text{par}} v_1$, $e_2 \mapsto_{\text{par}} v_2$, and $[v_1, v_2/x_1, x_2]e \mapsto_{\text{par}} v$.

**Proof.** Since $v \text{ val}$, the given derivation must consist of one or more steps. We proceed by induction on the derivation of the first step, $\text{let}(e_1; e_2; x_1 \cdot x_2 \cdot e) \mapsto_{\text{par}} e'$. For Rule (42.3d), we have $e_1 \text{ val}$ and $e_2 \text{ val}$, and $e' = [e_1, e_2/x_1, x_2]e$, so we may take $v_1 = e_1$ and $v_2 = e_2$ to complete the proof. The other cases follow easily by induction. \hfill \square

**Lemma 42.2.** If $\text{let}(e_1; e_2; x_1 \cdot x_2 \cdot e) \mapsto_{\text{seq}}^* v$ with $v \text{ val}$, then there exists $v_1 \text{ val}$ and $v_2 \text{ val}$ such that $e_1 \mapsto_{\text{seq}} v_1$, $e_2 \mapsto_{\text{seq}} v_2$, and $[v_1, v_2/x_1, x_2]e \mapsto_{\text{seq}} v$.

**Proof.** Similar to the proof of Lemma 42.2. \hfill \square
Theorem 42.3 (Implicit Parallelism). The sequential and parallel dynamics coincide: for all \( v \) \( \text{val} \), \( e \mapsto^*_{\text{seq}} v \iff e \mapsto^*_{\text{par}} v \).

Proof. From left to right it is enough to prove that if \( e \mapsto^*_{\text{seq}} e' \mapsto^*_{\text{par}} v \) with \( v \) \( \text{val} \), then \( e \mapsto^*_{\text{par}} v \). This may be shown by induction on the derivation of \( e \mapsto^*_{\text{seq}} e' \). If \( e \mapsto^*_{\text{seq}} e' \) by Rule (42.2c), then by Rule (42.3d) we have \( e \mapsto^*_{\text{par}} e' \), and hence \( e \mapsto^*_{\text{par}} v \). If \( e \mapsto^*_{\text{seq}} e' \) by Rule (42.2a), then we have \( e = \text{let}(e_1; e_2; x_1 . x_2 . e) \), \( e' = \text{let}(e'_1; e'_2; x_1 . x_2 . e) \), and \( e_1 \mapsto^*_{\text{seq}} e'_1 \). By Lemma 42.1 on the preceding page there exists \( v_1 \) \( \text{val} \) and \( v_2 \) \( \text{val} \) such that \( e'_1 \mapsto^*_{\text{par}} v_1, e_2 \mapsto^*_{\text{par}} v_2 \), and \( [v_1, v_2 / x_1, x_2] e \mapsto^*_{\text{par}} v \). By induction we have \( e_1 \mapsto^*_{\text{par}} v_1 \), and hence \( e \mapsto^*_{\text{par}} v \). The other cases are handled similarly.

From right to left, it is enough to prove that if \( e \mapsto^*_{\text{par}} e' \mapsto^*_{\text{seq}} v \) with \( v \) \( \text{val} \), then \( e \mapsto^*_{\text{seq}} v \). We proceed by induction on the derivation of \( e \mapsto^*_{\text{par}} e' \). Rule (42.3d) carries over directly to the sequential case by Rule (42.2c). Consider Rule (42.3a). We have \( \text{let}(e_1; e_2; x_1 . x_2 . e) \mapsto_{\text{par}} \text{let}(e'_1; e'_2; x_1 . x_2 . e), e_1 \mapsto_{\text{par}} e'_1, \) and \( e_2 \mapsto_{\text{par}} e'_2 \). By Lemma 42.2 on the previous page we have that there exists \( v_1 \) \( \text{val} \) and \( v_2 \) \( \text{val} \) such that \( e'_1 \mapsto^*_{\text{seq}} v_1, e'_2 \mapsto^*_{\text{seq}} v_2, \) and \( [v_1, v_2 / x_1, x_2] e \mapsto^*_{\text{seq}} v \). By induction we have \( e_1 \mapsto^*_{\text{seq}} v_1 \) and \( e_2 \mapsto^*_{\text{seq}} v_2, \) and hence \( e \mapsto^*_{\text{seq}} v \), as required. The other cases are handled similarly. \( \square \)

Theorem 42.3 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.

42.2 Cost Semantics

In this section we define a parallel cost semantics that assigns a cost graph to the evaluation of an expression. Cost graphs are defined by the following grammar:

\[
\text{Cost} \quad c \ ::= \quad 0 \quad \text{zero cost} \\
\quad | \quad 1 \quad \text{unit cost} \\
\quad | \quad c_1 \otimes c_2 \quad \text{parallel combination} \\
\quad | \quad c_1 \oplus c_2 \quad \text{sequential combination}
\]

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:

\[
wk(c) = \begin{cases} 
0 & \text{if } c = 0 \\
1 & \text{if } c = 1 \\
wk(c_1) + wk(c_2) & \text{if } c = c_1 \otimes c_2 \\
wk(c_1) + wk(c_2) & \text{if } c = c_1 \oplus c_2 
\end{cases} \tag{42.4}
\]

The depth is defined by the following equations:

\[
dp(c) = \begin{cases} 
0 & \text{if } c = 0 \\
1 & \text{if } c = 1 \\
\max(dp(c_1), dp(c_2)) & \text{if } c = c_1 \otimes c_2 \\
\max(dp(c_1), dp(c_2)) & \text{if } c = c_1 \oplus c_2 
\end{cases} \tag{42.5}
\]

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 12 we introduced cost semantics as a means of assigning time complexity to evaluation. The proof of Theorem 12.7 on page 96 shows that \( e \Downarrow^k v \) iff \( e \rightarrow^k v \). That is, the step complexity of an evaluation of \( e \) to a value \( v \) is just the number of transitions required to derive \( e \rightarrow^* v \). Here we use cost graphs as the measure of complexity, then relate these cost graphs to the transition semantics given in Section 42.1 on page 377.

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 semantics. By definition we arrange that \( e \Downarrow^0 e \) when \( e \) val. The cost assignment for \( \text{let} \) is given by the following rule:

\[
\frac{e_1 \Downarrow^{c_1} v_1 \quad e_2 \Downarrow^{c_2} v_2 \quad [v_1, v_2 / x_1, x_2] e \Downarrow^c v}{\text{let}(e_1; e_2; x_1. x_2. e) \Downarrow^{(c_1 \otimes c_2) \oplus 1 \oplus c} v}
\] (42.6)

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 semantics of other language constructs is specified in a similar manner, using only sequential combination so as to isolate the source of parallelism to the \( \text{let} \) construct.

The link between the cost semantics and the transition semantics 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 42.4 (Work Efficiency).** If \( e \Downarrow^c v \), then \( e \rightarrow^w v \) and \( e \rightarrow^d v \), where \( w = wk(c) \) and \( d = dp(c) \). Conversely, if \( e \rightarrow^w v \) and \( e \rightarrow^d v \), where \( v \) val, then \( e \Downarrow^c v \) for some cost graph \( c \) 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 (42.6). By induction we have \( e_1 \rightarrow^{w_1}_{\text{seq}} v_1 \), \( e_2 \rightarrow^{w_2}_{\text{seq}} v_2 \), and \( [v_1, v_2 / x_1, x_2] e \rightarrow^w v \), where \( w_1 = wk(c_1) \), \( w_2 = wk(c_2) \), and \( w = wk(c) \). By pasting together derivations we obtain a derivation

\[
\begin{align*}
\text{let}(e_1; e_2; x_1. x_2. e) &\rightarrow^{w_1}_{\text{seq}} \text{let}(v_1; e_2; x_1. x_2. e) \\
&\rightarrow^{w_2}_{\text{seq}} \text{let}(v_1; v_2; x_1. x_2. e) \\
&\rightarrow_{\text{seq}} [v_1, v_2 / x_1, x_2] e \\
&\rightarrow^w_{\text{seq}} v.
\end{align*}
\] (42.10)
Noting that \( wk((c_1 \otimes c_2) \oplus 1 \oplus c) = w_1 + w_2 + 1 + w \) completes the proof. Similarly, we have by induction that \( e_1 \mapsto_{\text{d1}} v_1 \), \( e_2 \mapsto_{\text{d2}} v_2 \), and \( e \mapsto_{\text{d}} v \), 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:

\[
\text{let}(e_1; e_2; x_1 \cdot x_2, e) \mapsto_{\text{par}} \text{let}(v_1; e_2'; x_1 \cdot x_2, e) \quad (42.11)
\]

\[
\mapsto_{\text{par}} \text{let}(v_1; v_2; x_1 \cdot x_2, e)
\]

\[
\mapsto_{\text{par}} [v_1, v_2/x_1, x_2]e
\]

Calculating \( dp((c_1 \otimes c_2) \oplus 1 \oplus c) = \max(d_1, d_2) + 1 + d \) completes the proof.

The second part is proved by induction on \( w \) (respectively, \( d \)) to obtain the required cost derivation. If \( w = 0 \), then \( e = v \) and hence \( e \Downarrow^0 v \). If \( w = w' + 1 \), then it is enough to show that if \( e \mapsto_{\text{seq}} e' \) and \( e' \Downarrow^c v \) with \( wk(c') = w' \), then \( e \Downarrow^c v \) for some \( c \) such that \( wk(c) = w \). We proceed by induction on the derivation of \( e \mapsto_{\text{seq}} e' \). Consider Rule (42.2c). We have \( e = \text{let}(e_1; e_2; x_1 \cdot x_2, e_0) \) with \( e_1 \) val and \( e_2 \) val, and \( e' = [e_1, e_2/x_1, x_2]e_0 \). By definition \( e_1 \Downarrow^0 e_1 \) and \( e_2 \Downarrow^0 e_2 \), since \( e_1 \) and \( e_2 \) are values. It follows that \( e \Downarrow^{(0 \otimes 0) \oplus 1 \oplus c} v \) by Rule (42.6). But \( wk((0 \otimes 0) \oplus 1 \oplus c') = 1 + wk(c') = 1 + w' = w \), as required. The remaining cases for sequential derivations follow a similar pattern. Turning to the parallel derivations, consider Rule (42.3a), in which we have \( e = \text{let}(e_1; e_2; x_1 \cdot x_2, e_0) \mapsto_{\text{par}} \text{let}(e'_1; e'_2; x_1 \cdot x_2, e_0) = e' \), with \( e_1 \mapsto_{\text{par}} e'_1 \) and \( e_2 \mapsto_{\text{par}} e'_2 \). We have by the outer inductive assumption that \( e' \Downarrow^c v \) for some \( c' \) such that \( dp(c') = d' \), and we are to show that \( e \Downarrow^c v \) for some \( c \) such that \( dp(c) = 1 + d' = d \). It follows from the form of \( e' \) and the determinacy of evaluation that \( c' = (c'_1 \otimes c'_2) \oplus 1 \oplus c_0 \), where \( c'_1 \Downarrow^{c'_1} v_1 \), \( c'_2 \Downarrow^{c'_2} v_2 \), and \( v_1, v_2/x_1, x_2, e_0 \Downarrow^{c_0} v \). It follows by the inner induction that \( e_1 \Downarrow^{c_1} v_1 \) for some \( c_1 \) such that \( dp(c_1) = dp(c'_1) + 1 \), and that \( e_2 \Downarrow^{c_2} v_2 \) for some \( c_2 \) such that \( dp(c_2) = dp(c'_2) + 1 \). But then \( e \Downarrow^c v \), where \( c = (c_1 \otimes c_2) \oplus 1 \oplus c_0 \). Calculating, we obtain

\[
\begin{align*}
dp(c) &= \max(dp(c'_1) + 1, dp(c'_2) + 1) + 1 + dp(c_0) \\
&= \max(dp(c'_1), dp(c'_2)) + 1 + 1 + dp(c_0) \\
&= dp((c'_1 \otimes c'_2) \oplus 1 \oplus c_0) + 1 \\
&= dp(c') + 1 \\
&= d' + 1 \\
&= d,
\end{align*}
\]
which completes the proof.

Theorem 42.4 on page 382 is the basis for saying that $L\{\text{and}\}$ is work-efficient—the computations performed in any execution, sequential or parallel, are precisely those that must be performed according to the sequential semantics. This is in contrast to speculative parallelism, as discussed in Chapter 41, in which we may schedule a task for execution whose outcome is not needed to determine the overall result of the computation.

### 42.3 Vector Parallelism

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 *vector* of values of a specified type. The primitive operations on vectors 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 vector simultaneously, forming a vector of the results.

We will consider here a very simple language, $L\{\text{vec}\}$, of vector operations to illustrate the main ideas.

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$\text{vec}(\tau)$</td>
<td>$\tau\text{vec}$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$\text{vec}(e_0, \ldots, e_{n-1})$</td>
<td>$[e_0, \ldots, e_{n-1}]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{sub}(e_1; e_2)$</td>
<td>$e_1 [e_2]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{rpl}(e_1; e_2)$</td>
<td>$\text{rpl}(e_1; e_2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{len}(e)$</td>
<td>$\text{len}(e)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{idx}(e)$</td>
<td>$\text{idx}(e)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{map}(e_1; x. e_2)$</td>
<td>$\langle e_2 \mid x \in e_1 \rangle$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{cat}(e_1; e_2)$</td>
<td>$\text{cat}(e_1; e_2)$</td>
</tr>
</tbody>
</table>

The expression $\text{vec}(e_0, \ldots, e_{n-1})$ evaluates to an $n$-vector whose elements are given by the expressions $e_0, \ldots, e_{n-1}$. The operation $\text{sub}(e_1; e_2)$ retrieves the element of the vector given by $e_1$ at the index given by $e_2$. The operation $\text{rpl}(e_1; e_2)$ creates a vector whose length is given by $e_1$ consisting solely of the element given by $e_2$. The operation $\text{len}(e)$ returns the number
of elements in the vector given by \( e \). The operation \( \text{idx}(e) \) creates a vector of length \( n \) (given by \( e \)) whose elements are 0, \ldots, \( n-1 \). The operation \( \text{map}(e_1; x. e_2) \) computes the vector whose \( i \)th element is the result of evaluating \( e_2 \) with \( x \) bound to the \( i \)th element of the vector given by \( e_1 \). The operation \( \text{cat}(e_1; e_2) \) concatenates two vectors of the same type.

The static semantics of these operations is given by the following typing rules:

\[
\begin{align*}
\Gamma &\vdash e_0 : \tau \quad \cdots \quad \Gamma &\vdash e_{n-1} : \tau \\
\Gamma &\vdash \text{vec}(e_0, \ldots, e_{n-1}) : \text{vec}(\tau)
\end{align*}
\] (42.21a)

\[
\begin{align*}
\Gamma &\vdash e_1 : \text{vec}(\tau) \quad \Gamma &\vdash e_2 : \text{nat} \\
\Gamma &\vdash \text{sub}(e_1; e_2) : \tau
\end{align*}
\] (42.21b)

\[
\begin{align*}
\Gamma &\vdash e_1 : \text{nat} \quad \Gamma &\vdash e_2 : \tau \\
\Gamma &\vdash \text{rpl}(e_1; e_2) : \text{vec}(\tau)
\end{align*}
\] (42.21c)

\[
\begin{align*}
\Gamma &\vdash e : \text{vec}(\tau) \\
\Gamma &\vdash \text{len}(e) : \text{nat}
\end{align*}
\] (42.21d)

\[
\begin{align*}
\Gamma &\vdash e : \text{nat} \\
\Gamma &\vdash \text{idx}(e) : \text{vec}(\text{nat})
\end{align*}
\] (42.21e)

\[
\begin{align*}
\Gamma &\vdash e_1 : \text{vec}(\tau) \quad \Gamma, x : \tau \vdash e_2 : \tau' \\
\Gamma &\vdash \text{map}(e_1; x. e_2) : \text{vec}(\tau')
\end{align*}
\] (42.21f)

\[
\begin{align*}
\Gamma &\vdash e_1 : \text{vec}(\tau) \quad \Gamma &\vdash e_2 : \text{vec}(\tau) \\
\Gamma &\vdash \text{cat}(e_1; e_2) : \text{vec}(\tau)
\end{align*}
\] (42.21g)

The cost semantics of these primitives is given by the following rules:

\[
\begin{align*}
&
\begin{array}{c}
\mathit{e}_0 \downarrow^{c_0} v_0 \quad \cdots \quad \mathit{e}_{n-1} \downarrow^{c_{n-1}} v_{n-1}
\end{array} \\
&\text{vec}(v_0, \ldots, v_{n-1}) \downarrow^{\oplus_{i=0}^{n-1} c_i} \text{vec}(v_0, \ldots, v_{n-1})
\tag{42.22a}
\end{align*}
\]

\[
\begin{align*}
&
\begin{array}{c}
\mathit{e}_1 \downarrow^{c_1} \text{vec}(v_0, \ldots, v_{n-1}) \\
\mathit{e}_2 \downarrow^{c_2} \text{num}[I]
\end{array} \\
&\text{sub}(e_1; e_2) \downarrow^{\mathit{c}_1 \mathit{c}_2 \oplus \oplus_{i=0}^{n-1} I} v_i \\
&\mathit{e}_1 \downarrow^{c_1} \text{num}[n] \quad \mathit{e}_2 \downarrow^{c_2} v
\tag{42.22b}
\end{align*}
\]

\[
\begin{align*}
&
\begin{array}{c}
\mathit{e}_1 \downarrow^{c_1} \text{num}[n] \\
\mathit{e}_2 \downarrow^{c_2} v
\end{array} \\
&\text{rpl}(e_1; e_2) \downarrow^{\mathit{c}_1 \mathit{c}_2 \oplus \oplus_{i=0}^{n-1} I} \text{vec}(v_i, \ldots, v)
\tag{42.22c}
\end{align*}
\]

\[
\begin{align*}
&
\begin{array}{c}
\mathit{e} \downarrow^{c} \text{vec}(v_0, \ldots, v_{n-1})
\end{array} \\
&\text{len}(e) \downarrow^{c \oplus \text{I}} \text{num}[n]
\tag{42.22d}
\end{align*}
\]

\[
\begin{align*}
&
\begin{array}{c}
\mathit{e} \downarrow^{c} \text{num}[n]
\end{array} \\
&\text{idx}(e) \downarrow^{c \oplus \oplus_{i=0}^{n-1} I} \text{vec}(0, \ldots, n-1)
\tag{42.22e}
\end{align*}
\]
The cost semantics for vector operations may be validated by introducing a sequential and parallel cost semantics and extending the proof of Theorem 42.4 on page 382 to cover this extension.

## 42.4 Provable Implementations

Theorem 42.4 on page 382 states that the cost semantics accurately models the dynamics of the parallel let construct, whether executed sequentially or in parallel. This validates the cost semantics from the point of view of the dynamics of \( L\{\text{and}\} \), 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 **provable implementation** is one for which we may establish an asymptotic bound on the actual execution time once these overheads are taken into account.

For the purposes of this chapter, we define a **symmetric multiprocessor**, or SMP, to be a shared-memory multiprocessor with an interconnection network that implements a synchronization construct equivalent to a parallel-fetch-and-add instruction in which any number of processors may simultaneously add a value to a shared memory location, retrieving the previous contents, while ensuring that each processor obtains the result it would obtain in some sequential ordering of their execution. Most multiprocessors implement an instruction of expressive power equivalent to the fetch-and-add to provide a foundation for parallel programming. In the following analysis we assume that the fetch-and-add instruction takes constant time, but the result can be adjusted (as noted below) to account for the overhead of implementing it under more relaxed assumptions about the processor network.
The main result relating the abstract cost to its concrete realization on a \( p \)-processor SMP is an application of Brent’s Principle, which describes how to implement arithmetic expressions on a parallel processor.

**Theorem 42.5.** If \( e \Downarrow \mathcal{C} v \) with \( \operatorname{wk}(c) = w \) and \( dp(c) = d \), then \( e \) may be evaluated on a \( p \)-processor SMP in time \( O(\max(w/p, d)) \).

Since the work always dominates the depth, if \( p = 1 \), then the theorem reduces to the statement that \( e \) may be evaluated in time \( O(w) \), the sequential complexity of the expression. That is, the work cost is asymptotically realizable on a single processor machine. For the general case the theorem tells us that we can never evaluate \( e \) in fewer steps than its depth cost, since this is the critical path length, and, for computations with shallow depth, we can achieve the best-possible result of dividing up the work evenly among the \( p \) processors.

Theorem 42.5 suggests a characterization of those problems for which having a great degree of parallelism (more processing elements) improves the running time. For a computation of depth \( d \) and work \( w \), we can make good use of parallelism whenever \( w/p > d \), which occurs when the parallelizability ratio, \( w/d \), is at least \( p \). In a highly sequential program the work is directly proportional to the depth, and so the parallelizability is constant. This implies that increasing \( p \) does not speed up the computation. On the other hand, a highly parallelizable computation is one with constant depth, or depth \( d \) proportional to \( \lg w \). Such programs have a high parallelizability ratio, and hence are amenable to speedup by increasing the number of available processors. It is worth stressing that it is not known whether all problems admit a parallelizable solution or not. 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 an important open problem in complexity theory to characterize which problems are parallelizable, and which are not.

The proof of Theorem 42.5 amounts to a design for the implementation of \( L\{ \) and \( \} \). A critical ingredient is scheduling the workload onto the \( p \) processors so as to maximize their utilization. This is achieved by maintaining a shared worklist of tasks that have been created by evaluation of a parallel \lit{let} construct, all of which must be completed to determine the final outcome of the computation. (Here we make use of shared memory so that all processors have access to the central worklist.) Execution is divided into rounds. At the end of each round a processor may complete execution, in
which case further work can be scheduled onto it; it may continue execution into the next round; or it may fork two additional tasks to be scheduled for execution, blocking until they complete.

To start the next round the processors must collectively assign work to themselves so that if sufficient work is available, then all \( p \) processors will be assigned work. Assume that we have at least \( p \) units of work remaining to be done at any given time (otherwise just consider all remaining work in what follows). Each step of execution on each processor consists of executing an instruction of \( \mathcal{L}\{\text{and}\} \). After this step a task may either be complete, or may continue with further execution, or may fork two new tasks as a result of executing a parallel \( \text{let} \) instruction, or it may join two completed tasks into one. The synchronization required for a join may be implemented on an SMP by allocating a data structure to each (dynamic) join point, and arranging that the parallel threads signal their completion by atomically posting their result to this data structure. The first thread to complete stores its result in this data structure (atomically, to avoid race conditions). When the second thread completes, it continues from the join point, passing along its own result and that of the first thread to complete.

Theorem 42.5 on the preceding page may also be extended to the vector operations discussed in Section 42.3 on page 384. The proof requires that we specify an algorithm to implement each of the operations in the time bounds specified by the cost semantics in accordance with the theorem.

To get an idea of what is involved, let us consider how to implement the operation \( \text{idx}(e) \) on a \( p \)-processor SMP. We wish to show, consistently with Theorem 42.5 on the preceding page, that this operation may be implemented in time \( O(\max(n/p, 1)) \), where \( e \) evaluates to \( n \). This may be achieved as follows. First, reserve, in constant time, an uninitialized region of \( n \) words of memory for the vector to be created by this operation. To initialize this memory, we assign responsibility for a segment of size \( n/p \) to each of the \( p \) processors, which then execute in parallel to fill in the required values. To do this we must assign to processor \( i \) the starting point, \( n_i \), of the \( i \)th segment. The starting points are calculating by constructing, in constant time, the vector of numbers \( 0, \ldots, p-1 \), each of which is then multiplied by \( n/p \) to obtain the required vector \( n_0, \ldots, n_{p-1} \). Processor \( i \) will then initialize the segment starting at \( n_i \) to the numbers \( n_i, n_i + 1, \ldots, n_i + (n/p) - 1 \). Each processor required \( O(n/p) \) to perform this, and all processors may execute in parallel without further coordination, achieving the required bound. Note that had we specified, say, a unit cost for the index operation, we would have been unable to extend the proof of Theorem 42.5 on the previous page, because it is not possible to write the required \( O(n) \) data
42.5 Exercises

items in $O(1)$ time.

42.5 Exercises
Part XVI

Concurrency
Chapter 43

Process Calculus

So far we have mainly studied the static and dynamic semantics 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. There are many forms of process calculi, differing in technical details and in emphasis. We will consider the best-known formalism, which is called the \( \pi \)-calculus. The development will proceed in stages, starting with simple action models, then extending to interacting concurrent processes, and finally to the synchronous and asynchronous variants of the \( \pi \)-calculus itself.

Our presentation of the \( \pi \)-calculus 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 a choice of events. Other forms of process include parallel composition, the introduction of a communication channel, and, in the asynchronous case, a send on a channel. The basic form of event is the ability to read (and, in the synchronous case, write) on a channel. Events are combined by a non-deterministic choice operator. Even the choice operator can be eliminated in favor of a protocol for treating a parallel composition of events as a non-deterministic choice among them.
43.1 Actions and Events

Our treatment of concurrent interaction is based on the notion of an event, which specifies the set of 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 read and a write on a common channel. The processes synchronize on these complementary actions, after which they may 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.

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>P</td>
<td>await(E)</td>
<td>$E$</td>
</tr>
<tr>
<td>Event</td>
<td>E</td>
<td>null</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>choice(E₁; E₂)</td>
<td>$E₁ + E₂$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rcv<a href="P">a</a></td>
<td>$?a . P$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>snd<a href="P">a</a></td>
<td>$!a . P$</td>
</tr>
</tbody>
</table>

The variables $a$, $b$, and $c$ range over channels, which serve as synchronization sites between processes.

We will handle events modulo structural congruence, written $P₁ ≡ P₂$ and $E₁ ≡ E₂$, respectively, which is the strongest equivalence relation closed under the following rules:

\[
\begin{align*}
E \equiv E' \\
$E \equiv $$E'$
\end{align*}
\]

\[
\begin{align*}
E₁ \equiv E₁' \quad E₂ \equiv E₂' \\
E₁ + E₂ \equiv E₁' + E₂'
\end{align*}
\]

\[
\begin{align*}
P \equiv P' \\
?a . P \equiv ?a . P'
\end{align*}
\]

\[
\begin{align*}
P \equiv P' \\
!a . P \equiv !a . P'
\end{align*}
\]

\[
\begin{align*}
E + 0 \equiv E
\end{align*}
\]

\[
\begin{align*}
E₁ + E₂ \equiv E₂ + E₁
\end{align*}
\]

\[
\begin{align*}
E₁ + (E₂ + E₃) \equiv (E₁ + E₂) + E₃
\end{align*}
\]
The importance of imposing structural congruence on sequential processes is that it enables us to think of an event as having the form of a finite sum of send or receive events, with the sum of zero events being the null event, 0.

An illustrative example of Robin 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.

\[ V = \$ (\langle 2p \rangle . (1\text{tea} . V + \langle 2p \rangle . (1\text{cof} . V))) \]

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.

### 43.2 Concurrent Interaction

We enrich the language of processes with concurrent composition.

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>$P$</td>
<td>$\text{await} (E)$</td>
<td>$$ E$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{stop}$</td>
<td>$1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{par}(P_1; P_2)$</td>
<td>$P_1 \parallel P_2$</td>
</tr>
</tbody>
</table>

The process 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 1 with $\$ 0$, the process that awaits the event that will never occur, but we prefer to treat the inert process as a primitive concept.

Structural congruence for processes is enriched by the following rules governing the inert process and concurrent composition of processes:

\[ P \parallel 1 \equiv P \quad \text{(43.2a)} \]

\[ P_1 \parallel P_2 \equiv P_2 \parallel P_1 \quad \text{(43.2b)} \]
Concurrent Interaction

\[ P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3 \]  

(43.2c)

\[
\begin{align*}
P_1 & \equiv P'_1 \\
P_2 & \equiv P'_2 \\
P_1 \parallel P_2 & \equiv P'_1 \parallel P'_2
\end{align*}
\]  

(43.2d)

Up to structural equivalence 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 is the process \( 1 \).

The dynamic semantics of concurrent interaction is defined by an action-indexed family of transition judgements, \( P \xrightarrow{\alpha} P' \), where \( \alpha \) is an action as specified by the following grammar:

\[
\begin{array}{c|c}
\text{Category} & \text{Item} \\
\hline
\text{Action} & \alpha \\
\hline
\end{array}
\]

\[
\begin{align*}
\alpha ::= & \text{rcv}[a] \quad ?a \\
& \text{snd}[a] \quad !a \\
& \text{sil} \quad \epsilon
\end{align*}
\]

The action label on a transition specifies the effect of an execution step on the environment in which it occurs. The receive action, \( ?a \), and the send action, \( !a \), are complementary. Two concurrent processes may interact whenever they announce complementary actions, resulting in a silent transition, which is labelled by the silent action, \( \text{sil} \).

\[
\begin{align*}
P_1 \equiv P'_1 & \quad P'_1 \xrightarrow{\alpha} P'_2 \quad P'_2 \equiv P_2 \\
P_1 & \xrightarrow{\alpha} P_2
\end{align*}
\]  

(43.3a)

\[
\begin{align*}
\$ (\!a . P + E) & \xrightarrow{\!a} P
\end{align*}
\]  

(43.3b)

\[
\begin{align*}
\$ (?a . P + E) & \xrightarrow{?a} P
\end{align*}
\]  

(43.3c)

\[
\begin{align*}
P_1 \parallel P_2 & \xrightarrow{\alpha} P'_1 \parallel P_2 \\
P_1 & \xrightarrow{\alpha} P'_1
\end{align*}
\]  

(43.3d)

\[
\begin{align*}
P_1 & \xrightarrow{\!a} P'_1 \\
P_2 & \xleftarrow{?a} P'_2 \\
P_1 \parallel P_2 & \xrightarrow{\alpha} P'_1 \parallel P'_2
\end{align*}
\]  

(43.3e)
Rules (43.3b) and (43.3c) specify that any of the events on which a process is synchronizing may occur. Rule (43.3e) synchronizes two processes that take complementary actions.

As an example, let us consider the interaction of the vending machine, \( V \), with the user process, \( U \), defined as follows:

\[
U = \$ \!2p. \$ !2p. \$ ?cof. 1.
\]

Here is a trace of the interaction between \( V \) and \( U \):

\[
V || U \xrightarrow{!tea.} \xrightarrow{!2p.} \xrightarrow{!cof.} \xrightarrow{?cof.} 1.
\]

These steps are justified, respectively, by the following pairs of labelled transitions:

\[
U \xrightarrow{!2p.} U' = \$ !2p. \$ ?cof. 1
\]

\[
V \xrightarrow{?2p.} V' = (!tea. V + !2p. \$ !cof. V)
\]

\[
U' \xrightarrow{!2p.} U'' = \$ ?cof. 1
\]

\[
V' \xrightarrow{?2p.} V'' = !cof. V
\]

\[
U'' \xrightarrow{?cof} 1
\]

\[
V'' \xrightarrow{!cof} V
\]

We have suppressed uses of structural congruence in the above derivations to avoid clutter, but it is important to see its role in managing the non-deterministic choice of events by a process.

### 43.3 Replication

Some presentations of process calculus 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

\[
* P \equiv P \parallel * P.
\]
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 dynamic semantics to model the details of replication more closely:

\[ *P \rightsquigarrow P \parallel *P. \]

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

\[ A = *\$(?a.P(\$(!a.1))), \]

in which we have replaced occurrences of \( X \) within \( P \) by an initiator process that signals the event \( a \) to the activator. Observe that the activator, \( A \), is structurally congruent to the process \( A' \parallel A \), where \( A' \) is the process

\[ \$(?a.P(\$(!a.1))). \]

To start process \( P \) we concurrently compose the activator, \( A \), with an initiator process, \( \$(!a.1) \). Observe that

\[ A \parallel \$(!a.1) \rightsquigarrow A \parallel P(!a.1), \]

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:

\[
\begin{align*}
V_1 &= *\$(?v.V_2) \\
V_2 &= \$(?2p.(!tea.V_0 + ?2p.(!cof.V_0))) \\
V_0 &= \$(!v.1)
\end{align*}
\]

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, \( V_0 \), is simulated by the concurrent composition \( V_0 \parallel V_1 \).
43.4 Private Channels

It is often desirable to isolate interactions among a group of concurrent processes from those among another group of processes. This can be achieved by creating a private channel that is shared among those in the group, and which is inaccessible from all other processes. This may be modeled by enriching the language of processes with a construct for creating a new channel:

\[
\text{Process } P ::= \text{new}(a \cdot P) \quad \nu(a \cdot P)
\]

As the syntax suggests, this is a binding operator in which the channel \(a\) is bound within \(P\).

Structural congruence is extended with the following rules:

\[
\frac{P =_{\alpha} P'}{P \equiv P'} \quad (43.7a)
\]

\[
\frac{P \equiv P'}{\nu(a \cdot P) \equiv \nu(a \cdot P')} \quad (43.7b)
\]

\[
\frac{a \notin P_2}{\nu(a \cdot P_1) \parallel P_2 \equiv \nu(a \cdot P_1 \parallel P_2)} \quad (43.7c)
\]

The last rule, called *scope extrusion*, is not strictly necessary at this stage, but will be important in the treatment of communication in the next section.

The dynamic semantics is extended with one additional rule permitting steps to take place within the scope of a binder.

\[
\frac{P \xrightarrow{\alpha} P' \quad a \notin \alpha}{\nu(a \cdot P) \xrightarrow{\alpha} \nu(a \cdot P')} \quad (43.8)
\]

No process may interact with \(\nu(a \cdot P)\) along the newly-allocated channel, for to do so would require knowledge of the private channel, \(a\), which is chosen, by the magic of \(\alpha\)-equivalence, to be distinct from all other channels in the system.

As an example, let us consider again the non-recursive definition of the vending machine. 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 \(\nu(\nu(v \cdot V_0 \parallel V_1))\), 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:

\[
\nu(v \cdot V_0 || V_1) \implies \nu(v \cdot V_2) \equiv \nu(v \cdot V_2 || U)
\]

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.

### 43.5 Synchronous Communication

The concurrent process calculus presented in the preceding section models synchronization based on the willingness of two processes to undertake complementary actions. A natural extension of this model is to permit data to be passed from one process to another as part of synchronization. Since we are abstracting away from the computation occurring within a process, it would not make much sense to consider, say, passing an integer during synchronization. A more interesting possibility is to permit passing channels, so that new patterns of connectivity can be established as a consequence of inter-process synchronization. This is the core idea of the \( \pi \)-calculus.

The syntax of events is changed to account for communication by generalizing send and receive events as specified in the following grammar:

\[
\begin{array}{l}
\text{Category} & \text{Item} & \text{Abstract} & \text{Concrete} \\
\text{Event} & E & ::= & \text{rcv}[a](x \cdot P) & ?a(x) \cdot P \\
& & & \text{snd}[a;b](P) & !a(b) \cdot P \\
\end{array}
\]

The event \( ?a(x) \cdot P \) binds the variable \( x \) within the process expression \( P \). The rest of the syntax remains as described earlier in this chapter.

The syntax of actions is generalized along similar lines, with both the send and receive actions specifying the data communicated by the action.

\[
\begin{array}{l}
\text{Category} & \text{Item} & \text{Abstract} & \text{Concrete} \\
\text{Action} & a & ::= & \text{rcv}[a](b) & ?a(b) \\
& & & \text{snd}[a](b) & !a(b) \\
\end{array}
\]

The action \( !a(b) \) represents a write, or send, of a channel, \( b \), along a channel, \( a \). The action \( ?a(b) \) represents a read, or receive, along channel, \( a \), of another channel, \( b \).
Interaction in the $\pi$-calculus consists of synchronization on the concurrent availability of complementary actions on a channel, passing a channel from the sender to the receiver.

\[
\begin{align*}
\frac{!a(b) . (P + E)}{P} & \quad \text{(43.9a)} \\
\frac{?a(x) . (P + E)}{[b/x]P} & \quad \text{(43.9b)} \\
\frac{P_1 !a(b) \rightarrow P'_1 \quad P_2 ?a(b) \rightarrow P'_2}{P_1 \parallel P_2 \rightarrow P'_1 \parallel P'_2} & \quad \text{(43.9c)}
\end{align*}
\]

In contrast to pure synchronization, the message-passing form of interaction is fundamentally asymmetric — the receiver continues with the channel passed by the sender substituted for the bound variable of the action. Rule (43.9b) may be seen as “guessing” that the received data will be $b$, which is substituted into the resulting process.

### 43.6 Polyadic Communication

So far communication is limited to sending and receiving a single channel along another channel. It is often useful to consider more flexible forms of communication in which zero or more channels are communicated by a single interaction. Transmitting no data corresponds to a pure signal on a channel in which the mere fact of the communication is all that is transmitted between the sender and the receiver. Transmitting more than one channel corresponds to a packet in which a single interaction communicates a finite number of channels from sender to receiver.

The polyadic $\pi$-calculus is the generalization of the $\pi$-calculus to admit communication of multiple channels between sender and receiver in a single interaction. The syntax of the polyadic $\pi$-calculus is a simple extension of the monadic $\pi$-calculus in which send and receive events, and their corresponding actions, are generalized as follows:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event $E$</td>
<td>$\text{rcv}<a href="x_1,%5Cldots,x_k,P">a</a>$</td>
<td>$?a(x_1,\ldots,x_k).P$</td>
<td>$a(b_1,\ldots,b_k).P$</td>
</tr>
<tr>
<td></td>
<td>$\text{snd}<a href="P">a; b_1,\ldots,b_k</a>$</td>
<td>$!a(b_1,\ldots,b_k).P$</td>
<td>$a(b_1,\ldots,b_k)$</td>
</tr>
<tr>
<td>Action $a$</td>
<td>$\text{rcv}<a href="b_1,%5Cldots,b_k">a</a>$</td>
<td>$?a(b_1,\ldots,b_k)$</td>
<td>$a(b_1,\ldots,b_k)$</td>
</tr>
</tbody>
</table>
The index $k$ ranges over natural numbers. When $k$ is zero, the events model pure signals, and when $k > 1$, the events model communication of packets along a channel. There arises the possibility of sending more or fewer values along a channel than are expected by the receiver. To remedy this one may associate with each channel a unique arity $k \geq 0$, which represents the size of any packet that it may carry. The syntax of the polyadic $\pi$-calculus should then be restricted to respect the arity of the channel. We leave the specification of this refinement as an exercise for the reader.

The rules for structural congruence and interaction generalize in the evident manner to the polyadic case.

### 43.7 Mutable Cells as Processes

Let us consider a reference cell server that, when contacted on a pre-determined channel with an initial value and a response channel, creates a fresh cell that may be contacted on a dedicated channel that is returned on the response channel. The client may either receive from or send a value along the dedicated channel dedicated in order to retrieve or modify the current contents of the associated cell.

The reference server, when contacted on channel $r$ providing an initial contents and a response channel, creates a new cell server process and a new channel on which to contact it.

$$R(r) = * \$(\ ?r(x,k) . v(l) . (\ !c(x,l) . 1) \parallel \$ (\ !k(l) . 1))$$

The reference server, when provided an initial value, $x$, and a response channel, $k$, allocates a new channel that serves as the name of the newly allocated cell, then contacts the cell service, providing $x$ and $l$, to create a new cell, and sends $l$ back along the response channel.

The cell server, when contacted on channel $c$ providing an initial contents, $x$, and a channel, $l$, creates a server that may be contacted on channel $l$ to set and retrieve the contents of that cell.

$$C(c) = * \$(\ ?c(x,l) . (S(l) + G(x,l)))$$
$$S(l) = !l(x') . (\ !c(x',l) . 1)$$
$$G(x,l) = !l(x) . (\ !c(x,l) . 1)$$

The cell server listens on channel $c$ for an initial contents and a channel, $l$, and establishes a server that listens on channel $l$ for either a send or a receive. If a new value is received it creates a new cell server with that value.
as contents, but that may be contacted on the same channel. Otherwise it sends the current value on the same channel, and restarts the server loop.

The use the reference service in a process $P$, we concurrently compose $P$ with $R(r)$ and $C(c)$, where $r$ and $c$ are distinct channels dedicated to these services.

$$\nu(r.\nu(c.P \parallel R(r) \parallel C(c))).$$

The process $P$ allocates a response channel, and communicates with the reference server:

$$P = \nu(k.\nu(r(x_0,k).1 \parallel ?k(l) \ldots))$$

The process allocates a response channel, and sends it to the reference server, along with the initial contents of the cell. It then listens on the response channel for the channel on which to contact the cell, then proceeds (in the elided portion of the code) to interact with the cell along that channel.

### 43.8 Asynchronous Communication

This form of interaction is called *synchronous*, because both the sender and the receiver are blocked from further interaction until synchronization has occurred. On the receiving side this is inevitable, because the receiver cannot continue execution until the channel which it receives has been determined, much as the body of a function cannot be executed until its argument has been provided. On the sending side, however, there is no fundamental reason why notification is required; the sender could simply send the message along a channel without specifying how to continue once that message has been received. This “fire and forget” semantics is called *asynchronous* communication, in constrast to the *synchronous* form just described.

The *asynchronous* $\pi$-calculus is obtained by removing the synchronous send event, $\nu(a) . P$, and adding a new form of process, the asynchronous send process, written $\nu(a)$, which has no continuation after the send. The
syntax of the asynchronous $\pi$-calculus is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>$P$</td>
<td>$\text{snd}<a href="b">a</a>$</td>
<td>$!a(b)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{await}(E)$</td>
<td>$E$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{par}(P_1;P_2)$</td>
<td>$P_1 \parallel P_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{new}(a,P)$</td>
<td>$v(a.P)$</td>
</tr>
<tr>
<td>Event</td>
<td>$E$</td>
<td>$\text{null}$</td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{rcv}<a href="x.P">a</a>$</td>
<td>$?a(x).P$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{choice}(E_1;E_2)$</td>
<td>$E_1 + E_2$</td>
</tr>
</tbody>
</table>

Up to structural congruence, an event is just a choice of zero or more reads along any number of channels.

The dynamic semantics for the asynchronous $\pi$-calculus is defined by omitting Rule (43.9a), and adding the following rule for the asynchronous send process:

$$
!a(b) \xrightarrow{!a(b)} 1 \quad (43.10)
$$

One may regard the pending asynchronous write as a kind of buffer in which the message is held until a receiver is chosen.

In a sense the synchronous $\pi$-calculus is more fundamental than the asynchronous variant, because we may always mimic the asynchronous send by a process of the form $\$ !a(b) . 1$, which performs the send, and then becomes the inert process 1. In another sense, however, the asynchronous $\pi$-calculus is more fundamental, because we may encode a synchronous send by introducing a notification channel on which the receiver sends a message to notify the sender of the successful receipt of its message. This exposes the implicit communication required to implement synchronous send, and avoids it in cases where it is not needed (in particular, when the resumed process is just the inert process, as just illustrated).

To get an idea of what is involved in the encoding of the synchronous $\pi$-calculus in the asynchronous $\pi$-calculus, we sketch the implementation of an acknowledgement protocol that only requires (polyadic) asynchronous communication. A synchronous process of the form

$$
\nu(a.\$ ((!a(b).P) + E) \parallel \$ ((?a(x).Q) + F))
$$

is represented by the asynchronous process

$$
\nu(a.\nu(a_0.P' \parallel Q')),
$$
where \( a_0 \notin P, a_0 \notin Q \), and we define

\[
P' = !a(b, a_0) \parallel S (\pi a_0 \cdot P + E)
\]

and

\[
Q' = S (\pi a(x, x_0) \cdot (\pi x_0 \cdot Q) + F).
\]

The process that is awaiting the outcome of a send event along channel \( a \) instead sends the argument, \( b \), along with a newly allocated acknowledgement channel, \( a_0 \), along the channel \( a \), then awaits receipt of a signal in the form of a null message along \( a_0 \), then acts as the process \( P \). Correspondingly, the process that is awaiting a receive event along channel \( a \) must be prepared to receive, in addition, the acknowledgement channel, \( x_0 \), on which it sends an asynchronous signal back to the sender, and proceeds to act as the process \( Q \). It is easy to check that the synchronous interaction of the original process is simulated by several steps of execution of the translation into asynchronous form.

### 43.9 Definability of Input Choice

It turns out that we may simplify the asynchronous \( \pi \)-calculus even further by eliminating the non-deterministic choice of events by defining it in terms of parallel composition of processes. This means, in fact, that we may do away with the concept of an event entirely, and just have a very simple calculus of processes defined by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>( P )</td>
<td>( \text{snd}<a href="b">a</a> )</td>
<td>( !a(b) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{rcv}[a](x \cdot P) )</td>
<td>( ?a(x) \cdot P )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{stop} )</td>
<td>( 1 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{par}(P_1; P_2) )</td>
<td>( P_1 \parallel P_2 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{new}(a \cdot P) )</td>
<td>( \nu(a \cdot P) )</td>
</tr>
</tbody>
</table>

This reduces the language to three main concepts: channels, communication, and concurrent composition.

The elimination of non-deterministic choice is based on the following intuition. Let \( P \) be a process of the form

\[
S (\pi a_1(x_1) \cdot P_1 + \ldots + \pi a_k(x_k) \cdot P_k).
\]

Interaction with this process by a sending a channel, \( b \), along channel \( a_i \) involves two separable actions:
1. The transmitted value, $b$, must be substituted for $x_i$ in $P_i$ to obtain the resulting process, $[b/x_i]P_i$, of the interaction.

2. The other events must be “killed off”, since they were not chosen by the interaction.

Ignoring the second action for the time being, the first may be met by simply regarding $P$ as the following parallel composition of processes:

$$\nu(t).S_t || \nu(a_1(x_1)).P_1 || \ldots || \nu(a_k(x_k)).P_k,$$

When concurrently composed with a sending process $!a_i(b)$, this process interactions to yield $[b/x]P_i$, representing the same non-deterministic choice of interaction. However, the interaction fails to “kill off” the processes that were not chosen when the communication along $a_i$ was chosen.

To rectify this we modify the encoding of choice to incorporate a protocol for signalling the non-selected processes that they are not eligible to participate in any further communication events. This is achieved by associating a fresh channel with each receive event group of the form illustrated by $P$ above, and arranging that if any of the receiving processes is chosen, then the others become “zombies” that are disabled from further interaction. The process $P$ is represented by the process $P'$ given by the expression

$$v(t).S_t || \nu(a_1(x_1)).P'_1 || \ldots || \nu(a_k(x_k)).P'_k),$$

where $P'_i$ is the process

$$v(s.v(f) || !t(s,f) || ?s() . (F_t || P_i) || ?f() . (F_t || !a_i(x_i))) .$$

The process $S_t$ signals success when contacted on channel $t$,

$$S_t = ?t(s,f) . !s()$$

and the process $F_t$ signals failure when contacted on channel $t$,

$$F_t = ?t(s,f) . !f().$$

The process $P'$ allocates a new channel that is shared by all of the processes participating in the encoding of the process $P$. It then creates $k + 1$ processes, one for each summand, and a “success” process that mediates the protocol. The summands all wait for communication on their respective channels, and the mediating process signals success when contacted. When a concurrently executing process interacts with $P'$ by sending a channel $b$
to $P'_i$ along channel $a_i$, the protocol is initiated. First, the process $P'_i$ sends a newly allocated success and failure channel to the mediator process, and awaits further communication along these channels. (The new channels serve to identify this particular interaction of $P'$ with its environment.) The mediator signals success, and terminates. The signal activates the receive event along the success channel of $P'_i$, which then activates a new mediator, the “failure” process, to replace the original, “success” process, and also activates $P_i$ since this summand has been chosen for the interaction. All other summands remain active, receiving communications on their respective channels, with the concurrently executing mediator being the “failure” process. Should any of these summands be selected for communication, it is their job as zombies to die off after ensuring that the failing mediator is reinstated (for the sake of the other zombie processes) and re-sending the received message so that it may be propagated to a “living” recipient (that is, one that has not been disabled by a previous interaction with one of its cohort).

43.10 Exercises
Chapter 44

Monadic Concurrency

In this chapter we utilize the process calculus presented in Chapter 43 to derive a uniform treatment of several seemingly disparate concepts: mutable storage, speculative parallelism, input/output, process creation, and interprocess communication. The unifying theme is to use a process calculus to give an account of context-sensitive execution. For example, inter-process communication necessarily involves the execution of two processes, each in a context that includes the other. The two processes synchronize, and continue execution separately after their rendezvous.

44.1 Framework

The language $L\{\text{conc}\}$ is an extension of $L\{\text{cmd}\}$ (described in Chapter 48) with an additional level of processes, which represent concurrently executing agents. The syntax of $L\{\text{conc}\}$ is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$\text{cmd}(\tau)$</td>
<td>$\tau\text{cmd}$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$\text{cmd}(m)$</td>
<td>$\text{cmd}(m)$</td>
</tr>
<tr>
<td>Comm</td>
<td>$m$</td>
<td>$\text{return}(e)$</td>
<td>$\text{return }e$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid \text{letcmd}(e;x.m)$</td>
<td>$\text{letcmd}(x) \text{ be } e \text{ in } m$</td>
</tr>
<tr>
<td>Proc</td>
<td>$p$</td>
<td>$\text{proc}<a href="m">a</a>$</td>
<td>${a : m}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid \text{par}(p_1; p_2)$</td>
<td>$p_1 \parallel p_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\mid \text{new}<a href="x.p">\tau</a>$</td>
<td>$\nu(x:\tau.p)$</td>
</tr>
</tbody>
</table>

The basic form of process is $\text{proc}[a](m)$, consisting of a single command, $m$, labelled with a symbol, $a$, that serves to identify it. We may also form
the parallel composition of processes, and generate a new symbol for use within a process.

As always, we identify syntactic objects up to $\alpha$-equivalence, so that bound names may always be chosen so as to satisfy any finitary constraint on their occurrence. As in Chapter 43, we also identify processes up to structural congruence, which specifies that parallel composition is commutative and associative, and that new symbol generation may have its scope expanded to encompass any parallel process, subject only to avoidance of capture.

In the succeeding sections of this chapter, the language $\mathcal{L}\{\text{conc}\}$ will be extended to model various forms of computational phenomena. In each case we will enrich the language with new forms of command, representing primitive capabilities of the language, and new forms of process, used to model the context in which commands are executed. In this respect it is misleading to think of processes as necessarily having to do with concurrent execution and synchronization! Rather, what processes provide is a simple, uniform means of describing the context in which a command is executed. This can include concurrent interaction (synchronization) in the familiar sense, but is not limited to this case.

The static semantics of $\mathcal{L}\{\text{conc}\}$ extends that of $\mathcal{L}\{\text{cmd}\}$ (see Chapter 48) to include the additional level of processes. Let $\Sigma$ range over finite sets of judgements of the form $a : \tau$, where $a$ is a symbol and $\tau$ is a type, such that no symbol is the subject of more than one such judgement in $\Sigma$. We define the judgement $p \ ok$ by the following rules:

\begin{align*}
\Sigma; \Gamma \vdash m \sim \tau \\
\Sigma, a : \text{proc}(\tau); \Gamma \vdash \text{proc}[a](m) \ ok
\end{align*} (44.1a)

\begin{align*}
\Sigma; \Gamma \vdash p_1 \ ok \\
\Sigma; \Gamma \vdash p_2 \ ok \\
\Sigma; \Gamma \vdash \text{par}(p_1; p_2) \ ok
\end{align*} (44.1b)

\begin{align*}
\Sigma, a : \tau; \Gamma \vdash p \ ok \\
\Sigma; \Gamma \vdash \text{new}[\tau](a,p) \ ok
\end{align*} (44.1c)

\begin{align*}
\Sigma; \Gamma \vdash p' \ ok \\
\Sigma; \Gamma \vdash p \ ok
\end{align*} (44.1d)

Rule (44.1a) specifies that a process of the form $\text{proc}[a](m)$ is well-formed if $m$ is a command yielding a value of type $\tau$, where $a$ is a process identifier of type $\tau$. The type $\text{proc}(\tau)$ is the type of process identifiers returning a value of type $\tau$. Rule (44.1b) states that a parallel composition of processes
is well-formed if both processes are well-formed. Rule (44.1c) enriches $\Sigma$ with a new symbol with a type $\tau$ chosen so that $p$ is well-formed under this assumption. Finally, Rule (44.1d) states that typing respects structural congruence. Ordinarily such a rule is left implicit, but we state it explicitly for emphasis.

Each extension of $\mathcal{L}\{\text{conc}\}$ considered below may introduce new forms of process governed by new formation and execution rules.

The dynamic semantics of $\mathcal{L}\{\text{conc}\}$ is defined by judgements of the form $p \mapsto p'$, where $p$ and $p'$ are processes. Execution of processes includes structural normalization, may apply to any active process, may occur within the scope of a newly introduced symbol, and respects structural congruence:

\[
\frac{m \mapsto m'}{\text{proc}[a](m) \mapsto \text{proc}[a](m')}
\]

Rule (44.2b) specifies that a process whose execution has completed normally announces this fact to the ambient context by offering the returned value labelled with the process’s identifier. This allows for other processes to notice that the process labelled $a$ has terminated, and to recover its returned value.

In the rest of this chapter we consider various forms of computation, each of which gives rise to new rules for process execution. These rules generally have the form of transitions of the form

\[
{\{a : m\} \xrightarrow{\alpha} \nu(a_{1 : \tau_{1}}, \ldots, \nu(a_{j : \tau_{j}} \parallel \nu(p_{1} \parallel \ldots \parallel p_{k})))},
\]

where $j, k \geq 0$ and $\alpha$ is an action appropriate to that form of computation.
44.2 Input/Output

Character input and output are readily modeled in \( L \{ \text{conc} \} \) by considering input and output ports to be channels on which we may transmit characters.

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comm</td>
<td>( m ) := getc() getc()</td>
<td>putc(( e )) putc(( e ))</td>
<td></td>
</tr>
</tbody>
</table>

The static semantics assumes that we have a type \( \text{char} \) of characters:

\[
\Sigma \Gamma \vdash \text{getc()} \sim \text{char} \tag{44.3a}
\]

\[
\Sigma \Gamma \vdash e : \text{char} \\
\Sigma \Gamma \vdash \text{putc}(e) \sim \text{char} \tag{44.3b}
\]

Given two distinguished ports, in and out, the dynamic semantics of character input/output may be given by the following rules:

\[
\{ a : \text{getc()} \} \xrightarrow{?\text{in}(c)} \{ a : \text{return } c \} \tag{44.4a}
\]

\[
\{ a : \text{putc}(c) \} \xrightarrow{!\text{out}(c)} \{ a : \text{return } c \} \tag{44.4b}
\]

As a technical convenience, Rule (44.3b) specifies that putc returns the character that it sent to the output.

44.3 Mutable Cells

Here we develop a representation of mutable storage in \( L \{ \text{conc} \} \) in which each reference cell is a process that enacts a protocol for retrieving and altering its contents. The process \( \langle l : e \rangle \), where \( e \) is a value of some type \( \tau \), represents a mutable cell at location \( l \) with contents \( e \) of type \( \tau \). This process is prepared to send the value \( e \) along the channel named \( l \), once again becoming the same process. It is also prepared to receive a value along channel \( l \), which becomes the new contents of the reference cell with location \( l \). Thus we may think of a reference cell as a “server” that emits the current contents of the cell, and that may respond to requests to change its contents.
To model reference cells as processes we extend the grammar of $L\{\text{conc}\}$ to incorporate reference types as described in Chapter 38 and to introduce a new form of process representing a reference cell:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>ref($\tau$)</td>
<td>$\tau$ ref</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>loc[$l$]</td>
<td>$l$</td>
</tr>
<tr>
<td>Comm</td>
<td>$m$</td>
<td>new<a href="$e$"></a></td>
<td>new<a href="$e$"></a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>get($e$)</td>
<td>! $e$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set($e_1$;$e_2$)</td>
<td>$e_1 := e_2$</td>
</tr>
<tr>
<td>Proc</td>
<td>$p$</td>
<td>ref<a href="$e$">$l$</a></td>
<td>⟨$l$ : $e$⟩</td>
</tr>
</tbody>
</table>

The process ⟨$l$ : $e$⟩ represents a mutable cell at location $l$ with contents $e$, where $e$ is a value.

The static semantics of reference cells is essentially as described in Chapter 48, transposed to the setting of $L\{\text{conc}\}$. The typing rule for references is given as follows:

$$\Sigma, l : \tau \text{ref} \vdash e : \tau \quad \Sigma, l : \tau \text{ref} \vdash e \text{ val} \quad \Sigma, l : \tau \text{ref} \vdash \langle l : e \rangle \text{ ok}$$ (44.5)

The process ⟨$l$ : $e$⟩ is well-formed if the assumed type of $l$ is $\tau$ ref, where $e$ is of type $\tau$ under the full set of typing assumptions for locations.

The dynamic semantics of mutable storage is specified in $L\{\text{conc}\}$ by the following rules:\footnote{For the sake of concision we have omitted the evident rules for evaluation of the constituent expressions of the various forms of command.}

$$e \text{ val}$$
$$\{a : \text{ new[] } (e)\} \mapsto \nu\{l : \tau \text{ ref }, \{a : \text{ return } l\} \parallel \langle l : e \rangle\}$$ (44.6a)

$$e \text{ val}$$
$$\{a : l \mapsto \nu\{e \mapsto \{a : \text{ return } e\}$$ (44.6b)

$$e \text{ val}$$
$$\{a : l := e\} \mapsto \{a : \text{ return } e\}$$ (44.6c)

$$e \text{ val}$$
$$\langle l : e \rangle \mapsto \langle l : e \rangle$$ (44.6d)

$$e \text{ val} \quad e' \text{ val}$$
$$\langle l : e \rangle \mapsto \langle l : e' \rangle$$ (44.6e)
Rule (44.6a) gives the semantics of \texttt{new}, which allocates a new location, \( l \), which is returned to the calling process, and spawns a new process consisting of a reference cell at location \( l \) with contents \( e \). Rule (44.6b) specifies that the execution of the process \{\( a : ! l \)\} consists of synchronizing with the reference cell at location \( l \) to obtain its contents, continuing with the value so obtained. Rule (44.6c) specifies that execution of \{\( a : l := e \)\} synchronizes with the reference cell at location \( l \) to specify its new contents, \( e' \). Rules (44.6d) and (44.6e) specify that a reference cell process, \( \langle l : e \rangle \), may interact with other processes via the location \( l \), by either sending the contents, \( e \), of \( l \) to a receiver without changing its state, or receiving its new contents, \( e' \), from a sender, and changing its contents accordingly.

It is instructive to reconsider the proof of type safety for reference cells given in Chapter 38. Whereas in Chapter 38 the execution state for a command, \( m \), has the form \( m \odot \mu \), where \( \mu \) is a memory mapping locations to values, here the execution state for \( m \) is a process that, up to structural congruence, has the form

\[
v(l_1: \tau_1 \text{ref}. \ldots . l_k: \tau_k \text{ref}. \langle l_1 : e_1 \rangle \parallel \ldots \parallel \langle l_k : e_k \rangle \parallel \{ a : m \}) . \quad (44.7)
\]

The memory has been decomposed into a set of active locations, \( l_1, \ldots , l_k \), and a set of processes \( \langle l_1 : e_1 \rangle , \ldots , \langle l_k : e_k \rangle \) governing the active locations.

It will turn out to be an invariant of the dynamic semantics that each active location is governed by exactly one process, but the static semantics of processes given by Rules (44.1) are not sufficient to ensure it. (This is as it should be, because the stated property is special to the semantics of reference cells, and not a general property of all possible uses of the process calculus.) The static semantics is sufficient to ensure that if a process of the form (44.7) is well-formed, then for each \( 1 \leq i \leq j \),

\[
l_1 : \tau_1 \text{ref}, \ldots, l_k : \tau_k \text{ref} \vdash e_i : \tau_i .
\]

As discussed in Chapter 38 this condition is necessary for type preservation, because memories may contain cyclic references.

The static semantics of processes is enough to ensure preservation; all that is required is that the contents of each location be type-consistent with its declared type. The static semantics is not, however, sufficient to ensure progress, for we may have fewer reference cell processes than declared locations, and hence the program may “get stuck” referring to the contents of a location, \( l \), for which there is no process of the form \( \langle l : e \rangle \) with which to interact. One prove that the following property is an invariant of the
dynamic semantics in the sense that if \( p \) satisfies this condition and is well-formed according to Rules (44.1), and \( p \rightarrow q \), then \( q \) also satisfies the same condition:

**Lemma 44.1.** If \( p \equiv v(l: \tau \text{ ref. } p') \) and \( p \rightarrow q \), then \( q \equiv \langle l : c \rangle \parallel q' \) for some process \( q' \) and value \( e \).

For the proof of progress, observe that by inversion of Rules (44.1) and (44.5), if \( p \text{ ok} \), where

\[
p \equiv v(l_1: \tau \text{ ref. } \ldots v(l_k: \tau \text{ ref. } q \parallel \{a : m\})),
\]

where \( l \) occurs in \( m \), then

\[
p \equiv v(l: \tau \text{ ref. } p')
\]

for some \( p' \). This, together with Lemma 44.1, ensures that we may make progress in the case that \( m \) has the form \( ! l \) or \( l := e' \) for some \( e' \).

### 44.4 Futures

The semantics of reference cells given in the preceding section makes use of concurrency to model mutable storage. By relaxing the restriction that the content of a cell be a value, we open up further possibilities for exploiting concurrency. In this section we model the concept of a *future*, a memoized, speculatively executed suspension, in the context of the language \( \mathcal{L}\{\text{conc}\} \).

The syntax of futures is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau )</td>
<td>( \text{fut}(\tau) )</td>
<td>( \tau \text{ fut} )</td>
</tr>
<tr>
<td>Expr</td>
<td>( e )</td>
<td>( \text{loc}[l] )</td>
<td>( l )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{pid}[a] )</td>
<td>( a )</td>
</tr>
<tr>
<td>Comm</td>
<td>( m )</td>
<td>( \text{fut}(e) )</td>
<td>( \text{fut}(e) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{syn}(e) )</td>
<td>( \text{syn}(e) )</td>
</tr>
<tr>
<td>Proc</td>
<td>( p )</td>
<td>( \text{fut}[\text{wait}]<a href="a">l</a> )</td>
<td>( [l : \text{wait}(a)] )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{fut}[\text{done}]<a href="e">l</a> )</td>
<td>( [l : \text{done}(e)] )</td>
</tr>
</tbody>
</table>

Expressions are enriched to include *locations* of futures, and *process identifiers*, or *pid’s*, for synchronization. The command \( \text{fut}(e) \) creates a cell whose value is determined by evaluating \( e \) simultaneously with the calling process. The command \( \text{syn}(e) \) synchronizes with the future determined
by \( e \), returning its value once it is available. A future is represented by a process that may be in one of two states, corresponding to whether the computation of its value is pending or finished. A future in the \textit{wait} state has the form \( \text{fut} [\text{wait}] [l] (a) \), indicating that the value of the future at location \( l \) will be determined by the result of executing the process with pid \( a \). A future in the \textit{done} state has the form \( \text{fut} [\text{done}] [l] (e) \), indicating that the value of the future at location \( l \) is \( e \).

The static semantics of futures consists of the evident typing rules for the commands \( \text{fut}(e) \) and \( \text{syn}(e) \), together with rules for the new forms of process:

\[
\begin{array}{c}
\Sigma \Gamma \vdash e : \tau \\
\Sigma \Gamma \vdash \text{fut}(e) \sim \tau\text{fut}
\end{array}
\]  
(44.8a)

\[
\begin{array}{c}
\Sigma \Gamma \vdash e : \tau\text{fut} \\
\Sigma \Gamma \vdash \text{syn}(e) \sim \tau
\end{array}
\]  
(44.8b)

\[
\begin{array}{c}
\Sigma \Gamma \vdash l : \tau\text{proc} \\
\Sigma \Gamma \vdash l : \tau\text{fut}
\end{array}
\]  
(44.8c)

\[
\begin{array}{c}
\Sigma \Gamma \vdash a : \tau\text{proc} \\
\Sigma \Gamma \vdash a : \tau\text{proc}
\end{array}
\]  
(44.8d)

\[
\begin{array}{c}
\Sigma \vdash l : \tau\text{fut} \\
\Sigma \vdash a : \tau\text{proc}
\end{array}
\]  
(44.8e)

\[
\begin{array}{c}
\Sigma \vdash [l : \text{wait}(a)] \text{ok}
\end{array}
\]  
(44.8f)

The dynamic semantics of futures is specified by the following rules:

\[
\begin{array}{c}
\{a : \text{fut}(e)\} \\
\rightarrow
\end{array}
\]  
(44.9a)

\[
\begin{array}{c}
\nu(l : \tau\text{fut}) \cdot \nu(b : \tau\text{proc}.\{a : \text{return } l\} \parallel [l : \text{wait}(b)] \parallel \{b : \text{return } e\})
\end{array}
\]  
(44.9b)

\[
\begin{array}{c}
\{a : \text{syn}(l)\} \overset{?l(e)}{\rightarrow} \{a : \text{return } e\}
\end{array}
\]  
(44.9c)

\[
\begin{array}{c}
[l : \text{wait}(a)] \overset{?a(e)}{\rightarrow} [l : \text{done}(e)]
\end{array}
\]  
(44.9d)
Rule (44.9a) specifies that a future is created in the wait state pending termination of the process that evaluates its argument. Rule (44.9b) specifies that we may only retrieve the value of a future once it has reached the done state. Rules (44.9c) and (44.9d) specify the behavior of futures. A future changes from the wait to the done state when the process that determines its contents has completed execution. Observe that Rule (44.9c) synchronizes with the process labelled $b$ by waiting for that process to announce its termination with its returned value, as described by Rule (44.2b). A future in the done state repeatedly offers its contents to any process that may wish to synchronize with it.

### 44.5 Fork and Join

The semantics of futures given in Section 44.4 on page 415 may be seen as a combination of the more primitive concepts of forking a new process, synchronizing with, or joining, another process, creating a reference cell to hold the state of the future, and sum types to represent the state of the future (either waiting or done). In this section we will focus on the fork and join primitives that underly the semantics of futures.

The syntax of $\mathcal{L}\{\text{conc}\}$ is extended with the following constructs:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$\text{proc}(\tau)$</td>
<td>$\tau\text{proc}$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$\text{pid}[a]$</td>
<td>$a$</td>
</tr>
<tr>
<td>Comm</td>
<td>$m$</td>
<td>$\text{fork}(m)$</td>
<td>$\text{fork}(m)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{join}(e)$</td>
<td>$\text{join}(e)$</td>
</tr>
</tbody>
</table>

The static semantics is given by the following rules:

$$
\Sigma \Gamma \vdash m \sim \tau \quad \Rightarrow \quad \Sigma \Gamma \vdash \text{fork}(m) \sim \tau\text{proc} \quad (44.10a)
$$

$$
\Sigma \Gamma \vdash e : \tau \text{proc} \quad \Rightarrow \quad \Sigma \Gamma \vdash \text{join}(e) \sim \tau \quad (44.10b)
$$

The dynamic semantics is given by the following rules:

$$
\{ a : \text{fork}(m) \} \leftarrow v(b. \{ a : \text{return} b \} \parallel \{ b : m \}) \quad (44.11a)
$$
Rule (44.11a) creates a new process executing the given command, and returns the pid of the new process to the calling process. Rule (44.11b) synchronizes with the specified process, passing its return value to the caller when it has completed.

### 44.6 Synchronization

When programming with multiple processes it is necessary to take steps to ensure that they interact in a meaningful manner. For example, if two processes have access to a reference cell representing the current balance in a bank account, it is important to ensure that updates by either process are atomic in that they are not compromised by any action of the other process. Suppose that one process is recording accrued interest by increasing the balance by \( r \) \%, and the other is recording a debit of \( n \) dollars. Each proceeds by reading the current balance, performing a simple arithmetic computation, and storing the result back to record the result. However, we must ensure that each operation is performed in its entirety without interference from the other in order to preserve the semantics of the transactions.

To see what can go wrong, suppose that both processes read the balance, \( b \), then each calculate their own version of the new balance, \( b_1 = b + r \times b \) and \( b_2 = b - n \), and then both store their results in some order, say \( b_1 \) followed by \( b_2 \). The resulting balance, \( b_2 \), reflects the debit of \( n \) dollars, but not the interest accrual! If the stores occur in the opposite order, the new balance reflects the interest accrued, but not the debit. In either case the answer is wrong!

The solution is to ensure that a read-and-update operation is completed in its entirety without affecting or being affected by the actions of any other process. One way to achieve this is to use an mvar, which is a reference cell that may, at any time, either hold a value or be empty.\(^2\) Thus an mvar may be in one of two states: full or empty, according to whether or not it holds a value. A process may take the value from a full mvar, thereby rendering it empty, or put a value into an empty mvar, thereby rendering it full with that value. No process may take a value from an empty mvar, nor may a process

\(^2\)The name “mvar” is admittedly cryptic, but is relatively standard. Mvar’s are also known as mailboxes, since their behavior is similar to that of a postal delivery box.
put a value to a full mvar. Any attempt to do so blocks progress until the state of the mvar has been changed by some other process so that it is once again possible to make progress. This simple primitive is sufficient to implement many higher-level constructs such as communication channels, as we shall see shortly.

The syntax of mvar’s is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau )</td>
<td>( \text{mvar}(\tau) )</td>
<td>( \tau \text{mvar} )</td>
</tr>
<tr>
<td>Comm</td>
<td>( m )</td>
<td>( \text{mvar}(e) )</td>
<td>( \text{mvar}(e) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{take}(e) )</td>
<td>( \text{take}(e) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{put}(e_1; e_2) )</td>
<td>( \text{put}(e_1; e_2) )</td>
</tr>
<tr>
<td>Proc</td>
<td>( p )</td>
<td>( \text{mvar}[\text{full}][l](e) )</td>
<td>( [l : \text{full}(e)] )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{mvar}<a href="l">\text{empty}</a> )</td>
<td>( [l : \text{empty}] )</td>
</tr>
</tbody>
</table>

The static semantics for commands is analogous to that for reference cells, and is omitted. The rules governing the two new forms of process are as follows:

\[
\begin{align*}
\Sigma \vdash l : \tau \text{mvar} & \quad \Sigma; \Gamma \vdash e : \tau \\
\Sigma; \Gamma \vdash [l : \text{full}(e)] & \text{ok} \tag{44.12a}
\end{align*}
\]

\[
\begin{align*}
\Sigma \vdash l : \tau \text{mvar} & \quad \Sigma; \Gamma \vdash [l : \text{empty}] & \text{ok} \tag{44.12b}
\end{align*}
\]

The dynamic semantics of mvars is given by the following transition rules:

\[
\begin{align*}
\{a : \text{mvar}(e)\} & \xrightarrow{e \text{ val}} \nu(l : \tau \text{mvar} \cdot \{a : \text{return } l\} \mid [l : \text{full}(e)]) \tag{44.13a}
\end{align*}
\]

\[
\begin{align*}
\{a : \text{take}(l)\} & \xrightarrow{?}(e) \{a : \text{return } e\} \tag{44.13b}
\end{align*}
\]

\[
\begin{align*}
\{a : \text{put}(l; e)\} & \xrightarrow{!}(e) \{a : \text{return } e\} \tag{44.13c}
\end{align*}
\]

\[
\begin{align*}
[l : \text{full}(e)] & \xrightarrow{!}(e) [l : \text{empty}] \tag{44.13d}
\end{align*}
\]

\[
\begin{align*}
[l : \text{empty}] & \xrightarrow{?}(e) [l : \text{full}(e)] \tag{44.13e}
\end{align*}
\]
Rules (44.13d) and (44.13e) enforce the protocol ensuring that only one process at a time may access the contents of an mvar. If a full mvar synchronizes with a \texttt{take} (Rule (44.13b)), then its state changes to empty, precluding further reads of its value. Conversely, if an empty mvar synchronizes with a \texttt{put} (Rule (44.13e)), then its state changes to full with the value specified by the \texttt{put}.

Using mvar’s it is straightforward to implement communication channels over which processes may send and receive values of some specified type, $\tau$. To be specific, a \textit{channel} is just an mvar containing a queue of messages maintained in the order in which they were received. Sending a message on a channel adds (atomically!) a message to the back of the queue associated with that channel, and receiving a message from a channel removes (again, atomically) a message from the front of the queue. We leave a full development of channels as an instructive exercise for the reader.

44.7 Excercises
Part XVII

Modularity
Chapter 45

Separate Compilation and Linking

45.1  Linking and Substitution

45.2  Exercises
Chapter 46

Basic Modules
Chapter 47

Parameterized Modules
Part XVIII

Modalities
Chapter 48

Monads

In this chapter we isolate a crucial idea from Chapter 37, the use of a modality to distinguish pure expressions from impure commands. In Chapter 37 the distinction between pure and impure is based solely on whether assignment to variables is permitted or not. Here we distinguish two modes based on the general concept of a computational effect, of which assignment to variables is but one example. While it is difficult to be precise about what constitutes an effect, a rough-and-ready rule is any behavior that constrains the order of execution beyond that the requirements imposed by the flow of data. For example, since the order in which input or output is performed clearly matters to the meaning of a program, these operations may be classified as effects. Similarly, mutation of data structures (as described in Chapter 38) is clearly sensitive to the order in which they are executed, and so mutation should also be classified as an effect.

The trouble with computational effects is precisely that they constrain the order of evaluation. This inhibits the use parallelism (Chapter 42) or laziness (Chapter 40), and generally makes it harder to reason about the behavior of a program. But it should ideally be possible to take advantage of these concepts when effects are not used, rather than always planning for the possibility that they might be used. We draw a modal distinction between two forms of expression:

1. The pure expressions, or terms, that are executed solely for their value, and that may engender no effects.

2. The impure expressions, or commands, that are executed for their value and their effect.

The mode distinction gives rise to a new form of type, called the lax modal-
The Lax Modality, or monad, whose elements are unevaluated commands. These commands can be passed as pure data, or activated for use by a special form of command.

### 48.1 The Lax Modality

The syntax of $L\{\text{cmd}\}$ is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$\text{cmd}(\tau)$</td>
<td>$\tau$ cmd</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$x$</td>
<td>$x$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{cmd}(m)$</td>
<td>$\text{cmd}(m)$</td>
</tr>
<tr>
<td>Comm</td>
<td>$m$</td>
<td>$\text{return}(e)$</td>
<td>$\text{return}\ e$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{let}\ \text{cmd}(e;x.m)$</td>
<td>$\text{let cmd}(e;\ x.m) \ \text{be}\ e \ \text{in}\ m$</td>
</tr>
</tbody>
</table>

The language $L\{\text{cmd}\}$ distinguishes two modes of expression, the pure (effect-free) expressions, and the impure (effect-capable) commands. The modal type $\text{cmd}(\tau)$ consists of suspended commands that, when evaluated, yield a value of type $\tau$. The expression $\text{cmd}(m)$ introduces an unevaluated command as a value of modal type. The command $\text{return}(e)$ returns the value of $e$ as its value, without engendering any effects. The command $\text{let}\ \text{cmd}(e;x.m)$ activates the suspended command obtained by evaluating the expression $e$, then continues by evaluating the command $m$. This form sequences evaluation of commands so that there is no ambiguity about the order in which effects occur during evaluation.

The static semantics of $L\{\text{cmd}\}$ consists of two forms of typing judgement, $e : \tau$, stating that the expression $e$ has type $\tau$, and $m \sim \tau$, stating that the command $m$ only yields values of type $\tau$. Both of these judgement forms are considered with respect to hypotheses of the form $x : \tau$, which states that a variable $x$ has type $\tau$. The rules defining the static semantics of $L\{\text{cmd}\}$ are as follows:

$$ \Gamma \vdash m \sim \tau \quad \Gamma \vdash \text{cmd}(m) : \text{cmd}(\tau) \quad (48.1a) $$

$$ \Gamma \vdash e : \tau \quad \Gamma \vdash \text{return}(e) \sim \tau \quad (48.1b) $$

$$ \Gamma \vdash e : \text{cmd}(\tau) \quad \Gamma, x : \tau \vdash m \sim \tau' \quad \Gamma \vdash \text{let}\ \text{cmd}(e; x.m) \sim \tau' \quad (48.1c) $$

The dynamic semantics of an instance of $L\{\text{cmd}\}$ is specified by two transition judgements:
48.2 Exceptions

1. **Evaluation of expressions**, \( e \mapsto e' \).

2. **Execution of commands**, \( m \mapsto m' \).

The rules of expression evaluation are carried over from the effect-free setting without change. There is, however, an additional form of value, the encapsulated command:

\[
\text{cmd}(m) \quad \text{val}
\]

(48.2)

Observe that \( \text{cmd}(m) \) is a value regardless of the form of \( m \). This is because the command is not executed, but only encapsulated as a form of value.

The rule of execution enforce the sequential execution of commands.

\[
\frac{e \mapsto e'}{\text{return}(e) \mapsto \text{return}(e')}
\]

(48.3a)

\[
\frac{e \quad \text{val}}{\text{return}(e) \quad \text{final}}
\]

(48.3b)

\[
\frac{e \mapsto e'}{\text{letcmd}(e;x.m) \mapsto \text{letcmd}(e';x.m)}
\]

(48.3c)

\[
\frac{m_1 \mapsto m'_1}{\text{letcmd}(\text{cmd}(m_1);x.m_2) \mapsto \text{letcmd}(\text{cmd}(m'_1);x.m_2)}
\]

(48.3d)

\[
\frac{\text{return}(e) \quad \text{final}}{\text{letcmd}(\text{cmd}(\text{return}(e));x.m) \mapsto [e/x]m}
\]

(48.3e)

Rules (48.3a) and (48.3c) specify that the expression part of a return or let command is to be evaluated before execution can proceed. Rule (48.3b) specifies that a return command whose argument is a value is a final state of command execution. Rule (48.3d) specifies that a letcomp activates an encapsulated command, and Rule (48.3e) specifies that a completed command passes its return value to the body of the let.

### 48.2 Exceptions

What if a command raises an exception? We may think of raising an exception as an alternate form of return from a command. Correspondingly, we may think of an exception handler as an alternate form of monadic bind.
that is sensitive to both the normal and the exceptional return from a command. The language $L\{\text{comm exc}\}$ extends $L\{\text{cmd}\}$ with exceptions in this style. The grammar is as follows:

$$\begin{array}{lll}
\text{Category} & \text{Item} & \text{Abstract} & \text{Concrete} \\
\text{Comm} & m & := & \text{raise}[\tau](e) \rightarrow \text{raise}(e) \\
 & & | & \text{letcomp}(e;x.m_1;y.m_2) \rightarrow \text{let cmd}(x) \text{ be } e \text{ in } m_1 \text{ ow}(y) \text{ in } m_2 \\
\end{array}$$

The command $\text{raise}(e)$ that raises an exception with value $e$, and the command

$$\text{let cmd}(x) \text{ be } e \text{ in } m_1 \text{ ow}(y) \text{ in } m_2$$

generalizes the monadic bind to account for exceptions. Specifically, it executes the encapsulated command specified by the expression $e$. If it returns normally, then the return value is bound to $x$ and the command $m_1$ is executed, much as before. If, instead, execution of the encapsulated command results in an exception, the associated value is bound to $y$ and the command $m_2$ is executed. The monadic bind construct of $L\{\text{cmd}\}$ may be regarded as short-hand for the command

$$\text{let cmd}(x) \text{ be } e \text{ in } m \text{ ow}(y) \text{ in } \text{raise}(y),$$

which propagates any exception that may be raised during execution of the command $e$ specified by $e$.

The static semantics of these constructs is given by the following rules:

$$\Gamma \vdash e : \tau_{\text{exn}} \quad \frac{}{\Gamma \vdash \text{raise}[\tau](e) \sim \tau} \quad (48.4a)$$

$$\begin{array}{llll}
\Gamma \vdash e : \text{cmd}(\tau) & \Gamma, x : \tau \vdash m_1 \sim \tau' & \Gamma, y : \tau_{\text{exn}} \vdash m_2 \sim \tau' \\
\Gamma \vdash \text{letcomp}(e;x.m_1;y.m_2) \sim \tau' & (48.4b) \\
\end{array}$$

The dynamic semantics of these commands consists of a transition system of the form $m \rightarrow m'$ defined by the following rules:

$$\begin{array}{ll}
\frac{e \rightarrow e'}{\text{raise}[\tau](e) \rightarrow \text{raise}[\tau](e')} & (48.5a) \\
\frac{e \rightarrow e'}{\text{letcomp}(e;x.m_1;y.m_2) \rightarrow \text{letcomp}(e';x.m_1;y.m_2)} & (48.5b) \\
\frac{m \rightarrow m'}{\text{letcomp}(\text{cmd}(m);x.m_1;y.m_2) \rightarrow \text{letcomp}(\text{cmd}(m');x.m_1;y.m_2)} & (48.5c) \\
\end{array}$$
48.3 Derived Forms

The bind construct imposes a sequential evaluation order on commands, according to which the encapsulated command is executed prior to execution of the body of the bind. This gives rise to a familiar programming idiom, called sequential composition, which we now derive from the lax modality.

Since there are only two constructs for forming commands, the bind and the return command, it is easy to see that a command of type \( \tau \) always has the form

\[
\text{let cmd(x_1) be \( e_1 \) in \ldots let cmd(x_n) be \( e_n \) in return \( e \)},
\]

where \( e_1 : \tau_1 \text{ cmd}, \ldots, e_n : \tau_n \text{ cmd}, \) and \( x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau \). The dynamic semantics of \( L\{\text{cmd}\} \) specifies that this is evaluated by evaluating the expression, \( e_1 \), to an encapsulated command, \( m_1 \), then executing \( m_1 \) for its value and effects, then passing this value to \( e_2 \), and so forth, until finally the value determined by the expression \( e \) is returned.

To execute \( m_1 \) and \( m_2 \) in sequence, where \( m_2 \) may refer to the value of \( m_1 \) via a variable \( x_1 \), we may write

\[
\text{let cmd(x_1) be \( m_1 \) in \( m_2 \)}.
\]

This encapsulates, and then immediate activates, the command \( m_1 \), binding it value to \( x_1 \), and continuing by executing \( m_2 \). More generally, to execute a sequence of commands in order, passing the value of each to the next, we may write

\[
\text{let cmd(x_1) be \( m_1 \) in \ldots let cmd(x_{k-1}) be \( m_{k-1} \) in \( m_k \)}.
\]

Notationally, this quickly gets out of hand. We therefore introduce the do syntax, which is reminiscent of the notation used in many imperative programming languages. The binary do construct, \( \{x \leftarrow m_1 ; m_2\} \), stands for the command

\[
\text{let cmd(x) be \( m_1 \) in \( m_2 \)}.
\]
which executes the commands $m_1$ and $m_2$ in sequence, passing the value of $m_1$ to $m_2$ via the variable $x$. The general do construct,

$$\text{do}\{x_1 \leftarrow m_1; \ldots; x_k \leftarrow m_k; \text{return } e\},$$

is defined by iteration of the binary do as follows:

$$\text{do}\{x_1 \leftarrow m_1; \ldots \text{do}\{x_k \leftarrow m_k; \text{return } e\}\ldots\}.$$ 

This notation is reminiscent of that used in many well-known programming languages. The point here is that sequential composition of commands arises from the presence of the lax modality in the language. In other words conventional imperative programming languages are implicitly structured by this type, even if the connection is not made explicit.

### 48.4 Monadic Programming

The modal separation of expressions from commands ensures that the semantics of expression evaluation is not compromised by the possibility of effects. One consequence of this restriction is that it is impossible to define an expression $x : \tau_{\text{cmd}} \vdash \text{run } x : \tau$ whose behavior is to unbundle the command bound to $x$, execute it, and return its value as the value of the entire expression. For if such an expression were to exist, expression evaluation would engender effects, ruining the very distinction we are trying to preserve!

The only way for a command to occur inside of an expression is for it to be encapsulated as a value of modal type. To execute such a command it is necessary to bind it to a variable using the bind construct, which is itself a form of command. This is the essential means by which effects are confined to commands, and by which expressions are ensured to remain pure. Put another way, it is impossible to define an expression $\text{run } e$ of type $\tau$, where $e : \tau_{\text{cmd}}$, whose value is the result of running the command encapsulated in the value of $e$. There is, however, a command $\text{run } e$ defined by

$$\text{let cmd}(x) \text{ be } e \text{ in return } x,$$

which executes the encapsulated command and returns its value.

Now consider the extension of $L\{\text{cmd}\}$ with function types. Recall from Chapter 13 that a function has the form $\lambda(x : \tau.e)$, where $e$ is a (pure) expression. In the context of $L\{\text{cmd}\}$ this implies that no function may engender an effect when applied! For example, it is not possible to write a function of the form $\lambda(x : \text{unit.print } "\text{hello}" )$ that, when applied, outputs the string hello to the screen.
This may seem like a serious limitation, but this apparent “bug” is actually an important “feature.” To see why, observe that the type of the foregoing function would, in the absence of the lax modality, be something like \( \text{unit} \rightarrow \text{unit} \). Intuitively, a function of this type is either the identity function, the constant function returning the null tuple (this is, in fact, the identity function), or a function that diverges or incurs an error when applied (in the presence of such possibilities). But, above all, it cannot be the function that prints hello.

However, let us consider the closely related type \( \text{unit} \rightarrow (\text{unit cmd}) \). This is the type of functions that, when applied, yield an encapsulated command, of type unit. One such function is

\[
\lambda (x: \text{unit.cmd}(\text{print "hello"})).
\]

This function does not output to the screen when applied, since no pure function can have an effect, but it does yield a command that, when executed, performs this output. Thus, if \( e \) is the above function, then the command

\[
\text{let cmd(_) be } e(\langle \rangle) \text{ in return } \langle \rangle
\]

executes the encapsulated command yielded by \( e \) when applied, engendering the intended effect, and returning the trivial element of unit type.

The importance of this example lies in the distinction between the type \( \text{unit} \rightarrow \text{unit} \), which can only contain uninteresting functions such as the identity, and the type \( \text{unit} \rightarrow (\text{unit cmd}) \), which reveals in its type that the result of applying it is an encapsulated command that may, when executed, engender an arbitrary effect. In short, the type reveals the reliance on effects. The function type retains its meaning, and, in combination with the lax modality, provides a type of procedures that yield a command when applied. A procedure call is implemented by combining function application with the modal bind operation in the manner illustrated by expression (48.6).

A particular case arises when exceptions are regarded as effects. Doing so has the advantage that a value of type \( \text{nat} \rightarrow \text{nat} \) is, even in the presence of exceptions, a function that, when applied to a natural number, returns a natural number. If a function can raise an exception when called, then it must be given the weaker type \( \text{nat} \rightarrow \text{nat cmd} \), which specifies that, when applied, it yields an encapsulated computation that, when executed, may raise an exception. Two such functions cannot be directly composed, since their types are no longer compatible. Instead we must explicitly sequence
their execution. For example, to compose $f$ and $g$ of this type, we may write

$$\lambda(x:\text{nat}. \text{do}\{y \leftarrow \text{run } g(x); z \leftarrow \text{run } f(y); \text{return } z\}).$$

Here we have used the do syntax introduced in Chapter 48, which according to our conventions above, implicitly propagates exceptions arising from the application of $f$ and $g$ to their surrounding context.

This distinction may be regarded as either a boon or a bane, depending on how important it is to indicate in the type whether a function might raise an exception when called. For programmer-defined exceptions one may wish to draw the distinction, but the situation is less clear for other forms of run-time errors. For example, if division by zero is to be regarded as a form of exception, then the type of division must be

$$\text{nat} \rightarrow \text{nat} \rightarrow \text{nat cmd}$$

to reflect this possibility. But then one cannot then use division in an ordinary arithmetic expression, because its result is not a number, but an encapsulated command. One response to this might be to consider division by zero, and other related faults, not as handle-able exceptions, but rather as fatal errors that abort computation. In that case there is no difference between such an error and divergence: the computation never terminates, and this condition cannot be detected during execution. Consequently, operations such as division may be regarded as partial functions, and may therefore be used freely in expressions without taking special pains to manage any errors that may arise.

### 48.5 Exercises
Chapter 49

Comonads

Monads arise naturally for managing effects that both influence and are influenced by the context in which they arise. This is particularly clear for storage effects, whose context is a memory mapping locations to values. The semantics of the storage primitives makes reference to the memory (to retrieve the contents of a location) and makes changes to the memory (to change the contents of a location or allocate a new location). These operations must be sequentialized in order to be meaningful (that is, the precise order of execution matters), and we cannot expect to escape the context since locations are values that give rise to dependencies on the context. As we shall see in Chapter 44 other forms of effect, such as input/output or interprocess communication, are naturally expressed in the context of a monad.

By contrast the use of monads for exceptions as in Chapter 48 is rather less natural. Raising an exception does not influence the context, but rather imposes the requirement on it that a handler be present to ensure that the command is meaningful even when an exception is raised. One might argue that installing a handler influences the context, but it does so in a nested, or stack-like, manner. A new handler is installed for the duration of execution of a command, and then discarded. The handler does not persist across commands in the same sense that locations persist across commands in the case of the state monad. Moreover, installing a handler may be seen as restoring purity in that it catches any exceptions that may be raised and, assuming that the handler does not itself raise an exception, yields a pure value. A similar situation arises with fluid binding (as described in Chapter 35). A reference to a symbol imposes the demand on the context to provide a binding for it. The binding of a symbol may be changed, but only
for the duration of execution of a command, and not persistently. Moreover, the reliance on symbol bindings within a specified scope confines the impurity to that scope.

The concept of a \textit{comonad} captures the concept of an effect that \textit{imposes a requirement} on its context of execution, but that does not persistently alter that context beyond its execution. Computations that rely on the context to provide some capability may be thought of as impure, but the impurity is confined to the extent of the reliance—outside of this context the computation may be once again regarded as pure. One may say that monads are appropriate for \textit{global}, or \textit{persistent}, effects, whereas comonads are appropriate for \textit{local}, or \textit{ephemeral}, effects.

\section{A Comonadic Framework}

The central concept of the comonadic framework for effects is the \textit{constrained typing judgement}, $e : \tau [\chi]$, which states that an expression $e$ has type $\tau$ (as usual) provided that the context of its evaluation satisfies the constraint $\chi$. The nature of constraints varies from one situation to another, but will include at least the trivially true constraint, $\top$, and the conjunction of constraints, $\chi_1 \land \chi_2$. We sometimes write $e : \tau$ to mean $e : \tau [\top]$, which states that expression $e$ has type $\tau$ under no constraints.

The syntax of the comonadic framework, $\mathcal{L}\{\text{comon}\}$, is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$\boxdot [\chi] (\tau)$</td>
<td>$\boxed{\chi \cdot \tau}$</td>
</tr>
<tr>
<td>Const</td>
<td>$\chi$</td>
<td>$tt$</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{and}(\chi_1; \chi_2)$</td>
<td>$\chi_1 \land \chi_2$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$\text{box}(e)$</td>
<td>$\text{box}(e)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{unbox}(e)$</td>
<td>$\text{unbox}(e)$</td>
</tr>
</tbody>
</table>

A type of the form $\boxed{\chi \cdot \tau}$ is called a \textit{comonad}; it represents the type of unevaluated expressions that impose constraint $\chi$ on their context of execution. The constraint $\top$ is the trivially true constraint, and the constraint $\chi_1 \land \chi_2$ is the conjunction of two constraints. The expression $\text{box}(e)$ is the introduction form for the comonad, and the expression $\text{unbox}(e)$ is the corresponding elimination form.

The judgement $\chi$ true expresses that the constraint $\chi$ is satisfied. This judgement is partially defined by the following rules, which specify the
meanings of the trivially true constraint and the conjunction of constraints.

\[
\begin{align*}
\text{true} & \quad \text{true} \\
\chi_1 \text{ true} & \quad \chi_2 \text{ true} \quad \text{and} (\chi_1;\chi_2) \text{ true} \\
\text{and} (\chi_1;\chi_2) \text{ true} & \quad \chi_1 \text{ true} \\
\text{and} (\chi_1;\chi_2) \text{ true} & \quad \chi_2 \text{ true}
\end{align*}
\]  

(49.1a) (49.1b) (49.1c) (49.1d)

We will make use of hypothetical judgements of the form \( \chi_1 \text{ true}, \ldots, \chi_n \text{ true} \vdash \chi \text{ true} \), where \( n \geq 0 \), expressing that \( \chi \) is derivable from \( \chi_1, \ldots, \chi_n \), as usual.

The static semantics is specified by generic hypothetical judgements of the form

\[
x_1 : \tau_1 [\chi_1], \ldots, x_n : \tau_n [\chi_n] \vdash e : \tau [\chi].
\]

As usual we write \( \Gamma \) for a finite set of hypotheses of the above form.

The static semantics of the core constructs of \( \mathcal{L} \{ \text{comon} \} \) is defined by the following rules:

\[
\frac{\chi' \vdash \chi}{\Gamma, x : \tau [\chi] \vdash x : \tau [\chi']}
\]  

(49.2a)

\[
\frac{\Gamma \vdash e : \tau [\chi]}{\Gamma \vdash \text{box}(e) : \Box [\chi] [\tau [\chi']]} 
\]  

(49.2b)

\[
\frac{\Gamma \vdash e : \Box [\chi] [\tau [\chi']]}{\Gamma \vdash \text{unbox}(e) : \tau [\chi']}
\]  

(49.2c)

Rule (49.2b) states that a boxed computation has comonadic type under an arbitrary constraint. This is valid because a boxed computation is a value, and hence imposes no constraint on its context of evaluation. Rule (49.2c) states that a boxed computation may be activated provided that the ambient constraint, \( \chi' \), is at least as strong as the constraint \( \chi \) of the boxed computation. That is, any requirement imposed by the boxed computation must be met at the point at which it is unboxed.

Rules (49.2) are formulated to ensure that the constraint on a typing judgement may be strengthened arbitrarily.

**Lemma 49.1** (Constraint Strengthening). If \( \Gamma \vdash e : \tau [\chi] \) and \( \chi' \vdash \chi \), then \( \Gamma \vdash e : \tau [\chi'] \).
49.1 A Comonadic Framework

Proof. By rule induction on Rules (49.2).

Intuitively, if a typing holds under a weaker constraint, then it also holds under any stronger constraint as well.

At this level of abstraction the dynamic semantics of $L\{\text{comon}\}$ is trivial.

\[
\begin{align*}
\text{box}(e) & \vdash \text{val} \\
\frac{e \mapsto e'}{\text{unbox}(e) \mapsto \text{unbox}(e')} \\
\frac{\text{unbox}(\text{box}(e)) \mapsto e}
\end{align*}
\]

In specific applications of $L\{\text{comon}\}$ the dynamic semantics will also specify the context of evaluation with respect to which constraints are to be interpreted.

The role of the comonadic type in $L\{\text{comon}\}$ is explained by considering how one might extend the language with, say, function types. The crucial idea is that the comonad isolates the dependence of a computation on its context of evaluation so that such constraints do not affect the other type constructors. For example, here are the rules for function types expressed in the context of $L\{\text{comon}\}$:

\[
\begin{align*}
\Gamma, x : \tau \, \text{tt} \vdash e_2 : \tau \, \text{tt} \\
\Gamma \vdash \text{lam}(\tau; \tau_2) \, \text{arr}(\tau_1; \tau_2) \, [\chi] \\
\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \, [\chi] \\
\Gamma \vdash e_2 : \tau_2 \, [\chi] \\
\Gamma \vdash \text{ap}(e_1; e_2) : \tau \, [\chi]
\end{align*}
\]

These rules are formulated so as to ensure that constraint strengthening remains admissible. Rule (49.4a) states that a $\lambda$-abstraction has type $\tau_1 \rightarrow \tau_2$ under any constraint $\chi$ provided that its body has type $\tau_2$ under the trivially true constraint, assuming that its argument has type $\tau_1$ under the trivially true constraint. By demanding that the body be well-formed under no constraints we are, in effect, insisting that its body be boxed if it is to impose a constraint on the context at the point of application. Under a call-by-value evaluation order, the argument $x$ will always be a value, and hence imposes no constraints on its context.

Let the expression unbox_app$(e_1; e_2)$ be an abbreviation for unbox$(\text{ap}(e_1; e_2))$, which applies $e_1$ to $e_2$, then activates the result. The derived static semantics...
for this construct is given by the following rule:

\[
\Gamma \vdash e_1 : \tau_1 \rightarrow \Box_{\chi} \tau [\chi'] \quad \Gamma \vdash e_2 : \tau_2 [\chi'] \quad \chi' \vdash \chi
\]

\[
\Gamma \vdash \text{unbox_app}(e_1; e_2) : \tau [\chi']
\]

(49.5)

In words, to apply a function with impure body to an argument, the ambient constraint must be strong enough to type the function and its argument, and must be at least as strong as the requirements imposed by the body of the function. We may view a type of the form \(\tau_1 \rightarrow \Box_{\chi} \tau_2\) as the type of functions that, when applied to a value of type \(\tau_1\), yield a value of type \(\tau_2\) engendering local effects with requirements specified by \(\chi\).

Similar principles govern the extension of \(L\{\text{comon}\}\) with other types such as products or sums.

### 49.2 Comonadic Effects

In this section we discuss two applications of \(L\{\text{comon}\}\) to managing local effects. The first application is to exceptions, using constraints to specify whether or not an exception handler must be installed to evaluate an expression so as to avoid an uncaught exception error. The second is to fluid binding, using constraints to specify which symbols must be bound during execution so as to avoid accessing an unbound symbol. The first may be considered to be an instance of the second, in which we think of the exception handler as a distinguished symbol whose binding is the current exception continuation.

#### 49.2.1 Exceptions

To model exceptions we extend \(L\{\text{comon}\}\) as follows:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const (\chi)</td>
<td>::= (\uparrow)</td>
<td>(\uparrow)</td>
<td></td>
</tr>
<tr>
<td>Expr (e)</td>
<td>::= (\text{raise}<a href="e">\tau</a>)</td>
<td>(\text{raise}(e))</td>
<td></td>
</tr>
<tr>
<td>\null</td>
<td>\null</td>
<td>(\text{handle}(e_1; x. e_2))</td>
<td>(\text{try } e_1 \text{ ow } x \Rightarrow e_2)</td>
</tr>
</tbody>
</table>

The constraint \(\uparrow\) specifies that an expression may raise an exception, and hence that its context is required to provide a handler for it.

The static semantics of \(L\{\text{comon}\}\) is extended with the following rules:

\[
\Gamma \vdash e : \tau_{\text{exc}} [\chi] \quad \chi \vdash \uparrow
\]

\[
\Gamma \vdash \text{raise}[\tau](e) : \tau[\chi]
\]

(49.6a)
\[
\Gamma \vdash e_1 : \tau [\chi \land \top] \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau [\chi]
\]
\[
\Gamma \vdash \text{handle}(e_1; x . e_2) : \tau [\chi]
\]
(49.6b)

Rule (49.6a) imposes the requirement for a handler on the context of a \texttt{raise} expression, in addition to any other conditions that may be imposed by its argument. (The rule is formulated so as to ensure that constraint strengthening remains admissible.) Rule (49.6b) transforms an expression that requires a handler into one that may or may not require one, according to the demands of the handling expression. If \(e_2\) does not demand a handler, then \(\chi\) may be taken to be the trivial constraint, in which case the overall expression is pure, even though \(e_1\) is impure (may raise an exception).

The dynamic semantics of exceptions is as given in Chapter 28. The interesting question is to explore the additional assurances given by the comonadic type system given by Rules (49.6). Intuitively, we may think of a stack as a constraint transformer that turns a constraint \(\chi\) into a constraint \(\chi'\) by composing frames, including handler frames. Then if \(e\) is an expression of type \(\tau\) imposing constraint \(\chi\) and \(k\) is a \(\tau\)-accepting stack transforming constraint \(\chi\) into constraint \(\top\), then evaluation of \(e\) on \(k\) cannot yield an uncaught exception. In this sense the constraints reflect the reality of the execution behavior of expressions.

To make this precise, we define the judgement \(k : \tau [\chi]\) to mean that \(k\) is a stack that is suitable as an execution context for an expression \(e : \tau [\chi]\). The typing rules for stacks are as follows:

\[
\epsilon : \tau [\top]
\]
(49.7a)

\[
k : \tau' [\chi'] 
\quad f : \tau [\chi] \Rightarrow \tau' [\chi']
\]
\[
k; f : \tau [\chi]
\]
(49.7b)

Rule (49.7a) states that the empty stack must not impose any constraints on its context, which is to say that there must be no uncaught exceptions at the end of execution. Rule (49.7b) simply specifies that a stack is a composition of frames. The typing rules for frames are easily derived from the static semantics of \(L\{\text{comon}\}\). For example,

\[
x : \tau_{exn} \vdash e : \tau [\chi]
\]
\[
\text{handle}(\neg; x . e) : \tau [\chi \land \top] \Rightarrow \tau [\chi]
\]
(49.8)

This rule states that a handler frame transforms an expression of type \(\tau\) demanding a handler into an expression of type \(\tau\) that may, or may not, demand a handler, according to the form of the handling expression.
The formation of states is defined essentially as in Chapter 27.

\[ k : \tau [\chi], e : \tau [\chi] \quad k \triangleright e \quad \text{ok} \quad (49.9a) \]

\[ k : \tau [\chi], e : \tau [\chi] \quad e \quad \text{val} \quad k \triangleleft e \quad \text{ok} \quad (49.9b) \]

Observe that a state of the form \( e \triangleright \text{raise}(e) \), where \( e \text{ val} \), is ill-formed, because the empty stack is well-formed only under no constraints on the context.

Safety ensures that no uncaught exceptions can arise. This is expressed by defining final states to be only those returning a value to the empty stack.

\[ e \text{ val} \quad \frac{}{e \triangleleft e \quad \text{final}} \quad (49.10) \]

In contrast to Chapter 28, we do not consider an uncaught exception state to be final!

**Theorem 49.2 (Safety).**

1. If \( s \text{ ok} \) and \( s \mapsto s' \), then \( s' \text{ ok} \).

2. If \( s \text{ ok} \) then either \( s \text{ final} \) or there exists \( s' \) such that \( s \mapsto s' \).

**Proof.** These are proved by rule induction on the dynamic semantics and on the static semantics, respectively, proceeding along standard lines. \( \square \)

### 49.2.2 Fluid Binding

Using comonads we may devise a type system for fluid binding that ensures that no unbound symbols are accessed during execution. This is achieved by regarding the mapping of symbols to their values to be the context of execution, and introducing a form of constraint stating that a specified symbol must be bound in the context.

Let us consider a comonadic static semantics for \( L\{\text{fluid}\} \) defined in Chapter 35. For this purpose we consider atomic constraints of the form \( \text{bd}(a) \), stating that the symbol \( a \) has a binding.

The static semantics of fluid binding consists of judgements of the form \( \Gamma \vdash \Sigma e : \tau [\chi] \), where \( \Sigma \) assigns types to the fluid-bound symbols.

\[ \chi \vdash \text{bd}(a) \quad \frac{}{\Gamma \vdash \Sigma, e : \tau [\chi]} \quad (49.11a) \]
\[
\frac{
\Gamma \vdash_{\Sigma;\tau} e_1 : \tau[\chi]\quad \Gamma \vdash_{\Sigma;\tau} e_2 : \tau[\chi \land \text{bd}(a)]
}{
\Gamma \vdash_{\Sigma;\tau} \text{put}[a](e_1; e_2) : \tau[\chi]
}\quad (49.11b)
\]

Rule (49.11a) records the demand for a binding for the symbol \(a\) incurred by retrieving its value. Rule (49.11b) propagates the fact that the symbol \(a\) is bound to the body of the fluid binding.

The dynamic semantics is as specified in Chapter 35. The safety theorem for the comonadic type system for fluid binding states that no unbound symbol error may ever arise during execution. We define the judgement \(\theta \models \chi\) to mean that \(a \in \text{dom}(\theta)\) whenever \(\chi \vdash \text{bd}(a)\).

**Theorem 49.3** (Safety).

1. If \(\vdash_{\Sigma} e : \tau[\chi]\) and \(e \overset{\theta}{\mapsto} e'\), then \(\vdash_{\Sigma} e' : \theta[\chi]\).

2. If \(\vdash_{\Sigma} e : \tau[\chi]\) and \(\theta \models \chi\), then either \(e \text{ val}\) or there exists \(e'\) such that \(e \overset{\theta}{\mapsto} e'\).

The comonadic static semantics may be extended to account for dynamic symbol generation. The main difficulty is to manage the interaction between the scopes of symbols and their occurrences in types. First, it is straightforward to define the judgement \(\Sigma \vdash \chi\text{ constr}\) to mean that \(\chi\) is a constraint involving only those symbols \(a\) such that \(\Sigma \vdash a : \tau\) for some \(\tau\). Using this we may also define the judgement \(\Sigma \vdash \tau\text{ type}\) analogously. This judgement is used to impose a restriction on symbol generation to ensure that symbols do not escape their scope:

\[
\frac{
\Gamma \vdash_{\Sigma;\tau} e : \tau\quad \Sigma \vdash \tau\text{ type}
}{
\Gamma \vdash_{\Sigma} \text{new}[\sigma](a.e) : \tau}
\quad (49.12)
\]

This imposes the requirement that the result type of a computation involving a dynamically generated symbol must not mention that symbol. Otherwise the type \(\tau\) would involve a symbol that makes no sense with respect to the ambient symbol context, \(\Sigma\).

For example, an expression such as

\[
\text{new } a : \text{nat in put } a \text{ is } z \text{ in } \lambda (x : \text{nat. box}(\ldots \text{get } a \ldots))
\]

is ill-typed. The type of the \(\lambda\)-abstraction must be of the form \(\text{nat} \to \square_{\chi} \tau\), where \(\chi \vdash \text{bd}(a)\), reflecting the dependence of the body of the function on the binding of \(a\). This type is propagated through the fluid binding for \(a\), since it holds only for the duration of evaluation of the \(\lambda\)-abstraction itself, which is immediately returned as its value. Since the type of the \(\lambda\)-abstraction involves the symbol \(a\), the second premise of Rule (49.12) is not
met, and the expression is ill-typed. This is as it should be, for we cannot guarantee that the dynamically generated symbol replacing \( a \) during evaluation will, in fact, be bound when the body of the function is executed.

However, if we move the binding for \( a \) into the scope of the \( \lambda \)-abstraction,

\[
\text{new } a : \text{nat in } \lambda (x : \text{nat}. \text{box} (\text{put } a \text{ is } z \text{ in } \ldots \text{get } a \ldots)),
\]

then the type of the \( \lambda \)-abstraction may have the form \( \text{nat} \rightarrow \Box^\chi \tau \), where \( \chi \) need not constrain \( a \) to be bound. The reason is that the fluid binding for \( a \) discharges the obligation to bind \( a \) within the body of the function. Consequently, the condition on Rule (49.12) is met, and the expression is well-typed. Indeed, each evaluation of the body of the \( \lambda \)-abstraction initializes the fresh copy of \( a \) generated during evaluation, so no unbound symbol error can arise during execution.

### 49.3 Exercises
Part XIX

Equivalence
Chapter 50

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 $L\{\text{nat} \rightarrow \}$ of Chapter 14 in which we can express addition as the function $\text{plus}$, the expressions

$$\lambda(x:\text{nat}. \lambda(y:\text{nat}. \text{plus}(x)(y)))$$

and

$$\lambda(x:\text{nat}. \lambda(y:\text{nat}. \text{plus}(y)(x)))$$

are equal. In other words, the addition function as programmed in $L\{\text{nat} \rightarrow \}$ 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 $L\{\text{nat} \rightarrow\}$.

## 50.1 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 $\lambda$-abstraction, then very few expressions would come out as equivalent. But such considerations seem silly, because they conflict with the intuition that the significance of an expression lies in its contribution to the outcome of a computation, and not to the process of obtaining that outcome. In short, if two expressions make the same contribution to the outcome of a complete program, then they ought to be regarded as equal.

We must fix what we mean by a complete program. Two considerations inform the definition. First, the dynamic semantics of $L\{\text{nat} \rightarrow\}$ 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 $\text{nat}$, 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 $\text{nat}$—
that is, it is a complete program with a hole in it. The meta-variable $C$ stands for any expression context.

Replacement is the process of filling a hole in an expression context, $C$, with an expression, $e$, which is written $C\{e\}$. Importantly, the free variables of $e$ that are exposed by $C$ are captured by replacement (which is why replacement is not a form of substitution, which is defined so as to avoid capture). If $C$ is a program context, then $C\{e\}$ is a complete program iff all free variables of $e$ are captured by the replacement. For example, if $C = \lambda (x{:}\text{nat}. \circ)$, and $e = x + x$, then

$$C\{e\} = \lambda (x{:}\text{nat}. x + x).$$

The free occurrences of $x$ in $e$ are captured by the $\lambda$-abstraction as a result of the replacement of the hole in $C$ by $e$.

We sometimes write $C\{\circ\}$ to emphasize the occurrence of the hole in $C$. Expression contexts are closed under composition in that if $C_1$ and $C_2$ are expression contexts, then so is $C\{\circ\}$: $C\{\circ\} = C_1\{C_2\{\circ\}\}$, and we have $C\{e\} = C_1\{C_2\{e\}\}$. The trivial, or identity, expression context is the “bare hole”, written $\circ$, for which $\circ\{e\} = e$.

The static semantics of expressions of $L\{\text{nat} \to\}$ is extended to expression contexts by defining the typing judgement $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ so that if $\Gamma \vdash e : \tau$, then $\Gamma' \vdash C\{e\} : \tau'$. This judgement may be inductively defined by a collection of rules derived from the static semantics of $L\{\text{nat} \to\}$ (for which see Rules (14.1)). Some representative rules are as follows:

\begin{align*}
\circ : (\Gamma \triangleright \tau) & \rightsquigarrow (\Gamma \triangleright \tau) \\
C : (\Gamma \triangleright \tau) & \rightsquigarrow (\Gamma' \triangleright \tau) \\
\text{s}(C) : (\Gamma \triangleright \tau) & \rightsquigarrow (\Gamma' \triangleright \text{nat}) \\
C : (\Gamma \triangleright \tau) & \rightsquigarrow (\Gamma' \triangleright \text{nat}) \\
\text{natrec} C \{z \Rightarrow e_0 | s(x) \text{ with } y \Rightarrow e_1\} & : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau') \\
\text{natrec} e \{z \Rightarrow C_0 | s(x) \text{ with } y \Rightarrow e_1\} & : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')
\end{align*}
Observational Equivalence

Γ′ ⊢ e : τ′  C_1 : (Γ' ∪ τ) ∼ (Γ', x : nat, y : τ' ∪ τ')
Γ ⊢ e₀ : τ  C_1(Γ ∪ x : nat, y : τ': τ') (50.1e)

Γ' ⊢ e : τ  Γ' ⊢ e₀ : τ  C_1 : (Γ ∪ τ) ∼ (Γ' ∪ τ')  Natrec_e {z ⇒ e₀ | s(x) with y ⇒ C_1} : (Γ' ∪ τ) ∼ (Γ' ∪ τ') (50.1f)

Γ' ⊢ e : τ  C_1 : (Γ ∪ τ) ∼ (Γ' ∪ τ')  Γ' ⊢ e₀ : τ  Γ' ⊢ e₀ : τ  C_1(e₀) : (Γ ∪ τ) ∼ (Γ' ∪ τ') (50.1g)

Γ' ⊢ e : τ  C_2 : (Γ ∪ τ) ∼ (Γ' ∪ τ')  Γ' ⊢ e : τ  C_2 : (Γ ∪ τ) ∼ (Γ' ∪ τ') (50.1h)

Lemma 50.1. If C : (Γ ∪ τ) ∼ (Γ' ∪ τ'), then Γ' ⊆ Γ, and if Γ ⊢ e : τ, then Γ' ⊢ C{e} : τ'.

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 50.2. If C : (Γ ∪ τ) ∼ (Γ' ∪ τ'), and C' : (Γ' ∪ τ₂) ∼ (Γ'' ∪ τ''), then C'{C{∅}} : (Γ ∪ τ) ∼ (Γ'' ∪ τ'').

Lemma 50.3. If C : (Γ ∪ τ) ∼ (Γ' ∪ τ') and x ∉ dom(Γ), then C : (Γ, x : σ ∪ τ) ∼ (Γ', x : σ ∪ τ').

Proof. By induction on Rules (50.1).

A complete program is a closed expression of type nat.

Definition 50.1. We say that two complete programs, e and e', are Kleene equivalent, written e ∼ e', iff there exists n ≥ 0 such that e →* π and e' →* π.

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 0 ≠ 1.

Definition 50.2. Suppose that Γ ⊢ e : τ and Γ ⊢ e' : τ are two expressions of the same type. We say that e and e' are observationally equivalent, written e ≡ e' : τ [Γ], iff C{e} ≃ C{e'} for every program context C : (Γ ∪ τ) ∼ (∅ ∪ 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,

$$\text{if } e \mathcal{E} e' : \tau \ [\Gamma], \text{ then } C\{e\} \mathcal{E} C\{e'\} : \tau' \ [\Gamma']$$

for every expression context $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$. Such a family of relations is consistent iff $e \mathcal{E} e' : \text{nat} \ [\emptyset]$ implies $e \simeq e'$.

**Theorem 50.4.** 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 $C : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \text{nat})$, then by congruence $C\{e\} \mathcal{E} C\{e'\} : \text{nat} \ [\emptyset]$, and hence by consistency $C\{e\} \simeq C\{e'\}$.

A closing substitution, $\gamma$, for the typing context $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ is a finite function assigning closed expressions $e_1 : \tau_1, \ldots, e_n : \tau_n$ to $x_1, \ldots, x_n$, respectively. We write $\hat{\gamma}(e)$ for the substitution $[e_1, \ldots, e_n/x_1, \ldots, x_n]e$, and write $\gamma : \Gamma$ to mean that if $x : \tau$ occurs in $\Gamma$, then there exists a closed expression, $e$, such that $\gamma(x) = e$ and $e : \tau$. We write $\gamma \equiv \gamma' : \Gamma$, where $\gamma : \Gamma$ and $\gamma' : \Gamma$, to express that $\gamma(x) \equiv \gamma'(x) : \Gamma(x)$ for each $x$ declared in $\Gamma$.

**Lemma 50.5.** If $e \equiv e' : \tau \ [\Gamma]$ and $\gamma : \Gamma$, then $\hat{\gamma}(e) \equiv \hat{\gamma}(e') : \tau$. Moreover, if $\gamma \equiv \gamma' : \Gamma$, then $\hat{\gamma}(e) \equiv \hat{\gamma}'(e') : \tau$.

**Proof.** Let $C : (\emptyset \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \text{nat})$ be a program context; we are to show that $C\{\hat{\gamma}(e)\} \simeq C\{\hat{\gamma}(e')\}$. Since $C$ has no free variables, this is equivalent to showing that $\hat{\gamma}(C\{e\}) \simeq \hat{\gamma}(C\{e'\})$. Let $D$ be the context

$$\lambda (x_1 : \tau_1 \ldots \lambda (x_n : \tau_n, C\{\circ\})) (e_1) \ldots (e_n),$$

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 50.3 on the facing page we have $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma \triangleright \text{nat})$, from which it follows.
directly that $D : (\Gamma \triangleright \tau) \rightsimeq (\emptyset \triangleright \text{nat})$. Since $e \equiv e' : \tau \mid \Gamma$, we have $D \{e\} \simeq D \{e'\}$. But by construction $D \{e\} \simeq \hat{\gamma}(C \{e\})$, and $D \{e'\} \simeq \hat{\gamma}(C \{e'\})$, so $\hat{\gamma}(C \{e\}) \simeq \hat{\gamma}(C \{e'\})$. Since $C$ is arbitrary, it follows that $\hat{\gamma}(e) \equiv \hat{\gamma}(e') : \tau$.

Defining $D'$ similarly to $D$, but based on $\gamma'$, rather than $\gamma$, we may also show that $D' \{e\} \simeq D' \{e'\}$, and hence $\hat{\gamma}'(e) \equiv \hat{\gamma}'(e') : \tau$. Now if $\gamma \equiv \gamma' : \Gamma$, then by congruence we have $D \{e\} \equiv D' \{e\} : \text{nat}$, and $D \{e'\} \equiv D' \{e'\} : \text{nat}$. It follows that $D \{e\} \equiv D' \{e\} : \text{nat}$, and so, by consistency of observational equivalence, we have $D \{e\} \simeq D' \{e\}$, which is to say that $\hat{\gamma}(e) \equiv \hat{\gamma}'(e') : \tau$.

Theorem 50.4 on the previous page licenses the principle of proof by coinduction: to show that $e \equiv e' : \tau \mid \Gamma$, it is enough to exhibit a consistent congruence, $E$, such that $e \, E \, e' : \tau \mid \Gamma$. 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.

### 50.2 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 $\tau$ to a function that merely returns it. The active uses are those that operate on the expression itself; these are the elimination forms associated with the type of that expression. For the purposes of distinguishing two expressions, it is only the active uses that matter; the passive uses merely manipulate expressions at arm’s length, affording no opportunities to distinguish one from another.

This leads to the definition of extensional equivalence alluded to in the introduction.

**Definition 50.3.** Extensional equivalence is a family of relations $e \sim e' : \tau$ between closed expressions of type $\tau$. It is defined by induction on $\tau$ as follows:

- $e \sim e' : \text{nat}$ \quad iff \quad $e \equiv e'$
- $e \sim e' : \tau_1 \rightarrow \tau_2$ \quad iff \quad if $e_1 \sim e'_1 : \tau_1$, then $e \, (e_1) \sim e' \, (e'_1) : \tau_2$

The definition of extensional equivalence at type nat licenses the following principle of proof by nat-induction. To show that $E \ (e, e')$ whenever $e \sim e' : \text{nat}$, it is enough to show that
50.3 Extensional and Observational Equivalence coincide

In this section we prove the coincidence of observational and extensional equivalence.

**Lemma 50.6** (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 $\tau$. If $\tau = \text{nat}$, 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. \qed

**Lemma 50.7** (Consistency). If $e \sim e' : \text{nat}$, then $e \simeq e'$.

**Proof.** By nat-induction (without appeal to the inductive hypothesis). If $e \mapsto^* z$ and $e' \mapsto^* z$, then $e \simeq e'$; if $e \mapsto^* s(d)$ and $e' \mapsto^* s(d')$ then $e \simeq e'$. \qed

**Theorem 50.8** (Reflexivity). If $\Gamma \vdash e : \tau$, then $e \sim e : \tau [\Gamma]$. 

October 16, 2009 DRAFT 18:42
Proof. We are to show that if $\Gamma \vdash e : \tau$ and $\gamma \sim \gamma' : \Gamma$, then $\hat{\gamma}(e) \sim \hat{\gamma'}(e') : \tau$. The proof proceeds by induction on typing derivations; we consider a few representative cases.

Consider the case of Rule (13.4a), in which $\tau = \tau_1 \rightarrow \tau_2$, $e = \lambda(x : \tau_1. e_2)$ and $e' = \lambda(x : \tau_1. e_2')$. Since $e$ and $e'$ are values, we are to show that

$$\lambda(x : \tau_1. \hat{\gamma}(e_2)) \sim \lambda(x : \tau_1. \hat{\gamma'}(e_2')) : \tau_1 \rightarrow \tau_2.$$

Assume that $e_1 \sim e_1' : \tau_1$; we are to show that $[e_1/x]\hat{\gamma}(e_2) \sim [e_1'/x]\hat{\gamma'}(e_2') : \tau_2$. Let $\gamma_2 = \gamma|x \mapsto e_1$ and $\gamma_2' = \gamma'|x \mapsto e_1'$, and observe that $\gamma_2 \sim \gamma_2'$.

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 (14.1d), for which we are to show that

$$\text{natrec}(\hat{\gamma}(e); \hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1)) \sim \text{natrec}(\hat{\gamma'}(e'); \hat{\gamma}(e_0'); x.y.\hat{\gamma'}(e_1')) : \tau.$$

By the induction hypothesis applied to the first premise of Rule (14.1d), we have

$$\hat{\gamma}(e) \sim \hat{\gamma'}(e') : \text{nat}.$$

We proceed by nat-induction. It suffices to show that

$$\text{natrec}(z; \hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1)) \sim \text{natrec}(z; \hat{\gamma}(e_0'); x.y.\hat{\gamma'}(e_1')) : \tau,$$  \hspace{1cm} (50.2)

and that

$$\text{natrec}(s(\overline{n}); \hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1)) \sim \text{natrec}(s(\overline{n}); \hat{\gamma}(e_0'); x.y.\hat{\gamma'}(e_1')) : \tau,$$  \hspace{1cm} (50.3)

assuming

$$\text{natrec}(\overline{n}; \hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1)) \sim \text{natrec}(\overline{n}; \hat{\gamma}(e_0'); x.y.\hat{\gamma'}(e_1')) : \tau.$$  \hspace{1cm} (50.4)

To show (50.2), by Lemma 50.6 on the preceding page it is enough to show that $\hat{\gamma}(e_0) \sim \hat{\gamma}(e_0') : \tau$. This is assured by the outer inductive hypothesis applied to the second premise of Rule (14.1d).

To show (50.3), define

$$\delta = \gamma|x \mapsto \overline{n}|y \mapsto \text{natrec}(\overline{n}; \hat{\gamma}(e_0); x.y.\hat{\gamma}(e_1))$$

and

$$\delta' = \gamma'|x \mapsto \overline{n}|y \mapsto \text{natrec}(\overline{n}; \hat{\gamma}(e_0'); x.y.\hat{\gamma'}(e_1')).$$

By (50.4) we have $\delta \sim \delta' : \Gamma, x : \text{nat}, y : \tau$. Consequently, by the outer inductive hypothesis applied to the third premise of Rule (14.1d), and Lemma 50.6 on the previous page, the required follows. $\Box$

18:42 Draft October 16, 2009
Corollary 50.9 (Termination). If $e : \tau$, then there exists $e' \text{ val}$ such that $e \rightarrow^* e'$.

Symmetry and transitivity of extensional equivalence are easily established by induction on types; extensional equivalence is therefore an equivalence relation.

Lemma 50.10 (Congruence). If $C_0 : (\Gamma \triangleright \tau) \leadsto (\Gamma_0 \triangleright \tau_0)$, and $e \sim e' : \tau [\Gamma]$, then $C_0\{e\} \sim C_0\{e'\} : \tau_0 [\Gamma_0]$.

Proof. By induction on the derivation of the typing of $C_0$. We consider a representative case in which $C_0 = \lambda (x : \tau_1. C_2)$ so that $C_0 : (\Gamma \triangleright \tau) \leadsto (\Gamma_0 \triangleright \tau_1 \rightarrow \tau_2)$ and $C_2 : (\Gamma \triangleright \tau) \leadsto (\Gamma_0, x : \tau_1 \triangleright \tau_2)$. Assuming $e \sim e' : \tau [\Gamma]$, we are to show that

$$C_0\{e\} \sim C_0\{e'\} : \tau_1 \rightarrow \tau_2 [\Gamma_0],$$

which is to say

$$\lambda (x : \tau_1. C_2\{e\}) \sim \lambda (x : \tau_1. C_2\{e'\}) : \tau_1 \rightarrow \tau_2 [\Gamma_0].$$

We know, by induction, that

$$C_2\{e\} \sim C_2\{e'\} : \tau_2 [\Gamma_0, x : \tau_1].$$

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 50.3 on page 456 it is enough to show that

$$\hat{\gamma}_1(C_2\{e\}) \sim \hat{\gamma}'_1(C_2\{e'\}) : \tau_2,$$

which follows immediately from the inductive hypothesis.

\[\square\]

Theorem 50.11. If $e \sim e' : \tau [\Gamma]$, then $e \equiv e' : \tau [\Gamma]$.

Proof. By Lemmas 50.7 on page 457 and 50.10, and Theorem 50.4 on page 455.

\[\square\]

Corollary 50.12. If $e : \text{nat}$, then $e \equiv n : \text{nat}$, for some $n \geq 0$.

Proof. By Theorem 50.8 on page 457 we have $e \sim e : \tau$. Hence for some $n \geq 0$, we have $e \sim n : \text{nat}$, and so by Theorem 50.11, $e \equiv n : \text{nat}$. 

\[\square\]
Lemma 50.13. For closed expressions $e : \tau$ and $e' : \tau$, if $e \equiv e' : \tau$, then $e \sim e' : \tau$.

Proof. We proceed by induction on the structure of $\tau$. If $\tau = \text{nat}$, consider the empty context to obtain $e \simeq e'$, and hence $e \sim e' : \text{nat}$. 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 50.11 on the preceding page we have $e_1 \equiv e'_1 : \tau_1$, and hence by congruence of observational equivalence it follows that $e(e_1) \equiv e'(e'_1) : \tau_2$, from which the result follows by induction.

Theorem 50.14. If $e \equiv e' : \Gamma$, then $e \sim e' : \Gamma$.

Proof. Assume that $e \equiv e' : \Gamma$, and that $\gamma \sim \gamma' : \Gamma$. By Theorem 50.11 on the previous page we have $\hat{\gamma} \equiv \hat{\gamma}' : \Gamma$, so by Lemma 50.5 on page 455 $\hat{\gamma}(e) \equiv \hat{\gamma}'(e') : \tau$. Therefore, by Lemma 50.13, $\hat{\gamma}(e) \sim \hat{\gamma}(e') : \tau$.

Corollary 50.15. $e \equiv e' : \tau [\Gamma]$ iff $e \sim e' : \tau [\Gamma]$.

Theorem 50.16. If $\Gamma \vdash e \equiv e' : \tau$, then $e \sim e' : \tau [\Gamma]$, and hence $e \equiv e' : \tau [\Gamma]$.

Proof. By an argument similar to that used in the proof of Theorem 50.8 on page 457 and Lemma 50.10 on the previous page, then appealing to Theorem 50.11 on the preceding page.

Corollary 50.17. If $e \equiv e' : \text{nat}$, then there exists $n \geq 0$ such that $e \rightarrow^* n$ and $e' \rightarrow^* n$.

Proof. By Theorem 50.16 we have $e \sim e' : \text{nat}$, and hence $e \simeq e'$.

50.4 Some Laws of Equivalence

In this section we summarize some useful principles of observational equivalence for $L \{ \text{nat} \rightarrow \}$. For the most part these may be proved as laws of extensional equivalence, and then transferred to observational equivalence by appeal to Corollary 50.15. 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.
50.4 Some Laws of Equivalence

50.4.1 General Laws

Extensional equivalence is indeed an equivalence relation: it is reflexive, symmetric, and transitive.

\[
\begin{align*}
e & \cong e : \tau [\Gamma] \\
e' & \cong e : \tau [\Gamma] \\
e & \cong e' : \tau [\Gamma] \\
\end{align*}
\]

(50.5a)

Reflexivity is an instance of a more general principle, that all definitional equivalences are observational equivalences.

\[
\begin{align*}
\Gamma \vdash e \equiv e' : \tau \\
e & \cong e' : \tau [\Gamma] \\
\end{align*}
\]

(50.6a)

This is called the principle of symbolic evaluation.

Observational equivalence is a congruence: we may replace equals by equals anywhere in an expression.

\[
\begin{align*}
e & \cong e' : \tau [\Gamma] \quad C : (\Gamma \triangleright \tau) \leadsto (\Gamma' \triangleright \tau') \\
C\{e\} \cong C\{e'\} : \tau' [\Gamma'] \\
\end{align*}
\]

(50.7a)

Equivalence is stable under substitution for free variables, and substituting equivalent expressions in an expression gives equivalent results.

\[
\begin{align*}
\Gamma \vdash e : \tau \\
\Gamma \vdash e_2 \equiv e'_2 : \tau' [\Gamma, x : \tau] \\
[e/x]e_2 \equiv [e/x]e'_2 : \tau' [\Gamma] \\
\end{align*}
\]

(50.8a)

50.4.2 Extensionality Laws

Two functions are equivalent if they are equivalent on all arguments.

\[
\begin{align*}
e(x) & \cong e'(x) : \tau_2 [\Gamma, x : \tau_1] \\
e & \cong e' : \tau_1 \rightarrow \tau_2 [\Gamma] \\
\end{align*}
\]

(50.9)

Consequently, every expression of function type is equivalent to a λ-abstraction:

\[
e \cong \lambda(x : \tau_1. e(x)) : \tau_1 \rightarrow \tau_2 [\Gamma]
\]

(50.10)
50.4.3 Induction Law

An equation involving a free variable, $x$, of type $\text{nat}$ can be proved by induction on $x$.

$$\frac{\left[\pi/x\right]e \equiv \left[\pi/x\right]e' : \tau \, [\Gamma] \, (\text{for every } n \in \mathbb{N})}{e \equiv e' : \tau \, [\Gamma, x : \text{nat}]} \quad (50.11a)$$

To apply the induction rule, we proceed by mathematical induction on $n \in \mathbb{N}$, which reduces to showing:

1. $\left[z/x\right]e \equiv \left[z/x\right]e' : \tau \, [\Gamma]$, and
2. $\left[s(\pi)/x\right]e \equiv \left[s(\pi)/x\right]e' : \tau \, [\Gamma]$, if $\left[\pi/x\right]e \equiv \left[\pi/x\right]e' : \tau \, [\Gamma]$.

50.5 Exercises
Chapter 51

Equational Reasoning for PCF

In this Chapter we develop the theory of observational equivalence for \( \mathcal{L}\{\text{nat} \rightarrow\} \). The development proceeds long lines similar to those in Chapter 50, 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.

51.1 Observational Equivalence

The definition of observational equivalence, along with the auxiliary notion of Kleene equivalence, are defined similarly to Chapter 50, but modified to account for the possibility of non-termination.

The collection of well-formed \( \mathcal{L}\{\text{nat} \rightarrow\} \) contexts is inductively defined in a manner directly analogous to that in Chapter 50. Specifically, we define the judgement \( C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau') \) by rules similar to Rules (50.1), modified for \( \mathcal{L}\{\text{nat} \rightarrow\} \). (We leave the precise definition as an exercise for the reader.) When \( \Gamma \) and \( \Gamma' \) are empty, we write just \( C : \tau \rightsquigarrow \tau' \).

A complete program is a closed expression of type \( \text{nat} \).

Definition 51.1. We say that two complete programs, \( e \) and \( e' \), are Kleene equivalent, written \( e \simeq e' \), iff for every \( n \geq 0 \), \( e \rightarrow^* \overline{p} \) iff \( e' \rightarrow^* \overline{p} \).

Kleene equivalence is easily seen to be an equivalence relation and to be closed under converse evaluation. Moreover, \( \overline{0} \not\equiv \top \), and, if \( e \) and \( e' \) are both divergent, then \( e \simeq e' \).

Observational equivalence is defined as in Chapter 50.
51.2 Extensional Equivalence

**Definition 51.2.** We say that \( \Gamma \vdash e : \tau \) and \( \Gamma \vdash e' : \tau \) are observationally, or contextually, equivalent iff for every program context \( C : (\Gamma \triangleright \tau) \leadsto (\emptyset \triangleright \text{nat}) \), \( C \{ e \} \simeq C \{ e' \} \).

**Theorem 51.1.** Observational equivalence is the coarsest consistent congruence.

*Proof.* See the proof of Theorem 50.4 on page 455.

**Lemma 51.2 (Substitution and Functionality).** If \( e \equiv e' : \tau [\Gamma] \) and \( \gamma : \Gamma \), then \( \hat{\gamma}(e) \equiv \hat{\gamma}(e') : \tau \). Moreover, if \( \gamma \equiv \gamma' : \Gamma \), then \( \hat{\gamma}(e) \equiv \hat{\gamma}'(e) : \tau \) and \( \hat{\gamma}(e') \equiv \hat{\gamma}'(e') : \tau \).

*Proof.* See Lemma 50.5 on page 455.

### 51.2 Extensional Equivalence

**Definition 51.3.** Extensional equivalence, \( e \sim e' : \tau \), between closed expressions of type \( \tau \) is defined by induction on \( \tau \) as follows:

\[
\begin{align*}
\begin{array}{l}
e \sim e' : \text{nat} \quad \text{iff} \quad e \simeq e' \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{l}
e \sim e' : \tau_1 \rightarrow \tau_2 \quad \text{iff} \quad e_1 \sim e'_1 : \tau_1 \text{ implies } e_e(e_1) \sim e'_e(e'_1) : \tau_2 \\
\end{array}
\end{align*}
\]

Formally, extensional equivalence is defined as in Chapter 50, 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 \( \hat{\gamma}(e) \sim \hat{\gamma}'(e') : \tau \) whenever \( \gamma \sim \gamma' : \Gamma \).

**Lemma 51.3 (Strictness).** If \( e : \tau \) and \( e' : \tau \) are both divergent, then \( e \sim e' : \tau \).

*Proof.* By induction on the structure of \( \tau \). If \( \tau = \text{nat} \), 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 51.4 (Converse Evaluation).** Suppose that \( e \sim e' : \tau \). If \( d \mapsto e \), then \( d \sim e' : \tau \), and if \( d' \mapsto e' \), then \( e \sim d' : \tau \).
51.3 Extensional and Observational Equivalence Coincide

As a technical convenience, we enrich $\mathcal{L}\{\text{nat} \to\}$ with bounded recursion, with abstract syntax $\text{fix}^m[\tau](x.e)$ and concrete syntax $\text{fix}^m x : \tau \, \text{is} \, e$, where $m \geq 0$. The static semantics of bounded recursion is the same as for general recursion:

$$\Gamma, x : \tau \vdash e : \tau \quad \Rightarrow \quad \Gamma \vdash \text{fix}^m[\tau](x.e) : \tau \quad \text{(51.1a)}$$

The dynamic semantics of bounded recursion is defined as follows:

$$\text{fix}^0[\tau](x.e) \mapsto \text{fix}^0[\tau](x.e) \quad \text{(51.2a)}$$

$$\text{fix}^{m+1}[\tau](x.e) \mapsto [\text{fix}^m[\tau](x.e) / x]e \quad \text{(51.2b)}$$

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 is stated and proved in Section 51.4 on page 468 below.

Theorem 51.5 (Fixed Point Induction). Suppose that $x : \tau \vdash e : \tau$. If

$$(\forall m \geq 0) \ \text{fix}^m x : \tau \, \text{is} \, e \sim \text{fix}^m x : \tau \, \text{is} \, e' : \tau,$$

then $\text{fix} x : \tau \, \text{is} \, e \sim \text{fix} x : \tau \, \text{is} \, e' : \tau$.

Proof. Define an applicative context, $A$, to be either a hole, $\circ$, or an application of the form $A(e)$, where $A$ is an applicative context. (The typing judgement $A : \rho \rightsquigarrow \tau$ is a special case of the general typing judgment for contexts.) Define extensional equivalence of applicative contexts, written $A \approx A' : \rho \rightsquigarrow \tau$, by induction on the structure of $A$ as follows:

1. $\circ \approx \circ : \rho \rightsquigarrow \rho$;
2. if $A \approx A' : \rho \rightsquigarrow \tau_2 \to \tau$ and $e_2 \sim e'_2 : \tau_2$, then $A(e_2) \approx A'(e'_2) : \rho \rightsquigarrow \tau$. 

October 16, 2009   Draft   18:42
We prove by induction on the structure of $\tau$, if $A \approx A' : \rho \rightsquigarrow \tau$ and

for every $m \geq 0$, $A\{\text{fix}^m x : \rho \text{ is } e\} \sim A'\{\text{fix}^m x : \rho \text{ is } e'\} : \tau$,  \hspace{1cm} (51.3)

then

$A\{\text{fix} x : \rho \text{ is } e\} \sim A'\{\text{fix} x : \rho \text{ is } e'\} : \tau$.  \hspace{1cm} (51.4)

Choosing $A = A' = \circ$ (so that $\rho = \tau$) completes the proof.

If $\tau = \text{nat}$, then assume that $A \approx A' : \rho \rightsquigarrow \text{nat}$ and (51.3). By Definition 51.3 on page 464, we are to show

$A\{\text{fix} x : \rho \text{ is } e\} \simeq A'\{\text{fix} x : \rho \text{ is } e'\}$.  

By Corollary 51.14 on page 471 there exists $m \geq 0$ such that

$A\{\text{fix} x : \rho \text{ is } e\} \simeq A\{\text{fix}^m x : \rho \text{ is } e\}$.  

By (51.3) we have

$A\{\text{fix} x : \rho \text{ is } e\} \simeq A'\{\text{fix} x : \rho \text{ is } e'\}$.  

By Corollary 51.14 on page 471

$A'\{\text{fix}^m x : \rho \text{ is } e'\} \simeq A'\{\text{fix} x : \rho \text{ is } e'\}$.  

The result follows by transitivity of Kleene equivalence.

If $\tau = \tau_1 \rightarrow \tau_2$, then by Definition 51.3 on page 464, it is enough to show

$A\{\text{fix} x : \rho \text{ is } e\}(e_1) \sim A'\{\text{fix} x : \rho \text{ is } e'\}(e'_1) : \tau_2$

whenever $e_1 \sim e'_1 : \tau_1$. Let $A_2 = A(e_1)$ and $A'_2 = A'(e'_1)$. It follows from (51.3) that for every $m \geq 0$

$A_2\{\text{fix}^m x : \rho \text{ is } e\} \sim A'_2\{\text{fix}^m x : \rho \text{ is } e'\} : \tau_2$.  

Noting that $A_2 \approx A'_2 : \rho \rightsquigarrow \tau_2$, we have by induction

$A_2\{\text{fix} x : \rho \text{ is } e\} \sim A'_2\{\text{fix} x : \rho \text{ is } e'\} : \tau_2$

as required.  \hspace{1cm} \checkmark

**Lemma 51.6** (Reflexivity). \textit{If $\Gamma \vdash e : \tau$, then $e \sim e : \tau [\Gamma]$.}
51.3 Extensional and Observational Equivalence . . . 467

Proof. The proof proceeds along the same lines as the proof of Theorem 50.8 on page 457. The main difference is the treatment of general recursion, which is proved by fixed point induction. Consider Rule (15.1g). Assuming \( \gamma \sim \gamma' : \Gamma \), we are to show that

\[
\text{fix} x : \tau \text{ is } \hat{\gamma}(e) \sim \text{fix} x : \tau \text{ is } \hat{\gamma'}(e') : \tau.
\]

By Theorem 51.5 on page 465 it is enough to show that, for every \( m \geq 0 \),

\[
\text{fix}^m x : \tau \text{ is } \hat{\gamma}(e) \sim \text{fix}^m x : \tau \text{ is } \hat{\gamma'}(e') : \tau.
\]

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 51.4 on page 464, it is enough to show that \( \hat{\gamma}(e_1) \sim \hat{\gamma'}(e'_1) : \tau \), where

\[
e_1 = [\text{fix}^m x : \tau \text{ is } \hat{\gamma}(e)/x] \hat{\gamma}(e), \quad \text{and} \quad (51.5)
\]

\[
e'_1 = [\text{fix}^m x : \tau \text{ is } \hat{\gamma'}(e')/x] \hat{\gamma'}(e'). \quad (51.6)
\]

But this follows directly from the inner and outer inductive hypotheses. For by the outer inductive hypothesis, if

\[
\text{fix}^m x : \tau \text{ is } \hat{\gamma}(e) \sim \tau : [\text{fix}^m x : \tau \text{ is } \hat{\gamma}(e)]
\]

then

\[
[\text{fix}^m x : \tau \text{ is } \hat{\gamma'}(e)/x] \hat{\gamma'}(e) \sim \tau : [\text{fix}^m x : \tau \text{ is } \hat{\gamma}(e)/x] \hat{\gamma}(e)
\]

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 51.7 (Congruence). If \( C_0 : (\Gamma \vdash \tau) \leadsto (\Gamma_0 \vdash \tau_0) \), and \( e \sim e' : \tau : [\Gamma] \), then \( C_0\{e\} \sim C_0\{e'\} : \tau_0 : [\Gamma_0] \).

Proof. By induction on the derivation of the typing of \( C_0 \), following along similar lines to the proof of Lemma 51.6 on the preceding page. \( \square \)
Theorem 51.8. If \( e \sim e' : \tau [\Gamma] \), then \( e \equiv e' : \tau [\Gamma] \).

Proof. By consistency and congruence of extensional equivalence. \(\square\)

Lemma 51.9. If \( e \equiv e' : \tau \), then \( e \sim e' : \tau \).

Proof. By induction on the structure of \( \tau \). If \( \tau = \text{nat} \), then the result is immediate, since the empty expression context is a program context. If \( \tau = \tau_1 \rightarrow \tau_2 \), 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 51.8 \( e_1 \equiv e'_1 : \tau_1 \), and hence by Lemma 51.2 on page 464 \( e(e_1) \equiv e'(e'_1) : \tau_2 \), from which the result follows by induction. \(\square\)

Theorem 51.10. If \( e \equiv e' : \tau [\Gamma] \), then \( e \sim e' : \tau [\Gamma] \).

Proof. Assume that \( e \equiv e' : \tau [\Gamma] \). Suppose that \( \gamma \sim \gamma' : \Gamma \). By Theorem 51.8 we have \( \gamma \equiv \gamma' : \Gamma \), and so by Lemma 51.2 on page 464 we have

\[
\hat{\gamma}(e) \equiv \hat{\gamma}'(e') : \tau.
\]

Therefore by Lemma 51.9 we have

\[
\hat{\gamma}(e) \sim \hat{\gamma}'(e') : \tau.
\]

\(\square\)

Corollary 51.11. \( e \equiv e' : \tau [\Gamma] \) iff \( e \sim e' : \tau [\Gamma] \).

51.4 Compactness

The principle of fixed point induction is derived from a critical property of \( L\{\text{nat} \rightarrow \} \), 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 51.13 on the next page) makes use of the stack machine for \( \mathcal{L}\{\text{nat} \rightarrow \} \) defined in Chapter 27, augmented with the following transitions for bounded recursive expressions:

\[
\text{fix}^0 x : \tau \text{ise} \quad \text{for} \quad k \geq 0
\]

\[
\text{fix}^0 x : \tau \text{ise} \quad \text{for} \quad k \geq 0
\]

(51.7a)

(51.7b)

It is straightforward to extend the proof of correctness of the stack machine (Corollary 27.4 on page 242) to account for bounded recursion.

To get a feel for what is involved in the compactness proof, consider first the factorial function, \( f \), in \( \mathcal{L}\{\text{nat} \rightarrow \} \):

\[
\text{fix} f : \text{nat} \rightarrow \text{nat} \text{is} \lambda (x : \text{nat}. \text{if} z \Rightarrow s(z) \mid s(x') \Rightarrow x \ast f(x')}).
\]

Obviously evaluation of \( f(n) \) requires \( n \) recursive calls to the function itself. This means that, for a given input, \( n \), we may place a bound, \( m \), on the recursion that is sufficient to ensure termination of the computation. This can be expressed formally using the \( m \)-bounded form of general recursion,

\[
\text{fix}^m f : \text{nat} \rightarrow \text{nat} \text{is} \lambda (x : \text{nat}. \text{if} z \Rightarrow s(z) \mid s(x') \Rightarrow x \ast f(x')}).
\]

Call this expression \( f^{(m)} \). It follows from the definition of \( f \) that if \( f(\overline{n}) \xrightarrow{*} \overline{p} \), then \( f^{(m)}(\overline{n}) \xrightarrow{*} \overline{p} \) 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 \( \tau = \text{nat} \rightarrow (\text{nat} \rightarrow \text{nat}) \), given by the expression

\[
\text{fix} p : \tau \text{is} \lambda (x : \text{nat}. \text{if} z \Rightarrow \text{id} \mid s(x') \Rightarrow s \circ (p(x'))}),
\]

where \( \text{id} = \lambda (y : \text{nat}. y) \) is the identity, \( e' \circ e = \lambda (x : \tau. e'(e(x))) \) is composition, and \( s = \lambda (x : \text{nat}. s(x)) \) is the successor function. The application \( a(\overline{n}) \) terminates after three transitions, regardless of the value of \( n \), resulting in a \( \lambda \)-abstraction. When \( n \) is positive, the result contains a residual copy of \( a \) itself, which is applied to \( n - 1 \) as a recursive call. The \( m \)-bounded version of \( a \), written \( a^{(m)} \), is also such that \( a^{(m)}(\overline{n}) \) terminates in three steps, provided that \( m > 0 \). But the result is not the same, because the residuals of \( a \) appear as \( a^{(m-1)} \), rather than as \( a \) itself.

Turning now to the proof, it is helpful to introduce some notation. Suppose that \( x : \tau \vdash e_x : \tau \) for some arbitrary abstractor \( x.e_x \). Define \( f^{(w)} = \)
fix \( \tau \times e \), and \( f^{(m)} = \text{fix}^m \times \tau \times e \), 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 51.12.** If \([f^{(m)}/y]k \triangleright [f^{(m)}/y]e \rightarrow^* e \triangleleft \bar{\pi}\), where \( e \neq y \), then \([f^{(m+1)}/y]k \triangleright [f^{(m+1)}/y]e \rightarrow^* e \triangleleft \bar{\pi}\).

**Proof.** By induction on the definition of the transition judgement for \( K \{ \text{nat} \rightarrow \} \).

**Theorem 51.13** (Compactness). Suppose that \( y : \tau \vdash e : \text{nat} \) where \( y \notin f^{(\omega)} \). If \([f^{(\omega)}/y]e \rightarrow^* \pi\), then there exists \( m \geq 0 \) such that \([f^{(m)}/y]e \rightarrow^* \pi\).

**Proof.** We prove simultaneously the stronger statements that if

\[ [f^{(\omega)}/y]k \triangleright [f^{(\omega)}/y]e \rightarrow^* e \triangleleft \bar{\pi}, \]

then for some \( m \geq 0 \),

\[ [f^{(m)}/y]k \triangleright [f^{(m)}/y]e \rightarrow^* e \triangleleft \bar{\pi}, \]

and

\[ [f^{(\omega)}/y]k \triangleleft [f^{(\omega)}/y]e \rightarrow^* e \triangleleft \pi, \]

then for some \( m \geq 0 \),

\[ [f^{(m)}/y]k \triangleleft [f^{(m)}/y]e \rightarrow^* e \triangleleft \bar{\pi}. \]

(Note that if \([f^{(\omega)}/y]e \text{ val}\), then \([f^{(m)}/y]e \text{ val}\) for all \( m \geq 0 \).) The result then follows by the correctness of the stack machine (Corollary 27.4 on page 242).

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:

\[ [f^{(\omega)}/y]k \triangleright f^{(\omega)} \rightarrow [f^{(\omega)}/y]k \triangleright [f^{(\omega)}/x]e_x \rightarrow^* e \triangleleft \bar{\pi}. \]
51.5 Co-Natural Numbers

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
\[
[f(m)/y]k \triangleright [f(m)/x]e_x \mapsto^* e < \pi.
\]

By Lemma 51.12 on the facing page
\[
[f(m+1)/y]k \triangleright [f(m)/x]e_x \mapsto^* e < \pi
\]
and we need only observe that
\[
[f(m+1)/y]k \triangleright f(s(x)/y)k \triangleright [f(m)/x]e_x
\]
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 \(z \neq y\)
\[
[f(\omega)/y]k \triangleright fixz: \tau isd_\omega \mapsto [f(\omega)/y]k \triangleright [fixz: \tau isd_\omega/z]d_\omega \mapsto^* e < \pi.
\]
where \(d_\omega = [f(\omega)/y]d\). By induction there exists \(m \geq 0\) such that
\[
[f(m)/y]k \triangleright [fixz: \tau isd_m/z]d_m \mapsto^* e < \pi,
\]
where \(d_m = [f(m)/y]d\). But then by Lemma 51.12 on the preceding page we have
\[
[f(m+1)/y]k \triangleright [fixz: \tau isd_{m+1}/z]d_{m+1} \mapsto^* e < \pi,
\]
where \(d_{m+1} = [f(m+1)/y]d\), from which the result follows directly. \(\square\)

**Corollary 51.14.** 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 51.13 on the facing page to obtain \(m\), and note that the required Kleene equivalence follows. \(\square\)

## 51.5 Co-Natural Numbers

In Chapter 15 we considered a variation of \(L\{nat \rightarrow\}\) with the co-natural numbers, \(conat\), as base type. This is achieved by specifying that \(s(e)\) 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, \(\omega\), by \(fix x: conat is s(x)\), which consists of an infinite stack of successors. Since
the successor is interpreted lazily, \(\omega\) evaluates to a value, namely \(s(\omega)\), 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 \(\omega\).

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 \(\text{conat}\) must be formulated to account for laziness. Rather than being defined \textit{inductively} as the strongest relation closed under specified conditions, we define it \textit{coinductively} as the weakest relation consistent with analogous conditions. We may then show that two expressions are related using the principle of \textit{proof by coinduction}.

If \(\text{conat}\) 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 \(e \rightarrow^* z\), then \(e' \rightarrow^* z\), and \textit{vice versa}; and (b) if \(e \rightarrow^* s(e_1)\), then \(e' \rightarrow^* s(e'_1)\), and \textit{vice versa}. Note well that we do not require anything of \(e_1\) and \(e'_1\) in the second clause. This means that \(0 \simeq 1\), yet we retain consistency in that \(0 \not\simeq 1\).

Corollary 51.14 on the previous page can be proved for the co-natural numbers by essentially the same argument.

The definition of extensional equivalence at type \(\text{conat}\) is defined to be the \textit{weakest} equivalence relation, \(E\), between closed terms of type \(\text{conat}\) satisfying the following \textit{conat-consistency conditions}: if \(e \text{ E } e' : \text{conat}\), then

1. If \(e \rightarrow^* z\), then \(e' \rightarrow^* z\), and \textit{vice versa}.

2. If \(e \rightarrow^* s(e_1)\), then \(e' \rightarrow^* s(e'_1)\) with \(e_1 \text{ E } e'_1 : \text{conat}\), and \textit{vice versa}.

It is immediate that if \(e \sim e' : \text{conat}\), 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 \(\text{conat}\), then \(e \sim e' : \text{conat}\)—simply because the preceding two conditions are vacuously true in this case.

This is an example of the more general principle of \textit{proof by \text{conat}-coinduction}. To show that \(e \sim e' : \text{conat}\), it suffices to exhibit a relation, \(E\), such that

1. \(e \text{ E } e' : \text{conat}\), and

2. \(E\) satisfies the \textit{conat-consistency conditions}.
If these requirements hold, then $E$ is contained in extensional equivalence at type $\conat$, and hence $e \sim e' : \conat$, as required.

As an application of $\conat$-coinduction, let us consider the proof of Theorem 51.5 on page 465. The overall argument remains as before, but the proof for the type $\conat$ must be altered as follows. Suppose that $A \approx A' : \rho \rightsquigarrow \conat$, and let $a = A\{\text{fix } x : \rho \text{ is } e\}$ and $a' = A'\{\text{fix } x : \rho \text{ is } e'\}$. Writing $a^{(m)} = A\{\text{fix } m \times x : \rho \text{ is } e\}$ and $a'^{(m)} = A'\{\text{fix } m \times x : \rho \text{ is } e'\},$ assume that

$$\text{for every } m \geq 0, \ a^{(m)} \sim a'^{(m)} : \conat.$$  

We are to show that $a \sim a' : \conat$.

Define the functions $p_n$ for $n \geq 0$ on closed terms of type $\conat$ by the following equations:

$$p_0(d) = d$$
$$p_{n+1}(d) = \begin{cases} d' & \text{if } p_n(d) \rightsquigarrow^* s(d') \\ \text{undefined} & \text{otherwise} \end{cases}$$

For $n \geq 0$, let $a_n = p_n(a)$ and $a'_n = p_n(a')$. Correspondingly, let $a^{(m)}_n = p_n(a^{(m)})$ and $a'^{(m)}_n = p_n(a'^{(m)})$. Define $E$ to be the strongest relation such that $a_n E a'_n : \conat$ for all $n \geq 0$. We will show that the relation $E$ satisfies the $\conat$-consistency conditions, and so it is contained in extensional equivalence. Since $a E a' : \conat$ (by construction), the result follows immediately.

To show that $E$ is $\conat$-consistent, suppose that $a_n E a'_n : \conat$ for some $n \geq 0$. We have by Corollary 51.14 on page 471 $a_n \simeq a^{(m)}_n$, for some $m \geq 0$, and hence, by the assumption, $a_n \simeq a'^{(m)}_n$, and so by Corollary 51.14 on page 471 again, $a'^{(m)}_n \simeq a'_n$. Now if $a_n \rightsquigarrow^* s(b_n)$, then $a^{(m)}_n \rightsquigarrow^* s(b^{(m)}_n)$ for some $b^{(m)}_n$, and hence there exists $b'^{(m)}_n$ such that $a'^{(m)}_n \rightsquigarrow^* b'^{(m)}_n$, and so there exists $b'_n$ such that $a'_n \rightsquigarrow^* s(b'_n)$. But $b_n = p_{n+1}(a)$ and $b'_n = p_{n+1}(a')$, and we have $b_n E b'_n : \conat$ by construction, as required.

### 51.6 Exercises

1. Call-by-value variant, with recursive functions.
Chapter 52

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 class of programs we may write, and also to limit the class 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 $L\{\rightarrow \forall\}$ under a call-by-name interpretation.

52.1 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 $\forall (t \cdot t \rightarrow t)$. What function could it be? When instantiated at a type $\tau$ it should evaluate to a function $g$ of type $\tau \rightarrow \tau$ that, when further applied to a value $v$ of type $\tau$ returns a value $v'$ of type $\tau$. 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 \( \tau \), and \( f \) must therefore be the polymorphic identity.

Suppose that \( f \) is a function of type \( \forall (t.t) \). What function could it be? A moment’s thought reveals that it cannot exist at all! For it must, when instantiated at a type \( \tau \), return a value of that type. But not every type has a value (including this one), so this is an impossible assignment. The only conclusion is that \( \forall (t.t) \) is an empty type.

Let \( N \) be the type of polymorphic Church numerals introduced in Chapter 23, namely \( \forall (t.t \rightarrow (t \rightarrow t) \rightarrow t) \). What are the values of this type? Given any type \( \tau \), and values \( z : \tau \) and \( s : \tau \rightarrow \tau \), the expression

\[
f[\tau](z)(s)
\]

must yield a value of type \( \tau \). Moreover, it must behave uniformly with respect to the choice of \( \tau \). What values could it yield? The only way to build a value of type \( \tau \) is by using the element \( z \) and the function \( s \) passed to it. A moment’s thought reveals that the application must amount to the \( n \)-fold composition

\[
s(s(\ldots s(z)\ldots)).
\]

That is, the elements of \( N \) are in one-to-one correspondence with the natural numbers.

### 52.2 Observational Equivalence

The definition of observational equivalence given in Chapters 50 and 51 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 \( \mathcal{L}\{\rightarrow \forall\} \), 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 \( \mathcal{L}\{\rightarrow \forall\} \) 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 \( \mathcal{L}\{\rightarrow \forall\} \) with a base type, \( 2 \), containing two constants, \( \texttt{tt} \) and \( \texttt{ff} \), 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) \( e \rightarrow^* \texttt{tt} \) and \( e' \rightarrow^* \texttt{tt} \); or (b) \( e \rightarrow^* \texttt{ff} \) and \( e' \rightarrow^* \texttt{ff} \).
52.2 Observational Equivalence

This is obviously an equivalence relation, and it is immediate that \( \text{tt} \neq \text{ff} \), 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

\[
C : (\Delta; \Gamma \triangleright \tau) \leadsto (\Delta'; \Gamma' \triangleright \tau'),
\]

which states that \( C \) is an expression context that, when filled with an expression \( \Delta; \Gamma \vdash e : \tau \) yields an expression \( \Delta'; \Gamma' \vdash C\{e\} : \tau \). (We leave the precise definition of this judgement, and the verification of its properties, as an exercise for the reader.)

**Definition 52.1.** Two expressions of the same type are observationally equivalent, written \( e \sim e' : \tau \left[ \Delta; \Gamma \right] \), iff \( C\{e\} \simeq C\{e'\} \) whenever \( C : (\Delta; \Gamma \triangleright \tau) \leadsto (\emptyset \triangleright 2) \).

**Lemma 52.1.** 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. \( \square \)

**Lemma 52.2.**

1. If \( e \equiv e' : \tau \left[ \Delta, t; \Gamma \right] \) and \( \rho \) type, then \( \rho/t \equiv \rho/t\ e \equiv \rho/t\ e' : \rho/t\ \tau \left[ \Delta, \rho/t; \Gamma \right] \).

2. If \( e \equiv e' : \tau \left[ \emptyset; \Gamma, x : \sigma \right] \) and \( d : \sigma \), then \( \left[ d/x \right] e \equiv \left[ d/x \right] e' : \tau \left[ \emptyset; \Gamma \right] \). Moreover, if \( d \equiv d' : \sigma \), then \( \left[ d'/x \right] e \equiv \left[ d'/x \right] e' : \tau \left[ \emptyset; \Gamma \right] \), and similarly for \( e' \).

**Proof.**

1. Let \( C : (\Delta; \rho/t; \Gamma \triangleright [\rho/t] \tau) \leadsto (\emptyset \triangleright 2) \) be a program context. We are to show that

\[
C\{[\rho/t] e\} \simeq C\{[\rho/t] e'\}.
\]

Since \( C \) is closed, this is equivalent to

\[
[\rho/t] C\{e\} \simeq [\rho/t] C\{e'\}.
\]
Let \( C' \) be the context \( \Lambda (t.C\{\circ\}) [\rho] \), and observe that

\[
C' : (\Delta, t; \Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright 2).
\]

Therefore, from the assumption,

\[
C'\{e\} \simeq C'\{e'\}.
\]

But \( C'\{e\} \simeq [\rho / t] C\{e\} \), and \( C'\{e'\} \simeq [\rho / t] C\{e'\} \), from which the result follows.

2. By an argument essentially similar to that for Lemma 50.5 on page 455. 
\( \square \)

### 52.3 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 \( L\{\rightarrow \forall\} \) is somewhat more complex than for \( L\{\text{nat} \rightarrow\} \). 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 \( \rho \). But this runs into two problems, one technical, the other conceptual. The same device will be used to solve both problems.

The technical problem stems from impredicativity. In Chapter 50 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 $\rho$ and $\rho'$, and treat the type variable $t$ as standing for any relation (of a suitable class) between $\rho$ and $\rho'$. 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 class of “admissible” binary relations between closed expressions. The conditions are imposed to ensure that logical equivalence and observational equivalence coincide.

**Definition 52.2 (Admissibility).** A relation $R$ between expressions of types $\rho$ and $\rho'$ is admissible, written $R : \rho \leftrightarrow \rho'$, iff it satisfies two requirements:

1. Respect for observational equivalence: if $R(e, e')$ and $d \equiv e : \rho$ and $d' \equiv e' : \rho'$, 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 $\delta$ is a type substitution that assigns a closed type to each type variable $t \in \Delta$. A type substitution, $\delta$, induces a substitution function, $\hat{\delta}$, on types given by the equation

$$\hat{\delta}(\tau) = \left[\delta(t_1), \ldots, \delta(t_n) / t_1, \ldots, t_n\right] \tau,$$

and similarly for expressions. Substitution is extended to contexts pointwise by defining $\hat{\delta}(\Gamma)(x) = \hat{\delta}(\Gamma(x))$ for each $x \in \text{dom}(\Gamma)$.

Let $\delta$ and $\delta'$ be two type substitutions of closed types to the type variables in $\Delta$. A relation assignment, $\eta$, between $\delta$ and $\delta'$ is an assignment of an admissible relation $\eta(t) : \delta(t) \leftrightarrow \delta'(t)$ to each $t \in \Delta$. The judgement $\eta : \delta \leftrightarrow \delta'$ states that $\eta$ is a relation assignment between $\delta$ and $\delta'$. 

October 16, 2009 Draft 18:42
Logical equivalence is defined in terms of its generalization, called parameterized logical equivalence, written \( e \sim e' : \tau [\eta : \delta \leftrightarrow \delta'] \), defined as follows.

**Definition 52.3** (Parameterized Logical Equivalence). The relation \( e \sim e' : \tau [\eta : \delta \leftrightarrow \delta'] \) is defined by induction on the structure of \( \tau \) by the following conditions:

- \( e \sim e' : t [\eta : \delta \leftrightarrow \delta'] \) iff \( \eta(t)(e,e') \)
- \( e \sim e' : 2 [\eta : \delta \leftrightarrow \delta'] \) iff \( e \simeq e' \)
- \( e \sim e' : \tau_1 \rightarrow \tau_2 [\eta : \delta \leftrightarrow \delta'] \) iff \( e_1 \sim e'_1 : \tau_1 [\eta : \delta \leftrightarrow \delta'] \) implies \( e(e_1) \sim e'(e'_1) : \tau_2 [\eta : \delta \leftrightarrow \delta'] \)
- \( e \sim e' : \forall (t,\tau) [\eta : \delta \leftrightarrow \delta'] \) iff for every \( \rho, \rho', \) and every \( R : \rho \rightarrow \rho', \)
  \( e[\rho] \sim e'[\rho'] : \tau [\eta[t \leftarrow R] : \delta[t \leftarrow \rho] \leftrightarrow \delta'[t \leftarrow \rho']] \)

Logical equivalence is defined in terms of parameterized logical equivalence by considering all possible interpretations of its free type- and expression variables. An expression substitution, \( \gamma \), for a context \( \Gamma \), is an substitution of a closed expression \( \gamma(x) : \Gamma(x) \) to each variable \( x \in \text{dom}(\Gamma) \). An expression substitution, \( \gamma : \Gamma \), induces a substitution function, \( \hat{\gamma} \), defined by the equation

\[
\hat{\gamma}(e) = [\gamma(x_1), \ldots, \gamma(x_n)/x_1, \ldots, x_n]e,
\]

where the domain of \( \Gamma \) consists of the variables \( x_1, \ldots, x_n \).

The relation \( \gamma \sim \gamma' : \Gamma [\eta : \delta \leftrightarrow \delta'] \) is defined to hold iff \( \text{dom}(\gamma) = \text{dom}(\gamma') = \text{dom}(\Gamma) \), and \( \gamma(x) \sim \gamma'(x) : \Gamma(x) [\eta : \delta \leftrightarrow \delta'] \) for every variable, \( x \), in their common domain.

**Definition 52.4** (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 assignment \( \delta \) and \( \delta' \) of closed types to type variables in \( \Delta \), and every relation assignment \( \eta : \delta \leftrightarrow \delta' \), if \( \gamma \sim \gamma' : \Gamma [\eta : \delta \leftrightarrow \delta'] \), then \( \hat{\gamma}(\delta)(e) \sim \hat{\gamma}'(\delta')(e') : \tau [\eta : \delta \leftrightarrow \delta'] \).

When \( e, e' \), and \( \tau \) are closed, then this definition states that \( e \sim e' : \tau \) iff \( e \sim e' : \tau [\emptyset : \emptyset \leftarrow \emptyset] \), so that logical equivalence is indeed a special case of its generalization.

**Lemma 52.3** (Closure under Converse Evaluation). Suppose that \( e \sim e' : \tau [\eta : \delta \leftrightarrow \delta'] \). If \( d \leftarrow e \), then \( d \sim e' : \tau \), and if \( d' \leftarrow e' \), then \( e \sim d' : \tau \).
52.3 Logical Equivalence

Proof. By induction on the structure of $\tau$. When $\tau = t$, the result holds by the definition of admissibility. Otherwise the result follows by induction, making use of the definition of the transition relation for applications and type applications.

Lemma 52.4 (Respect for Observational Equivalence). Suppose that $e \sim e' : \tau [\eta : \delta \leftrightarrow \delta']$. If $d \cong e : \hat{\delta}(\tau)$ and $d' \cong e' : \hat{\delta}'(\tau)$, then $d \sim d' : \tau [\eta : \delta \leftrightarrow \delta']$.

Proof. By induction on the structure of $\tau$, relying on the definition of admissibility, and the congruence property of observational equivalence. For example, if $\tau = \forall (t.\sigma)$, 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, $d[\rho] \cong e[\rho] : [\rho/t]\hat{\delta}(\sigma)$,

$$d'[\rho] \cong e'[\rho] : [\rho'/t]\hat{\delta}'(\sigma).$$

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 52.5. The relation $e \sim e' : \tau [\eta : \delta \leftrightarrow \delta']$ is an admissible relation between closed types $\hat{\delta}(\tau)$ and $\hat{\delta}'(\tau)$.

Proof. By Lemmas 52.3 on the facing page and 52.4.

Logical Equivalence respects observational equivalence.

Corollary 52.6. 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 52.2 on page 477 and Corollary 52.5.

Lemma 52.7 (Compositionality). Suppose that

$$e \sim e' : \tau [\eta[t \mapsto R] : \delta[t \mapsto \hat{\delta}(\rho)] \leftrightarrow \delta'[t \mapsto \hat{\delta}'(\rho)]],$$

where $R : \hat{\delta}(\rho) \leftrightarrow \hat{\delta}'(\rho)$ is such that $R(d, d')$ holds iff $d \sim d' : \rho [\eta : \delta \leftrightarrow \delta']$. Then $e \sim e' : [\rho/t]\tau [\eta : \delta \leftrightarrow \delta']$. 
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 \( \rho, \rho' \), and \( R : \rho \leftrightarrow \rho' \),
\[
[\rho/t] \hat{\gamma}(\delta(e)) \sim [\rho'/t] \hat{\gamma}'(\delta'(e)) : \tau [\eta_\ast : \delta_\ast \leftrightarrow \delta'_\ast],
\]
where \( \eta_\ast = \eta[t \mapsto R], \delta_\ast = \delta[t \mapsto \rho], \) and \( \delta'_\ast = \delta'[t \mapsto \rho'] \). Since
\[
\Lambda(t : \hat{\gamma}(\delta(e))) [\rho] \mapsto^* [\rho/t] \hat{\gamma}(\delta(e))
\]
and
\[
\Lambda(t : \hat{\gamma}'(\delta'(e))) [\rho'] \mapsto^* [\rho'/t] \hat{\gamma}'(\delta'(e)),
\]
the result follows by Lemma 52.3 on page 480.

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
\[
\hat{\gamma}(\delta(e)) \sim \hat{\gamma}'(\delta'(e)) : \forall (t. \tau) [\eta : \delta \leftrightarrow \delta']
\]
Let \( \hat{\rho} = \hat{\delta}(\rho) \) and \( \hat{\rho}' = \hat{\delta}'(\rho) \). Define the relation \( R : \hat{\rho} \leftrightarrow \hat{\rho}' \) by \( R(d, d') \) iff \( d \sim d' : \rho [\eta : \delta \leftrightarrow \delta'] \). By Corollary 52.5 on the previous page, this relation is admissible.
By the definition of logical equivalence at polymorphic types, we obtain
\[
\hat{\gamma}(\delta(e)) [\hat{\rho}] \sim \hat{\gamma}'(\delta'(e)) [\hat{\rho}'] : \tau [\eta[t \mapsto R] : \delta[t \mapsto \hat{\rho}] \leftrightarrow \delta'[t \mapsto \hat{\rho}']].
\]
By Lemma 52.7 on the preceding page
\[
\hat{\gamma}(\delta(e)) [\hat{\rho}] \sim \hat{\gamma}'(\delta'(e)) [\hat{\rho}'] : [\rho/t] \tau [\eta : \delta \leftrightarrow \delta']
\]
52.3 Logical Equivalence

But
\[
\hat{\gamma}(\hat{\delta}(e))[\hat{\rho}] = \hat{\gamma}(\hat{\delta}(e))[\hat{\delta}(\hat{\rho})] \tag{52.1}
\]
\[
= \hat{\gamma}(\hat{\delta}(e[\hat{\rho}])) \tag{52.2}
\]
and similarly
\[
\hat{\gamma}'(\hat{\delta}'(e))[\hat{\rho}'] = \hat{\gamma}'(\hat{\delta}'(e))[\hat{\delta}'(\hat{\rho})] \tag{52.3}
\]
\[
= \hat{\gamma}'(\hat{\delta}'(e[\hat{\rho}])) \tag{52.4}
\]
from which the result follows.

\[\square\]

**Corollary 52.9.** If \( e \cong e' : \tau | \Delta; \Gamma \), then \( e \sim e' : \tau | \Delta; \Gamma \).

**Proof.** By Theorem 52.8 on the preceding page \( e \sim e : \tau | \Delta; \Gamma \), and hence by Corollary 52.6 on page 481, \( e \sim e' : \tau | \Delta; \Gamma \). \[\square\]

**Lemma 52.10 (Congruence).** If \( e \sim e' : \tau | \Delta; \Gamma \) and \( C : (\Delta; \Gamma \triangleright \tau) \leadsto (\Delta'; \Gamma' \triangleright \tau') \), then \( C\{e\} \sim C\{e'\} : \tau | \Delta'; \Gamma' \).

**Proof.** By induction on the structure of \( C \), following along very similar lines to the proof of Theorem 52.8 on the preceding page. \[\square\]

**Lemma 52.11 (Consistency).** Logical equivalence is consistent.

**Proof.** Follows immediately from the definition of logical equivalence. \[\square\]

**Corollary 52.12.** If \( e \sim e' : \tau | \Delta; \Gamma \), then \( e \cong e' : \tau | \Delta; \Gamma \).

**Proof.** By Lemma 52.11 Logical equivalence is consistent, and by Lemma 52.10, it is a congruence, and hence is contained in observational equivalence. \[\square\]

**Corollary 52.13.** Logical and observational equivalence coincide.

**Proof.** By Corollaries 52.9 and 52.12. \[\square\]

If \( d : \tau \) and \( d \mapsto e \), then \( d \sim e : \tau \), and hence by Corollary 52.12, \( d \cong e : \tau \). 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, though it cannot be omitted at such an early stage.
52.4 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 52.14.** Let $e : \forall(t.t \rightarrow t)$ be arbitrary, and let $id : \Lambda(t.\Lambda(x.t.x))$. Then $e \simeq id : \forall(t.t \rightarrow t)$.

**Proof.** By Corollary 52.13 on the previous page it is sufficient to show that $e \sim id : \forall(t.t \rightarrow t)$. Let $\rho$ and $\rho'$ be arbitrary closed types, let $R : \rho \leftrightarrow \rho'$ be an admissible relation, and suppose that $e_0 R e'_0$. We are to show

$$e[\rho](e_0) R id[\rho](e'_0),$$

which, given the definition of $id$, 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 52.8 on page 482 we have $e \sim e : \forall(t.t \rightarrow t)$. 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 $\mathcal{L}\{\rightarrow\forall\}$. The proof of definability in each case consisted of showing that the type and its associated introduction and elimination forms are encodable in $\mathcal{L}\{\rightarrow\forall\}$. The encodings are correct in the (weak) sense that the dynamic semantics of these constructs as given in the earlier chapters is derivable from the dynamic semantics of $\mathcal{L}\{\rightarrow\forall\}$ 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, \texttt{unit}, in \(L\{\to \forall\}\), as defined in Chapter 23 by the following equations:

\[
\text{unit} = \forall (r. r \to r) \\
\langle \rangle = \Lambda (r. \lambda (x: r. x))
\]

It is easy to see that \(\langle \rangle : \text{unit}\) according to these definitions. But this merely says that the type \texttt{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 52.14 on the preceding page! We say that the type \texttt{unit} is strongly definable within \(L\{\to \forall\}\).

Continuing in this vein, let us examine the definition of the binary product type in \(L\{\to \forall\}\), also given in Chapter 23:

\[
\tau_1 \times \tau_2 = \forall (r. (\tau_1 \to \tau_2 \to r) \to r) \\
\langle e_1, e_2 \rangle = \Lambda (r. \lambda (x: \tau_1 \to \tau_2 \to r. x(e_1)(e_2))) \\
e \cdot 1 = e[\tau_1] (\lambda (x: \tau_1. \lambda (y: \tau_2. x))) \\
e \cdot r = e[\tau_2] (\lambda (x: \tau_1. \lambda (y: \tau_2. y)))
\]

It is easy to check that \(\langle e_1, e_2 \rangle \cdot 1 \cong e_1 : \tau_1\) and \(\langle e_1, e_2 \rangle \cdot r \cong e_2 : \tau_2\) 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 \(L\{\to \forall\}\). We will make use of a lemma governing the behavior of the elements of the product type whose proof relies on Theorem 52.8 on page 482.

**Lemma 52.15.** 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 52.13 on page 483, we let \(\rho\) and \(\rho'\) be arbitrary closed types, and let \(R : \rho \leftrightarrow \rho'\) be an admissible relation between them. Suppose further that

\[h \sim h' : \tau_1 \to \tau_2 \to t [\eta : \delta \leftrightarrow \delta'],\]

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 52.8 on page 482 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 \( e \cdot 1 \cong e_1 : \tau_1 \) and \( e \cdot x \cong e_2 : \tau_2 \). We wish to show that \( e \cong \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \). From Lemma 52.15 on the previous page it is easy to deduce that \( e \cong \langle e \cdot 1, e \cdot x \rangle : \tau_1 \times \tau_2 \) 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. \( \text{natiter } z \{ z \Rightarrow e_0 | s(x) \Rightarrow e_1 \} \cong e_0 : \rho. \)
2. \( \text{natiter } s(e) \{ z \Rightarrow e_0 | s(x) \Rightarrow e_1 \} \cong [\text{natiter } e \{ z \Rightarrow e_0 | s(x) \Rightarrow e_1 \} / x] e_1 : \rho. \)
3. Suppose that \( x : \text{nat} \vdash r(x) : \rho \). If
   (a) \( r(z) \cong e_0 : \rho \), and
   (b) \( r(s(e)) \cong [r(e) / x] e_1 : \rho, \)

then for every \( e : \text{nat} \), \( r(e) \cong \text{natiter } e \{ z \Rightarrow e_0 | s(x) \Rightarrow e_1 \} : \rho. \)
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 \( \bar{n} \). We then argue for unicity of the iterator by mathematical induction on \( n \geq 0 \).

**Lemma 52.16.** If \( e : \text{nat} \), then either \( e \equiv z : \text{nat} \), or there exists \( e' : \text{nat} \) such that \( e \equiv s(e') : \text{nat} \). Consequently, there exists \( n \geq 0 \) such that \( e \equiv \bar{n} : \text{nat} \).

**Proof.** By Theorem 52.8 on page 482 we have \( e \sim e : \text{nat} \). Define the relation \( R : \text{nat} \leftrightarrow \text{nat} \) to be the strongest relation such that \( d R d' \) iff either \( d \equiv z : \text{nat} \) and \( d' \equiv z : \text{nat} \), or \( d \equiv s(d_1) : \text{nat} \) and \( d' \equiv s(d_1') : \text{nat} \) and \( d_1 R d_1' \). It is easy to see that \( z R z \), and if \( e R e' \), then \( s(e) R s(e') \). Letting zero = \( z \) and \( \text{succ} = \lambda (x : \text{nat} . s(x)) \), we have

\[
e[nat](\text{zero})\cdot(\text{succ}) \sim e[nat](\text{zero})\cdot(\text{succ}).
\]

The result follows by the induction principle arising from the definition of \( R \) as the strongest relation satisfying its defining conditions. \( \square \)

A straightforward extension of this argument shows that, up to observational equivalence, inductive and coinductive types are strongly definable in \( \mathcal{L}\{\to \forall\} \).

### 52.5 Exercises
Part XX

Working Drafts of Chapters
Appendix A

Polarization

Up to this point we have frequently encountered arbitrary choices in the dynamic semantics of various language constructs. For example, when specifying the dynamics of pairs, we must choose, rather arbitrarily, between the lazy semantics, in which all pairs are values regardless of the value status of their components, and the eager semantics, 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) semantics, 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. 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 27. 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.

A.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.
A.2 Focusing

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 \( L^{\pm}\{\text{nat} \to\} \) is a polarized formulation of \( L\{\text{nat} \to\} \) in which the syntax of types is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pos. Type</td>
<td>( \tau^+ )</td>
<td>( \text{dn}(\tau^-) )</td>
<td>( \downarrow \tau^- )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{nat}</td>
<td>\text{nat}</td>
</tr>
<tr>
<td>Neg. Type</td>
<td>( \tau^- )</td>
<td>( \text{up}(\tau^+) )</td>
<td>( \uparrow \tau^+ )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{parr}(\tau^+_1;\tau^-_2)</td>
<td>( \tau^+_1 \to \tau^-_2 )</td>
</tr>
</tbody>
</table>

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

\[
\tau^+_1 \to (\tau^+_2 \to (\ldots (\tau^+_{n-1} \to \tau^-_n) )))
\]

directly, but to form a left-iterated function type requires shifting,

\[
\downarrow (\tau^+_1 \to \tau^-_2) \to \tau^-
\]

to turn the negative function type into a positive type. Conversely, shifting is needed to define a function whose range is positive, \( \tau^+_1 \to \uparrow \tau^+_2 \).

A.2 Focusing

The syntax of \( L^{\pm}\{\text{nat} \to\} \) is motivated by the polarization of its types. For each polarity we have a class of values and a class of continuations with
which we may create (neutral) computations.

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pos. Value</td>
<td>$v^+$</td>
<td>$z$</td>
<td>$z$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$s(v^+)$</td>
<td>$s(v^+)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{del}^-(e)$</td>
<td>$\text{del}^-(e)$</td>
</tr>
<tr>
<td>Pos. Cont.</td>
<td>$k^+$</td>
<td>ifz($e_0;x.e_1$)</td>
<td>ifz($e_0;x.e_1$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{force}^-(k^-)$</td>
<td>$\text{force}^-(k^-)$</td>
</tr>
<tr>
<td>Neg. Value</td>
<td>$v^-$</td>
<td>lam[$\tau^+$(x.e)]</td>
<td>$\lambda(x:\tau^+.e)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{del}^+(v^+)$</td>
<td>$\text{del}^+(v^+)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{fix}(x.v^-)$</td>
<td>$\text{fix} x is v^-$</td>
</tr>
<tr>
<td>Neg. Cont.</td>
<td>$k^-$</td>
<td>ap($v^+;k^-$)</td>
<td>ap($v^+;k^-$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{force}^+(x.e)$</td>
<td>$\text{force}^+(x.e)$</td>
</tr>
<tr>
<td>Computation</td>
<td>$e$</td>
<td>$\text{ret}(v^-)$</td>
<td>$\text{ret}(v^-)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{cut}^+(v^+;k^+)$</td>
<td>$v^+ \triangleright k^+$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{cut}^-(v^-;k^-)$</td>
<td>$v^- \triangleright k^+$</td>
</tr>
</tbody>
</table>

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 $\text{del}^-(e)$, and we may consider a positive value to be a negative value using $\text{del}^+(v^+)$. The positive continuations include the conditional branch, sans argument, and the negative continuations include application sites for functions consisting of a positive argument value and a continuation for the negative result. In addition we include positive continuations to force the computation of a suspended negative value, and to extract an included positive value. Computations, which correspond to machine states, consist of returned negative values (these are final states), states passing a positive value to a positive continuation, and states passing a negative value to a negative continuation. General recursion appears as a form of negative value; the recursion is unrolled when it is made the subject of an observation.

### A.3 Statics

The static semantics of $\mathcal{L}^\pm\{\text{nat} \rightarrow \}$ consists of a collection of rules for deriving judgements of the following forms:

- Positive values: $\Gamma \vdash v^+: \tau^+$.
- Positive continuations: $\Gamma \vdash k^+: \tau^+ > \gamma^-$.
A.3 Statics

- Negative values: $\Gamma \vdash v^- : \tau^-$. 
- Negative continuations: $\Gamma \vdash k^- : \tau^- > \gamma^-$. 
- Computations: $\Gamma \vdash e : \gamma^-$. 

Throughout $\Gamma$ is a finite set of hypotheses of the form $x_1 : \tau_1^+, \ldots, x_n : \tau_n^+$, for some $n \geq 0$, and $\gamma^-$ is any negative type.

The typing rules for continuations specify both an argument type (on which values they act) and a result type (of the computation resulting from the action on a value). The typing rules for computations specify that the outcome of a computation is a negative type. All typing judgements specify that variables range over positive types. (These restrictions may always be met by appropriate use of shifting.)

The static semantics of positive values consists of the following rules:

\[
\begin{align*}
\Gamma, x : \tau^+ & \vdash x : \tau^+ & (A.1a) \\
\Gamma & \vdash z : \text{nat} & (A.1b) \\
\Gamma & \vdash v^+ : \text{nat} \\
\Gamma & \vdash s(v^+) : \text{nat} & (A.1c) \\
\Gamma & \vdash e : \tau^- \\
\Gamma & \vdash \text{del}^-(e) : \downarrow \tau^- & (A.1d)
\end{align*}
\]

Rule (A.1a) specifies that variables range over positive values. Rules (A.1b) and (A.1c) specify that the values of type nat are just the numerals. Rule (A.1d) specifies that a suspended computation (necessarily of negative type) is a positive value.

The static semantics of positive continuations consists of the following rules:

\[
\begin{align*}
\Gamma & \vdash e_0 : \gamma^- \\
\Gamma, x : \text{nat} & \vdash e_1 : \gamma^- \\
\Gamma & \vdash \text{ifz}(e_0; x. e_1) : \text{nat} > \gamma^- & (A.2a) \\
\Gamma & \vdash k^- : \tau^- > \gamma^- \\
\Gamma & \vdash \text{force}^-(k^-) : \downarrow \tau^- > \gamma^- & (A.2b)
\end{align*}
\]

Rule (A.2a) governs the continuation that chooses between two computations according to whether a natural number is zero or non-zero. Rule (A.2b)
specifies the continuation that forces a delayed computation with the specified negative continuation.

The static semantics of negative values is defined by these rules:

\[
\begin{align*}
\Gamma, x : \tau_1^+ \vdash e : \tau_2^- & \quad (A.3a) \\
\Gamma \vdash \lambda(x : \tau_1^+. e) : \tau_1^+ \rightarrow \tau_2^- \\
\Gamma \vdash v^+ : \tau^+ & \quad (A.3b) \\
\Gamma \vdash \text{del}^+(v^+) : \uparrow \tau^+ \\
\Gamma, x : \downarrow \tau^- \vdash v^- : \tau^- & \quad (A.3c) \\
\Gamma \vdash \text{fix} \ x \ is \ v^- : \tau^- 
\end{align*}
\]

Rule (A.3a) specifies the static semantics of a \( \lambda \)-abstraction whose argument is a positive value, and whose result is a computation of negative type. Rule (A.3b) specifies the inclusion of positive values as negative values. Rule (A.3c) specifies that negative types admit general recursion.

The static semantics of negative continuations is defined by these rules:

\[
\begin{align*}
\Gamma \vdash v_1^+ : \tau_1^+ & \quad (A.4a) \\
\Gamma \vdash k_2^+ : \tau_2^+ > \gamma^- \\
\Gamma \vdash \text{ap}(v_1^+; k_2^+) : \tau_1^+ \rightarrow \tau_2^+ > \gamma^- \\
\Gamma, x : \tau^+ \vdash e : \gamma^- & \quad (A.4b) \\
\Gamma \vdash \text{force}^+(x.e) : \uparrow \tau^+ > \gamma^- \\
\Gamma \vdash v^- : \tau^- & \quad (A.4c) \\
\Gamma \vdash \text{ret}(v^-) : \tau^- \\
\Gamma \vdash v^+ : \tau^+ & \quad (A.5a) \\
\Gamma \vdash k^+ : \tau^+ > \gamma^- \\
\Gamma \vdash v^+ > k^+ : \gamma^- \\
\Gamma \vdash v^- : \tau^- & \quad (A.5b) \\
\Gamma \vdash k^- : \tau^- > \gamma^- \\
\Gamma \vdash v^- > k^- : \gamma^- \\
\Gamma \vdash v^- > k^- : \gamma^- & \quad (A.5c)
\end{align*}
\]

Rule (A.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 (A.4b) specifies the continuation that passes a positive value, viewed as a negative value, to a computation.

The static semantics of computations is given by these rules:

\[
\begin{align*}
\Gamma \vdash v^- : \tau^- & \quad (A.5a) \\
\Gamma \vdash \text{ret}(v^-) : \tau^- \\
\Gamma \vdash v^+ : \tau^+ & \quad (A.5b) \\
\Gamma \vdash k^+ : \tau^+ > \gamma^- \\
\Gamma \vdash v^+ > k^+ : \gamma^- \\
\Gamma \vdash v^- : \tau^- & \quad (A.5c) \\
\Gamma \vdash k^- : \tau^- > \gamma^- \\
\Gamma \vdash v^- > k^- : \gamma^- 
\end{align*}
\]

Rule (A.5a) specifies the basic form of computation that simply returns the negative value \( v^- \). Rules (A.5b) and (A.5c) specify computations that pass a value to a continuation of appropriate polarity.
A.4 Dynamics

The dynamics of \( L^\pm \{ \text{nat} \arrow \} \) 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 dynamic semantics consists of the following rules:

\[
\begin{align*}
    z \triangleright \text{ifz}(e_0; x.e_1) & \mapsto e_0 \quad \text{(A.6a)} \\
    s(v^+) \triangleright \text{ifz}(e_0; x.e_1) & \mapsto [v^+/x]e_1 \quad \text{(A.6b)} \\
    \text{del}^-(e) \triangleright \text{force}^-(k^-) & \mapsto e; k^- \quad \text{(A.6c)} \\
    \lambda (x:\tau^+.e) \triangleright \text{ap}(v^+; k^-) & \mapsto [v^+/x]e; k^- \quad \text{(A.6d)} \\
    \text{del}^+(v^+) \triangleright \text{force}^+(x.e) & \mapsto [v^+/x]e \quad \text{(A.6e)} \\
    \text{fix} x \triangleright v^- & \mapsto [\text{del}^-(\text{fix} x \triangleright v^-)/x]v^- \triangleright k^- \quad \text{(A.6f)}
\end{align*}
\]

These rules specify the interaction between values and continuations.

Rules (A.6) make use of two forms of substitution, \([v^+/x]e\) and \([v^+/x]v^-\), which are defined as in Chapter 7. 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).

\[
\begin{align*}
    \text{ret}(v^-);k_0^- & = v^- \triangleright k_0^- \quad \text{(A.7a)} \\
    k^-;k_0^- & = k_1^- \\
    (v^- \triangleright k^-);k_0^- & = v^- \triangleright k_1^- \quad \text{(A.7b)} \\
    k^+;k_0^- & = k_1^+ \\
    (v^+ \triangleright k^+);k_0^- & = v^+ \triangleright k_1^+ \quad \text{(A.7c)} \\
    e_0;k^- & = e'_0 \mid e_1;k^- = e'_1 \\
    \text{ifz}(e_0; x.e_1);k^- & = \text{ifz}(e'_0; x.e'_1) \quad \text{(A.7d)}
\end{align*}
\]
\[
\begin{align*}
\text{force}^{-}(k^{-}) ; k^{-}_0 &= \text{force}^{-}(k^{-}_1) \\
A.7e \\
\text{ap}(v^+;k^{-}) ; k^{-}_0 &= \text{ap}(v^+;k^{-}_1) \\
A.7f \\
x \mid e; k_0^{-} &= e' \\
\text{force}^{+}(x.e) ; k_0^{-} &= \text{force}^{+}(x.e') \\
A.7g
\end{align*}
\]

Rules \((A.7d)\) and \((A.7g)\) make use of the parametric general judgement defined in Chapter 3 to express that the composition is defined uniformly in the bound variable.

### A.5 Safety

The proof of preservation for \(L^\pm \{\text{nat} \rightarrow \}\) reduces to the proof of the typing properties of substitution and composition.

**Lemma A.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. \(\square\)

**Lemma A.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 \(\Gamma \vdash k^-_0 ; k^-_1 : \tau^- > \gamma^-_1\).
A.6 Definability

Proof. Simultaneously, by induction on the derivations of the first premises of each clause of the lemma.

Theorem A.3 (Preservation). If $\Gamma \vdash e : \gamma^-$ and $e \rightarrow e'$, then $\Gamma \vdash e' : \gamma^-$.  

Proof. By induction on transition, appealing to inversion for typing and Lemmas A.1 on the preceding page and A.2 on the facing page.

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 A.4 (Progress). If $\Gamma \vdash e : \gamma^-$, then either $e = \text{ret}(v^-)$ for some $v^-$, or there exists $e'$ such that $e \rightarrow e'$.

A.6 Definability

The syntax of $L^\pm\{\text{nat} \rightarrow\}$ 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!

This chapter under construction . . . .

A.7 Exercises