Concepts, Techniques, and Models
of Computer Programming

with Practical Applications in
Distributed Computing and Intelligent Agents

PETER VAN ROY¹
Université catholique de Louvain, Belgium
Swedish Institute of Computer Science, Sweden

SEIF HARIDI²
Royal Institute of Technology (KTH), Sweden
Swedish Institute of Computer Science, Sweden

August 21, 2001

¹Email: pvr@info.ucl.ac.be, Web: http://www.info.ucl.ac.be/~pvr
²Email: seif@it.kth.se, Web: http://www.it.kth.se/~seif
WARNING

---

THIS MATERIAL IS CHANGING RAPIDLY AND SHOULD NOT BE TAKEN AS DEFINITIVE

---

ALL COMMENTS WELCOME
Contents

List of Figures xiv
List of Tables xix
Abstract xxii
Preface xxiii

I Introduction 1

1 Goals and Organization 3

1.1 Goals of the book ........................................ 5
  1.1.1 Programming as an engineering discipline ........ 5
  1.1.2 Designing abstractions ................................ 7
  1.1.3 Distributed systems and intelligent agents .......... 7

1.2 Main features ............................................ 7

  1.2.1 Formalism used ...................................... 7
  1.2.2 Uniform structuring principles ...................... 8
  1.2.3 Dataflow computation ................................ 10
  1.2.4 Computation models and expressiveness ............ 10
  1.2.5 Relevance of distribution and intelligence ........ 12

1.3 How to read this book .................................... 13

  1.3.1 Core chapters ...................................... 15
  1.3.2 Extension chapters ................................... 16
  1.3.3 Semantics .......................................... 17
  1.3.4 Appendices .......................................... 18

1.4 Computation models covered .............................. 18

  1.4.1 General computation models ......................... 18
  1.4.2 Extended computation models ......................... 19
  1.4.3 Using the models .................................... 21

1.5 How to teach from this book ................................ 21

  1.5.1 Courses ........................................... 22
  1.5.2 Software ........................................... 23
## CONTENTS

2 Example Applications ........................................... 25
  2.1 A bouncing ball ............................................ 25
    2.1.1 The architecture .................................... 26
    2.1.2 Concepts and techniques used ....................... 27
  2.2 A chat room .................................................. 28
    2.2.1 The architecture .................................... 29
    2.2.2 Making it open and distributed .................... 30
    2.2.3 More on fault tolerance .............................. 30
    2.2.4 Concepts and techniques used ....................... 31
  2.3 An intelligent minesweeper .............................. 32
    2.3.1 Description .......................................... 32
    2.3.2 The architecture .................................... 33
    2.3.3 Playing the game .................................... 34
    2.3.4 Concepts and techniques used ....................... 35
  2.4 Exercises .................................................... 35

3 Introduction to Programming .................................. 37
  3.1 A calculator ............................................... 37
  3.2 Variables ................................................... 38
  3.3 Functions .................................................. 38
  3.4 Lists ....................................................... 40
  3.5 Functions over lists ..................................... 41
  3.6 Correctness ................................................ 43
  3.7 Complexity ................................................ 44
  3.8 Lazy evaluation ............................................ 46
  3.9 Higher-order programming ................................ 47
  3.10 Concurrency ............................................... 49
  3.11 Dataflow .................................................. 49
  3.12 State ...................................................... 50
  3.13 Nondeterminism ......................................... 52
  3.14 Atomicity ................................................ 53
  3.15 Exercises .................................................. 54

II General Computation Models .................................. 57

4 Declarative Computation Model ............................... 59
  4.1 Defining practical programming languages ............... 60
    4.1.1 Language syntax ..................................... 61
    4.1.2 Language semantics ................................ 64
  4.2 The declarative computation model ....................... 69
    4.2.1 Single-assignment store and dataflow variables ....... 69
    4.2.2 Kernel language ...................................... 71
    4.2.3 Computations ........................................ 72

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
CONTENTS

4.2.4 Environments ................................................. 72
4.2.5 Exceptions ............................................... 73
4.3 Basic concepts .................................................. 73
  4.3.1 Variable declarations and environments ................. 73
  4.3.2 The interactive interface ................................ 74
  4.3.3 Basic data types and the type hierarchy ............... 75
  4.3.4 Expressions ............................................. 77
  4.3.5 Dynamic typing ......................................... 77
  4.3.6 Binding and testing variables .......................... 79
  4.3.7 Dataflow suspension .................................... 81
  4.3.8 Implicit variable initialization ........................ 81
4.4 Language constructs ........................................... 84
  4.4.1 Basic statements ....................................... 85
  4.4.2 Lexical scoping ......................................... 85
  4.4.3 The skip statement .................................... 86
  4.4.4 The if statement ....................................... 87
  4.4.5 Procedures ............................................. 88
  4.4.6 Procedural abstraction ................................ 91
  4.4.7 Anonymous procedures ................................. 93
  4.4.8 Nesting notation ....................................... 93
  4.4.9 Functions ................................................ 95
  4.4.10 The case statement ................................... 99
  4.4.11 Exception handling (the try and raise statements) .... 103
  4.4.12 Environment trimming and last call optimization ...... 105
  4.4.13 Binding and testing variables (general case) ........ 106
4.5 Exercises ...................................................... 110

5 Declarative Programming Techniques .......................... 113
  5.1 What is declarativeness? .................................... 115
    5.1.1 Specification languages ................................ 117
    5.1.2 Implementing components in the declarative model ... 118
    5.1.3 The richness of declarative programming ............ 118
  5.2 Computation models and program structure .................. 119
    5.2.1 Picking the right model ................................ 121
    5.2.2 Using different models together ...................... 124
  5.3 Iterative computation ....................................... 125
    5.3.1 Iteration with numbers ................................ 126
    5.3.2 Using local procedures ............................... 126
    5.3.3 General pattern of iteration .......................... 129
    5.3.4 Last call optimization ............................... 133
  5.4 Recursive computation ....................................... 134
  5.5 Thinking inductively ....................................... 136
  5.6 Thinking about efficiency .................................. 137
  5.7 Records, lists, and trees .................................. 139

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.7.1 Programming with lists</td>
<td>139</td>
</tr>
<tr>
<td>5.7.2 Parsing with an accumulating parameter</td>
<td>155</td>
</tr>
<tr>
<td>5.7.3 Programming with difference lists</td>
<td>159</td>
</tr>
<tr>
<td>5.7.4 Programming with trees</td>
<td>164</td>
</tr>
<tr>
<td>5.7.5 Drawing trees</td>
<td>165</td>
</tr>
<tr>
<td>5.7.6 The declarative \texttt{for} loop</td>
<td>170</td>
</tr>
<tr>
<td>5.8 Higher-order programming</td>
<td>172</td>
</tr>
<tr>
<td>5.8.1 Defining generic procedures</td>
<td>173</td>
</tr>
<tr>
<td>5.8.2 Traversing and folding trees</td>
<td>175</td>
</tr>
<tr>
<td>5.8.3 New control structures</td>
<td>177</td>
</tr>
<tr>
<td>5.8.4 New control structures (with delayed evaluation)</td>
<td>178</td>
</tr>
<tr>
<td>5.8.5 Lazy execution with programmed triggers</td>
<td>178</td>
</tr>
<tr>
<td>5.8.6 Currying</td>
<td>180</td>
</tr>
<tr>
<td>5.8.7 More higher-order programming</td>
<td>181</td>
</tr>
<tr>
<td>5.9 User-defined data types</td>
<td>182</td>
</tr>
<tr>
<td>5.9.1 Concrete data types</td>
<td>182</td>
</tr>
<tr>
<td>5.9.2 Abstract data types</td>
<td>184</td>
</tr>
<tr>
<td>5.9.3 Declarative game boards</td>
<td>187</td>
</tr>
<tr>
<td>5.10 Security</td>
<td>187</td>
</tr>
<tr>
<td>5.10.1 Basic concepts</td>
<td>189</td>
</tr>
<tr>
<td>5.10.2 Language properties</td>
<td>189</td>
</tr>
<tr>
<td>5.10.3 Implementing the basic concepts</td>
<td>190</td>
</tr>
<tr>
<td>5.10.4 Capabilities</td>
<td>193</td>
</tr>
<tr>
<td>5.10.5 Programming secure applications</td>
<td>193</td>
</tr>
<tr>
<td>5.11 Relation to other declarative models</td>
<td>196</td>
</tr>
<tr>
<td>5.11.1 Relation to logic programming</td>
<td>196</td>
</tr>
<tr>
<td>5.11.2 Relation to functional programming</td>
<td>199</td>
</tr>
<tr>
<td>5.11.3 Discussion</td>
<td>201</td>
</tr>
<tr>
<td>5.12 Nondeclarative needs</td>
<td>201</td>
</tr>
<tr>
<td>5.12.1 Programming with exceptions</td>
<td>201</td>
</tr>
<tr>
<td>5.12.2 Modules and standalone compilation</td>
<td>203</td>
</tr>
<tr>
<td>5.12.3 Text input/output with files</td>
<td>207</td>
</tr>
<tr>
<td>5.12.4 Text input/output with interactive windows</td>
<td>209</td>
</tr>
<tr>
<td>5.12.5 Stateless data input/output with files</td>
<td>211</td>
</tr>
<tr>
<td>5.13 More on efficiency</td>
<td>213</td>
</tr>
<tr>
<td>5.13.1 Reflections on optimization</td>
<td>213</td>
</tr>
<tr>
<td>5.13.2 Memory management</td>
<td>214</td>
</tr>
<tr>
<td>5.13.3 Garbage collection is not magic</td>
<td>216</td>
</tr>
<tr>
<td>5.14 Limitations of declarative programming</td>
<td>218</td>
</tr>
<tr>
<td>5.15 Exercises</td>
<td>219</td>
</tr>
<tr>
<td>5.16 Perspectives</td>
<td>222</td>
</tr>
</tbody>
</table>

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
## CONTENTS

### 6 Declarative Concurrency

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>The declarative model with concurrency</td>
<td>223</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Threads</td>
<td>224</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Nondeterminism</td>
<td>226</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Exceptions</td>
<td>227</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Triggers</td>
<td>229</td>
</tr>
<tr>
<td>6.1.5</td>
<td>The model’s “systolic” character</td>
<td>230</td>
</tr>
<tr>
<td>6.2</td>
<td>Threads</td>
<td>231</td>
</tr>
<tr>
<td>6.2.1</td>
<td>The <strong>thread</strong> statement</td>
<td>232</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Threads and the browser</td>
<td>233</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Dataflow computation with threads</td>
<td>234</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Thread scheduling and priority</td>
<td>236</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Operations on threads</td>
<td>238</td>
</tr>
<tr>
<td>6.3</td>
<td>Order-determining concurrency</td>
<td>240</td>
</tr>
<tr>
<td>6.4</td>
<td>Synchronization</td>
<td>241</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Usefulness of dataflow execution</td>
<td>243</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Futures and I-structures</td>
<td>244</td>
</tr>
<tr>
<td>6.5</td>
<td>Streams</td>
<td>245</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Eager stream communication</td>
<td>246</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Using thread priorities to tune resource consumption</td>
<td>248</td>
</tr>
<tr>
<td>6.5.3</td>
<td>Managing resources and improving throughput</td>
<td>249</td>
</tr>
<tr>
<td>6.5.4</td>
<td>Skipping over stream elements</td>
<td>250</td>
</tr>
<tr>
<td>6.5.5</td>
<td>“Active objects” with streams</td>
<td>252</td>
</tr>
<tr>
<td>6.5.6</td>
<td>Limitations of stream communication</td>
<td>253</td>
</tr>
<tr>
<td>6.6</td>
<td>Lazy execution</td>
<td>254</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Lazy stream communication</td>
<td>256</td>
</tr>
<tr>
<td>6.6.2</td>
<td>Improving throughput</td>
<td>258</td>
</tr>
<tr>
<td>6.6.3</td>
<td>Lazy reading of a file</td>
<td>259</td>
</tr>
<tr>
<td>6.6.4</td>
<td>The Hamming problem</td>
<td>261</td>
</tr>
<tr>
<td>6.6.5</td>
<td>List operations</td>
<td>262</td>
</tr>
<tr>
<td>6.6.6</td>
<td>List comprehensions</td>
<td>264</td>
</tr>
<tr>
<td>6.6.7</td>
<td>By-need synchronization</td>
<td>265</td>
</tr>
<tr>
<td>6.7</td>
<td>Other programming techniques</td>
<td>266</td>
</tr>
<tr>
<td>6.7.1</td>
<td>Coroutines</td>
<td>270</td>
</tr>
<tr>
<td>6.7.2</td>
<td>Termination detection</td>
<td>271</td>
</tr>
<tr>
<td>6.7.3</td>
<td>Concurrent composition</td>
<td>272</td>
</tr>
<tr>
<td>6.8</td>
<td>Soft real-time programming</td>
<td>272</td>
</tr>
<tr>
<td>6.8.1</td>
<td>Basic operations</td>
<td>272</td>
</tr>
<tr>
<td>6.8.2</td>
<td>Ticking</td>
<td>274</td>
</tr>
<tr>
<td>6.9</td>
<td>Exercises</td>
<td>277</td>
</tr>
</tbody>
</table>

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
7 Relational Programming 281
  7.1 The declarative model with nondeterminism ............... 282
  7.2 Simple examples .................................. 284
  7.3 Programming techniques ................................ 285
    7.3.1 Generate-and-test .............................. 286
    7.3.2 Elaborations of generate-and-test .................. 287
    7.3.3 Database queries ................................ 287
    7.3.4 Parsing ambiguous grammars ........................ 287
    7.3.5 Type checking .................................. 288
    7.3.6 Lazy execution .................................. 288
  7.4 Case studies ........................................ 288
    7.4.1 The $n$-queens problem ............................ 288
  7.5 Exercises .......................................... 290

8 Encapsulated State 291
  8.1 What is state? ...................................... 293
    8.1.1 Declarative state ................................ 293
    8.1.2 Encapsulated state ................................ 294
  8.2 State and system building ............................. 295
    8.2.1 System properties ............................... 296
    8.2.2 Component-based programming ....................... 296
    8.2.3 Object-oriented programming ....................... 297
    8.2.4 The importance of libraries ....................... 297
  8.3 The declarative model with explicit state ................. 298
    8.3.1 Cells ........................................... 298
    8.3.2 Relation to declarative programming ............... 300
  8.4 Abstract data types .................................. 302
    8.4.1 Different ADT structures .......................... 302
    8.4.2 Encapsulation with chunks ........................ 304
    8.4.3 Security revisited ................................ 304
  8.5 Indexed collections .................................. 305
    8.5.1 Arrays .......................................... 305
    8.5.2 Dictionaries .................................... 306
    8.5.3 Choosing a collection ............................. 307
  8.6 Iterative computation ................................. 309
    8.6.1 Iterators ........................................ 309
  8.7 Thinking about state .................................. 309
  8.8 Programming with state ................................ 311
    8.8.1 Stateful versus declarative programming .......... 311
    8.8.2 Array algorithms ................................ 311
    8.8.3 Hash tables ..................................... 311
    8.8.4 Transitive closure ................................ 311
  8.9 Component-based programming and system design ............. 317
    8.9.1 Components in general ............................ 318

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
8.9.2 System decomposition and computation models .......... 318
8.9.3 What happens at the interface .......................... 319
8.9.4 System structure ........................................ 324
8.9.5 Development methodology ............................... 326
8.9.6 Maintainable systems .................................... 328
8.10 Case studies ............................................... 329
  8.10.1 Word frequencies (with stateful dictionary) .......... 329
  8.10.2 Generating random numbers ............... 331
  8.10.3 “Word of Mouth” simulation ......................... 335
  8.10.4 Swiftfile .............................................. 338
  8.10.5 Forward chaining rule-based system .................. 338
8.11 Limitations of stateful programming .................... 338
  8.11.1 The real world is parallel ......................... 338
  8.11.2 The real world is distributed ....................... 338
8.12 Exercises .................................................. 339

9 Object-Oriented Programming ............................. 341
  9.1 Motivations ................................................. 342
    9.1.1 Inheritance ........................................... 342
    9.1.2 Encapsulated state and inheritance ................. 343
    9.1.3 Objects and classes ................................ 344
  9.2 Classes as complete ADTs .................................. 345
    9.2.1 Examples .............................................. 345
    9.2.2 Defining classes ...................................... 346
    9.2.3 Initializing attributes and features ................ 349
    9.2.4 First-class messages .................................. 350
    9.2.5 First-class attributes and features ................ 352
  9.3 Classes as incremental ADTs ............................. 354
    9.3.1 Inheritance ........................................... 354
    9.3.2 Static and dynamic binding .......................... 356
    9.3.3 Controlling visibility ............................... 356
    9.3.4 Delegation ............................................ 361
    9.3.5 Checking class membership ........................... 362
  9.4 Principal programming techniques ....................... 363
    9.4.1 Parameterized classes ................................ 364
    9.4.2 Multiple inheritance .................................. 365
    9.4.3 Rules of thumb for multiple inheritance .......... 372
    9.4.4 Limitations of class diagrams ...................... 372
  9.5 Relation to other computation models .................... 373
    9.5.1 Object-based and component-based programming .... 373
    9.5.2 Higher-order programming ............................ 373
    9.5.3 Should everything be an object? .................... 376
  9.6 Constructing the object system ......................... 379
    9.6.1 Abstraction diagram ................................... 379
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.6.2 Constructing classes</td>
<td>381</td>
</tr>
<tr>
<td>9.6.3 Constructing objects</td>
<td>383</td>
</tr>
<tr>
<td>9.6.4 Constructing and instantiating metaclasses</td>
<td>383</td>
</tr>
<tr>
<td>9.6.5 Stateful classes</td>
<td>384</td>
</tr>
<tr>
<td>9.7 Case studies</td>
<td>384</td>
</tr>
<tr>
<td>9.8 Exercises</td>
<td>384</td>
</tr>
<tr>
<td>10 Concurrency and State</td>
<td>385</td>
</tr>
<tr>
<td>10.1 The declarative model with concurrency and state</td>
<td>389</td>
</tr>
<tr>
<td>10.2 Approaches to programming with concurrency</td>
<td>389</td>
</tr>
<tr>
<td>10.2.1 Four practical approaches</td>
<td>389</td>
</tr>
<tr>
<td>10.2.2 Implementing coroutines</td>
<td>391</td>
</tr>
<tr>
<td>10.2.3 The nondeterministic concurrent model</td>
<td>392</td>
</tr>
<tr>
<td>10.3 Message passing</td>
<td>396</td>
</tr>
<tr>
<td>10.3.1 Asynchronous communication with ports</td>
<td>396</td>
</tr>
<tr>
<td>10.3.2 A simple active object with ports</td>
<td>398</td>
</tr>
<tr>
<td>10.3.3 A concurrent queue with ports</td>
<td>399</td>
</tr>
<tr>
<td>10.3.4 Active objects</td>
<td>401</td>
</tr>
<tr>
<td>10.3.5 Networks of active objects</td>
<td>402</td>
</tr>
<tr>
<td>10.3.6 Event manager with active objects</td>
<td>402</td>
</tr>
<tr>
<td>10.3.7 Active objects sharing one thread</td>
<td>407</td>
</tr>
<tr>
<td>10.3.8 A thread abstraction with termination detection</td>
<td>408</td>
</tr>
<tr>
<td>10.3.9 Eliminating sequential dependencies</td>
<td>411</td>
</tr>
<tr>
<td>10.4 Atomic actions</td>
<td>412</td>
</tr>
<tr>
<td>10.4.1 Simulating a slow network</td>
<td>412</td>
</tr>
<tr>
<td>10.4.2 Locks</td>
<td>414</td>
</tr>
<tr>
<td>10.4.3 Implementing locks</td>
<td>414</td>
</tr>
<tr>
<td>10.4.4 Using locks to build concurrent stateful ADTs</td>
<td>417</td>
</tr>
<tr>
<td>10.4.5 A concurrent stack and queue</td>
<td>418</td>
</tr>
<tr>
<td>10.4.6 Concurrent object-oriented programming</td>
<td>420</td>
</tr>
<tr>
<td>10.4.7 Tuple spaces (&quot;Linda&quot;)</td>
<td>420</td>
</tr>
<tr>
<td>10.4.8 Monitors</td>
<td>423</td>
</tr>
<tr>
<td>10.4.9 Implementing monitors</td>
<td>424</td>
</tr>
<tr>
<td>10.4.10 Bounded buffers</td>
<td>426</td>
</tr>
<tr>
<td>10.5 Threaded state and named state</td>
<td>426</td>
</tr>
<tr>
<td>10.6 Memory management</td>
<td>429</td>
</tr>
<tr>
<td>10.6.1 Finalization</td>
<td>429</td>
</tr>
<tr>
<td>10.6.2 Laziness and external resources</td>
<td>430</td>
</tr>
<tr>
<td>10.7 Case studies</td>
<td>431</td>
</tr>
<tr>
<td>10.7.1 A bouncing ball application</td>
<td>431</td>
</tr>
<tr>
<td>10.7.2 Multi-agent systems</td>
<td>438</td>
</tr>
<tr>
<td>10.7.3 SugarScape</td>
<td>439</td>
</tr>
<tr>
<td>10.8 Exercises</td>
<td>440</td>
</tr>
</tbody>
</table>

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
### 11 Representative Languages

11.1 Erlang and concurrent programming ........................................... 446
   11.1.1 Computation model ..................................................... 446
   11.1.2 The `receive` operation ............................................... 447
11.2 Haskell and functional programming ........................................ 450
   11.2.1 Computation model ..................................................... 451
11.3 Java and object-oriented programming ..................................... 451
   11.3.1 The Java language ..................................................... 451
   11.3.2 Computation model ..................................................... 451
11.4 Prolog and logic programming ............................................... 452
   11.4.1 Context ................................................................. 453
   11.4.2 The Prolog language .................................................. 453
   11.4.3 Translating Prolog into a relational program ...................... 454
11.5 Exercises .............................................................................. 458

### III Specialized Computation Models

12 Graphic User Interface Programming ........................................... 461
13 Distributed Programming .......................................................... 463
14 Constraint Programming .................................................................. 465

### IV Semantics

15 Language Semantics ..................................................................... 469
   15.1 Historical background ........................................................ 469
   15.2 The concurrent stateful model .............................................. 470
      15.2.1 Store ........................................................................... 470
      15.2.2 Abstract syntax ........................................................ 471
      15.2.3 Structural rules .......................................................... 472
      15.2.4 Sequential and concurrent composition .......................... 473
      15.2.5 Variable introduction .................................................. 473
      15.2.6 Imposing equality (tell) .............................................. 474
      15.2.7 Conditional statements (ask) ....................................... 477
      15.2.8 Procedural abstraction ................................................. 478
      15.2.9 Encapsulated state ..................................................... 479
      15.2.10 By-need synchronization .............................................. 480
      15.2.11 Exception handling .................................................... 486
      15.2.12 Variable substitution .................................................. 488
   15.3 Eight computation models ..................................................... 489
   15.4 Kernel extensions .................................................................... 491
      15.4.1 Names ........................................................................... 491
CONTENTS

E Failure Model 525

F Mozart System Properties and Limitations 527
  F.1 Memory properties and limitations ............................................. 527
    F.1.1 Maximum memory size ......................................................... 528
    F.1.2 Garbage collection .............................................................. 528
    F.1.3 Compiler memory usage ......................................................... 529
    F.1.4 Loaded modules are permanent ............................................. 529
    F.1.5 Environment trimming .......................................................... 529
  F.2 Numeric limitations ................................................................. 530
  F.3 Distribution limitations and modifications ............................... 531
    F.3.1 Performance limitations ...................................................... 531
    F.3.2 Functionality limitations .................................................... 531
    F.3.3 Modification ................................................................. 532

Bibliography 532

Index 540
List of Figures

1.1 Road map of the book ........................................ 14
1.2 The main computation models and how they are related ...... 20

2.1 The bouncing ball application .................................. 26
2.2 Architecture of the bouncing ball application ................ 27
2.3 The chat room application ...................................... 28
2.4 Architecture of the chat room application .................... 30
2.5 Handling a server crash ........................................ 31
2.6 The minesweeper application ................................... 33
2.7 Architecture of the minesweeper application ................ 34

3.1 Calculating the fifth row of Pascal’s triangle .................. 41
3.2 A simple example of dataflow execution ....................... 50
3.3 All possible executions of the first nondeterministic example . 52
3.4 A possible execution of the second nondeterministic example . 53

4.1 The context-free approach to language syntax .................. 62
4.2 The kernel approach to language semantics .................... 65
4.3 Translation approaches to language semantics .................. 68
4.4 The declarative computation model ............................ 69
4.5 The type hierarchy ............................................ 76
4.6 The Browser .................................................. 79
4.7 Tree insertion using the if statement ............................ 99
4.8 Tree insertion using the case statement ......................... 100
4.9 Merging sorted lists using the case statement ................. 102

5.1 Structure of the chapter ......................................... 114
5.2 A classification of declarative programming ................... 115
5.3 A declarative operation inside a general computation .......... 120
5.4 Finding roots using Newton’s method (first version) .......... 125
5.5 Finding roots using Newton’s method (second version) ........ 127
5.6 Finding roots using Newton’s method (third version) .......... 128
5.7 Finding roots using Newton’s method (fourth version) ........ 129
5.8 Finding roots using Newton’s method (fifth version) .......... 130
5.9 Sorting with mrgesort ......................................... 146
5.10 Declarative dictionary (with linear list) ....................... 152

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
5.11 Word frequencies (with declarative dictionary) ............. 153
5.12 Internal structure of binary tree dictionary in WordFreq (partial) 154
5.13 Declarative dictionary (with binary tree) ...................... 166
5.14 Tree drawing constraints ........................................ 167
5.15 An example tree .................................................. 167
5.16 Tree drawing algorithm ........................................... 169
5.17 The example tree displayed with the tree drawing algorithm .... 170
5.18 The For iterator .................................................... 177
5.19 Checking a binary tree ............................................. 178
5.20 Checking a binary tree (with delayed evaluation) ............. 179
5.21 Checking a binary tree lazily (using andthen) .................. 179
5.22 A simple graphical I/O interface for text ....................... 210

6.1 The concurrent declarative model ............................... 225
6.2 Causal orders of sequential and concurrent executions ....... 227
6.3 A concurrent map function ....................................... 234
6.4 A concurrent Fibonacci function ................................. 234
6.5 Thread creations for the call (Fib 6) ............................ 235
6.6 The Oz Panel showing thread creation in (Fib 26 X) ........... 236
6.7 Operations on threads ............................................. 239
6.8 Tree drawing algorithm with order-determining concurrency . 240
6.9 Pipeline of active objects implementing a sieve ................ 253
6.10 How can two clients send to the same server? They cannot! .. 254
6.11 Lazy buffer (naive definition) ................................... 259
6.12 Lazy buffer (correct definition for infinite lists) ............ 259
6.13 Lazy buffer (correct definition for finite lists) ............... 260
6.14 The by-need synchronization protocol ......................... 268
6.15 Procedures, coroutines, and threads ............................ 270
6.16 Concurrent composition ........................................... 272
6.17 A simple ’Ping Pong’ program .................................. 273
6.18 A standalone ’Ping Pong’ program ............................... 274
6.19 A standalone ’Ping Pong’ program that exits cleanly .......... 275

7.1 Two digit counting with depth-first search ...................... 284
7.2 The n-queens problem (when n = 4) ............................ 289
7.3 Solving the n-queens problem with relational programming .. 290

8.1 The declarative model with explicit state ....................... 299
8.2 Choosing an indexed collection .................................. 307
8.3 Transitive closure (first declarative version) .................. 313
8.4 Transitive closure (stateful version) ............................ 314
8.5 Transitive closure (second declarative version) ................. 315
8.6 A directed graph and its transitive closure ..................... 315
8.7 An application structured as a hierarchical graph ............ 317
8.8 Example of interfacing different execution models ............. 320

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
8.9 Application structure – static and dynamic ............... 325
8.10 Word frequencies (with stateful dictionary) .......... 330

9.1 An example of a class: Counter ............................ 344
9.2 Illegal and legal class hierarchies ......................... 353
9.3 The meaning of “private” .................................. 357
9.4 A parameterized sorting class .............................. 364
9.5 Class diagram of the graphics package ................. 367
9.6 Drawing in the graphics package ......................... 369
9.7 Class diagram with an association ........................ 370
9.8 Queue (object-oriented version with stateless list) ...... 377
9.9 Queue (object-oriented version with object list) ........ 378
9.10 Abstractions in object-oriented programming .......... 381
9.11 An example of class construction ....................... 382
9.12 An example of object construction ..................... 383

10.1 The declarative model with concurrency and state .................. 388
10.2 Implementing coroutines ................................... 392
10.3 Connecting two clients using a stream merger ............. 394
10.4 Implementing nondeterministic choice (naive version) ...... 396
10.5 Implementing nondeterministic choice (full version) .... 397
10.6 Queue (naive version with ports) .......................... 399
10.7 Queue (correct version with ports) ....................... 401
10.8 Event manager with active objects ....................... 403
10.9 Adding functionality with inheritance .................... 404
10.10 Batching a list of messages and procedures ............. 405
10.11 Active objects sharing one thread ....................... 407
10.12 Screenshot of the Ping-Pong application .................. 408
10.13 The Ping-Pong application: active objects in one thread .... 409
10.14 A thread abstraction with termination detection ........ 410
10.15 Lock (non-entrant version without exception handling) .... 415
10.16 Lock (non-entrant version with exception handling) ...... 415
10.17 Lock (reentrant version with exception handling) ........ 416
10.18 Queue (declarative version) ............................. 417
10.19 Queue (sequential stateful version) ...................... 418
10.20 Queue (concurrent stateful version with lock) .......... 419
10.21 Queue (concurrent object-oriented version with lock) .... 420
10.22 Queue (concurrent stateful version with exchange) ...... 421
10.23 Tuple space (object-oriented version) .................... 422
10.24 Monitor (naive version) .................................. 425
10.25 Monitor (full version) .................................... 427
10.26 Architecture of the bouncing ball application ......... 431

15.1 The kernel language of the concurrent stateful model .......... 471

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
B.1 Graph representation of the infinite list c1=a|b|c1 ........ 511
# List of Tables

4.1 The declarative kernel language .......................... 72
4.2 The *local* statement for declaring variables .......... 73
4.3 The syntax of interactive statements .................. 74
4.4 Expressions for calculating with numbers .............. 76
4.5 The equality and equality test operations ............. 79
4.6 Variable-declaration equality with escaped variables .. 82
4.7 Basic statements ................................... 84
4.8 The *skip* and *if* statements ........................ 87
4.9 Procedures ......................................... 88
4.10 Anonymous procedures .............................. 92
4.11 Procedural and functional nesting .................... 93
4.12 Functions ......................................... 95
4.13 The *andthen* and *orelse* operators ................. 96
4.14 Basic form of the *case* statement ................... 99
4.15 The *case* statement ................................ 101
4.16 Exceptions ......................................... 102
4.17 Equality (unification) and equality test (entailment check) ........ 106

5.1 A descriptive declarative kernel language .............. 116
5.2 The parser's input language (which is a token sequence) .. 156
5.3 The parser's output language (which is a value) .......... 157
5.4 Types relevant to security ............................ 190
5.5 Translating a declarative statement to logic .......... 197
5.6 Functors ........................................... 204

6.1 The concurrent declarative kernel language ............ 225
6.2 Correspondence between variable and communications channel .. 231
6.3 Threads ............................................ 232
6.4 Classifying synchronization ........................... 242

7.1 The *choice* statement ................................ 282

8.1 The kernel language with explicit state ............... 299
8.2 Cell operations ..................................... 299

9.1 Classes ............................................. 347

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
10.1 The kernel language with concurrency and state .................. 388
10.2 The \texttt{waitor} operation: nondeterministic choice .......... 394
10.3 Port operations ............................................. 396
15.1 Eight computation models ...................................... 490

B.1 Characters \textit{(character syntax)} .................................. 502
B.2 Operations on numbers ........................................... 503
B.3 Some character operations ....................................... 504
B.4 Literals .......................................................... 504
B.5 Atoms \textit{(character syntax)} ..................................... 505
B.6 Some atom operations ............................................. 506
B.7 Records and tuples ............................................... 506
B.8 Some record operations .......................................... 508
B.9 Some tuple operations ............................................ 509
B.10 Lists ............................................................. 509
B.11 Some list operations ............................................. 510
B.12 Strings \textit{(character syntax)} ................................... 512
B.13 Some virtual string operations ................................. 512

C.1 Interactive statements ............................................. 516
C.2 Statements and expressions ........................................ 516
C.3 Nestable constructs (no declarations) ............................ 517
C.4 Nestable declarations ............................................. 517
C.5 Terms and patterns ................................................. 518
C.6 Other nonterminals needed for statements and expressions .... 519
C.7 Operators with precedence and associativity ..................... 520
C.8 Keywords .......................................................... 520
C.9 Variables, atoms, strings, and characters \textit{(character syntax)} . 520
C.10 Nonterminals needed for character syntax ........................ 521
C.11 Integers and floating point numbers \textit{(character syntax)} ...... 521

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Abstract

This book gives a broad and deep view of practical computer programming as a unified engineering discipline based on a sound scientific foundation. It brings the student a comprehensive and up-to-date presentation of all major programming concepts and techniques. The concepts and techniques are organized into computation models, or less accurately, programming paradigms. We define the precise concept of computation model to capture the intuitive concept of programming paradigm. The models are situated in a uniform framework that has a complete and simple formal semantics. We examine the relationships between the models and show how and why to use different models together in the same program.

The simplest computation models cover the domain of declarative programming, which includes logic programming (deterministic and nondeterministic) and functional programming (strict and lazy). We add explicit state, leading to component-based programming and object-oriented programming. We show the different ways to use concurrency, including declarative concurrency, message passing, and atomic actions. We explain the models of the languages Erlang, Haskell, Java, and Prolog. We then present three specialized models of intrinsic interest, for user interface programming, dependable distributed programming, and constraint programming.

The book is suitable for undergraduate and graduate courses in programming techniques, programming models, constraint programming, distributed programming, and semantics. It emphasizes scalable techniques useful in real programs. All models are fully implemented for practical programming. There is an accompanying Open Source software development package, the Mozart Programming System, that can run all the program fragments in the text. The package has an interactive incremental development environment and a production-quality implementation for Unix and Windows platforms.

The book and software package are the fruits of a research collaboration by the Mozart Consortium, which groups the Swedish Institute of Computer Science (SICS) in Stockholm, Sweden, the Universität des Saarlandes in Saarbrücken, Germany, the Université catholique de Louvain (UCL) in Louvain-la-Neuve, Belgium, and related institutions. Both authors have been involved in this collaboration since its beginning. Peter Van Roy is professor in the Department of Computing Science and Engineering at UCL and part-time SICS member. Seif Haridi is professor in the Department of Teleinformatics at the Royal Institute of Technology (KTH), Stockholm, Sweden, and SICS chief scientific advisor.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Preface

“... although I have searched every library in Europe for the works of Doctor Tarr and Professor Fether, I have, up to the present day, utterly failed in my endeavors to procure a copy.”
– The System of Doctor Tarr and Professor Fether, Edgar Allan Poe (1809–1849)

This is not a programming language book, although it gives insights about many languages. This is not a programming techniques book, although it teaches many techniques. Rather, this is a programming models book: we present many different models of computation, what you can do with them, how they are related, and how they can be used together in the same program. Whole books have been written about each of these models. We try to give the essentials of each in just a chapter or two, from a practical rather than theoretical point of view. It turns out that the models have surprisingly many things in common with each other, so that one chapter per model really does suffice to explain what is “special” about the model.

It is our experience that students who learn about these models and how they fit together become better programmers and better scientists. The current trend is to restrict a student’s education to one or two models. The most extreme case is where a single, rather complex model, namely object-oriented programming, is used as a general-purpose tool with which all problems should be solved. Everything that might be needed is put in this model: abstract and concrete classes, interfaces, static and dynamic binding, final classes, concurrency, monitors, and synchronized methods, public, private, and protected attributes, single and multiple inheritance, and so forth. All design techniques are tailored towards this one model, almost to the exclusion of all else. In our view, this is fundamentally wrong in at least three ways. It makes understanding programming concepts harder, it makes programs harder to write, and it makes programs harder to reason about.

We find that it is much better for the model to have a small well-factored set of concepts and a few powerful ways to combine them. Programming is largely the invention of new abstractions. The model should make it easy to build new abstractions from existing ones. There are many advantages to this approach: programs are simpler to understand, more general, and easier to reason about.
This is not a new idea—it has been used before in computer science and in many other disciplines. The novelty of this book is the breadth and depth with which it is applied. The choice of which concepts and ways of combining to take is important. This book presents one such choice; we call the most complete model that results the unified model. We present many other useful models that are stepping stones to the unified model.

In this view, object-oriented programming is still important, but it is just one example of a useful abstraction. There are many others. For example, it is realized nowadays that the notion of “component-based programming” is important too. In one of its popular forms, it keeps the concepts of encapsulation, instantiation, and compositionality, like object-oriented programming, but leaves out inheritance. Component-based programming is a practical way to structure large programs. It is not natural to introduce component-based programming in the context of a purely object-oriented model. As we will see, it is more natural to introduce components as a new concept in a simple imperative model, i.e., a declarative computation model extended with state.

Why this book

Two important goals of our research are to find out what are the basic concepts of computer programming from the viewpoint of the programmer and to combine them into a single, simple computation model. After a decade of work, we have achieved these goals to some degree. This book presents these results in a practical and pedagogical way. This is not a theoretical book, although its results are theoretically well-founded. This is first of all a practical and down-to-earth guide, with many realistic examples and lots and lots of working code. Its main purpose is to teach real programming. We have tried to make the code representative of what is used in real-world programs. We have avoided programming styles and toy examples that do not “scale up”. We have chosen self-contained examples that can be expanded into full-blown applications with relatively little effort.

Design methodology

The design methodology that led to the models defined in this book is quite simple in its basic form. Two key ideas are to always question the design and to always have a working implementation. The language and its implementation are designed as a tower of abstractions. The developers continuously introduce new abstractions as solutions to practical problems. Sometimes a small inadequacy of the system with respect to a practical problem leads to a major restructuring. The burden of proof is on the developer proposing the abstraction: he must prototype it and show an application for which it is necessary. The net effect of a new abstraction must be either to simplify the system or to greatly increase its expressive power. If this seems to be the case, then intense discussion takes place
among all developers to simplify the abstraction as much as possible. Often it
vanishes: it can be completely expressed without modifying the system. This is
not always possible. Sometimes it is better to modify the system: to extend it or
to replace an existing abstraction by a new one.

The decision whether to accept an abstraction is made according to sever-
al criteria including aesthetic ones. Two major acceptance criteria are related
to implementation and formalization. The abstraction is acceptable only if its
implementation is efficient and its formalization is simple.

This methodology extends the approaches put forward by Hoare, Ritchie, and
Thompson [43, 75, 94]. Hoare advocates designing a program and its specification
concurrently. He also explains the importance of having a simple core language.
Ritchie advises having the designers and others actually use the system during
the development period. In Mozart this is possible because the development
environment is part of the run-time system. Thompson shows the power of a
well-designed abstraction. The success of Unix was made possible due to its
simple, powerful, and appropriate abstractions.

With respect to “traditional” software design processes, this methodology is
closest to exploratory programming [85]. The main defect of exploratory program-
ming, that it results in systems with ill-defined structure, is avoided by the way
the abstractions are refined and by the requirement of simple formalization.

The two-step process of first generating abstractions and then selecting among
them is analogous to the basic process of evolution. In evolution, an unending
source of different individuals is followed by a filter, survival of the fittest [24].
In the analogy, the individuals are abstractions and the filters are the acceptance
criteria of implementation and formalization. Some abstractions thrive (e.g.,
compositionality with lexical scoping), others die (e.g., the “generate and test”
approach to constraint programming is dead, being replaced by “propagate and
search”), others are born and mature (e.g., dynamic scope, which is currently
under discussion), and others become instances of more general ones (e.g., deep
guards, once basic, are now implemented with computation spaces).

History and acknowledgements

The ideas in this book did not come easily. They came after more than a decade
of discussion, programming, evaluation, throwing out the bad, and bringing in
the good and convincing others that it is good. We are lucky to have had a
coherent vision among the developers for such a long period. Thanks to this, we
have been able to make progress.

Many people contributed ideas, implementations, tools, and applications. Our
main research vehicle and “testbed” of new ideas is the Mozart system, which im-
plements the Oz language. The system’s main designers and developers are and
were (in alphabetic order): Per Brand, Denys Duchier, Donatien Grolaux, Seif
Haridi, Dragan Havelka, Martin Henz, Erik Klintskog, Leif Kornstaedt, Michael

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Mehl, Martin Müller, Tobias Müller, Anna Neiderud, Konstantin Popov, Ralf Scheidhauer, Christian Schulte, Gert Smolka, Peter Van Roy, and Jörg Würtz. Other important contributors are and were (in alphabetic order): Iliès Alouini, Thorsten Brunklaus, Raphaël Collet, Sameh El-Ansary, Nils Franzen, Simon Lindblom, Benjamin Lorenz, and Andreas Simon. In addition, we would like to thank our indirect contributors, power users, and developers “in spe”. Their legacy is not just the software quality but also scientific results, tools, course notes, applications, and encouragement. They include: Hassan Aït-Kaci, Mostafa Al-Metwallly, Andreas Franke, Claire Gardent, Fredrik Holmgren, Sverker Janson, Valentin Mesaros, Johan Montelius, Joachim Niehren, Mathias Picker, Andreas Podelski, Christophe Ponsard, Mahmoud Rafea, Juris Reinfelds, Thomas Sjöland, and Jean Vanderdonckt. Finally, we would like to thank all the contributors to the users@mozart-oz.org mailing list and the comp.lang.functional and comp.lang.prolog newsgroups, whose animated discussions have resulted in some very nice contributions to the book. We apologize to anyone we may have inadvertently omitted.

For this book, we would especially like to thank Donatien Grolaux for the GUI case studies (Section ??) and Raphaël Collet for the Minesweeper game (Sections 2.3 and ??) and for his work on Chapter 15 (Language Semantics).

How did we manage to keep the result so simple with such a large crowd working together? No miracle, but the consequence of a strong vision and a carefully crafted design methodology that took more than a decade to create and polish. Around 1990, some of us came together with already strong systems building and theoretical backgrounds. These people initiated the ACCLAIM project, funded by the European Union (1991–94). For some reason, this project became a focal point. Three important milestones among many were the papers by Janson & Haridi in 1991 [51] (multiple paradigms in AKL), by Smolka in 1995 [84] (building abstractions in Oz), and by Haridi et al in 1998 [40] (dependable open distribution in Oz). After ACCLAIM, two laboratories continued working together on the Oz ideas: the Programming Systems Lab (DFKI, Universität des Saarlandes, and Collaborative Research Center SFB 378) in Saarbrücken, Germany, and the Intelligent Systems Laboratory (Swedish Institute of Computer Science), in Stockholm, Sweden. The Oz language was originally developed by the Programming Systems Lab, led by Gert Smolka. In 1996 they were joined by the Department of Computing Science and Engineering (Université catholique de Louvain), in Louvain-la-Neuve, Belgium, when the first author moved there. Together the three laboratories formed the Mozart Consortium with its neutral Web site http://www.mozart-oz.org so that the work would not be tied down to a single institution.

This book was written using LaTeX, flex, xfig, xv, and Mozart, all running on the Linux operating system. The first author thanks the Walloon Region of Belgium for their generous support of the Oz/Mozart work at UCL in the PIRATES project.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
What’s missing

There are two main topics missing from the book:

- **Static typing.** The formalism used in this book is dynamically typed. Despite the advantages of static typing for program verification and implementation, we barely mention it. Section 4.3.5 explains why.

- **Specialized programming techniques.** The set of programming techniques is too vast to explain in one book of modest size. In addition to the general techniques explained in this book, each problem domain has its own particular techniques. This book does not cover all of them; attempting to do so would double or triple its size. To make up for this lack, we point the reader to some good books that treat particular problem domains: artificial intelligence techniques [77], algorithms [23], object-oriented design patterns [34], multi-agent programming [105], and numerical techniques [73].
Part I

Introduction
Chapter 1

Goals and Organization

“A programming language is like a natural, human language in that it favors certain metaphors, images, and ways of thinking.”
– Mindstorms: Children, Computers, and Powerful Ideas, Seymour Papert (?–)

Six blind sages were shown an elephant and met to discuss their experience. “It’s wonderful,” said the first, “an elephant is like a snake: slender and flexible.” “No, no, not at all,” said the second, “an elephant is like a tree: sturdily planted on the ground.” “Marvelous,” said the third, “an elephant is like a wall.” “Incredible,” said the fourth, “an elephant is a tube filled with water.” “What a strange and piecemeal beast this is,” said the fifth. “Strange indeed,” said the sixth, “but there must be some underlying harmony. Let us investigate the matter further.”
– Freely adapted from a traditional fable.

The purpose of this book is to give a broad and deep view of practical computer programming, as a unified engineering discipline based on a sound scientific foundation. There are many different ways to program a computer. One approach to study computer programming is to study programming languages. But there are a tremendously large number of languages; so large that it is impractical to study them all. How can we tackle this immensity? We can pick a small number of languages that are representative of different programming styles. But this gives little insight into programming as a unified discipline. This book uses another approach.

We focus on programming concepts and the techniques to use them, not on programming languages. The concepts are organized in terms of computation models. A computation model consists of a set of data types, operations on them, and a language to write programs that use these operations. Each computation model has its own set of techniques for programming and reasoning about programs. The number of different computation models that are known to be useful
is much smaller than the number of programming languages. This book covers both widely-used models as well as some lesser-used models. The criterion for presenting a model is whether it is useful in practice.

The term computation model makes precise the imprecise notion of “programming paradigm”. In the rest of the book, we will talk about computation models and not programming paradigms.

Each computation model has a core language that we call its kernel language. The kernel languages are introduced in a progressive way, by adding concepts one by one. All have a complete and simple formal semantics. Often, just adding one new concept makes a world of difference in programming. For example, adding encapsulated state to functional programming allows to do object-oriented programming. In like fashion, we will see that most models, even apparently divergent ones, are in fact closely related. It follows that it is quite natural for different models to be used together in one program. This is not a new idea. For example, the designers of Lisp and Scheme have long advocated a similar view. However, this book applies it in a much broader way than was previously done.

From the vantage point of computation models, the book also sheds new light on important problems in computer science. We present three such areas, namely graphic user interface design, robust distributed programming, and intelligent agents. We show how the judicious combined use of several computation models can help solve some of the difficult problems of these areas.

Languages mentioned

We mention a number of programming languages in the book and relate them to particular computation models. Some of the languages we mention are representative of particular models. Java and Smalltalk are based on an object-oriented model. Haskell and ML are based on a functional model. Prolog and Mercury are based on a logic model. Not all interesting languages can be so classified. We mention some other languages for their own merits. For example, Lisp and Scheme pioneered many of the concepts presented here. Erlang is functional, inherently concurrent, and fault tolerant.

We single out four languages as representatives of important computation models: Erlang, Haskell, Java, and Prolog. We identify what computation model each realizes in terms of the uniform framework of the book. For more information about them we refer readers to other books. Because of space limitations, we are not able to mention all interesting languages. Omission of a language does not imply any kind of value judgement.

Related books

Among programming books, the book by Abelson & Sussman [1, 2] is the closest in spirit to ours. It covers functional programming, imperative (i.e., stateful) programming, and introduces objects, concurrency, and logic programming. We consider the present book as a successor to Abelson & Sussman. It goes deeper
into concurrency and nondeterminism, objects, and inheritance, and also covers components, dataflow execution, distributed programming, constraint programming, and user interface design. It gives a formal semantics to these models, thus putting them on a solid foundation. The focus of the two books is different: the present book emphasizes the relationships between the computation models and practical techniques in the models while Abelson & Sussman emphasizes designing language features and building interpreters and virtual machines for them.

The Leda language deserves a mention, if only because its designer Timothy Budd realized some of the important issues early on [10]. But the language is disappointing. It only scratches the surface of what multiparadigm programming is all about. It has functions and objects, just a little logic, and no concurrency. The Leda book does not explain how to use the paradigms together, does not explain in any deep way how they are related, and gives no semantics.

1.1 Goals of the book

1.1.1 Programming as an engineering discipline

The main goal of the book is to teach programming as a true engineering discipline. An engineering discipline consists of a set of practical techniques and standard guidelines, firmly reposing on a scientific theory. Teaching an engineering discipline consists of two parts: teaching the fundamental concepts (the science) and teaching the current tools (the technology). Knowing the science prepares the student for future developments. Knowing the tools prepares the student for the present.

We consider that engineering is the proper analogy for the discipline of programming. This does not mean that the book is intended only for engineering students. On the contrary, anyone doing programming is an "engineer" in the sense of this book.

Programming is more than a craft

By programming we mean the step from specification to running program, which consists in designing the application architecture and its abstractions and coding them into a programming language. Up to now, programming has been taught more as a craft than as an engineering discipline. It is usually taught in the context of one (or a few) programming languages (e.g., Java, complemented with Haskell, Scheme, or Prolog). The historical accidents of the particular languages chosen are interwoven together so closely with the fundamental concepts that the two cannot be separated. There is a confusion between tools and concepts. As a consequence, different schools of thought have developed, based on different programming "paradigms": object-oriented, logic, functional, etc. The unity of programming as a single discipline has been lost.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Teaching programming in this fashion is like giving one course on building wooden bridges and (possibly) one course on building iron bridges. Engineers would implicitly consider the restriction to wood or iron as fundamental and would not think of using other materials or even of using wood and iron together.

The result is that programs often suffer from poor design. We give an example based on Java, but the problem exists in all languages to some degree. Concurrency in Java is complex to use and expensive in computational resources. Because of this limitation, Java-taught programmers conclude that concurrency is a fundamentally complex and expensive concept. Program specifications are designed around the concurrency restrictions, often in a contorted way. But these restrictions are not fundamental at all. There are forms of concurrency that are quite useful and yet as easy to program with as sequential programs. Furthermore, it is possible to implement threads almost as cheaply as procedure calls. If the programmer were taught about concurrency in the correct way, then he or she would be able to specify for and program in systems without concurrency restrictions (including improved versions of Java).

The kernel approach

Current languages scale up to programs of millions of lines of code. How can we separate the languages’ fundamental concepts, which underlie this scalability, from their historical accidents? The kernel approach shows one way. Full languages are translated into a small kernel language that has a minimal number of programmer-significant elements. This gives both programmer and student a clear insight into what the language does. The kernel language has a simple formal semantics, which gives a solid foundation to the programmer’s intuition and the programming techniques built on top of it. Because all languages translate into a subset of one kernel language, the underlying unity of programming is regained.

Reducing a complex phenomenon to its primitive elements is characteristic of the scientific method. It is a successful approach that is used in all the exact sciences. It gives a deep understanding that has predictive power. For example, materials science lets one design all bridges (whether made of wood, iron, or anything else) and predict their behavior in terms of simple concepts such as force and energy and their conservation laws.

The two approaches most similar to the kernel approach are the foundational calculus and the virtual machine. We find that neither is suitable for engineering. A foundational calculus, like the λ-calculus or π-calculus, reduces programming to a minimal number of elements, regardless of whether these are relevant for practical programming. This is important for the theory of computation, but practice shows that it is largely irrelevant to programmers. A virtual machine, which defines a language in terms of an implementation on an idealized machine, is useful for implementors. Its concepts are close to hardware and again are largely irrelevant to programmers.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
1.2 Main features

1.2.1 Formalism used

A major feature of the book is that it uses a single formalism for presenting all computation models and programs. Using a single formalism allows us to go much further than we could otherwise. Since the models have many concepts in common, introducing them is simplified within a single formalism. The effort...
needed by the student is reduced and the intellectual harvest is increased. The formalism we chose is the Oz language and its computation model. Why did we choose Oz? There are four reasons why it is a good choice:

- Because it was explicitly designed to support different computation models. For over a decade we have been analyzing different computation models and bringing together their concepts. Oz was the main programming vehicle of this research. It incorporates many of the most advanced ideas in programming language research.

- Because it has a complete implementation that is efficient and robust. The implementation, called Mozart Programming System, runs on both Unix and Windows platforms. It has an interactive, incremental development environment with many tools. All the program fragments in the book can be run directly in this environment. This gives immediate feedback to the student. Having rapid feedback is important when learning new concepts.

- Because its concepts have been tested in real use. It has profited from ten years of experience in application development, in both university and industrial projects, and it has deployed applications.

- Because it has a complete and simple formal semantics. This allows the student to deepen his knowledge of the different models more than if they were just introduced informally. The formal semantics leaves no stone unturned; it explains precisely what happens in every case. For example, how does dataflow execution interact with lazy execution? The semantics gives the answer.

In addition, Oz has a simple, well-factored syntax. Appendix C gives the complete syntax including all programmer conveniences (“syntactic sugar”) in just seven simple pages. We know of no other formalism that covers so much ground so well.

### 1.2.2 Uniform structuring principles

All the computation models in the book support a small set of program structuring principles that always apply, with no exceptional cases. Following these principles maximizes the expressive power of the models while keeping them deceptively simple. These principles are defined and used throughout the book. As a guide to these discussions, we briefly summarize the most important principles:

- **Full compositionality**: any statement can be executed in the context of any other statement. All contexts are equal. For example, a class can be defined inside a procedure and vice versa, with the expected behavior.

---

2For example, many students, already proficient Java programmers, have told us that they first understood what Java objects really are through this formalism (see Chapter 9).

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
• **Full concurrency support**: concurrency is supported as a practical tool to structure programs. This may seem obvious, but it has several consequences that are rarely supported. First, executing a statement in the current thread or a new thread behaves identically except for differences due to changes in execution order (this is analogous to higher-orderness, see below). Second, both threads and synchronization are very cheap, so they can be used whenever needed for program structure. Finally, threads and synchronization are easy to use (little or no extra notation is needed).

• **Full higher-orderness**: any statement can be made into a procedure, i.e., a “packet of work”. This separates the statement’s definition from its execution. Another part of the program can then decide whether and when to execute it. Executing the procedure has identical behavior to executing the statement originally.

• **Encapsulated state**: by using encapsulated state, components can be defined whose behavior changes over time. A state is a sequence of values in time. An encapsulated state is a state inside a component, that (1) survives in between invocations of the component and (2) does not affect the component’s interface in any way. The state is implemented as an updatable “container”. Encapsulated state is used extensively in object-oriented programming.

• **Capability property**: references to language entities cannot be forged. A statement’s initial references are determined by lexical scoping, i.e., by inspecting the program text. The only way to get additional references is if they are passed explicitly during execution. This is sometimes called “connectivity begets connectivity”. This supports modular programming in a strong sense: it is impossible for one statement to look inside another statement without permission. The capability property is particularly important when writing secure programs that are supported by a secure implementation.

• **Identity of compile time and run time**: the distinction between “compile time” and “run time” exists for performance optimization and reduces the expressiveness of the language. To a first approximation there should be no distinction: all operations should always be possible. This is important for long-lived applications, which because of their long life take over some characteristics of operating systems. This has two major consequences. First, all declarations are executable statements. For example, a class declaration creates a class value when executed. Second, the compiler is part of the run-time environment. It can be invoked at any time by a program.

Experience shows that these principles can be applied uniformly while still allowing an efficient implementation.
1.2.3 Dataflow computation

A unique feature of the book is the extensive use of dataflow as a basic mechanism for organizing concurrent programs. Dataflow execution is part of the concurrency support explained in the previous section. We say a computation is dataflow if it waits until its inputs are available before continuing. With dataflow, synchronization and communication become implicit, i.e., they require no additional instructions in the program text. This means that, to a first approximation, the dataflow aspect can be completely ignored by the programmer. Concurrent computations communicate through shared dataflow variables. It is invisible to the programmer whether or not a calculation temporarily blocks while waiting for a value to arrive.

The dataflow principle is well-known in parallel programming. There it is used to decouple computations so that they can be executed in parallel, resulting in increased performance. Dataflow is the basis of several powerful parallel computing architectures and languages [69]. Less well-known are the advantages of dataflow for concurrent programming (managing computations that consist of logically separate parts that evolve independently) and for structuring programs (building hierarchies of components that interact with as few dependencies as possible). One of the goals of the book is to make these aspects of dataflow more well-known.

Dataflow greatly simplifies many concurrent programming tasks. For example, synchronizing on availability of data, which is what dataflow provides, is implicit. Mutual exclusion is made simple by using streams of dataflow variables. Many other examples are given in the book and in [8, 82]. From a theoretical viewpoint, the advantages of dataflow are due to the fact that it is based on the monotonic behavior of dataflow variables: they can be bound to exactly one value. Synchronization on dataflow variables is monotonic. A thread that is ready to execute will stay ready until it executes. This is in contrast to imperative variables, which are nonmonotonic. For them, concurrent programming requires complex mechanisms such as monitors to coordinate one thread with another. Chapter 10 compares the two approaches for common concurrent programming tasks.

1.2.4 Computation models and expressiveness

The computation models of this book have simple formal definitions. Yet they are designed with a very different purpose than other formal calculi such as the Turing machine, the λ-calculus, or the π-calculus. Our focus is on the needs of practical programming rather than the needs of mathematical analysis. We are interested in languages with a small number of programmer-significant elements. The criterium is not just the number of elements, but rather how easy it is to write useful programs and to do practical reasoning about their properties. This approach is quite different from the foundational approach.
1.2 Main features

Foundational calculi

In computer science theory, it is common practice to design languages and computation models with a very small number of basic concepts, in order to study computation itself. For example, the Turing machine is a small language and computation model, but it is equivalent or superior in programming power to all computation models that have been implemented. That is, any program for one of these models can be implemented on a Turing machine and vice versa. The λ-calculus is a small language in which the only operations are defining functions and calling them. It is a major theoretical result that anything that can be programmed in this calculus can also be programmed on a Turing machine, and vice versa. The π-calculus is a small language that was designed to study concurrency. The π-calculus is very powerful, in the sense that it has all the basic concepts of concurrent programming.

The Turing machine, the λ-calculus, and the π-calculus are important examples of calculi that have been designed to study the possibilities and limits of computer programming. But they are too low-level to be practical programming languages. Concepts that are simple to the programmer require complicated encodings in them. For example, procedure calls can be implemented on a Turing machine, but they are complicated. From the viewpoint of the mathematical study of computation, this is not a defect of the Turing machine. It is a virtue, since it reduces complex concepts to their most primitive components. Mathematical study is simplified if the languages are simple.

Role of expressiveness and reasoning

The computation models of this book are designed not just with formal simplicity in mind (although it is important), but on the basis of how a programmer can express himself/herself and reason within the model. We find that adding a new element to a computation model is a two-edged sword. It introduces new forms of expression, making some programs simpler, but it also makes reasoning about programs harder. For example, by adding encapsulated state to a functional programming model we can express the full range of object-oriented programming techniques. However, reasoning about object-oriented programs is harder than reasoning about functional programs. Functional programming is about calculating values with mathematical functions. Neither the values nor the functions change over time. Encapsulated state is one way to model things that change over time: it provides a container whose content can be updated. The very power of this concept makes it harder to reason about.

There are many different practical computation models, with different levels of expressiveness, different programming techniques, and different ways of reasoning about them. We find that each model has its domain of application. This book explains many of these models, how they are related, how to program in them, and how to combine them to greatest advantage.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Importance of combining models

Each computation model was originally designed to be used in isolation. It might therefore seem like an aberration to use several of them together in the same program. We find that this is not at all the case. This is because models are not just monolithic “blocks” with nothing in common. On the contrary, they have much in common. For example, the differences between declarative & imperative models and concurrent & sequential models are very small compared to what they have in common. Because of this, it is easy to use several models together.

But even though it is technically possible, why would one want to use several models in the same program? The deep answer to this question is simple: because one does not program with models, but with programming concepts and ways to combine them. Depending on which concepts one uses, it is possible to consider that one is programming in a particular model. The model appears as a kind of epiphenomenon. Certain things become easy, other things become harder, and reasoning about the program is done in a particular way. It is quite natural for a well-written program to use different models. At this point, in Chapter 1, this answer may seem cryptic. It will become clear in later chapters.

An important principle we will see in this book is that concepts traditionally associated with one model can be used to great effect in more general models. For example, the concepts of lexical scoping, higher-order programming, and lazy evaluation, usually associated with functional programming, are useful in all models. This is well-known in the functional programming community. Functional languages have long been extended with explicit state (e.g., Scheme and ML) and more recently with concurrency (e.g., Concurrent ML and Concurrent Haskell).

1.2.5 Relevance of distribution and intelligence

The rapid pace of advances in computing technology has enormously increased the potential benefits of both distribution and intelligence. But current programming tools do not realize these benefits. On the one hand, the explosive development of the Internet is making it necessary to develop applications that are distributed, fault tolerant, and secure. Yet, most applications are still centralized, and a fortiori not fault tolerant or secure. This is because making them distributed requires specialized skills. On the other hand, the computational power of current systems (i.e., networked computers) makes it possible to put more intelligence in applications. Yet, with rare exceptions, this power is not used. Most personal computers remain idle most of the time, except for signal processing tasks (graphics and video).

To meet this double challenge, many concepts are needed. First, distribution requires concurrency. Distribution is when individual computers communicate through a network. This means that the system becomes parallel, and the simplest way to expose this to the programmer is by concurrency. Concurrency is the logical ability to have independent activities evolving simultaneously. But
1.3 How to read this book

Figure 1.1 gives a road map of the book, grouping chapters according to common themes. Chapter 1 is the first chapter you should read. The other chapters assume that some of the preceding chapters are known. Chapter 2 gives an overview of many concepts in terms of three realistic examples. The examples are carefully chosen to use techniques that scale up to real programs. We recommend you read this chapter to get a first idea of why it is important to choose the right programming language concepts. Chapter 3 is a gentle introduction to programming. This chapter introduces many of the basic concepts of the book in a progressive way. This chapter is important to build the reader’s intuition. Reading it carefully and interactively, executing the examples given, will pay dividends for the rest of the book.

Chapters 4–10 are the heart of the book. They define the different computation models, each with its programming and reasoning techniques, and show how the models relate to each other. Chapter 11 takes four languages that are representative of popular programming paradigms and situates them with respect to the models of the book. Chapters 12, 13, and 14 extend the concurrent stateful model of Chapter 10 to three important areas: user interfaces, distribution, and constraints. Chapters 15–16 define the semantics of the unified model both operationally and in terms of virtual machines. Appendices A–F give supporting background material.

Chapters 10, 13, and 14, when taken together, define the unified computation model. The unified model is rarely used in its complete form. Usually, subsets that have particular properties are used. The other chapters focus on important subsets of this model. They start with declarative programming (strict functional and deterministic logic programming) and successively introduce concurrency, laziness, nondeterminism (both the don’t know and don’t care forms), state, distribution, and constraint programming. At each level, the natural programming techniques are given for that level. All in all, a coherent, layered view is given of many programming models and how they fit together.

This book is intended to be a practical guide as well as teach theoretical concurrency, although essential, is not enough. Fault tolerance is needed as well because computers and parts of the network can fail. Security is important because many users with conflicting goals share networks and computers.

Second, many applications have greatly improved behavior if they have some “intelligence”. Intelligence is a broad term that covers many abilities. This book uses intelligence in a fairly narrow way, to mean just symbolic reasoning and machine learning. For example, an application can use a digital assistant to augment its performance or ease of use. An email reader can adapt to a user by learning the user’s habits, e.g., in which folder to put an incoming message. A computer game can help a player select good moves.
# I Introduction

1. Goals and organization  
2. Example applications  
3. Programming

# II. General Computation Models

4. Declarative model  
5. Declarative techniques  
6. Declarative concurrency  
7. Relational programming  
8. Encapsulated state  
9. Object orientation  
10. Concurrency and state  
11. Representative languages

# III. Specialized Computation Models

12. Graphic user interfaces  
13. Distributed programming  
14. Constraint programming

# IV. Semantics

15. Language semantics  
16. Virtual machines

---

Figure 1.1: Road map of the book
concepts. Many small programs and code fragments are given throughout. All syntactically-complete fragments can be run in the Mozart interactive interface. We suggest that you read the book accompanied by a running Mozart system. This is an excellent way to understand by doing. In addition, each chapter ends with a set of exercises. These exercises explore and extend what was said in the chapter. They range from very simple to rather substantial programming projects.

We now give a walk-through of the book, summarizing what each chapter is about.

1.3.1 Core chapters

- Chapter 4: Declarative Computation Model. This is the first model introduced in the book. Read Section 4.1 thoroughly, since it explains how we define the syntax and semantics of computation models throughout the book. The rest of the chapter defines the basic concepts and language constructs of the declarative model. They use a set of basic data types, which are defined in Appendix B. This appendix should be skimmed at first and consulted later as a reference.

- Chapter 5: Declarative Programming Techniques. The techniques shown here are a foundation for the whole book. This chapter gives a complete course in declarative programming, which is a surprisingly rich area, and it touches on most of the ideas to be developed in later chapters. Sections 5.1 and 5.2 give a brief introduction to the general computation models presented in the book: declarative (Chapter 5), concurrent declarative (Chapter 6), relational (Chapter 7), stateful (Chapters 8 and 9), and concurrent stateful (Chapter 10). These two sections give the basic intuition on how these models differ and how to use them together in the same program. While declarative programming has the simplest computation model and supports the simplest reasoning techniques, it is still practical (witness the vitality of the functional and logic programming communities). It provides insights that enhance understanding of the richer models given later on.

- Chapter 6: Declarative Concurrency. This chapter continues the previous chapter by adding concurrency and lazy execution. Declarative concurrency is truly concurrent as well as declarative (strange though this may seem). This is possible despite the nondeterminism introduced by concurrency because there is no observable nondeterminism. Lazy execution can be studied independently of concurrency, but we find it fits naturally in a concurrent declarative environment.

- Chapter 7: Relational Programming. This chapter extends declarative programming with explicit nondeterministic choice ("don’t know" nondeterminism). This adds a search component to programs, in the Prolog style.
Many problems are naturally specified with don’t know nondeterminism, and this model can execute them directly. This model is a prelude to the constraint-based model of Chapter 14.

- Chapter 8: Encapsulated State. This chapter introduces encapsulated state and builds abstract data types with it. Section 8.2 is a good discussion of why state is important when building systems that can evolve and learn. The chapter gives techniques for building software components using encapsulated state.

- Chapter 9: Object-Oriented Programming. This chapter introduces object-oriented programming, which is a particularly rich abstraction based on state. We define a simple yet powerful object system with multiple inheritance, static and dynamic binding, first-class messages, and fine control over visibility.

- Chapter 10: Concurrency and State. This chapter introduces the most general computation model of the book, which has both explicit state and concurrency. In contrast to Chapter 6, this model exposes the nondeterminism inherent in concurrency (“don’t care” nondeterminism). As a result, this model is one of the hardest to program with in its general form. Programs that use it can almost always be decomposed into parts that use simpler models. Section 10.1 is a good discussion of different concurrent computation models and their advantages and disadvantages.

- Chapter 11: Representative Languages. This chapter situates the computation models of four popular languages, Erlang, Haskell, Java, and Prolog, with respect to the models of the book. We choose these languages as examples of concurrent, functional, object-oriented, and logic programming paradigms.

### 1.3.2 Extension chapters

The preceding chapters were all concerned with general-purpose computation. The following chapters extend this to three particular areas.

- Chapter 12: Graphic User Interface Programming. This chapter introduces an approach for building graphic user interfaces that combines declarative and nondeclarative (“procedural” or “imperative”) techniques. The approach has a declarative base and is extended with concurrency and state exactly where needed. This combines the good properties of both the declarative and procedural approaches to graphic user interface design. This is a good example of the programming methodology of Chapters 4–10. It shows the power of starting with declarative programming techniques and complementing them with state and concurrency exactly where needed.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Chapter 13: Distributed Programming. This chapter introduces open distributed programming and fault tolerance. Distributed programming is only slightly more involved than concurrent programming, if approached in the right way. The main complications come from two sources: the computation is partitioned among separate processes and several independent computations can share resources (e.g., computers and networks). Partitioning introduces the problems of localized resources, limited network performance, and partial failure. Sharing introduces the problems of resource management and security.

Chapter 14: Constraint Programming. This chapter introduces constraint-based inferencing. This is a general technique for calculating with partial information and partially-specified algorithms. Constraints allow to express precisely what is known and search allows to overcome the lack of precise algorithms. Traditionally, constraint programming is considered as a branch of operations research, for solving combinatorial optimization problems. We show that it is actually much more general, giving a powerful set of tools for doing reasoning in the real world, where precision is often lacking.

1.3.3 Semantics

This part makes the preceding chapters more precise by giving two definitions of the unified model: a formal semantics and a virtual machine implementation.

Chapter 15: Language Semantics. This chapter defines a formal semantics of the unified computation model. The formalism is a simple one, inspired by the Structured Operational Semantics approach [108]. The semantics of each computation model in the book is a part of this chapter. It is done in layers. First comes the semantics for the kernel language of the concurrent stateful model. Subsets of this semantics suffice for all earlier models. Then three extensions are done. The first defines all the important language idioms (i.e., functions, lazy functions, pattern matching, loops, classes, objects, ports, and locks) in terms of the kernel language. The other two define distributed and constraint programming.

Chapter 16: Virtual Machines. This chapter defines different virtual machines to implement the concurrent stateful model. These virtual machines introduce registers and the graph model of the store. They permit rather precise calculations of a program’s memory usage.

The semantics can be used as a reference to clarify the subtle points in the discussions of the other chapters.
1.3.4 Appendices

There is a set of appendices to complete the information given in the chapters. The appendices contain enough information to make this a standalone programming book in addition to its role as a teaching book. They also give details that it would be boring to have in the main body of the text.

- Appendix A: Development Environment. This book is complemented by a fully-functional programming system, called Mozart, that has an interactive development environment. This appendix summarizes the most important abilities of the environment.

- Appendix B: Basic Data Types. This summarizes the most important data types and their operations.

- Appendix C: Language Syntax. This gives the complete syntax of the language used in the book, including all syntactic conveniences.

- Appendix D: Distribution Model. This gives a detailed definition of the distribution model.

- Appendix E: Failure Model. This gives a detailed definition of the failure model.

- Appendix F: Mozart System Properties and Limitations. This gives the link between the ideal computation models as presented in this book and their implementation in the Mozart system. All the differences between the ideal models and the implemented models are given.

1.4 Computation models covered

One of the distinguishing features of this book is that it covers many different computation models in a unified framework.

1.4.1 General computation models

The book covers the following general computation models:

- Declarative programming, also called stateless programming (Chapters 4 and 5). This model subsumes strict functional programming and deterministic logic programming.

- Order-determining concurrent programming (Section 6.3). This model is basically sequential; it has a restricted form of concurrency used solely to dynamically determine the order of operations in a sequential calculation.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
1.4 Computation models covered

- Concurrent declarative programming (Chapter 6). This model subsumes lazy functional programming and the declarative subset of concurrent logic programming. It is both concurrent and deterministic.

- Relational programming (Chapter 7). This model subsumes nondeterministic logic programming in the Prolog style.

- Stateful programming, also called imperative programming (Chapter 8). This model subsumes functional programming with state. Using state allows programs to learn from their past and change their behavior over time, while keeping exactly the same interface.

- Component-based programming (Section 8.9). This is an often-useful style of stateful programming which generalizes the notion of library or package. The basic concept is the component, which is a specification that can be instantiated to give modules, i.e., component instances. If the component is a compilation unit, we call it a functor.

- Object-oriented programming (Chapter 9). This is another often-useful style of stateful programming. The basic concept is the class, which is a specification that can be instantiated to give objects. A class incrementally defines an abstraction from other classes using inheritance.

- Nondeterministic concurrent programming (Section 10.2.3). This model subsumes pure concurrent logic programming. It is an intermediate step between declarative concurrency and stateful concurrency.

- Concurrent stateful programming (Chapter 10). This is the most expressive general-purpose model of the book. It subsumes concurrent object-oriented programming with passive objects. It also subsumes active objects with asynchronous message passing.

Figure 1.2 gives another view of these models, showing how they relate to each other. This is one possible overall view: “shuffling around” the basic operations of these models can give somewhat different views. None of these models is “better” than the others. Each has its domain of application. Any realistic program will usually have components programmed in many different models. For each model, we show how to program in it and we explain what its advantages and disadvantages are with respect to the other models.

1.4.2 Extended computation models

This book covers the following computation models that extend the general models for particular uses:

- Soft real-time programming (Section 6.8). This model is useful for real-time programming with timing deadlines that are respected most of the time.
Graphic user interface programming (Chapter 12). This model is a mixed declarative/procedural approach that uses each of the declarative and procedural approaches to best advantage. The static part and the well-delimited dynamic part of a user interface are specified declaratively. The other dynamic parts, which need high expressiveness, are specified procedurally. Partly because of this split, it is one of the most expressive yet concise tools that exist for building user interfaces.

Open robust transparent distributed programming (Chapter 13). This model extends the concurrent stateful model by extending all language entities with a distributed semantics. This semantics defines the network operations needed by each language entity. This is simple enough that it allows to control what are the network operations. This gives control over network operations and site dependencies, while maintaining the same language semantics as the concurrent stateful model. A given program has exactly the same behavior, modulo performance and resources, independent of its dis-
 distribution structure. This is an enormous help in writing robust distributed programs.

- Constraint programming (Chapter 14). This model permits to calculate with partial information. The model subsumes constraint logic programming and concurrent constraint programming.

- The unified computation model (Chapter 13 combined with Chapter 14). This is the most complete model of the book. It subsumes concurrent stateful programming, distribution, and constraints.

Chapter 15 gives the semantics of the unified model and subsets given in the book.

1.4.3 Using the models

With such a proliferation of computation models, it is natural to ask when to use each and how to use them together. Let us look at the general and the specialized models separately. Each specialized model has its own particular domain of application. The question then becomes: when is a particular domain of application relevant? For soft real time, distribution, and user interface programming, this is easy to answer. For constraints, it is more subtle, because many problems can be formulated as constraint problems. The basic question is whether there is only partial knowledge of the problem and its solution method. If so, then the constraint model is probably appropriate.

For the general models, the question is not about choosing the model, because all models are subsets of the concurrent stateful model of Chapter 10. Rather, the question is what programming techniques to use. Each technique has its natural place in one or more models. The basic principle is to use the least expressive model that is satisfactory, because this model makes reasoning about the program the simplest. This book introduces each programming technique in the earliest model in which it can be expressed. In later models, the technique can be used as is or in more general ways, with the new abilities of the model. Often, the same problem can be solved in several models, using different techniques. When this is so, we show the different solutions and explain the trade-offs involved in each.

1.5 How to teach from this book

The purpose of this book is to teach programming as an engineering discipline. We present programming concepts and techniques and how they can be organized into computation models. Section 1.5.1 explains the courses it is useful for. There is a substantial software package, the Mozart system, that accompanies the book. Section 1.5.2 introduces this package and explains how it can be used to support practical exercises.
1.5.1 Courses

This book can be used as textbook for several undergraduate and graduate computer science courses:

- An undergraduate course giving an introduction to the discipline of programming. Chapter 3 is the main guide, supplemented by material from Chapters 4–6, 8–10. Students should have some previous programming experience (e.g., a first programming course) and a minimum level of mathematical maturity (e.g., a first course on analysis, discrete mathematics, or algebra).

- A two- or three-course undergraduate sequence on the discipline of programming:
  1. First course: general introduction (Chapter 3) and declarative programming (Chapters 4–6). Depending on the students’ backgrounds, some material can be taken from Part IV.
  2. Second course: state and concurrency (Chapters 8–10). Depending on the students’ backgrounds, some material can be taken from the other chapters.
  3. Third course: extending models for particular situations. This includes relational programming (Chapter 7), specific programming languages (Chapter 11), graphic user interface programming (Chapter 12), distributed programming (Chapter 13), and constraint programming (Chapter 14).

Students should have some previous programming experience (e.g., a first programming course) and a minimum level of mathematical maturity (e.g., a first course on analysis, discrete mathematics, or algebra).

- An undergraduate course on distributed programming (Chapter 13). Students should have some programming experience with concurrent and object-oriented systems. For example, adequate prerequisites are Chapters 9 and 10. If desired, the book can be complemented with parts of other texts explaining distributed algorithms (e.g., [16], [62], or [93]).

- An undergraduate course on constraint programming (Chapters 4–7, Chapter 14). Students should have some previous programming experience (e.g., a first programming course with practical) and a minimum level of mathematical maturity (e.g., a first course on analysis, discrete mathematics, or algebra).

- An undergraduate or graduate course on computation models (the whole book, including the semantics). The depth of treatment can be adjusted depending on the students’ backgrounds.
Part of a graduate course on intelligent collaborative applications (the whole book, with emphasis on Part III). If desired, the book can be complemented by parts of texts on artificial intelligence (e.g., [77]) or multi-agent systems (e.g., [105]).

Part of an undergraduate course on semantics. Chapter 15 gives a real-sized example of how to define the semantics of a complete modern programming language. This chapter is complemented by previous chapters explaining each computation model (Chapters 4, 6, 8, and 10).

The book, while it has a solid theoretical underpinning (see, e.g., Chapter 15), is intended to give a practical education in these subjects. Each chapter has many program fragments, all of which can be executed on the Mozart system. Each chapter ends with a set of exercises that usually involve some programming. They can be solved on the Mozart system.

1.5.2 Software

A useful feature of the book is that all the program code given can be run on a software platform called the Mozart Programming System. Mozart is a full-featured production-quality programming system that comes with an interactive incremental development environment and a full set of tools. It compiles to an efficient platform-independent bytecode that runs on many Unix and Windows systems. Distributed programs can be run simultaneously on Unix and Windows systems. The Mozart Web site, http://www.mozart-oz.org, has complete information including downloadable binaries, documentation, scientific publications, source code, and mailing lists.

The Mozart system efficiently implements all the computation models covered in the book. This is ideal for comparing models, by writing programs to solve the same problem in different models, and for combining models, by writing programs that use different models together. Because each model is implemented efficiently, whole programs can be written in just one model. Other models can be brought in later, if needed, in a pedagogically justified way. For example, programs can be completely written in an object-oriented style, complemented by small declarative components where they are most useful.

The Mozart system is the result of a long-term development effort by the Mozart Consortium, an informal research and development collaboration of three laboratories. It has been under continuing development since 1991. The system is released with full source code under an Open Source license agreement. The first public release was in 1995. The first public release with distribution support was in 1999. This book uses version 1.2.0, released in May 2001.
Chapter 2

Example Applications

This chapter presents three small applications that show how to use different computation models together. Each application is inherently interesting and can be scaled up to become a full-blown application. We suggest you try each application in Mozart to get a feel for what it can do. This chapter explains the architectures of the applications and the programming concepts and techniques needed to build them. The source code is given in later chapters, when the concepts are introduced.

- The first application, in Section 2.1, is a simple animation: a bouncing ball. This application shows how to combine the techniques of object-orientation, concurrency, and higher-order programming. It also uses basic graphic user interface programming. Section 10.7.1 gives the full source code.

- The second application, in Section 2.2, is a robust distributed chat program. Its purpose is to introduce some of the abstractions used for building open, fault-tolerant distributed applications. Section ?? gives the full source code.

- The third application, in Section 2.3, is a minesweeper game with a digital assistant. Its purpose is to introduce the use of dataflow to simplify concurrent programming and the use of constraint reasoning to add intelligence to an application. Section ?? gives the full source code.

2.1 A bouncing ball

The first application simulates a set of bouncing balls. Before going into the design, we suggest that you run the program to get a feel for what it does. We assume that you have installed a Mozart system locally. Start the system with the command “oz”. This opens the interactive development environment. Load the file bounce.oz with the command CONTROL-X CONTROL-F bounce.oz. Run the file with the command CONTROL-. CONTROL-B. This opens two windows on the screen.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Figure 2.1: The bouncing ball application

Each window is a home for bouncing balls. Clicking the left button of the mouse in the window will create a bouncing “PacMan” ball of a new color. Each click creates a new ball. Figure 2.1 shows the situation after a few clicks. The balls move independently according to the law of gravity. That is, they appear to be following a parabolic path and bouncing when they hit the floor. Clicking the right button will remove the last ball that was created.

2.1.1 The architecture

What concepts do we need to build this application? Let us approach this question by looking closely at what the application does. Each ball is an active entity, independent of the others. The ball has a state, i.e., information that completely specifies it: its position, velocity, color, and the window it is located in. If each ball is an active entity, then we can say that it knows how to update its state, it knows how to draw itself, and it does draw itself repetitively. The ball has passive knowledge there (knowledge of how to update the state) but it is also active: the ball is doing something (calculating its new state and drawing itself repeatedly with a delay).

Each ball has two parts: an active part, with its own logic, and a passive part. One way to model the passive part is as an abstract data type. This consists of an encapsulated state, which is invisible from the outside, and a set of operations that are visible from the outside. Therefore the whole ball is an encapsulated active entity that can communicate with its surroundings. How does it communicate? If the program has other, similar entities to the ball, then communication is straightforward. The ball sends messages to other entities in
its environment, and these entities can send messages to the ball.

The program can host many balls. Therefore we need to manage them, i.e. create one new ball, destroy one ball, and destroy all balls (when the program terminates). This management is independent of the balls themselves. Therefore it is yet another active entity: a ball manager.

Finally, there is the graphics subsystem, which does the actual drawing on the screen. This is also an independent active entity. It manages the window, which displays the balls. It accepts commands from the human user and forwards them to the ball manager. Finally, it accepts drawing commands from the balls.

Figure 2.2 shows all the active entities in the bouncing ball example and how they communicate. In this figure, a circle is an active entity and an arrow is a communication link. The direction of the arrow gives the direction of the communication. The arrows are labelled with the possible messages that can be passed.

There are four kinds of active entities: the human, the graphics subsystem, the ball manager, and the ball. The ball contains both a passive and an active part. We would like to write the program so that it follows this structure as closely as possible. A good programming language should support this. Chapter 10 gives a program that uses “active objects”, which follows the architecture of Figure 2.2 very closely.

### 2.1.2 Concepts and techniques used

The bouncing ball application illustrates many principles of good program design. We summarize them here:
Encapsulation. It should be possible to encapsulate part of a program and give it a well-defined interface with the rest of the program. The language should support this style.

Concurrency. Often, the natural architecture of a program requires multiple active entities, that evolve independently. This should be supported by the language: there should not be any penalty for being concurrent when the architecture requires it. Concurrency should be both cheap and easy to program with.

Compositionality. Programs are built in layers. Each layer should obey the same laws.

Separation of concerns. Programs have to satisfy many concerns, some of which partially independent or contradictory. For example, a program may have to be both distributed and fault tolerant. The language should support a programming style that separates concerns.

Dependency clarity. The language should make dependencies between program parts clear and should not impose useless dependencies.

2.2 A chat room

Our second application is a distributed chat room. New users can register with the chat room and read all previous messages. New messages sent by any user will be seen by all users in the same order. Figure 2.3 shows each user’s interface.

To make the chat room realistic, we will make it distributed, open, and robust. By distributed we mean that it potentially runs on many processes. By open we mean that anyone on the Internet can enter and leave it at will. By robust we mean that there is a degree of fault tolerance: users are not perturbed by crashes of other users or even the chat room itself.
All in all, we have four major requirements for the chat room: it should have the right functionality, be distributed, be open, and be robust. How can we write a program that satisfies all these requirements? We will go over each requirement separately. To make things concrete, we assume that the chat room has a client/server structure. Users are the clients. Users register with the server. When the user sends a message, it goes to the server, which multicasts it to all known clients. The server keeps a record of all messages.

The first requirement is functionality. This one is easy to satisfy: just write a simple chat program that runs on one machine. But how do we add the other three requirements? The Mozart system gives us lots of support:

- **Distribution**: Oz is *network-transparent*: what a program does is independent of how it is distributed. That is, if we take the simple chat program and run parts of it on different processes, then it will still work.\(^1\)

- **Openness**: How do we run parts of a program on different processes? Oz lets us reference any language entity, for example, an object, with an ASCII string called a *ticket*. To connect two processes, it’s enough that a ticket is passed from one to the other.

- **Robustness**: How can the program resist crashes? Real fault tolerance is a complicated affair! Oz helps us by providing *fault detection* inside the Oz language, which lets us implement sophisticated fault tolerance inside the language.

It turns out that it is possible to build *generic tools* that satisfy all of these requirements at once. A single tool can take a simple chat room that runs in one process, and convert it into a distributed, open, robust chat room. The Mozart system provides two such tools, ClientServer and RedundantServer, that take any almost application with a client/server architecture and make it distributed, open, and robust. The major difference between the two tools is the degree of fault tolerance. ClientServer allows the client to reconnect dynamically to another server, if the server crashes. RedundantServer implements the server as two processes. If one process crashes, the other detects this and restarts it. This is invisible to the user. Chapter 13 shows how to write the chat server with either of these tools. Both tools are written entirely in Oz using the reflective failure model. The chapter explains the design of these tools and shows how to build your own powerful tools using the distribution model and failure model.

### 2.2.1 The architecture

The chat room has two parts: a server that stores the conversation and keeps track of all the clients, and a set of clients, each of which has a simple user

\(^1\)In Chapter 13, we will see that this is efficient: it results in network operations that are both few and predictable. We say that Oz is *network-aware*. 

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
interface that allows it to enter text and display the conversation. Each time a client enters a line of text, it sends it to the server, which then forwards it to all the clients (including the one that originated the text). This guarantees that each client sees exactly the same text. When a new client arrives, it registers itself with the server, which then sends it all the current text. Figure 2.4 shows the active entities in the chat room and how they communicate. In this figure, the objects could be either active or passive, and the messages can be sent either synchronously or asynchronously. When developing the application, we found that sending the messages synchronously gives the best “feel” because a user knows when his message has been accepted by the other clients.

2.2.2 Making it open and distributed

We make the chat room open by using the infrastructure of the World-Wide Web. With this infrastructure, local files can be given URLs, or Universal Resource Locators. URLs were first introduced with the World-Wide Web, but a URL gives a world-wide address that is useful for much more than just Web browsing. A file with a URL can be loaded from anywhere in the world. We store the chat room’s ticket in such a file. Then users can enter a chat room by knowing its URL.

2.2.3 More on fault tolerance

The ClientServer tool provides a simple kind of fault tolerance: it allows the server to ignore client crashes and it lets clients reconnect to another server when the server crashes. When the client tries to send to a crashed server, the window of Figure 2.5 appears. This window gives the user the option of trying to connect again to the same or other URL or to quit. Of course, when a new server comes
up the old conversation is lost. This is because the state of the crashed server is not transferred to the new server.

2.2.4 Concepts and techniques used

The chat room example introduces many of the important concepts in designing and building robust distributed applications in Oz. Here is a brief summary:

- The first principle of distribution in Oz is *network transparency*: the functionality of a program is the same independent of how it is distributed.

- The second principle of distribution in Oz is *network awareness*: network communication is simple and predictable. For example, calling a stationary object is done with a single message. Synchronizing on the result requires another message. Code for a class is transferred automatically upon first object access; no further transfers are needed. Network awareness allows the network behavior to be efficient.

- Openness is based on the concept of *ticket*, which is an ASCII string containing printable characters that represents a reference into a Mozart process. Any Mozart process can accept this string, thus transparently connecting the processes together.

- Tickets are useful because they can be transferred by means of a global addressing infrastructure such as the World-Wide Web’s URLs. For example, a URL can refer to a file containing a ticket. This allows applications to find each other anywhere in the world.

- There is a powerful two-step method to write distributed applications. First, write and test an application in one process. Second, feed this application as input to a *generic tool* that makes it distributed.

- Generic tools can make applications open, distributed, and robust all at once. The Mozart system provides many generic tools. Two of them, ClientServer and RedundantServer, are useful for client/server applications.
• A program fragment will often need resources, which are modules that are tied to a particular process. A functor is a software component specification; it defines the program’s source code along with the resources it needs. Installing the functor on a process will create a module and link it with the resources it needs.

• Compiled functors are first-class data structures in Oz. They can be created at run time and stored in other data structures. ChatClient.ozf and ChatServer.ozf are examples of compiled functors stored in files.

• The ClientServer tool takes any stateless client/server application, written as two functors, and makes it open, distributed, and partially fault tolerant: a client is able to detect a server failure and try to reconnect automatically. This is completely generic: the original two functors assume an application that runs within a single process.

• The RedundantServer tool takes any client/server application, including ones with stateful servers, and makes it distributed, open, and truly fault tolerant. The tool implements the server as two processes which each do the server’s calculations. Failure detection and recovery are done by the tool. The tool requires only two procedures that allow to access and set the server state.

• Tools like ClientServer and RedundantServer can be written completely within the language by using a reflective failure model.

2.3 An intelligent minesweeper

This section gives the architecture of a minesweeper game with a graphical user interface and a digital assistant. This application exploits several features of the Oz language, such as lexical scoping, symbolic data, concurrency, dataflow and constraint programming. Section 2.3.1 describes the game itself and its user interface. Section 2.3.2 presents the general architecture of the application and the implementation of the objects at the heart of the game.

2.3.1 Description

The minesweeper is a puzzle game. The playing field is a grid of squares, each of which may hide a mine. The total number of mines is given when the game starts. The objective is to find out where they are located.

The rules are pretty simple. (1) You uncover a square by clicking on it. If you explode a mine, the game is over. Otherwise, the number in the square gives the number of adjacent mines. (2) You mark a square as a mine by clicking on it with

---

2The minesweeper application and much of its explanation is due to Raphaël Collet.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
the middle button of the mouse. This helps to visualise your knowledge about the field. (3) You win the game when all the non-mined squares have been uncovered. To help you, a counter in the window indicates either the number of remaining mines to be marked, or the number of remaining squares to be uncovered. Our example do the latter.

Figure 2.6 shows the user interface during a game. The playing field has dimensions 10×10 and contains 20 mines. We can see that 39 squares have been uncovered and 8 squares have been marked with a cross. A message tells us that 41 squares are still to be uncovered.

The checkbuttons under the mine field reveal an important feature of our minesweeper application: the digital assistant. A digital assistant is a piece of software that helps a user to complete a task. Our assistant consists of intelligent agents that either infer new knowledge about the hidden mine field, or automatically play squares that are known safe. We call it intelligent because of its ability to solve nontrivial problems. The picture shows that the digital assistant is active because the buttons are checked. It has inferred, for instance, that the square at the tenth row and fourth column is a mine. The assistant also allows us to perform a local search by right-clicking on the square at the fifth row and sixth column. If we do this, it will infer that the clicked square is not a mine, and the autoplaying agent will uncover that square. An important property of the digital assistant is that it never infers wrong knowledge.

2.3.2 The architecture

Figure 2.7 gives an overview of the architecture of the application. It divides into three parts:

- The first part is a Game object, which implements the game itself. Interac-
The second part represents the human player. It comprises the human player and the user interface. The user interface also controls the digital assistant.

- The third part is the digital assistant. It comprises inference modules, which deduce new knowledge, and an Autoplay module, which plays automatically depending on the current knowledge. These modules are active objects.

The simplicity of the interaction between modules is what makes the architecture interesting. The digital assistant is completely independent of the rest; the game still works if it is left out.

### 2.3.3 Playing the game

We give scenarios showing how the game is played and how the digital assistant can help. The digital assistant deduces safe moves (squares with no mines). There are four levels of deduction:

- **No deduction.** The human player is completely on his own.

- **Zero propagation.** The digital assistant knows if a square has zero mines in it, that all neighboring squares are safe to play. This is a trivial deduction that is implemented by most existing minesweeper games.

- **Local deduction.** The digital assistant knows that the number in a square equals the sum of the number of mines in the neighboring squares. This performs significantly better than zero propagation.
Global deduction. The digital assistant performs global deductions on large parts of the playing field. That is, it exhaustively tries all possibilities, prunes away the impossible ones, and sees what common information exists for the others. This performs significantly better than local deduction.

At all levels, the player has the choice of letting the digital assistant make the safe moves automatically (called “autoplay”) or just indicating them on the board. With global deduction, it is often the case that the human player only has to do one or two moves in the whole game. All other safe moves are deduced by the digital assistant. The deduction is so strong that it can be used to deduce mathematical properties of the game itself. Experiments indicate that the game is only interesting if the number of mines is between 15% and 20% of the total number of squares. Below this percentage, only trivial deductions are needed most of the time and the human player almost always wins. Above this percentage, there are enough mines to block deduction, so the human player almost always loses.

2.3.4 Concepts and techniques used

Here is a summary of the concepts and techniques illustrated by the minesweeper application:

- Lexical scoping allows an object to hide data from other entities. This is the very simple and reliable way to implement a game object which hides a mine field that other entities try to guess.

- Several agents collaborate without knowing each other, simply by sharing data.

- Partial knowledge can be naturally represented by partially bound data, like records with unbound fields. Such a representation of knowledge makes programs easier to read and understand.

- Most “automatic” actions are implemented by means of dataflow. Dataflow is a natural way to block actions until some event occurs. When used with a state, it allows to implement agents whose activity can be switched on and off.

- A graphical library called QTk builds a user interface from its complete stateless definition. This gives a way to derive a user interface from data.

2.4 Exercises
Chapter 3

Introduction to Programming

“There is no royal road to geometry.”
– Euclid’s reply to Ptolemy, *Euclid* (c. 300 BC)

“Just follow the yellow brick road.”

This chapter gives a gentle, hands-on introduction to many of the programming concepts of this book. We assume you have had some previous exposure to programming. We also assume you have a computer in front of you on which the Mozart system is installed. We use the interactive interface of Mozart to introduce programming concepts in a progressive way.

This chapter only scratches the surface of the programming concepts and techniques we will see in this book. Later chapters extend it with many new concepts and new techniques.

3.1 A calculator

Let us start by using the system to do calculations. Start the Mozart system by typing:

```
oz
```

or by double-clicking a Mozart icon. This opens an editor window with two panes. In the top pane, type the following line:

```
{Browse 9999*9999}
```

Use the mouse to select this line. Now go to the *Oz* menu and select *Feed Region*. This feeds the selected text to the system. The system then does the calculation 9999*9999 and displays the result, 99980001, in a special window called the *browser*.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
3.2 Variables

While working with the calculator, we would like to remember an old result, so that we can use it later without retyping it. We can do this by declaring a variable:

```
declare v=9999*9999
```

This declares \( v \) and binds it to \( 99980001 \). We can use this variable later on:

```
(Browse v*v)
```

This displays the answer \( 9996000599960001 \).

Variables are just short-cuts for values. That is, they cannot be assigned more than once. But you can declare another variable with the same name as a previous one. This means that the old one is no longer accessible. But previous calculations, which used the old variable, are not changed. This is because there are in fact two concepts hiding behind the word “variable”:

- The variable identifier. This is what you type in. For example, a variable identifier can be the capital letter “\( V \)”. 
- The store variable. This is what the system uses to calculate with. It is part of the system’s memory, which we call its store.

The `declare` statement creates a new store variable and makes the variable identifier refer to it. Old calculations are not changed because they use another store variable.

3.3 Functions

Let us do a more involved calculation. Assume we want to calculate the factorial function \( n! \), which is defined as \( 1 \times 2 \times \cdots \times (n-1) \times n \). Factorial of 10 is:

```
(Browse 1*2*3*4*5*6*7*8*9*10)
```

This displays \( 3628800 \). What if we want to calculate the factorial of 100? We would like the system to do the tedious work of typing in all the integers from 1 to 100. We will do more: we will tell the system how to calculate the factorial of any \( n \). We do this by defining a function:

```
declare fun {Fact N}
    if N==0 then 1 else N*{Fact N-1} end
end
```

The keyword `declare` says we want to define something new. The keyword `fun` starts a new function. The function is called `Fact` and has one argument \( N \). The argument is a local variable, i.e., it is known only inside the function body. Each time we call the function a new variable is declared.
The function body is an instruction called an \textbf{if} expression. When the function is called then the \textbf{if} expression does the following steps:

- It first checks whether $N$ is equal to 0 by doing the test $N==0$.
- If the test succeeds, then the expression after the \textbf{then} is calculated. This just returns the number 1. This is because the factorial of 0 is 1.
- If the test fails, then the expression after the \textbf{else} is calculated. That is, if $N$ is not 0, then the expression $N\times\text{Fact }N-1$ is done. This expression uses \texttt{Fact}, the very function we are defining! This is called \textit{recursion}. It is perfectly normal and no cause for alarm. \texttt{Fact} is recursive because the factorial of $N$ is simply $N$ times the factorial of $N-1$. \texttt{Fact} uses the following mathematical definition of factorial:

\begin{align*}
0! & = 1 \\
n! & = n \times (n-1)! \text{ if } n > 0
\end{align*}

which is recursive.

Now we can try out the function:

\{Browse \{Fact 10\}\}

This should display 3628800 as before. This gives us confidence that \texttt{Fact} is doing the right calculation. Let us try a bigger input:

\{Browse \{Fact 100\}\}

This will display a \textit{huge} number:

\begin{verbatim}
933 26215 44394 41526 81699 23885 62667 00490
71596 82643 81621 46859 29638 95217 59999 32299
15608 94146 39761 56518 28625 36979 20827 22375
82511 85210 91686 40000 00000 00000 00000 00000
\end{verbatim}

This is an example of arbitrary precision arithmetic, sometimes called “infinite precision” although it is not infinite. The precision is limited by how much memory your system has. A typical personal computer can handle hundreds of thousands of digits. The skeptical reader will ask: is this huge number really the factorial of 100? How can we tell? Doing the calculation by hand would take a long time and probably be incorrect. We will see later on how to gain confidence that the system is doing the right thing.

\textbf{Combinations}

Let us write a function to calculate the number of combinations of $r$ items taken from $n$. This is equal to the number of subsets of size $r$ that can be made from
a set of size $n$. This is written $\binom{n}{r}$ in mathematical notation and pronounced “$n$ choose $r$”. It can be defined as follows using the factorial:

\[
\binom{n}{r} = \frac{n!}{r! (n-r)!}
\]

which leads naturally to the following function:

```
declare
fun {Comb N R}
   {Fact N} div ({Fact R}*{Fact N-R})
end
```

For example, (Comb 10 3) is 120, which is the number of ways that 3 items can be taken from 10. This is not the most efficient way to write Comb, but it is probably the simplest.

### Functional abstraction

The function Comb calls Fact three times. It is always possible to use existing functions to help define new functions. This principle is called functional abstraction because it uses functions to build abstractions. In this way, large programs are like onions, with layers upon layers of functions calling functions.

### 3.4 Lists

Now we can calculate functions of integers. But an integer is really not very much to look at. Say we want to calculate with lots of integers. For example, we would like to define one function that calculates the whole nth row in one swoop. The nth row has $n$ integers in it. We can do it by using lists of integers. A list is just a sequence of elements, bracketed at the left and right, like [1 2 3 4 5]. The empty list is written nil (and not []). Lists can be displayed just like numbers:

```
{Browse [1 2 3 4 5]}
```
3.5 Functions over lists

We create a list by means of the vertical bar “|” (pronounced “cons”):

```
declare
H=1
T=[2 3 4 5]
{Browse H|T}
```

This displays H|T, a new list whose head is 1 and whose tail is [2 3 4 5]. The new list is just [1 2 3 4 5]. We can take apart an existing list, getting back the head and tail:

```
declare
L=[5 6 7 8 9]
{Browse L.1}
{Browse L.2}
```

The operation L.1 gets back the head, namely 5. The operation L.2 gets back the tail, namely [6 7 8 9]. This uses the dot operator “.”, which is used to select the first or second parts of the cons pair. Another way to take apart a list is by using the `case` instruction, which gets both head and tail in one step:

```
declare
L=[5 6 7 8 9]
case L of H|T then {Browse H} {Browse T} end
```

This declares two local variables, H and T, and binds them to the head and tail of the list L. This displays 5 and [6 7 8 9], just like before. Local variables declared with a `case` are just like variables declared with `declare`, except that the variable exists only between the `then` and the `end`.

3.5 Functions over lists

Now that we can calculate with lists, let us define a function, \( \text{Pascal N} \), to calculate the nth row of Pascal’s triangle. Let us first understand how to do the

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
calculation by hand. Figure 3.1 shows how to calculate the fifth row from the fourth. Let us see how this works if each row is a list of integers. To calculate a row, we start from the previous row. We shift it left by one position and shift it right by one position. We then add the two shifted rows together. For example, take the fourth row:

\[
[1 \ 3 \ 3 \ 1]
\]

We shift this row left and right and then add them together:

\[
[1 \ 3 \ 3 \ 1 \ 0] \\
+ [0 \ 1 \ 3 \ 3 \ 1]
\]

Note that shifting left adds a zero to the right and shifting right adds a zero to the left. Doing the addition gives:

\[
[1 \ 4 \ 6 \ 4 \ 1]
\]

which is the fifth row. Now we can write a function to do the same operations. Here it is (for brevity we omit the obligatory declare):

\[
\text{fun \{Pascal N\}} \\
\text{\hspace{1cm} if \ N==1 \ then \ [1]} \\
\text{\hspace{1cm} else} \\
\text{\hspace{3.5cm} \{AddList \ \{ShiftLeft \ \{Pascal \ N-1\}\}} \\
\text{\hspace{3.5cm} \{ShiftRight \ \{Pascal \ N-1\}\}} \\
\text{\hspace{1cm} end} \\
\text{end}
\]

This does not completely solve the problem. We have to define three more functions: \text{ShiftLeft}, which shifts left by one position, \text{ShiftRight}, which shifts right by one position, and \text{AddList}, which adds two lists. Here are \text{ShiftLeft} and \text{ShiftRight}:

\[
\text{fun \{ShiftLeft L\}} \\
\text{\hspace{1cm} case \ L \ of \ H|T \ then} \\
\text{\hspace{3.5cm} H|\{ShiftLeft T\}} \\
\text{\hspace{1cm} else \ [0] \ end} \\
\text{end}
\]

\[
\text{fun \{ShiftRight L\} 0|L \ end}
\]

\text{ShiftRight just adds a zero to the left. \text{ShiftLeft} traverses \ L one element at a time and builds the output one element at a time. We have added an \text{else} to the case instruction. This is similar to an \text{else} in an \text{if}: it is executed if the pattern of the case does not match. That is, when \ L is empty then the output is \ [0], i.e., a list with just zero inside. Here is \text{AddList}:

\[
\text{fun \{AddList L1 L2\}} \\
\text{\hspace{1cm} case \ L1 \ of \ H1|T1 \ then} \\
\text{\hspace{4.5cm} \{AddList \ H1 \ \{AddList \ T1 \}} \\
\text{\hspace{4.5cm} \{AddList \ T1 \}} \\
\text{\hspace{1cm} case \ L2 \ of \ H2|T2 \ then} \\
\text{\hspace{4.5cm} \{AddList \ H2 \ \{AddList \ T2 \}} \\
\text{\hspace{4.5cm} \{AddList \ T2 \}}
\]
This uses two case instructions, one inside another, because we have to take apart both L1 and L2. By the way, Pascal, ShiftLeft, ShiftRight, and AddList should be fed to the system all at once with just one declare keyword prefacing all four functions (why?).

Now we can calculate any row of Pascal’s triangle. For example, the 20th row is:

\[
[1 \ 19 \ 171 \ 969 \ 3876 \ 11628 \ 27132 \ 50388 \ 75582 \ 92378 \ 92378 \ 75582 \ 50388 \ 27132 \ 11628 \ 3876 \ 969 \ 171 \ 19 \ 1]
\]

Is this answer correct? How can you tell? It looks right: it is symmetric (reversing the list gives the same list) and the first and second arguments are 1 and 19, which are right. Looking at Figure 3.1, it is easy to see that the second element of the nth row is always \(n - 1\) (it is always one more than the previous row and it starts out zero for the first row). In the next section, we will see how to reason about correctness.

### Top-down program development

Let us summarize the technique we used to write Pascal:

- The first step is to understand how to do the calculation by hand.
- The second step writes a function to solve the problem, assuming that some auxiliary functions (here, ShiftLeft, ShiftRight, and AddList) are known.
- The third step completes the solution by writing the auxiliary functions.

The technique of first writing the required function and filling in the blanks afterwards is known as top-down program development.

### 3.6 Correctness

A program is correct if it does what we would like it to do. How can we tell whether a program is correct? Usually it is impossible to duplicate the program’s calculation by hand. We need other ways. One simple way, which we used before, is to verify that the program is correct for outputs that we know. This increases confidence in the program. But it does not go very far. To verify correctness in general, we have to reason about the program. This means three things:

- We need a mathematical model of the operations of the programming language, defining what they should do. This model is called the semantics of the language.
• We need to define what we would like the program to do. Usually, this is a mathematical definition of the inputs that the program needs and the output that it calculates. This is called the program’s specification.

• We use mathematical techniques to reason about the program, using the semantics. We would like to demonstrate that the program satisfies the specification.

**Mathematical induction**

One very useful technique is mathematical induction. This proceeds in two steps. We first show that the program is correct for the simplest cases. Then we show that, if the program is correct for a given case, then it is correct for the next case. If we can make an ordered list of all possible cases, then we can conclude the program is always correct. This technique can be applied for integers and lists:

• For integers, the base case is 0 or 1, and for a given integer \(n\) the next case is \(n + 1\).

• For lists, the base case is \texttt{nil} (the empty list) or a list with one or a few elements, and for a given list \(T\) the next case is \(H|T\) (with no conditions on \(H\)).

Let us see how induction works for the factorial function:

• \(\texttt{Fact 0}\) returns the correct answer, namely 1.

• Assume that \(\texttt{Fact N-1}\) is correct. Then look at the call \(\texttt{Fact N}\). We see that the \texttt{if} instruction takes the \texttt{else} case, and calculates \(N \times \texttt{Fact N-1}\). By hypothesis, \(\texttt{Fact N-1}\) returns the right answer. Therefore, assuming that the multiplication is correct, \(\texttt{Fact N}\) also returns the right answer.

This reasoning uses the mathematical definition of factorial, namely \(n! = n \times (n - 1)!\) if \(n > 0\), and \(0! = 1\). Later in the book we will see more sophisticated reasoning techniques. But the basic approach is always the same: start with the language semantics and problem specification, and use mathematical reasoning to show that the program correctly implements the specification.

### 3.7 Complexity

The \texttt{Pascal} function we defined above gets very slow if we try to calculate higher-numbered rows. Row 20 takes a second or two. Row 30 takes many minutes. If you try it, wait patiently for the result. How come it takes this much time? Let us look again at the function \texttt{Pascal}:
fun {Pascal N}
  if N==1 then [1]
  else
    {AddList {ShiftLeft {Pascal N-1}}
    {ShiftRight {Pascal N-1}}}
  end
end

Calling {Pascal N} will call {Pascal N-1} two times. Therefore, calling {Pascal 30} will call {Pascal 29} twice, giving four calls to {Pascal 28}, eight to {Pascal 27}, and so forth, doubling with each lower row. This gives $2^{29}$ calls to {Pascal 1}, which is about half a billion. No wonder that {Pascal 30} is slow. Can we speed it up? Yes, there is an easy way: just call {Pascal N-1} once instead of twice. The second call gives the same result as the first, so if we could just remember it then one call would be enough. We can remember it by using a local variable. Here is a new function, FastPascal, that uses a local variable:

fun {FastPascal N}
  if N==1 then [1]
  else L in
    L={FastPascal N-1}
    {AddList {ShiftLeft L} {ShiftRight L}}
  end
end

We declare the local variable L by adding “L in” to the else part. This is just like using declare, except that the variable exists only between the else and the end. We bind L to the result of {FastPascal N-1}. Now we can use L wherever we need it. How fast is FastPascal? Try calculating row 30. This takes minutes with Pascal, but is done practically instantaneously with FastPascal. A lesson we can learn from this example is that using a good algorithm is more important than having the best possible compiler or fastest machine.

Exponential and polynomial functions

The execution time of a program as a function of input size, up to a constant factor, is called the program’s time complexity. The time complexity of {Pascal N} is proportional to $2^n$. This is an exponential function in n, which grows very quickly as n increases. What is the time complexity of {FastPascal N}? There are n recursive calls, and each call processes a list of average size $n/2$. Therefore its time complexity is proportional to $n^2$. This is a polynomial function in n, which grows at a much slower rate than an exponential function. Programs whose time complexity is exponential are impractical except for very small inputs. Programs whose time complexity is a low-order polynomial are practical.
3.8 Lazy evaluation

The functions we have written so far will do their calculation as soon as they are called. This is called *eager* evaluation. Another way to evaluate functions is called *lazy* evaluation.\(^1\) In lazy evaluation, a calculation is done only when the result is needed. Here is a simple lazy function that calculates a list of integers:

```
fun lazy {Ints N}
    N|{Ints N+1}
end
```

Calling `{Ints 0}` calculates the infinite list `0|1|2|3|4|5|...`. This looks like it is an infinite loop, but it is not. The `lazy` annotation ensures that the function will only be evaluated when it is needed. This is one of the advantages of lazy evaluation: we can calculate with potentially infinite data structures without any loop boundary conditions. For example:

```
L={Ints 0}
(Browse L)
```

This displays the following, i.e., nothing at all:

```
L<Future>
```

(The browser displays values but does not affect their calculation.) The “Future” annotation means that `L` has a lazy function attached to it. If the value of `L` is needed, then this function will be automatically called. Therefore to get more results, we have to do something that needs the list. For example:

```
(Browse L.1)
```

This displays the first element, namely 0. We can calculate with the list as if it were completely there:

```
case L of A|B|C|_ then {Browse A+B+C} end
```

This causes the first three elements of `L` to be calculated, and no more. What does it display?

Lazy calculation of Pascal’s triangle

Let us do something useful with lazy evaluation. We would like to write a function that calculates as many rows of Pascal’s triangle as are needed, but we do not know beforehand how many. That is, we have to look at the rows to decide when there are enough. Here is a lazy function that generates an infinite list of rows:

```
fun lazy {PascalList Row}
    Row|{PascalList
        {AddList {ShiftLeft Row}
            {ShiftRight Row}}}
end
```

\(^1\)These are sometimes called *supply-driven* and *demand-driven* evaluation, respectively.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Calling this function and browsing it will display nothing:

```
declare
L=\{PascalList [1]\}
{Browse L}
```

(The argument [1] is the first row of the triangle.) To display more results, they have to be needed:

```
{Browse L.1}
{Browse L.2.1}
```

This displays the first and second rows.

Instead of writing a lazy function, we could write a function that takes \( N \), the number of rows we need, and directly calculates those rows:

```
fun {PascalList2 N Row}
  if N==1 then [Row]
  else
    Row|{PascalList2 N-1 {AddList {ShiftLeft Row} {ShiftRight Row}}}
  end
end
```

We can display 10 rows by calling \{Browse {PascalList2 10 [1]}\}. But what if later on we decide that we need 11 rows? We would have to call \texttt{PascalList2} again, with argument 11. This would redo all the work of defining the first 10 rows. The lazy version avoids redoing all this work. It is always ready to continue where it left off.

### 3.9 Higher-order programming

We have written an efficient function, \texttt{FastPascal}, that calculates rows of Pascal's triangle. Now we would like to experiment with variations on Pascal's triangle. For example, instead of adding numbers to get each row, we would like to subtract them, exclusive-or them (to calculate just whether they are odd or even), or many other possibilities. One way to do this is to write a new version of \texttt{FastPascal} for each variation. But this quickly becomes tiresome. Can we somehow just have one generic version? This is indeed possible. Let us call it \texttt{GenericPascal}. Whenever we call it, we pass it the customizing function (adding, exclusive-oring, etc.) as an argument. The ability to pass functions as arguments is known as \textit{higher-order programming}.

Here is the definition of \texttt{GenericPascal}. It has one extra argument \texttt{Op} to hold the function that calculates each number:

```
fun {GenericPascal Op N}
  if N==1 then [1]
  else L in
end
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
L={\text{GenericPascal Op N-1}}\\\{\text{OpList Op \{ShiftLeft L\} \{ShiftRight L\}}\}
end
end

AddList is replaced by OpList. The extra argument Op is passed to OpList.
ShiftLeft and ShiftRight do not need to know Op, so we can use the old
versions. Here is the definition of OpList:

\[
\text{fun } \{\text{OpList Op L1 L2}\} \\
\quad \text{case } L1 \text{ of } \text{H1}\text{|T1 then} \\
\quad \quad \text{case } L2 \text{ of } \text{H2}\text{|T2 then} \\
\quad \quad \quad \{\text{Op H1 H2}\}|\{\text{OpList Op T1 T2}\} \\
\quad \text{else nil end end}
\]

Instead of doing an addition H1+H2, this version does \{\text{Op H1 H2}\}.

To try out this version, we need to define functions that we can pass as
arguments. For example, we can define the addition function:

\[
\text{fun } \{\text{Add X Y}\} \{\text{X+Y}\} \text{ end}
\]

Now we can run \{\text{GenericPascal Add 5}\}.\text{\textsuperscript{2}} This gives the fifth row exactly as
before. We can define FastPascal in terms of GenericPascal:

\[
\text{fun } \{\text{FastPascal N}\} \{\text{GenericPascal Add N}\} \text{ end}
\]

Let us define another function:

\[
\text{fun } \{\text{Xor X Y}\} \{\text{if X==Y then 0 else 1 end end}\}
\]

This does an exclusive-or operation, which is defined as follows:

\[
\begin{array}{ccc}
X & Y & \{\text{Xor X Y}\} \\
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

Exclusive-or lets us calculate the parity of each number in Pascal’s triangle, i.e.,
whether the number is odd or even. The numbers themselves are not calculated.
Calling \{\text{GenericPascal Xor N}\} gives the result:

\[
\begin{array}{cccccc}
1 \\
1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 \\
\end{array}
\]

\text{\textsuperscript{2}}We can also call \{\text{GenericPascal Number.´+´ 5}\}, since the plus operation ´+´
is part of the module Number. But modules are not introduced in this chapter.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
3.10 Concurrency

We would like our program to have several independent activities, each of which executes at its own pace. This is called concurrency. There should be no interference between the activities, unless the programmer decides that they need to communicate. This is how the real world works outside of the system. We would like to be able to do this inside the system as well.

We introduce concurrency by creating threads. A thread is simply an executing program like the functions we saw before. The difference is that a program can have more than one thread. Threads are created with the thread instruction. Do you remember how slow the original Pascal function was? We can call Pascal inside its own thread. This means that it will not keep other calculations from continuing. They may slow down, if Pascal really has a lot of work to do. This is because the threads share the same underlying computer. But none of the threads will stop. Here is an example:

```
thread P in
  P={Pascal 30}
  {Browse P}
end
{Browse 99*99}
```

This creates a new thread. Inside this new thread, we call {Pascal 30} and then browse the result. The new thread has a lot of work to do. But this does not keep the system from displaying 99*99 immediately.

3.11 Dataflow

What happens if an operation tries to use a variable that is not yet bound? From a purely aesthetic point of view, it would be nice if the operation would simply wait. Perhaps some other thread will bind the variable, and then the operation can continue. This civilized behavior is known as dataflow. Figure 3.2 gives a simple example: the two multiplications wait until their arguments are bound and the addition waits until the multiplications complete. As we will see later in the book, there are many good reasons to have dataflow behavior. For now, let us see how dataflow and concurrency work together. Take for example:

```
declare X in
thread {Delay 10000} X=99 end
{Browse start} {Browse X*X}
```

The multiplication X*X waits until X is bound. The first browse immediately displays start. The second browse waits for the multiplication, so it displays...
nothing yet. The \{Delay 10000\} call pauses for 10000 milliseconds (i.e., 10 seconds). \(x\) is bound only after it continues. When \(x\) is bound, then the multiplication continues and the second browse displays 9801. The two operations \(x=99\) and \(x\times x\) can be done in any order with any kind of delay; dataflow execution will always give the same result. The only effect a delay can have is to slow things down. For example:

\[
\text{declare } x \text{ in} \\
\text{thread } \{\text{Browse start}\} \{\text{Browse } x\times x\} \text{ end} \\
\{\text{Delay 10000} \} \; x=99
\]

This behaves exactly as before: the browser displays 9801 after 10 seconds. This illustrates two nice properties of dataflow. First, calculations work correctly independent of how they are partitioned between threads. Second, calculations are patient: they do not signal errors, but simply wait.

Adding threads and delays to a program can radically change a program’s appearance. But as long as the same operations are invoked with the same arguments, it does not change the program’s results at all. This is the key property of dataflow concurrency. This is why dataflow concurrency gives most of the advantages of concurrency without the complexities that are usually associated with it.

### 3.12 State

How can we let a function learn from its past? That is, we would like the function to have some kind of internal memory, which helps it do its job. Memory is needed for functions that can change their behavior and learn from their past. This kind of memory is called \textit{state}.\textsuperscript{3} Like for concurrency, state models an essential aspect of how the world works outside of the system. We would like to be able to do this in the system as well. Later in the book we will see deeper reasons for having state. For now, let us just see how it works.

\textsuperscript{3}In subsequent chapters, we call it \textit{encapsulated state} to emphasize that it is internal to the function.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
For example, let us take the \texttt{FastPascal} function we defined before, which calculates the nth row of Pascal's triangle. If it could somehow remember the results of previous calls, then it would not have to recalculate rows each time it is called. How do we add memory to \texttt{FastPascal}?

\section*{A memory cell}

There are lots of ways to define state. The simplest way is to define a single \textit{memory cell}. This is a kind of box in which you can put any contents. Many programming languages call this a “variable”. We call it a “cell” to avoid confusion with the variables we used before, which are more like mathematical variables, i.e., just short-cuts for values. There are three functions on cells: \texttt{NewCell} creates a new cell, \texttt{Assign} puts a new value in a cell, and \texttt{Access} gets the current value stored in the cell. For example:

\begin{verbatim}
declare C={NewCell 0}
{Assign C {Access C}+1}
{Browse {Access C}}
\end{verbatim}

This creates a cell \texttt{C} with initial content 0, adds one to the content, and then displays it.

\section*{A memory store}

Another way to define state is by means of a \textit{memory store}. A memory store is similar to the memory of a computer. It has a series of memory cells, numbered from 1 up to the maximum used so far. There are four functions on memory stores: \texttt{NewStore} creates a new store, \texttt{Put} puts a new value in a memory cell, \texttt{Get} gets the current value stored in a memory cell, and \texttt{Size} gives the highest-numbered cell used so far. For example:

\begin{verbatim}
declare S={NewStore}
{Put S 2 [22 33]}
{Browse {Get S 2}}
{Browse {Size S}}
\end{verbatim}

This stores [22 33] in memory cell 2, displays [22 33], and then displays 2.

\section*{Using the memory store}

For \texttt{FastPascal}, we can use a memory store to remember previously-calculated rows. Here’s how:

- First make a store \texttt{S} available to \texttt{FastPascal}.
- For the call \{\texttt{FastPascal}\}, let M be the number of rows stored in \texttt{S}, i.e., rows 1 up to M are in \texttt{S}.  

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
3.13 Nondeterminism

We have seen how to add concurrency and state to a program separately. What happens when a program has both? It turns out that having both at the same time is a tricky business, because the same program can give different results from one execution to the next. This is because the order in which threads access the state can change from one execution to the next. This variability is called nondeterminism. Nondeterminism by itself is not a problem; we already have it with concurrency. The difficulties occur when the nondeterminism shows up in the program, i.e., if it is observable. Here is an example:

```haskell
declare
c={NewCell 0}
thread
{Assign C 1}
end
thread
{Assign C 2}
end
```

What is the content of C after this program executes? Figure 3.3 shows the two possible executions of this program. Depending on which one is done, the final cell content can be either 1 or 2. The problem is that we cannot say which. This is a simple case of observable nondeterminism. Things can get much trickier. For example, let us use a cell to hold a counter that can be incremented by several threads:
3.14 Atomicity

What is the content of C after this program executes? It looks like each thread just adds 1 to the content, making it 2. But there is a surprise lurking: the final content can also be 1! How is this possible? Try to figure out why before reading on.

The content can be 1 because thread execution is interleaved. That is, threads take turns each executing a little. We have to assume that any possible interleaving can occur. For example, consider the execution of Figure 3.4. Both I and J are bound to 0. Then, since I+1 and J+1 are both 1, the cell gets assigned 1 twice. The final result is that the cell content is 1.

This is a simple example. More complicated programs have many more possible interleavings. Programming with concurrency and state together is largely a question of mastering the interleavings. In the history of computer technology, many famous and dangerous bugs were due to designers not realizing how difficult this really is. The Therac-25 radiation therapy machine is an infamous example. It sometimes gave its patients radiation doses that were thousands of times greater than normal, resulting in death or serious injury [61].

This leads us to a first lesson for programming with state and concurrency: if at all possible, do not use them together! It turns out that we often do not need both together. When a program does need to have both, it can almost always be designed so that their interaction is limited to a very small part of the program.

3.14 Atomicity

Let us think some more about how to program with concurrency and state. One way to make it easier is to use atomic operations. An operation is atomic if it
is all or nothing. That is, either it is completely done or not done at all. No intermediate situations, where it is partially done, can be observed.

With atomic operations we can solve the interleaving problem of the cell counter. The idea is to make sure that each thread body is atomic. To do this, we need a way to build atomic operations. We introduce a new language entity, called lock, for this. A lock has an inside and an outside. The programmer defines the instructions that are inside. A lock has the property that only one thread at a time can be executing inside. If a second thread tries to get in, then it will wait until the first gets out. Therefore what happens inside the lock is atomic.

We need two operations on locks. First, we create a new lock by calling the function NewLock. Second, we define the lock’s inside with the instruction lock L then ... end, where L is a lock. Now we can fix the cell counter:

\begin{verbatim}
declare
C={NewCell 0}
L={NewLock}
thread
   lock L then I in
      I={Access C}
      {Assign C I+1}
   end
end
thread
   lock L then J in
      J={Access C}
      {Assign C J+1}
   end
end
\end{verbatim}

In this version, the final result is always 2. Both thread bodies have to be guarded by the same lock, otherwise the undesirable interleaving can still occur. Do you see why?

### 3.15 Exercises

1. Section 3.3 defines the function Comb to calculate combinations. This function is not very efficient because it might require calculating very large factorials. The purpose of this exercise is to write a more efficient version of Comb.

   - As a first step, use the following alternative definition to write a more efficient function:

     \[
     \binom{n}{r} = \frac{n \times (n-1) \times \cdots \times (n-r+1)}{r \times (r-1) \times \cdots \times 1}
     \]

     Calculate the numerator and denominator separately and then divide them. Make sure that the result is 1 when \( r = 0 \).
• As a second step, use the following identity:

\[
\binom{n}{r} = \binom{n}{n-r}
\]

...to increase efficiency even more. That is, if \( r > n/2 \) then do the calculation with \( n - r \) instead of with \( r \).

2. Section 3.6 explains the basic ideas of program correctness and applies them to show that the factorial function defined in Section 3.3 is correct. In this exercise, apply the same ideas to the function `Pascal` of Section 3.5 to show that it is correct.

3. What does Section 3.6 say about programs whose time complexity is a high-order polynomial? Are they practical or not? What do you think?

4. Section 3.8 defines the lazy function `Ints` that lazily calculates an infinite list of integers. Let us define a function that calculates the sum of a list of integers:

```plaintext
fun {SumList L}
    case L of H|T then H+{SumList T}
    else 0 end
end
```

What happens if we call `{SumList {Ints 0}}`? Is this a good idea?

5. Section 3.9 explains how to use higher-order programming to calculate variations on Pascal’s triangle. The purpose of this exercise is to explore these variations.

• Calculate individual rows using subtraction, multiplication, and other operations. Why does using multiplication give a triangle with all zeroes? Try the following kind of multiplication instead:

```plaintext
fun {Mul1 X Y} (X+1)*(Y+1) end
```

What does the 10th row look like when calculated with `Mul1`?

• The following loop instruction will calculate and display 10 rows at a time:

```plaintext
for I in 1..10 do {Browse {FastPascal Op I}} end
```

Use this loop instruction to make it easier to explore the variations.

6. This exercise compares variables and cells. We give two code fragments. The first uses variables:
local X in
   X=23
local X in
   X=44
end
   {Browse X}
end

The second uses a cell:

local X in
   X={NewCell 23}
   {Assign X 44}
   {Browse {Access X}}
end

In the first, the identifier X refers to two different variables. In the second, X refers to a cell. What does Browse display in each fragment? Explain.

7. Write a version of FastPascal that uses a memory store, following the description given in Section 3.12. In your solution, you may assume that the four operations on memory stores are available in the system.

8. Define the four operations of a memory store in terms of a memory cell. The cell holds the store contents as a list of the form [[N1 X1] ... [Nn Xn]], where [Ni Xi] means that cell number Ni has content Xi.

9. Section 3.13 gives an example using a cell to store a counter that is incremented by two threads.

   - Try executing this example several times. What results do you get? Do you ever get the result 1? Why could this be?
   - Modify the example by adding calls to Delay in each thread. This changes the thread interleaving without changing what calculations the thread does. Can you devise a scheme that always results in 1?
   - Section 3.14 gives a version of the counter that never gives the result 1. What happens if you use the delay technique to try to get a 1 anyway?
Part II

General Computation Models
Chapter 4

Declarative Computation Model

“Non sunt multiplicanda entia praeter necessitatem.”
“Do not multiply entities beyond necessity.”
– Ockham’s Razor, William of Ockham (1285–1349?)

Programming encompasses three things:

• First, a computation model, which is where the computation is done. It contains the data entities, the operations that can be performed on them, and a language to express programs that do these operations.

• Second, a set of programming techniques to let you express the problems you want to solve as programs in the language.

• Third, a set of reasoning techniques to let you reason about programs, to increase confidence that they behave correctly.

The above definition of computation model is very general. What is a reasonable computation model? Intuitively, we will say that a reasonable model is one that can be used to solve many problems and that has straightforward and practical reasoning techniques. We will have more to say about this question later on. The first and simplest computation model we will study is declarative programming. For now, we define this as evaluating functions over partial data structures. This is sometimes called stateless programming, as opposed to stateful programming (also called imperative programming) which is explained in Chapter 8. As we will see, declarative programming is one of the most fundamental computation models. It encompasses the core ideas of the two main declarative “paradigms”, namely functional and logic programming. It encompasses programming with strict functions over complete values, as in Scheme and ML. It also encompasses deterministic logic programming, as in Prolog when search is not used.

Declarative programming is a rich area–most of the ideas of the more expressive computation models are already there, at least in embryonic form. We therefore present it in two chapters. The present chapter defines the computation
model and its programming language. The next chapter, Chapter 5, presents the programming techniques of this language. Later chapters enrich this basic model with concurrency (including lazy execution), state (including objects), distribution (including fault tolerance), and constraints (including search). The present chapter consists of four sections:

- Section 4.1 explains how to define the syntax and semantics of practical programming languages. Syntax is defined by a context-free grammar extended with language constraint. Semantics is defined by translating into a simple kernel language. These techniques will be used in this chapter and the rest of the book. This chapter uses them to define the declarative computation model.

- Section 4.2 presents a brief overview of the declarative computation model. This model calculates functions over partial values. This section introduces the concepts of single-assignment store and partial value.

- Section 4.3 introduces the basic concepts of the declarative model, namely variables, data types, binding and equality, and suspension. These concepts are introduced by means of many examples that can be run on the Mozart system. The section introduces the Mozart system’s interactive interface as well as the Browser, a tool for displaying data structures.

- Section 4.4 gives the declarative language constructs. This includes the basic constructs, namely `skip`, `if`, variable declaration and binding, procedures, and exceptions. This section also introduces the concepts of lexical scoping, procedural abstraction, anonymous procedures (i.e., lambda expressions), and environment trimming. Finally, the section defines two important linguistic abstractions, namely functions and the `case` statement, and an important syntactic short-cut, namely implicit variable initialization.

4.1 Defining practical programming languages

Programming languages are much simpler than natural languages, but they can still have a surprisingly rich syntax, set of grammatical constructs, and libraries. This is especially true for languages that are used to solve real-world problems, which we call practical languages. A practical language is like the toolbox of an experienced mechanic: there are many different tools for many different purposes and all tools are there for a reason.

This section sets the stage for the rest of the book by explaining how we will present the syntax (“grammar”) and semantics (“meaning”) of practical programming languages. With this foundation we will then be ready to present the
4.1 Defining practical programming languages

first computation model of the book in Section 4.2, namely the declarative computation model. We will continue to use these techniques throughout the book to define computation models.

4.1.1 Language syntax

The syntax of a language defines what are the legal programs, i.e., programs that can be successfully executed. At this stage we do not care what the programs are actually doing; that is semantics and will be handled in the next section.

Grammars

A grammar defines how to make ‘sentences’ out of ‘words’. Grammars can be used for natural languages, like English or Swedish, as well as for artificial languages, like programming languages. For programming languages, ‘sentences’ are usually called ‘statements’ and ‘words’ are usually called ‘tokens’. Just as words are made of letters, tokens are made of characters. This gives us two levels of structure:

- statement (‘sentence’) = sequence of tokens (‘words’)
- token (‘word’) = sequence of characters (‘letters’)

Grammars are useful both for defining statements and tokens.

Extended Backus-Naur Form

One of the most common notations for defining grammars is called Extended Backus-Naur Form (EBNF for short), after its inventors John Backus and Peter Naur. The EBNF notation distinguishes terminal symbols and nonterminal symbols. A terminal symbol is simply a token. A nonterminal symbol represents a sequence of tokens. The nonterminal is defined by means of a grammar rule, which shows how to expand it into tokens. For example, the following rule defines the nonterminal ⟨digit⟩:

⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

It says that ⟨digit⟩ represents one of the ten tokens 0, 1, ..., 9. The symbol “|” is read as “or”; it means to pick one of the alternatives. Grammar rules can themselves refer to other nonterminals. For example, we can define a nonterminal ⟨integer⟩ that defines how to write positive integers:

⟨integer⟩ ::= ⟨digit⟩ { ⟨digit⟩ }

This rule says that an integer is a digit followed by zero or more digits. The braces “{ ... }” mean to repeat whatever is inside any number of times, including zero.
Context-free grammar
(e.g., with EBNF)

- Is easy to read and understand
- Defines a superset of the language

+ 

Set of extra conditions

- Expresses restrictions imposed by the language
  (e.g., variables must be declared before use)
- Makes the grammar context-sensitive

Figure 4.1: The context-free approach to language syntax

How to read grammars

To read a grammar, start with any nonterminal symbol, say \( \text{integer} \). Reading the corresponding grammar rule from left to right gives a sequence of tokens according to the following scheme:

- Each terminal symbol encountered is added to the sequence.
- For each nonterminal symbol encountered, read its grammar rule and replace the nonterminal by the sequence of tokens that it expands into.
- Each time there is a choice (with |), pick any of the alternatives.

The grammar can be used both to verify that a statement is legal and to generate statements.

Context-free and context-sensitive grammars

The set of all the possible statements generated by a grammar and one nonterminal symbol is called a formal language, or language for short. For a given language, it is not always possible to find a grammar that generates it. Techniques to define grammars can be classified according to how expressive they are, i.e., how many languages they can define. For example, the EBNF notation given above defines a class of grammars called context-free grammars. They are so-called because the expansion of a nonterminal, e.g., \( \text{digit} \), is always the same no matter where it is used.

For most practical languages, it is usually not possible to give a context-free grammar that exactly defines all legal programs and no others. For example, in many languages a variable has to be declared before it is used. This condition cannot be expressed in a context-free grammar because the nonterminal that uses the variable must only allow using already-declared variables. This is a context dependency.

The syntax of most practical programming languages is defined in two parts (see Figure 4.1): as a context-free grammar complemented with a set of extra conditions imposed by the language. The context-free grammar is kept because
it is easy to read and understand. It has an important locality property: a
nonterminal symbol can be understood by examining only the rules needed to
define it; the (possibly much more numerous) rules that use it can be ignored.
The context-free grammar is corrected by imposing a set of extra conditions,
like the declare-before-use restriction on variables. Taking these conditions into
account gives a context-sensitive grammar.

Syntax notation used in this book

In this chapter and the rest of the book, each new data type and language con-
struct is introduced together with a small syntax diagram that shows how it fits
in the whole language. The syntax diagram gives grammar rules for a simple
context-free grammar. The diagrams define a sequence of tokens. The notation
is carefully designed to satisfy two basic principles:

- All grammar rules can stand on their own. No later information will ever
  invalidate a grammar rule. That is, we never give an incorrect grammar
  rule just to “simplify” the presentation.

- It is always clear by inspection when a grammar rule completely defines a
  nonterminal symbol or when it only gives a partial definition. A partial
  definition always ends in three dots “...”.

All the syntax diagrams are summarized in Appendix C. This appendix also
defines the syntax of tokens in terms of characters.

The tokens in the grammar rules may be separated by any amount of blank
space and comments. Blank space is one of the characters tab (character code
9), newline (code 10), vertical tab (code 11), form feed (code 12), carriage return
(code 13), and space (code 32). A comment is one of three possibilities:

- A sequence of characters starting from the character % (percent) until the
  end of the current line or file (whichever comes first).

- A sequence of characters starting from /* and ending with *//, inclusive.
  This kind of comment may be nested.

- The single character ? (question mark).

Here is an example of a syntax diagram with two grammar rules that illustrates
our notation:

\[
\begin{align*}
\text{<statement>} & \::= \text{skip} \mid \text{<expression>} = \text{<expression>} \mid ... \\
\text{<expression>} & \::= \text{<variable>} \mid \text{<integer>} \mid ...
\end{align*}
\]

These rules give partial definitions of two nonterminals, <statement> and <expression>.
The first rule says that a statement can be the keyword skip, or two expressions
separated by the equals symbol =, or something else. The second rule says that
an expression can be a variable, an integer, or something else. To avoid confusion
with the grammar rule's own syntax, a symbol that occurs literally in the text is always quoted with single quotes. For example, the equals symbol is shown as ‘=’. Keywords are not quoted, since for them no confusion is possible. A choice between different possibilities in the grammar rule is given by a vertical bar |. 

Here is a second example to give the remaining notation:

\[
\begin{align*}
\langle \text{statement} \rangle & ::= \ \textbf{if} \ \langle \text{expression} \rangle \ \textbf{then} \ \langle \text{statement} \rangle \\
& \{ \ \textbf{elseif} \ \langle \text{expression} \rangle \ \textbf{then} \ \langle \text{statement} \rangle \ \} \\
& \ [ \ \textbf{else} \ \langle \text{statement} \rangle \ \] \ \textbf{end} \ | \ ... \\
\langle \text{expression} \rangle & ::= \ `\{ ` \{ \langle \text{expression} \rangle \}+ `\} ` | ... \\
\langle \text{label} \rangle & ::= \ \textbf{unit} | \ \textbf{true} | \ \textbf{false} | \ \langle \text{variable} \rangle | \ \langle \text{atom} \rangle
\end{align*}
\]

The first rule defines the if statement. There is an optional sequence of elseif clauses, i.e., there can be any number of occurrences including zero. This is denoted by the braces \{ ... \}. This is followed by an optional else clause, i.e., it can occur zero or one times. This is denoted by the brackets [ ... ]. The second rule defines the syntax of explicit lists. They must have at least one element, e.g., \{5 6 7\} is valid but \{\} is not (note the space). This is denoted by \{ ... \}+. The third rule defines the syntax of record labels. This is a complete definition. There are five possibilities and no more will ever be given.

### 4.1.2 Language semantics

The semantics of a language defines what a program does when it executes. Ideally, the semantics should be defined in a simple mathematical structure that lets us reason about the program (including its correctness, execution time, and memory use) without introducing any irrelevant details. Can we achieve this for a practical language without making the semantics too complicated? The technique we use, which we call the kernel approach, gives an affirmative answer to this question.

Modern programming languages have evolved through more than five decades of experience in constructing programmed solutions to complex, real-world problems. Modern programs can be quite complex, reaching sizes measured in millions of lines of code, written by teams of human programmers over many years. In our view, languages that scale successfully to this level of complexity are successful in part because they model some essential aspects of how to construct complex programs. In this sense, these languages are not just arbitrary constructions of the human mind. We would therefore like to understand them in a scientific way, i.e., by explaining their behavior in terms of a simple underlying model. This is the deep motivation behind the kernel approach.

---

1. The figure of five decades is somewhat arbitrary. We measure it from the first working stored-program computer, the Manchester Mark I, which ran its first documented program on June 21, 1948 [83].

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Figure 4.2: The kernel approach to language semantics

The kernel approach

This book uses the kernel approach to define language semantics. In this approach, all language constructs are defined in terms of translations into a core language known as the kernel language. The kernel approach consists of two parts (see Figure 4.2):

- First, define a very simple language, called the kernel language. Ideally, this language should be easy to reason in. The kernel language and the data structures it manipulates together form the kernel computation model.

- Second, define a translation scheme from the full programming language to the kernel language. Each grammatical construct in the full language is translated into the kernel language. The translation should be as simple as possible. There are two kinds of translation, namely linguistic abstraction and syntactic sugar. Both are explained below.

The kernel approach is used throughout the book. Each computation model has its kernel language, which builds on its predecessor by adding one new concept. The first kernel language, which is presented in Section 4.2, is called the declarative kernel language.

The kernel approach leaves us free to define the semantics of the kernel language in any way we want, e.g., with an operational, axiomatic, or denotational semantics. This book mostly uses an operational semantics, supplemented with a few axiomatic or denotational concepts when they are useful. We choose the operational approach because it is the only one we know that is simple, practical, and works for all computation models of the book.
Throughout the book, we give an informal semantics for every new language construct and we often reason informally about programs. These informal presentations are always based on the formal operational semantics of the kernel language, which is given in Chapter 15.

Linguistic abstraction

Both programming languages and natural languages can evolve to meet their needs. When using a programming language, at some point we may feel the need to extend the language, i.e., to add a new linguistic construct. For example, the declarative model of this chapter has no looping constructs. Section 5.7.6 defines a for construct to express certain kinds of loops that are useful for writing declarative programs. The new construct is both an abstraction and an addition to the language syntax. We therefore call it a linguistic abstraction. A practical programming language consists of a set of linguistic abstractions.

There are two phases to defining a linguistic abstraction. First, define a new grammatical construct. Second, define its translation into the kernel language. The kernel language is not changed. This book gives many examples of useful linguistic abstractions, e.g., functions (fun), pattern matching (case), loops (for), lazy functions (fun lazy), list comprehensions, classes (class), and mailboxes (receive)\(^2\).

A good example of a linguistic abstraction is the concept of function, introduced with the fun construct in Section 4.4.9. We have already programmed with functions in the previous chapter. The declarative kernel language we introduce in this chapter has no functions but only procedures. Since functions are very useful to have in a language, we define them as a linguistic abstraction. We define a syntax for function definitions and function calls, and a translation into procedure definitions and procedure calls. The translation lets us answer all questions about functions. For example, what does \{F1 {F2 X} {F3 Y}\} mean exactly (nested function calls)? In what order are the three functions called?

Syntactic sugar

It is often convenient to provide a short-cut notation for frequently-occurring idioms. This notation is part of the language syntax and is defined by grammar rules. This notation is called syntactic sugar. Syntactic sugar is analogous to linguistic abstraction in that its meaning is defined precisely by translating it into the full language. But it should not be confused with linguistic abstraction: it does not provide a new abstraction, but just reduces program size and improves program readability.

Two substantial examples of syntactic sugar in Oz are implicit variable initialization, given in Section 4.3.8, and nesting notation, given in Section 4.4.8.

\(^2\)Linguistic abstractions can be added to the Mozart system with the gump parser-generator tool [57]. Using this tool is beyond the scope of this book.
Here we give a third example, namely syntactic sugar that is based on the \texttt{local} statement. Local variables can always be defined by using the statement \texttt{local x \texttt{in} \ldots \texttt{end}}. When this statement is used inside another, it is convenient to have syntactic sugar that lets us leave out the keywords \texttt{local} and \texttt{end}. Instead of:

\begin{verbatim}
if N==1 then [1]
else
  local L in
  ...
  end
end
\end{verbatim}

we can write:

\begin{verbatim}
if N==1 then [1]
else L in
  ...
end
\end{verbatim}

which is both shorter and more readable than the full notation.

**Further discussion of the kernel approach**

The language used in this book, Oz, has the additional property that the kernel language is a subset of the full language. The translation of a program into the kernel language gives exactly the same results as the program. This is useful both for learning the kernel language and for debugging programs.

The implementation of a practical language can implement certain constructs more efficiently than their representation in the kernel language, as long as the implementation has the same effect as the kernel language translation. For example, classes and objects in Oz are implemented more efficiently than their kernel definitions. This gives a second kernel language between the original kernel language and the practical language. The second kernel language is mostly ignored in this book, but it is an important part of an efficient implementation.

**Lifecycle of abstractions**

Any abstraction that we define has two phases in its lifecycle. When first we define it, it has no linguistic support. If at some point, we find that it is especially basic and useful, we can decide to give it linguistic support. It then becomes a linguistic abstraction. The second phase is the true sign that the abstraction has “made it big time”: it is so basic and useful that it should be as easy to use as possible. Most abstractions never reach the second phase because they are too specialized.
Other translation approaches

The kernel approach is an example of the translation approach to semantics, i.e., it is based on a translation from one language to another. Figure 4.3 shows the three ways that the translation approach has been used for defining programming languages:

- The kernel approach, used throughout the book, is intended for the programmer. Its concepts correspond directly to programming concepts.

- The calculus approach is intended for the mathematician. Examples are the Turing machine, the lambda calculus (underlying functional programming), first-order logic (underlying logic programming), and the pi calculus (to model concurrency). Because these calculi are intended for formal mathematical study, they have as few elements as possible.

- The machine approach is intended for the implementor. Programs are translated into an idealized machine, which is traditionally called an abstract machine or a virtual machine.\(^3\) It is relatively easy to translate idealized machine code into real machine code.

Because we focus on practical programming techniques, this book uses only the kernel approach.

\(^3\)Strictly speaking, a virtual machine is a software emulation of a real machine, running on the real machine, that is almost as efficient as the real machine. It achieves this efficiency by executing most virtual instructions directly as real instructions. The concept was pioneered by IBM in the early 1960's in the VM operating system. Because of the success of Java, which uses the term “virtual machine”, modern usage tends to blur the distinction between abstract and virtual machines.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
4.2 The declarative computation model

We define the declarative model in three steps. We first explain the single-assignment store and dataflow variables. We then introduce the kernel language and explain what a computation is. Finally, we introduce the concept of environment, which maps textual variable names to entities in the store. The rest of the chapter explains all the statements in the kernel language, the syntactic conveniences that we use to make programming easier, all the basic types and their operations.

4.2.1 Single-assignment store and dataflow variables

“They are neither values, nor uninitialized variables, nor yet nothing. May we not call them the ghosts of values yet to come...?”
– freely adapted from The Analyst, George Berkeley (1685–1753)

All data structures in the declarative model reside in a value store, i.e., a store...
that contains just values. A value is a mathematical constant. For example, the integer 314 is a value. Values can also be compound entities. For example, the list [1 2 3] and the record person(name:"George" age:25) are values. Having a value store is sufficient for functional programming, but not quite enough for the declarative model of this chapter.

For many reasons, we generalize slightly the value store to be a single-assignment store. This store contains dataflow variables, which can each be either unbound or bound. A data structure that may or may not contain unbound variables is called a partial value.

Dataflow variables

A dataflow variable starts out by being unbound. It can be bound to exactly one value. Once bound, it stays bound to the same value. Once bound, it is indistinguishable from the value. In the rest of the book, when we talk about variables we always mean dataflow variables. Dataflow variables are useful for many reasons. Some of these will become apparent only in later computation models. We will give one good reason now and put off explaining the others until Chapter 6. Consider the operations of declaring and binding a variable. Assume that we would like to declare a variable separately from binding it. Then what happens if we try to use the variable before it is bound, i.e., if we commit a use error? There are many ways a language can treat this kind of error:

1. The variable's content can be undefined, i.e. it is “garbage”: whatever is found in memory. No error message is given. The content can be used in calculations, giving a result that can be anything.

2. The variable can be initialized to a default value when it is declared, e.g., to 0 for an integer.

3. An error message can be given if there is an attempt to use the variable before it is bound.

4. An attempt to use the variable before it is bound will wait until the variable is bound.

These possibilities are listed in increasing order of “niceness”. The first is terrible, since a program with a use error can literally give any result. The second is a little better. If the program has a use error, then at least it will always give the same result, even if it is a wrong one. The third is even better. A program with a use error will signal this fact, instead of silently continuing to work. The fourth is the best. It gives the right result without programmer effort. The program will wait until the variable is bound, and then continue.

The real advantage of the fourth solution will become apparent when we do concurrent programming, i.e., programs with activities that run independently. If
4.2 The declarative computation model 71

we do two concurrent operations, say $A=23$ and $B=A+1$, then with the fourth solution this will always run correctly and result in $B=24$. It doesn’t matter whether $A=23$ is tried first or whether $B=A+1$ is tried first. With the other solutions, there is no guarantee of this. This property of order-independence makes possible the declarative concurrency of Chapter 6. It is at the heart of why dataflow variables are a good idea.

Partial values

Partial values are the bread and butter of declarative programming. All data structures in the declarative model are potentially partial values. When all their variables are bound, then they become complete values, which we just call values.

An unbound dataflow variable can be used as if it were bound. In other words, it can be put in data structures and passed to procedures. It is only when the program attempts to use the content that the computation waits until the variable is bound.

Let us give an example of dataflow variables and partial values. Assume the store contains three unbound variables $X$, $Y$, and $Z$. Doing the binding $X=[Y Z]$ binds $X$ to a list of the two variables $Y$ and $Z$. The list $[Y Z]$ is a partial value. If $Y$ is bound to 2 and $Z$ is bound to 3, then $X$’s value is $[2 3]$. $Y$ and $Z$ have been replaced by 2 and 3 in the partial value. Binding a variable always replaces it in all partial values that contain it. We summarize this in the following box.

Variables and partial values

A dataflow variable, also called variable, is an entity that resides in a single-assignment store, that is initially un-bound, and that can be bound to exactly one value. It can be bound to several partial values, as long as these partial values are compatible with each other. A partial value is a data structure that may contain unbound variables. When one of these variables is bound, then it is replaced by the partial value it is bound to. A complete value, also called value, is a data structure that does not contain any unbound variables.

4.2.2 Kernel language

The declarative computation model defines a simple language called the kernel language. Table 4.1 shows the syntax of a statement in the kernel language. The language is called kernel because it has no linguistic abstractions or syntactic conveniences. The kernel language contains the control constructs needed to express all possible executions.

For simplicity, Table 4.1 leaves out most of the basic types (integers, records, etc.) and their operations (arithmetic, comparisons, etc.). This information is

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Table 4.1: The declarative kernel language

given in Appendix B. We will refer to this appendix whenever we introduce a basic type.

4.2.3 Computations

In the computation model, a running program has at all times an execution state. Figure 4.4 shows the execution state, which consists of two parts: a statement \( S \) and a single-assignment store \( \sigma \). A running program is defined in terms of a computation. A computation is a sequence of execution states: \((S_0, \sigma_0) \rightarrow (S_1, \sigma_1) \rightarrow ...\). One transition in this sequence is called a computation step. A computation step is atomic, i.e., it is is done “all at once”; there are no visible intermediate states. In this chapter, all computations are sequential, i.e., the execution state contains exactly one statement, which is transformed by a linear sequence of computation steps.

4.2.4 Environments

A statement is executed in the context of an environment, which is a mapping from textual variable names, also called variable identifiers, to actual variables in the store. Another name for an executing statement is a thread. In later chapters we will have multiple threads that interact, which is called concurrency. For now, we have exactly one thread. The thread can access the store. It is able to manipulate the store by introducing new variables, by binding variables to partial values, and by reading variables’ values. A thread can access information only through the variables visible to it, directly or indirectly. By directly we mean that one of the thread’s statements references the variable. By indirectly we mean that the variable is in a data structure that is directly or indirectly accessible by the thread.
4.3 Basic concepts

The previous section gave a brief definition of the declarative model including the kernel language, with some motivation. The kernel language seems simple, but what do its constructs mean? This section and the next will answer that question. We first explain the basic concepts: variables, data types, binding and equality, and suspension. To make things clear, we give many code examples that can all be run directly on the Mozart system using the interactive interface. Along the way we introduce the Browser, an indispensable tool for displaying information in the store. Section 4.4 then explains the language constructs of the kernel language together with the concepts of lexical scoping and procedural abstraction.

4.3.1 Variable declarations and environments

There is one statement whose sole purpose is to declare new variable identifiers and introduce new variables corresponding to these identifiers. Its syntax is given in Table 4.2. To explain how it executes, consider the example

```
local X Y Z in (statement) end
```

which executes the following steps:

---

Table 4.2: The `local` statement for declaring variables

4.2.5 Exceptions

If a program raises an exception, is it still declarative? There are two viewpoints:

- A function raises an exception if its arguments are outside of its domain. If we consider the execution undefined outside of the domain, then raising an exception is no longer declarative.

- The exception is considered to be part of the function’s range. The arguments that cause the exception are therefore part of the function’s domain. In this case, the function is still declarative.

The first viewpoint is usually the easiest to reason with, although the second can be useful in certain circumstances. Section 5.12.1 explains how to program with exceptions in the declarative model. Section 6.1.3 explains what happens when the model is extended to be concurrent. Surprisingly, exceptions are no longer declarative when used in a concurrent setting.

---

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
First, it creates three new variables in the store. We call them \( \text{Unb1}, \text{Unb2}, \text{Unb3} \), just so that we can talk about them. These names are not the textual names of the variables, but are just three names that we pick so that we can show what the environment looks like.

Second, it creates a new environment associated with \( \langle \text{statement} \rangle \). The new environment extends the previous environment with mappings from the textual names \( X, Y, \) and \( Z \) to the corresponding new variables \( \text{Unb1}, \text{Unb2}, \text{Unb3} \). The new environment looks like \([..., X=\text{Unb1}, Y=\text{Unb2}, Z=\text{Unb3}]\), where \([...]\) is the old environment. The textual names are also called variable identifiers or just identifiers.

Third, it executes \( \langle \text{statement} \rangle \) in the new environment.

Finally, after \( \langle \text{statement} \rangle \) completes, the old environment is restored.

If the identifiers \( X, Y, \) or \( Z \) existed in the previous environment, then they will become inaccessible in \( \langle \text{statement} \rangle \). After \( \langle \text{statement} \rangle \) completes, they will become accessible again. In Table 4.2, the nonterminal symbol \( \langle \text{variable} \rangle \) stands for a variable identifier. There are two ways to write a variable identifier:

- An upper-case letter followed by zero or more alphanumeric characters (letters or numbers) or underscore characters.
- Any sequence of printable characters enclosed within ` (back-quote) characters, e.g., `this is a 25$\text{variable!}`.

A precise definition of identifier syntax is given in Appendix C. All newly-declared variables are unbound before any statement is executed. All variables must be declared explicitly, except in certain pattern matching constructs to be shown later.

### 4.3.2 The interactive interface

The Mozart system has an interactive interface that allows to introduce and execute program fragments. These fragments are not arbitrary; they have to respect
the syntax of interactive statements, which is given in Table 4.3. An *interactive statement* is either any legal statement or a new form, the `declare` statement. We assume that the user feeds interactive statements to the system one by one. The interactive interface allows to do much more than just feed statements; it has all the functionality needed for program development. Appendix A explains how to use the interactive interface. For now, we assume that the user just knows how to feed statements.

The interactive interface has a single, global environment. The `declare` statement augments this environment with new mappings. It follows that the `declare` statement can only be used interactively, not in separately-compiled programs. Feeding the following fragment:

```declare X Y Z```

creates three new variables in the store and augments the global environment permanently. We say that \( X, Y, Z \) are *global variables*. Feeding this declaration again will cause \( X, Y, \) and \( Z \) to map to fresh variables. A *fresh* variable is simply a variable that is different from all previously-existing variables.

There is a second form of `declare`. Feeding:

```declare X Y Z in (statement)```

behaves like a *local* declaration that augments the global environment permanently. It is like a *local* declaration whose *end* part is at the end of the interactive session.

In the rest of the book, all program examples given will be acceptable for feeding to the interactive interface. For reasons of brevity in the text, we often leave out the initial keyword “`declare`”. This keyword should be added if the program example needs to declare new variables.

### 4.3.3 Basic data types and the type hierarchy

When performing computations, programs manipulate many different values. We group possible values into sets called data types. A *data type* or *type* is a set of values. A value is “of a type” if it is in the type’s set. For example, programs can compute with integers, which are all of the integer type. The Mozart system comes with a set of predefined types, which are called *basic data types*. In addition, programs can define their own types, which are called *abstract data types*, ADT for short. In Chapter 5 and later chapters we show how to define ADTs.

The basic types in Mozart include numbers (integers, floats, and characters), literals (atoms and names), records, chunks, tuples, lists, strings, and virtual strings. These types are explained in Appendix B. In addition to these basic types, there are others including procedures, functions, functors, cells, ports, classes, and objects. Procedures and functions are introduced later in this chapter. The other types are given in later chapters.

The basic types in Mozart can be classified into a hierarchy. Figure 4.5 shows part of this hierarchy. In this hierarchy, each node represents a type. A type
Figure 4.5: The type hierarchy

Table 4.4: Expressions for calculating with numbers

represents the set of all values of the type. Any variable, if it ever gets a value, will be bound to a value of one of these types. The hierarchy is ordered by set inclusion, i.e., all values of a node’s type are also values of the parent node’s type. For example, all tuples are records and all lists are tuples. A few of the nodes in Figure 4.5, namely Number, Literal, and Bool, are completely defined by their children in the figure. That is, all numbers are either integers or floats. This is not true of the other nodes. E.g., there are tuples that are not literals, lists, or virtual strings. All descendants of a node are disjoint (i.e., share no values) except for two cases. List and VirtualString share the node String. Atom and String share the node Nil. Four types, Nil, Unit, True, and False, contain just one element, which is written nil, unit, true, and false respectively.
4.3.4 Expressions

An expression is a syntactic short-cut for a sequence of operations that returns a value. It is our first example of syntactic sugar, as defined in Section 4.1.2. It is different from a statement, which is also a sequence of operations but does not return a value. An expression can be used inside a statement whenever a value is needed. For example, 11*11 is an expression and X=11*11 is a statement. Table 4.4 shows the syntax of expressions that calculate with numbers. Later on we will see expressions for calculating with other data types. Expressions are built hierarchically, starting from basic expressions (e.g., variables and numbers) and combining them together. There are two ways to combine them: using operators (e.g., the addition 1+2+3+4) or using function calls (e.g., the square root {Sqrt 5.0}).

4.3.5 Dynamic typing

“During a period of exciting discovery or progress there is no time to plan the perfect headquarters.”
– Parkinson’s Law or The Pursuit of Progress, Cyril Northcote Parkinson (1909–1993)

The declarative computation model is dynamically typed. This means that the user does not have to explicitly declare types. When a variable is introduced, its type as well as its value are unknown. Only when the variable is bound to a value, does its type become determined. However, the language is still strongly typed, i.e., it does have types and their correct use is checked. The compiler tries to check types at compile time. But because of dynamic typing, not all can be checked in this way. Some checks are necessarily left for run time.

A strongly-typed language can be either dynamically-typed or statically-typed. Let us compare these two possibilities. In a dynamically-typed language, variables can be bound to entities of any type, so in general their type is known only at run time. In a statically-typed language, on the other hand, all variable types are known at compile time. The type can be declared by the programmer or inferred by the compiler. When designing a language, one of the major decisions to make is whether the language is to be dynamically typed, statically typed, or some mixture of both. What are the advantages and disadvantages of dynamic and static typing?

- Dynamic typing puts fewer restrictions on programs and programming. It allows many more programming techniques. It allows truly open programming, i.e., independently-written components can come together at run time and interact with as few assumptions as possible about each other. This flexibility is felt quite strongly in practice. The programmer spends much less time adjusting the program to fit the type system.

- Dynamic typing shortens the turnaround time between an idea and its
implementation. It allows to have an incremental development environment that is part of the run-time system. It allows to test programs or program fragments even when they are in an incomplete or inconsistent state.

- Static typing allows to catch many more program errors at compile time. The static type declarations are a partial specification of the program, i.e., they specify part of the program’s behavior. The compiler’s type checker verifies that the program satisfies this partial specification. This can be quite powerful. Modern static type systems can catch a surprising number of semantic errors.

- Static typing allows a more efficient implementation. Since the compiler has more information about what values a variable can contain, it can choose a more efficient representation. For example, if a variable is of boolean type, the compiler can implement it with a single bit. In a dynamically-typed language, the compiler cannot always deduce the type of a variable. When it cannot, then it usually has to allocate a full memory word, so that any possible value (or a pointer to a value) can be accommodated.

Unfortunately, the choice between dynamic and static typing is most often based on emotional (“gut”) reactions, not on rational argument. Adherents of dynamic typing relish the expressive freedom and rapid turnaround it gives them and criticize the reduced expressiveness of static typing. On the other hand, adherents of static typing emphasize the aid it gives them for writing correct and efficient programs and point out that it finds many program errors at compile time.

It turns out that both approaches are viable. Static typing is recommended when the programming techniques are well-understood and when efficiency is paramount. Dynamic typing is recommended for rapid development and when programs must be as flexible as possible, such as application prototypes, operating systems, and some artificial intelligence applications.

The choice between static or dynamic typing is not black or white. In each approach, a bit of the other can be added, gaining some of its advantages. For example, different kinds of polymorphism (where a variable might have values of several different types) add flexibility to statically-typed functional and object-oriented languages. It is an active research area to determine static type systems that capture as much as possible of the flexibility of dynamic type systems, while encouraging good programming style and still permitting compile time verification.

The computation models given in this book are all subsets of the Oz language, which is dynamically typed. A major goal of Oz was to allow the most possible programming techniques with as few restrictions as possible. The only way to achieve this goal is with dynamic typing.

When the programming techniques are known, then a possible next step is to design a static type system. While research in Oz and its programming techniques is still ongoing in the Mozart Consortium, the Alice project at the Universität des
4.3 Basic concepts

Table 4.5: The equality and equality test operations

| (statement)       | ::= (expression) ’=’ (expression) | ... |
| (expression)      | ::= (expression) ’==’ (expression) |
| (expression) ’\ne’ (expression) | | (variable) | (integer) | (float) | (character) | ... |

Figure 4.6: The Browser

Saarlandes in Saarbrücken is already taking this step. Alice is a statically-typed language that has some of the expressiveness of Oz.

4.3.6 Binding and testing variables

Information is added to the store by binding a variable. The simplest way to bind a variable is to use the operation =, which is called equality. Table 4.5 shows the syntax of equality. For example, feeding:

```
declare X in X=1
```

declares X as a new unbound variable and binds it to the integer 1. Declaring and binding can be done separately. For example, feeding:

```
declare X
```

declares X but does not bind it. It can be bound later on by feeding:

```
X=1
```

Making declaration and binding two separate operations is a key feature of the models in the book. It lets us do dataflow execution, as we will see later.

The Browser

The interactive interface has a tool, called the Browser, which allows to look into the store. This tool is available to the programmer as a procedure called Browse.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
The procedure Browse has one argument. It is called as \{Browse \langle expression\rangle\},
where \langle expression\rangle is any expression. It can display partial values and it will update the display whenever the partial values are bound more. Feeding the following:

\{Browse X\}
displays the value of X, namely 1. Feeding:

```
declare Y in
{Browse Y}
```
displays just the name of the variable, namely Y. No value is displayed. This means that Y is currently unbound. Figure 4.6 shows the browser window after these two operations. If Y is bound, e.g., by doing Y=2, then the browser will update its display to show this binding.

**Equality is not assignment**

The equality operation is not an assignment statement. Variables are not mutable containers, but just a way to reference values. In many languages, variables are mutable containers by default. As we will see throughout the book, this default is much too strong for many programming tasks. It makes reasoning about programs much harder. It turns out that mutable containers are not needed for many programming tasks. Chapters 5 and 6 show that much can be done without them. Chapter 8 introduces mutable containers (called “encapsulated state”) and shows exactly what additional expressive power they provide.

Equality just adds information to the store; it never changes any existing information. But adding information is more than just binding a variable to a value. For example, the argument order does not matter: binding X to 1 can be written as either X=1 or 1=X. The information is the same in either case. Another example is when X is already bound to a value. Feeding X=1 will verify that X is equal to 1. If X was previously bound to an incompatible value, i.e., any value different from 1, then an exception will be raised. Exception handling is described in Section 4.4.11.

**Equality test**

Often, we would like to know whether a variable is equal to some value without doing a binding. The operation \==, which is called equality test, does this. Table 4.5 shows the syntax of equality test. For example, assume X equals 1. Then feeding:

\{Browse X\==2\}
displays false. The operation \!= is like equality test, except that it gives the opposite result (true when == would give false, etc.).

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
The general case

It is possible to execute the equality \( X=Y \) and the equality test \( X==Y \) where both \( X \) and \( Y \) are any partial values (including unbound variables). The general cases of equality and equality test are explained in Section 4.4.13.

4.3.7 Dataflow suspension

What happens if one of the arguments of an operation is an unbound variable? Instead of flagging an error or trying to continue, the operation waits until all its arguments are bound before continuing. This behavior is called dataflow suspension. It is a common occurrence during concurrent executions, in which more than one instruction sequence can be executing at the same time. An independently-executing instruction sequence is called a thread. Programming with more than one thread is called concurrent programming; it is introduced in Chapter 6.

All the examples in this chapter execute in a single thread, i.e., they are sequential. Actually, each expression fed into the interactive interface executes in its own thread. This lets us give a simple example of concurrent execution. First, feed the following statement:

```
declare A B C in
C=A+B
{Browse C}
```

This will display nothing, since the instruction \( C=A+B \) suspends (both of its arguments are unbound). Now, feed the following statement:

```
A=10
```

This will bind \( A \), but the instruction \( C=A+B \) still suspends since \( B \) is still unbound. Finally, feed the following:

```
B=200
```

This displays 210 in the browser. Any operation, not just addition, will suspend if it does not get enough input information to calculate its result. For example, the equality test operation \( X==Y \) will suspend if it cannot decide whether or not \( X \) is equal to or different from \( Y \). This happens, e.g., if one or both of the variables are unbound.

Programming errors often result in dataflow suspensions. If you feed a statement that should display a result and nothing is displayed, then the probable cause of the problem is a suspended operation. Carefully check all operations to make sure that all their arguments are bound.

4.3.8 Implicit variable initialization

To make programs shorter and easier to read, we provide a syntactic short-cut to bind a variable immediately when it is declared. We call this a variable-declaration.
Table 4.6: Variable-declaration equality with escaped variables

equality. This is another example of syntactic sugar, as defined in Section 4.1.2. The idea is to put an equality operation between local and in. Instead of local $X$ in $X=10$ {Browse $X$} end, in which $X$ is mentioned three times, the short-cut lets one write local $X=10$ in {Browse $X$} end, which mentions $X$ only twice. A simple case is the following:

```
local
  $X=\langle\text{expression}\rangle$
in
  ...
end
```

This declares $X$ and binds it to the result of $\langle\text{expression}\rangle$. The general case is:

```
local
  $\text{Term}=\langle\text{expression}\rangle$
in
  ...
end
```

where $\text{Term}$ is any partial value. This declares all the variables in $\text{Term}$, and then binds $\text{Term}$ to the result of $\langle\text{expression}\rangle$. In both cases, the variables occurring on the left-hand side of the equality, i.e., $X$ or the variables in $\text{Term}$, are the ones declared. Table 4.6 shows the syntax: the declaration part of a local or declare can contain any statement. If the statement is an equality, then the variables on its left-hand side are implicitly declared. Here is an example:

```
local
  $Y=1$
in
  local
    $M=f(M~Y)$
    $[X~Y]=L$
    $L=[1~2]$
    $X=1$
in {Browse [M~L]} end
end
```

This example uses list and tuple types, which are explained in Appendix B. In the outer local statement, $Y$ is declared and bound to 1. Then, in the inner local statement, all variables on the left-hand sides of the equalities are introduced, i.e., $M$, $Y$, $X$, and $L$. This means that the outer variable $Y$ is invisible in the inner
local statement. The above example is equivalent to:

```plaintext
local Y in
  Y=1
  local M X Y L in
    M=f(M Y)
    [X Y]=L
    L=[1 2]
    X=1
    {Browse [M L]}
  end
end
```

Implicit variable initialization has two important properties. First, all variables are declared together before any statement is executed. Second, if a variable occurs twice in the left-hand side of an equality, then it is declared only once.

**Escaped variables**

In the inner `local`, if we want `Y` to denote the variable in the outer `local`, then we have to suppress the declaration of `Y` in the inner `local`. This can be done by using an exclamation mark as follows:

```plaintext
local
  Y=1
in
  local
    M=f(M Y)
    [X1 !Y]=L
    L=[1 2]
  in {Browse [M L]}
end
end
```

The notation `!Y` is called an escaped variable (see Table 4.6). It means that `Y` should not be declared. An exclamation mark `!` is only meaningful on the left-hand side of an initializing equality. If used in any other place, it will be ignored.

**Pattern matching**

Implicit variable initialization is convenient when taking apart a complex data structure. For example, if `T` is bound to the record `tree(key:a left:L right:R value:1)`, then just one equality is enough to extract all four fields:

```plaintext
local
  tree(key:A left:B right:C value:D)=T
```

4The exclamation mark `!` can be used in other situations where you want to suppress the introduction of new variables, for example in patterns, which we will see later on.
### 4.4 Language constructs

This section explains the different statements in the kernel language. A statement is a syntactic construct that expresses some form of computation. Statements can often be built from other statements, like bricks can be piled on top of one another to build a house. Statements are the building blocks of programs.

We have already seen three basic forms of statement in the declarative model, which are summarized in Section 4.4.1. In this section we will see six more statement forms, namely the skip statement (which does nothing), the proc statement (which defines procedures), the procedure call, the if statement (for
conditional execution), and the \texttt{try} and \texttt{raise} statements (for exceptions). Complementing these are two linguistic abstractions, for functions and pattern matching, that include the \texttt{fun} statement (which defines functions), the function call, and the \texttt{case} statement. With this full language, we can write concise and powerful algorithms in the declarative model. Chapter 5 explains the programming techniques of this model and adds a few non-declarative concepts needed to write standalone programs.

### 4.4.1 Basic statements

While introducing the basic data types of the declarative model, we have seen the following three basic forms of statement:

- Operations on data types. All operations, e.g., on numbers, records, lists, and so on, are primitive statements.

- Declaring variables: the \texttt{local} and \texttt{declare} statements. The \texttt{declare} statement is a syntactic short-cut for \texttt{local} that is used in the interactive interface; it is not part of the kernel language.

- Statement sequencing: doing one statement after another. The statement sequence \langle S1 \rangle \langle S2 \rangle is itself a single, bigger statement. Statement sequencing guarantees an order of execution, i.e., \langle S1 \rangle will be completed before starting \langle S2 \rangle. In particular, \langle S2 \rangle is not executed as long as \langle S1 \rangle suspends or if \langle S1 \rangle raises an exception. Exceptions are part of a general way of handling abnormal executions, which is explained in Section 4.4.11.

Table 4.7 gives the syntax of the basic statements.

### 4.4.2 Lexical scoping

As we saw in Section 4.3.1, a variable identifier, which is a sequence of characters, is a completely different beast from an unbound variable, which is a data structure in the store. A statement \langle S \rangle can contain many occurrences of variable identifiers. For each identifier occurrence, we can ask the question: where was this identifier declared? If the declaration is in some statement (part of \langle S \rangle or not) that textually surrounds (i.e., encloses) the occurrence, then we say that the declaration obeys \textit{lexical scoping} (also called \textit{static scoping}). Other kinds of scoping are possible, e.g., dynamic scoping. We will defer the discussion of dynamic scoping until later, since it is less generally useful than lexical scoping.

Identifier occurrences in a statement can be \textit{bound} or \textit{free} with respect to that statement. An identifier occurrence \textit{x} is \textit{bound} with respect to a statement \langle S \rangle if it is declared inside \langle S \rangle. There are several possibilities:

- A declaration statement (\texttt{local} or \texttt{declare}) that declares \textit{x}, is inside \langle S \rangle.
- Some other statement that declares \( x \), is inside \((S)\). We will introduce these statements later on. These statements include procedures, functors, and statements that contain class methods or patterns.

An identifier occurrence that is not bound is free. Free occurrences can only exist in incomplete program fragments, i.e., statements that cannot run. In a running program, it is always true that every identifier occurrence is bound.

### Bound identifiers and bound variables

Do not confuse a bound identifier with a bound variable!

A bound identifier does not exist at run time; it is a textual variable name that textually occurs inside a construct that declares it (e.g., a procedure definition or variable declaration). A bound variable exists at run time; it is a dataflow variable that is bound to a partial value.

Here is an example with free and bound variable identifiers:

```plaintext
local Arg1 Arg2 in
Arg1=111*111
Arg2=999*999
Res=Arg1+Arg2
end
```

In this statement, all variable identifiers are declared with lexical scoping. Identifiers \( \text{Arg1} \) and \( \text{Arg2} \) are bound and \( \text{Res} \) is free. What happens if you try to feed this statement? It gives a “variable \( \text{Res} \) not introduced” error since \( \text{Res} \) is not defined.\(^5\) To run this statement, it has to be part of a bigger statement that defines \( \text{Res} \). For example, it is possible to feed:

```plaintext
declare Res in
local Arg1 Arg2 in
Arg1=111*111
Arg2=999*999
Res=Arg1+Arg2
end
{Browse Res}
```

This executes without error since it defines \( \text{Res} \). It does not matter if \( \text{Res} \) was previously defined in the global environment, since among several nested declarations of an identifier, the innermost one overrides all others.

### 4.4.3 The \texttt{skip} statement

The statement \texttt{skip} is the empty statement. It can always be executed and its execution does nothing at all.

\(^5\)Unless \( \text{Res} \) was defined previously in the global environment. If \( \text{Res} \) is already bound to something else, you will get another kind of error. In that case, try using an identifier that was never used before in the interactive session.
4.4 Language constructs

| (statement) ::= skip |
| if (expression) then (statement) |
| { elseif (expression) then (statement) } |
| [ else (statement) ] end |
| ... |

Table 4.8: The **skip** and if statements

4.4.4 The if statement

The declarative model provides a simple form of conditional statement, whose syntax is given in Table 4.8. The statement

```
if (E) then (S1) else (S2) end
```

has the following informal semantics:

- First evaluate (E). If this gives **true**, then execute (S1).
- If (E) evaluates to **false**, then execute (S2).
- If (E) evaluates to any partial value that is not an unbound variable, then raise an exception.
- Otherwise, if (E) suspends or evaluates to an unbound variable, then the executing thread suspends until one of the above three cases applies.

This differs in two ways from the behavior of an if statement in conventional languages. First, because of dynamic typing, it is possible for (E) to evaluate to a non-boolean value, giving an error. Second, because of dataflow execution, it is possible for the if to suspend until it knows which alternative to take. This does not make a difference in sequential programs, but it will become very important when we introduce concurrency in Chapter 6.

The declarative model provides a number of boolean comparison functions. These include `==` and `\=`, which can compare any two values for equality, as well as the numeric comparisons `=<=`, `<`, `>=`, and `>`, which can compare two integers, two floats, or two atoms. Atoms are compared according to the lexicographic order of their print representations. In the following example, `Z` is bound to the maximum of `X` and `Y`:

```
declare X Y Z in
X=5 Y=10
if X>Y then Z=X else Z=Y end
{Browse Z}
```

Like all operations, comparison functions have correct dataflow behavior. For example:
**Table 4.9: Procedures**

```plaintext
declare X Y Z in
if X>=Y then Z=X else Z=Y end
{Browse Z}
```

Since \( x \) and \( y \) are unbound, the comparison \( x \geq y \) suspends and nothing is displayed.

An **if** statement with an **elseif** clause:

```plaintext
if \langle E1 \rangle then \langle S1 \rangle elseif \langle E2 \rangle then \langle S2 \rangle else \langle S3 \rangle end
```

is an abbreviation for two nested **if** statements:

```plaintext
if \langle E1 \rangle then \langle S1 \rangle else
  if \langle E2 \rangle then \langle S2 \rangle else \langle S3 \rangle end
end
```

An **if** statement missing the **else** clause:

```plaintext
if \langle E1 \rangle then \langle S1 \rangle end
```

is equivalent to:

```plaintext
if \langle E1 \rangle then \langle S1 \rangle else skip end
```

### 4.4.5 Procedures

Procedures are one of the most important basic building blocks of any language. In the declarative model, it is possible to take any statement and make it into in a procedure, which can be called in one line. For example, consider the statement:

```plaintext
local A=1 B=3 C=2 D RealSol X1 X2 in
D=B*B-4*A*C
if D>=0 then
  RealSol=true
  X1=(˜B+{Sqrt D})/(2.0*A)
  X2=(˜B-{Sqrt D})/(2.0*A)
else
  RealSol=false
  X1=˜B/(2.0*A)
  X2={Sqrt ˜D}/(2.0*A)
end
{Browse RealSol#X1#X2}
end
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
This calculates the solutions of the quadratic equation $x^2 + 3x + 2 = 0$. It uses the quadratic formula $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, which gives the two solutions of the equation $ax^2 + bx + c = 0$. The value $d = b^2 - 4ac$ is called the discriminant: if it is positive or zero, then there are two real solutions. Otherwise, the two solutions are conjugate complex numbers. The above statement can be converted into a procedure by using it as the body of a procedure definition and passing the free variables as arguments:

```plaintext
declare proc {QuadraticEquation A B C RealSol X1 X2}
  D=B*B-4*A*C
  in
    if D>=0 then
      RealSol=true
      X1=(˜B+{Sqrt D})/(2.0*A)
      X2=(˜B-{Sqrt D})/(2.0*A)
    else
      RealSol=false
      X1=˜B/(2.0*A)
      X2={Sqrt ˜D}/(2.0*A)
    end
  end
end
```

This procedure will solve any quadratic equation. Just call it with the equation’s coefficients as arguments:

```plaintext
declare RS X1 X2 in
  {QuadraticEquation 1 3 2 RS X1 X2}
  {Browse RS#X1#X2}
```

### Advantages of procedures

Using procedures has many advantages, as we will see throughout the book. The basic advantages are fourfold:

- A procedure is a means to *generalize* a solution to make it applicable to a whole class of similar problems, rather than just to one problem.
- A procedure is a means to *shorten* the program, i.e., a procedure can be defined once and called from several places.
- A procedure is a means to *structure* the program, i.e., procedures can bring together in the source text parts that conceptually belong together.
- A procedure is a means to *abstract* parts of a program. When calling the procedure, one can forget about how the procedure is defined. It is enough to remember what the procedure does, without the details of exactly how it does it.
Defining a procedure

In the declarative model, procedures are first-class: a procedure is a value that can be computed (“defining” the procedure), passed as argument to another procedure, stored in a data structure, and called (“invoked”). Table 4.9 shows the syntax of procedure definition and invocation. A procedure definition is executed at run time; it is not a compile time declaration. We now give an informal semantics for procedure definitions. Executing the definition:

```plaintext
declare MulByN N in
N=3
proc {MulByN X Y}
   Y=N*X
end
```
does two things:

1. It creates a closure for the procedure definition. A closure is a value that contains three parts:
   - The procedure body. In our example, this is \( Y = N \times X \).
   - A list of variable identifiers, the formal parameters, whose partial values are not known when the procedure is defined, but will be given when the procedure is called. Each time the procedure is called, new partial values are given. In our example, the formal parameters are \([X, Y]\), in that order.
   - An environment, the external frame, that maps all the body’s external references to their partial values in the store. An external reference is a free variable identifier that is not a formal parameter. That is, it is an identifier whose partial value is known when the procedure is defined, not when the procedure is called. In our example, the identifier \( N \) is an external reference. The external frame is \([N=3]\).

2. It binds \( \text{MulByN} \) to the closure. If \( \text{MulByN} \) is already bound to another value, then an exception is raised (see Section 4.4.11).

Calling a procedure

The procedure \( \text{MulByN} \) can then be called by giving partial values to the formal parameters. We give an informal semantics of a procedure call. Executing the call:

```plaintext
declare A B in
A=10
{MulByN A B}
{Browse B}
```
does five things:
1. It first checks that \texttt{MulByN} is bound to a closure with two formal arguments. If \texttt{MulByN} is unbound, then the call will suspend until it is bound to a value. If \texttt{MulByN} is bound to anything else than a two-argument closure, then an exception is raised.

2. It extends the current environment with the formal parameters. That is, it adds mappings from the formal parameters to their respective actual parameters. An \textit{actual parameter} is the partial value that is given in the call in the same position as the formal parameter in the definition. In our example, the environment before the call is \([..., A=10, B=\text{Unb1}],\) i.e., at the top it contains identifiers \(A\) and \(B\), and \(A\) maps to 10 and \(B\) maps to an unbound variable \(\text{Unb1}\). It might contain other mappings (\(\ldots\)), but these do not interest us. After adding the formal parameters \(X\) and \(Y\), the environment becomes \([A=10, B=\text{Unb1}, X=10, Y=\text{Unb1}]\).

3. It then extends the environment again with the external frame. That is, it adds all the mappings in the external frame to the environment. In our example the environment becomes \([... , A=10, B=\text{Unb1}, X=10, Y=\text{Unb1}, N=3]\).

4. It then executes the procedure body in the extended environment. In our example, this binds \(\text{Unb1}\) to 30. After execution the environment is therefore \([... , A=10, B=30, X=10, Y=30, N=3]\).

5. When the procedure body terminates, then it restores the original environment by removing all the new identifiers that have been added. As we will see later, the implementation actually restores the original environment somewhat earlier, namely when it enters the last call in the procedure body (see Section 4.4.12 for explanation). In our example the final environment is \([... , A=10, B=30]\).

A procedure has a unique identity, given by its unique closure, and is distinct from all other procedures. That is, procedures obey name equality (see Section 4.4.13). Two different procedure definitions always return different procedures, even if the definitions have exactly the same text. The same procedure definition, if executed twice, will return two different procedures.

### 4.4.6 Procedural abstraction

In Section 4.4.4, we have seen how to use an \texttt{if} statement to compute the maximum of two numbers or atoms. We can make this statement into a procedure by putting it inside a \texttt{proc} declaration. This is called \textit{abstracting} the code into a procedure. Here is what it looks like:

```scheme
local proc {Max X Y Z}
  if X>=Y then Z=X else Z=Y end
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
The proc declaration between local and in does implicit variable initialization just like the equalities: it declares the variable Max and binds it as well. The X, Y, and Z identifier occurrences inside the procedure Max are bound in the procedure. They are not the same as the other X, Y, and Z in the local statement, which are bound in that statement.

In general, any statement can be abstracted into a procedure. Doing so is called procedural abstraction. Any identifiers at all can be chosen as the procedure’s formal parameters. A reasonable choice is to take any subset of the statement’s free variables to be the formal parameters. For example, the free identifiers of the statement:

\[
\text{if } X \geq Y \text{ then } Z = X \text{ else } Z = Y \text{ end}
\]

are X, Y, and Z. Abstracting the statement by taking all these identifiers as formal parameters results in:

\[
\text{proc } \{\text{Max } X \ Y \ Z\} \ {\text{if } X \geq Y \text{ then } Z = X \text{ else } Z = Y \text{ end}}
\]

This procedure calculates the maximum of two numbers. Another way to abstract is by taking some of these identifiers, for example x and z. This gives:

\[
\text{proc } \{\text{LowerBound } X \ Z\} \ {\text{if } X \geq Y \text{ then } Z = X \text{ else } Z = Y \text{ end}}
\]

We have chosen another name for this procedure that is more intuitive. The procedure takes any number x and returns another number that is equal to x if possible, but always at least y. The value of y should be known when LowerBound is executed.

As we will see throughout the book, the concept of procedural abstraction goes a very long way. It underlies all the more powerful techniques for building abstractions and it is important for security.
4.4 Language constructs

4.4.7 Anonymous procedures

Why is a variable bound to a procedure in a different way from it being bound to a record, e.g., \(X = \text{foo}(a:12 \ b:13)\)? The answer is that there is no difference. The procedure definition is just a binding:

\[
P = \text{proc} \{ X_1 \ldots X_n \} \langle \text{statement} \rangle \text{end}
\]

The right-hand side of the equality defines an anonymous procedure value (see Table 4.10). The definition in Section 4.4.5, namely:

\[
\text{proc} \{ P X_1 \ldots X_n \} \langle \text{statement} \rangle \text{end}
\]

is just a syntactic short-cut. The \texttt{Max} example can be written as follows, using variable initialization and anonymous procedures:

\begin{verbatim}
local
Max = \text{proc} \{ X \ Y \ Z \}
    \text{if} X \geq Y \ \text{then} \ Z = X \ \text{else} \ Z = Y \ \text{end}
end
X = 5
Y = 10
Z
in
\{Max X Y Z\} \{Browse Z\}
end
\end{verbatim}

4.4.8 Nesting notation

So far, the language we have seen is flat, that is, there is no syntax for nested procedure calls. This can lead to verbose code. This section provides syntactic sugar for nested procedure calls that often results in significantly smaller and more readable programs. This is another example of syntactic sugar, as defined in Section 4.1.2.

Procedural nesting with nesting markers

Procedural nesting allows to combine several procedure calls into a single statement. As well as being shorter and often easier to read, this avoids having to

\[
\langle \text{expression} \rangle ::= \{ \cdot \langle \text{expression} \rangle \} \langle \text{expression} \rangle \cdot \mid \cdot \$
\]

Table 4.11: Procedural and functional nesting

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
explicitly declare some of the variables shared between the procedure calls. Pro-
cedural nesting gives syntactic support for nesting a procedure call inside another
statement at an expression position. So, in general:

```plaintext
can be written as:

{Q {P ... $ ...} ... }
```

This uses the dollar sign $ as a *nesting marker*. The variable Y is no longer need-
ed. Table 4.11 shows how expression syntax is extended to support procedural
nesting. There are two new kinds of expression: the nested procedure call and the
dollar sign.

The rule to convert nested syntax to flattened syntax is simple. First, a
procedure call nested inside another call is moved before this call. Second, declare
a fresh variable with one occurrence replacing the nested procedure call and the
other occurrence replacing the nesting marker.

**Functional nesting**

Functional nesting allows to write procedure calls as if they were function calls.
Any call `{P X1 ... XN R}` can be considered as a function, where the last argu-
ment R is considered to be the result. Then `{P X1 ... XN}` is a kind of function
call that can be inserted in any expression instead of R. Table 4.11 shows how
this extends expression syntax. So the notation:

```plaintext
{Q {P X1 ... XN} ... }
```

is equivalent to:

```plaintext
local R in
    {P X1 ... XN R}
    {Q R ... }
end
```

Functional nesting is the first step towards having full-fledged functions in the
language. It only goes part way because it is still not possible to *define* functions.
The next section gives a linguistic abstraction that does exactly that.

**Nesting in data structures**

There is one more rule to remember. It has to do with a nested call inside a data
structure (record, tuple, or list). For example:

```plaintext
Ys={F X} | {Map Xr F}
```

In this case the nested call(s) are done after the equality statement:

```plaintext
local Y in
    {P ... Y ... }
    {Q Y ... }
end
```
Table 4.12: Functions

\[
\text{local } Y \text{ Yr } \text{in}
\]
\[
Ys=Y|Yr
\]
\[
\{F \ X \ Y\}
\]
\[
\{\text{Map } Xr \ F \ Yr\}
\]
\[
\text{end}
\]

This example comes from the Map function, which is defined in the next section. The Insert function defined in Figure 4.7 gives another example.

Doing the expansion in this order makes many recursive procedures tail recursive, i.e., the last call in the procedure body is a recursive call to the procedure itself. As we will see in Section 4.4.12, a tail-recursive procedure executes with the same space and time efficiency as an iterative construct like a while loop.

### 4.4.9 Functions

The declarative model provides a linguistic abstraction for programming with functions. This is our first real example of a linguistic abstraction, as defined in Section 4.1.2. We have seen that a procedure call like:

\[\{P \ X1 \ldots \ XN \ R\}\]

can be used in a nested expression as a function call:

\[R=\{P \ X1 \ldots \ XN\}\]

A similar syntactic convenience is possible for function definitions, which are just syntactic short-cuts for procedure definitions. For example, the following function definition is possible:

\[
\text{local YY in}
\]
\[
Ys=Y|Yr
\]
\[
\{F X Y\}
\]
\[
\{\text{Map } Xr \ F \ Yr\}
\]
\[
\text{end}
\]

This example comes from the Map function, which is defined in the next section. The Insert function defined in Figure 4.7 gives another example.

Doing the expansion in this order makes many recursive procedures tail recursive, i.e., the last call in the procedure body is a recursive call to the procedure itself. As we will see in Section 4.4.12, a tail-recursive procedure executes with the same space and time efficiency as an iterative construct like a while loop.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Table 4.13: The andthen and orelse operators

\[
\text{(binaryOp) ::= andthen | orelse | ...}
\]

\[
\text{fun } \{ F \ X_1 \ldots \ X_N \} \ (\text{statement}) \ (\text{expression}) \ \text{end}
\]

where the function body ends in an expression. This corresponds to the following procedure definition:

\[
\text{proc } \{ F \ X_1 \ldots \ X_N \ R \} \ (\text{statement}) \ R=(\text{expression}) \ \text{end}
\]

The expression does not have to be at the end of the function body. For example, if the function body is an if statement, then each alternative of the if can end in an expression:

\[
\text{fun } \{ \text{Max} \ X \ Y \}
\quad \text{if } X>=Y \ \text{then} \ X \ \text{else} \ Y \ \text{end}
\quad \text{end}
\]

Similar rules apply for the case statement and other statements we will see later. For each statement there is a corresponding expression. Roughly speaking, the syntax of functions is similar to that of procedures, except that whenever an execution sequence in a procedure ends in a statement, the corresponding sequence in a function ends in an expression. Table 4.12 gives the function syntax. This table recapitulates all the statements we have seen and shows how to use them as expressions. This table also includes the case statement because it is used in the next chapter.

The operators andthen and orelse

Here is another example, using functions as arguments to functions. Consider the following function:

\[
\text{fun } \{ \text{AndThen} \ BP1 \ BP2 \}
\quad \text{if } \{BP1\} \ \text{then} \ \{BP2\} \ \text{else} \ false \ \text{end}
\quad \text{end}
\]

This takes two boolean functions and returns a boolean. It is a “smart” and operation: it returns true only if both boolean functions return true, but if the first function returns false, then it does not evaluate the second one. We show the intermediate steps in the transformation of this function to the kernel language. First, we make a procedure by introducing a result variable B:

\[
\text{proc } \{ \text{AndThen} \ BP1 \ BP2 \ B \}
\quad B=\text{if } \{BP1\} \ \text{then} \ \{BP2\} \ \text{else} \ false \ \text{end}
\quad \text{end}
\]

This intermediate step is a legal program in its own right. It still uses the if as an expression, though. Second, we move the B into the if expression to make an if statement:
The AndThen operation (and its relative, OrElse) is already provided in the language, by means of the keywords andthen and orelse (see Table 4.13). The former behaves like the function AndThen. The latter behaves like a smart or operation: it evaluates its second argument only if the first argument evaluates to false. To be precise, the expression:

\[(\text{expr1}) \text{ andthen } \text{expr2}\]

has identical semantics to the following:

\{AndThen
    \text{fun } \{\}\ (\text{expr1}) \text{ end}
    \text{fun } \{\}\ (\text{expr2}) \text{ end}\}

The Map function and tail recursion

As a final example, we show the well-known higher-order function Map. It is defined as follows:

\[\text{fun } \{\text{Map Xs F}\}
\text{case Xs}
\text{of nil then nil}
\text{[] X|Xr then } \{\text{F X}\}|\{\text{Map Xr F}\}
\text{end}\]
\text{end}\]

Here is an example call:

\{Browse \{\text{Map [1 2 3 4] fun } \{\}\ X*X end}\}\]

Functional languages are based on a formalism called the lambda calculus, whose basic operations are defining and evaluating functions. The anonymous function

\[\text{fun } \{\}\ X*X end\]

is identical to the lambda expression \(\lambda x. x*x\).

The above definition of Map is tail recursive in the declarative model but not in functional languages without dataflow variables, like Standard ML or Scheme. We can see why this is so by looking at the translation to the kernel language:

\[\text{proc } \{\text{Map Xs F Ys}\}
\text{case Xs}
\text{of nil then nil}
\text{[] X|Xr then Y Yr in}
\text{Ys=Y Yr}
\text{F X Y}
\text{Map Xr F Yr}\]
\text{end}\]
\text{end}\]
The result \( Y_r \) uses the dataflow variable \( Y_r \) as a “placeholder” for the result. It is passed to the recursive call \( \text{Map } X_r \mathcal{F} Y_r \). This follows the nesting rules of Section 4.4.8.

**Comparison with functional languages**

The functional notation is very similar to that of standard functional languages. There are higher-order functions and pattern matching is possible with the `case` statement (see next section). The main difference is the absence of currying (see Section 5.8.6). Furthermore, because of dataflow variables, the declarative model allows certain forms of tail recursion optimization that are not found in most functional languages, including Standard ML, Scheme, and Erlang.

Functions in the declarative model are strict, i.e., function arguments are evaluated before the function body is executed. There is another useful execution order, lazy evaluation, in which function arguments are evaluated only if their result is needed. Lazy evaluation is supported in a more expressive model, the concurrent declarative model, which is presented in Chapter 6.

**When to use functions**

The programmer has the choice of using procedural or functional notation, both for defining and calling an operation. The question is, when should functional notation be used? Here are some rules of thumb that we have found to be useful:

- Be consistent between definitions and calls. If you define a procedure \( P \), do not call it as a function, i.e., do not use functional nesting for procedures. Use instead procedural nesting with a nesting marker. Furthermore, if you define a function, call it as function.

- Use a function definition when the operation is a function in the mathematical sense, i.e., there is one output and possibly many inputs, and the output is a mathematical function of the input arguments.

- Use procedures in most other cases: when there are multiple outputs, when the output is part of an argument, when there are side-effects (e.g., due to using stateful data types), or when the result is nondeterministic. When using stateful data types, an object-oriented style is often appropriate, as explained in Chapter 8.

- Relaxing the previous rule is acceptable. That is, it is reasonable to use functions when there is a clear direction of information flow, even though the definition is not strictly functional. After all, functions are concise.
4.4 Language constructs

Table 4.14: Basic form of the `case` statement

\[
\text{fun} \{\text{Insert Key Value TreeIn}\} \\
\quad \text{if TreeIn=nil then } \text{tree(Key Value nil nil)} \\
\quad \text{else } \text{tree(K1 V1 T1 T2)=TreeIn in} \\
\quad \quad \text{if Key==K1 then } \text{tree(Key Value T1 T2)} \\
\quad \quad \text{elseif Key<K1 then} \\
\quad \quad \quad \text{tree(K1 V1 \{\text{Insert Key Value T1}\} T2)} \\
\quad \quad \text{else} \\
\quad \quad \quad \text{tree(K1 V1 T1 \{\text{Insert Key Value T2}\})} \\
\quad \text{end} \\
\quad \text{end}
\]

Figure 4.7: Tree insertion using the `if` statement

4.4.10 The `case` statement

The declarative model provides a linguistic abstraction, the `case` statement, that uses pattern matching to examine and take apart a data structure. Figure 4.14 shows the basic form of the `case` statement. If `X` has the form of the record `f(l1:X1 ... ln:Xn)`, then the variables `X1`, ..., `Xn` are declared and the `then` part is executed. Otherwise, the `else` part is executed. The `case` statement can be defined by translating to `if` and `local` statements. We show how to translate the basic form; the full form of Figure 4.15 can be translated as a nested series of basic forms. The following statement:

\[
\text{case } X \text{ of } H|T \text{ then (stmt1) else (stmt2) end}
\]

translates to:

\[
\text{if } \{\text{Label X}\}==`|` \text{ andthen } \{\text{Arity X}\}==[1 2] \text{ then } H|T=X \text{ in (stmt1)} \\
\text{else } \text{(stmt2) end}
\]

For records the translation is done in a similar way, see Exercises.

A simple example using the `case` statement is the insertion of elements into a binary tree. For our example, a binary tree is either empty, represented by `nil`, or is a tuple of the form `tree(Key Value TreeL TreeR)`, which represents a tree node where `Key` is the node’s key with corresponding value `Value`, and `TreeL` is the left subtree (whose keys are all less than `Key`), and `TreeR` is the right subtree (whose keys are all greater than `Key`). The function `{Insert Key Value TreeIn}` has three arguments. Given a `Key`, a `Value`, and a `tree`,
fun {Insert Key Value TreeIn}
  case TreeIn
    of nil then tree(Key Value nil nil)
     [] tree(K1 _ T1 T2) andthen Key==K1 then
       tree(Key Value T1 T2)
     [] tree(K1 V1 T1 T2) andthen Key<K1 then
       tree(K1 V1 {Insert Key Value T1} T2)
     [] tree(K1 V1 T1 T2) andthen Key>K1 then
       tree(K1 V1 T1 {Insert Key Value T2})
  end
end

Figure 4.8: Tree insertion using the case statement

TreeIn, Insert returns a new tree that is the input tree augmented with a node containing Key and Value.

One possible definition of Insert is shown in Figure 4.7. This Insert works by handling each case in the obvious way. The first if tests whether the tree is empty or not. If the tree is nonempty there is a second if that compares the input key with the node’s key. The elseif is used to shorten the program. In Figure 4.7, the local statement

local tree(K1 V1 T1 T2)=TreeIn in

... end

does two things: it takes apart a record (namely TreeIn) and it declares four new variables (namely K1, V1, T1, T2). We say that it does a pattern matching on TreeIn to extract the values of the newly-declared variables K1, V1, T1 and T2. Because pattern matching is so often useful, we support it with a linguistic abstraction: a case statement. Figure 4.8 shows the tree-insertion procedure using a case statement.

Syntax and semantics

Table 4.15 gives the syntax of the case statement. This table defines a new nonterminal symbol, ⟨pattern⟩, that defines the terms that are allowed for pattern matching. Intuitively, all expressions are allowed except those that call an operation. All non-escaped variables in a ⟨pattern⟩ are declared implicitly and their scope includes the statement after the then.

The case statement has the following semantics. Assume that the case expression evaluates to v. Then the case statement will match v against all patterns, one by one, starting from the first and trying the next one only if the previous one fails to match. Matching v against “⟨pattern⟩ [ andthen ⟨expression⟩ ]” is done by checking the following four possibilities, in the order given:
4.4 Language constructs

| (statement) ::= case (expression)  
|              of (pattern) [ andthen (expression) ] then (inStatement)  
|              { '[]' (pattern) [ andthen (expression) ] then (inStatement) }  
|              [ else (inStatement) ] end  
| ...  
| (inStatement) ::= [ { (declarationPart) }+ in ] (statement)  
| (pattern) ::= [ `!` ] (variable) | (atom) | (integer) | (float)  
|              | (string) | unit | true | false  
|              | (label) `( ' { [ (feature) `:` ] (pattern) } [ `...` ] `)` `.`  
|              | (pattern) (consBinOp) (pattern)  
|              | `[ ` ` { (pattern) }+ `] `.`  
| (consBinOp) ::= `#` | `|` |

Table 4.15: The case statement

- v matches (pattern) without binding any variable occurring in v. Then (expression) is evaluated, if it exists. If this gives true (or if it does not exist) then the corresponding statement is executed. If it gives false, then v is matched against the next pattern.

- v matches (pattern) but would bind some variables occurring in v. The thread suspends until these variables are bound.

- The matching of v and (pattern) fails. Then v is matched against the next pattern.

- v fails to match for all patterns. Then the else statement is executed, if it exists. If the else part does not exist, then an exception is raised.

Just as with implicit variable initialization (Section 4.3.8), it is possible to use an exclamation point ! to suppress the declaration of a new local variable. For example, in the following code:

```plaintext
case f(X1 X2) of f(!Y Z) then ... else ... end
```

x1 is matched is against the value of the variable Y, where Y is defined outside of the case statement. The case statement and its executing thread will suspend if x1 is insufficiently instantiated to decide the result of the matching.

The case expression can be a record structure. This allows matching on several arguments at once. Figure 4.9 gives an example, the function SMerge, which takes two sorted lists Xs and Ys and merges them into a single sorted list. For example \{SMerge [1 3 5] [2 4 6]\} returns [1 2 3 4 5 6]. By the way, it is an interesting exercise to translate SMerge into the kernel language.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
fun \{SMerge Xs Ys\} 
    case Xs#Ys 
    of nil#Ys then Ys 
        [] Xs#nil then Xs 
        [] (X\(Xr))#(Y\(Yr)) andthen X<\(Xr) then 
            X\{SMerge \(Xr) Ys\} 
        else 
            Y\{SMerge Xs Yr\} 
        end 
    end 

Figure 4.9: Merging sorted lists using the case statement

| (statement) ::= try (inStatement) |
| [ catch (pattern) then (inStatement) |
| { `[[]` (pattern) then (inStatement) } ] |
| [ finally (inStatement) ] end |
| raise (inExpression) end |
| ... |
| (inStatement) ::= [ { (declarationPart) }+ in ] (statement) |
| (inExpression) ::= [ { (declarationPart) }+ in [ (statement) ] ] (expression) |

Table 4.16: Exceptions
4.4.11 Exception handling (the try and raise statements)

The declarative model has an exception handling mechanism that allows to safeguard programs against exceptional and unpredictable situations at run time. The mechanism is extensible, i.e., it allows for user-defined exceptions, which lets the programmer foresee new exceptional situations. Any partial value except for an unbound variable can be an exception. To raise the exception E, execute the following statement:

\[
\text{raise E end}
\]

Here is a simple example, a procedure that evaluates some simple arithmetic expressions and displays the result:

\[
\text{proc \{Eval E}\}
\text{case E}
\text{of plus(X Y) then \{Browse X+Y\}}
\text{[] times(X Y) then \{Browse X*Y\}}
\text{else raise illFormedExpr(E) end}
\text{end}
\text{end}
\]

This example has one user-defined exception, namely the tuple illFormedExpr(E). Raised exceptions can be caught with the try statement, whose syntax is given in Table 4.16. Execution of \text{try \{S\} catch \ldots end} is equivalent to executing \(\{S\}\), if \(\{S\}\) does not raise an exception. If \(\{S\}\) raises exception \(E\) and \(E\) matches one of the patterns, then control is passed to the corresponding statement. Pattern matching is done exactly as with the case statement. If \(E\) does not match any pattern then the exception is propagated outside the try statement. It may be caught by another, enclosing try statement. In any case, if not caught by any other try statement, it is eventually caught by the system, which catches all escaping exceptions. Here is an example using our Eval procedure:

\[
\text{try}
\text{\{ForAll [\{plus(5 10) times(6 11) min(7 10)\}] Eval\}}
\text{catch illFormedExpr(E) then}
\text{\{Browse `*** Illegal expression `\#E\#` ***`\}}
\text{end}
\]

The ForAll operation on lists is explained in Appendix B. A try statement may also specify a final statement which is always executed, whether or not \(\{S\}\) raises an exception. This is useful when dealing with entities such as files which are external to the computation model. Assume that \(F\) is an opened file\(^6\), the procedure ProcessFile manipulates the file in some way, and the procedure CloseFile closes the file. The following program ensures that \(F\) is always closed after Process completes, whether or not it raises an exception:

\[
\text{try}
\text{\{ProcessFile F\}}
\]

\(^6\)We will see later how input/output is handled.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
finally {CloseFile F} end

Note that this try statement does not actually catch the exception; it just executes CloseFile whenever ProcessFile completes. It’s possible to combine both catching the exception and executing a final statement:

try
  {Process F}
catch X then
    {Browse "*** Exception '#X#' when processing file ***"}
  end
  finally {CloseFile F} end

System exceptions

The exceptions raised by the system are records with one of the three labels failure, error, and system:

- failure: indicates an attempt to perform an inconsistent equality operation (e.g., 1=2) in the store (see Section 4.4.13).
- error: indicates a runtime error which should not occur, such as applying a nonprocedure to some (e.g., {foo 1}) argument or adding an integer to an atom (e.g., X=1+a), etc.
- system: indicates a runtime condition occurring in the environment of the Mozart operating system process, e.g., an unforeseeable situation like a closed file or window or a failure to open a connection between two Mozart processes in distributed programming (see Chapter 13).

The exact format of Mozart system exceptions is in an experimental state\(^7\) and therefore the user is advised to rely only on the label, as in the following example:

fun {One} 1 end
fun {Two} 2 end
try {One}={Two}
catch
  failure(...) then {Show caughtFailure}
end

The pattern failure(...) catches any record whose label is failure. When an exception is raised but not handled, then an error message is printed in the Oz emulator window (for interactive execution) or on standard error (for standalone applications). In standalone applications the default behavior is that the whole application terminates. It is possible to change this behavior to something else that is more desirable for particular applications, by using the System module Property.

\(^7\)A large amount of debugging information is in the exception record.
4.4.12 Environment trimming and last call optimization

As we have seen before, a statement is always executed in the context of an environment. The environment contains all the variable identifiers in the statement. Variable identifiers are declared with the `local` statement: they are added to the environment when the statement starts and removed when it completes. This was explained in Section 4.3.1. This is quite conservative behavior; it turns out that many identifiers are needed for only part of the statement. For example:

```
local A B C in
  {P0 X A}
  {P1 A B}
  {P2 B C}
  {P3 C Y}
end
```

The identifier `A` is only needed for `P0` and `P1`. As soon as the call `{P1 A B}` is initiated, `A` can be safely removed from the environment. This makes the environment smaller, thus saving memory. In general, it is possible to remove identifiers from the environment as soon as they are no longer needed for the remainder of the statement. This technique is called *environment trimming*. Section 5.3.3 gives a complete worked-out example of environment trimming and last call optimization; here we explain just the basic concepts.

Environment trimming is needed to avoid memory leaks for systems that do garbage collection. This requires a bit of bookkeeping in the implementation since the garbage collector has to know exactly when each identifier is no longer needed. If environment trimming is not used, then the programmer has to remember explicitly when identifiers are no longer needed. This is a significant mental burden.

Environment trimming is also needed for programs with loops. Without it, the memory space needed by a loop is proportional to the number of iterations that the loop does. This is easy to see: each loop iteration introduces new identifiers. If these identifiers are only removed *after* the loop finishes, as the `local` statement does, then the whole loop needs memory space proportional to the number of iterations. The problem is compounded with nested loops. A system with this limitation would be able to run only toy programs.

A loop in the declarative model is implemented with a recursive procedure call. Here is a simple example:

```
proc {Loop10 I}
  if I==10 then skip
  else
    {Browse I} % Body of the loop
    {Loop10 I+1} % Go to next iteration
  end
end
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Table 4.17: Equality (unification) and equality test (entailment check)

Calling \{Loop10 0\} displays successive integers from 0 up to 9. Each loop iteration introduces a new identifier \(i\). The recursive call \{Loop10 \(i+1\)\} starts the next iteration. Since the new \(i\) is not needed after this call, it can be trimmed away just before the call. This fixes the loop problem. This is called tail recursion optimization, or, in its general form, last call optimization. The optimization is done in all procedures. All new identifiers introduced by the procedure are removed just before the procedure's last call. Any implementation of the declarative model must do at least the last call optimization to be practical.

4.4.13 Binding and testing variables (general case)

In Section 4.3.6 we have seen how to bind and test dataflow variables, using the equality (\(\{=\}\)) and equality test (\(\{==\}\) and \(\{\neq\}\)) operations as shown in Table 4.17. So far, we have seen only the simple cases of these operations. Let us now examine the general cases.

Binding a variable to a value is a special case of an operation called unification. The unification \(\langle\text{Term1}\rangle = \langle\text{Term2}\rangle\) will make the partial values \(\langle\text{Term1}\rangle\) and \(\langle\text{Term2}\rangle\) equal, if possible, by adding zero or more bindings to the store. For example, \(f(X Y) = f(1 2)\) does two bindings: \(X=1\) and \(Y=2\). If the two terms cannot be made equal, then an exception is raised. Unification exists because of partial values; if there would be only complete values then it would have no meaning.

Testing whether a variable is equal to a value is a special case of the entailment check and disentailment check operations. The entailment check \(\langle\text{Term1}\rangle == \langle\text{Term2}\rangle\) (and its opposite, the disentailment check \(\langle\text{Term1}\rangle \neq \langle\text{Term2}\rangle\)) is a two-argument boolean function that suspends until it is known whether \(\langle\text{Term1}\rangle\) and \(\langle\text{Term2}\rangle\) are equal or not equal. Entailment and disentailment check never do any binding.

Structural equality and name equality

In both unification and entailment check, an important role is played by the concept of equality of two values. We make a digression to explain two essentially different kinds of equality, namely structural equality and name equality.

In Figure 4.5, all the types in the subtrees Number and Record except Name obey structural equality. This means that values of these types are equal if and only if they are structurally identical. For example, two numbers are equal if
they have the same value. Two record values are equal if they have the same label, the same feature names, and the subtrees at these features are pairwise equal. Two lists are equal if both their heads and their tails are pairwise equal. Structural equality allows values to be equal even if they are copies occupying different memory locations.

The Name type obeys name equality. Appendix B explains this type. Values of the Name type are equal if and only if they have been created by the same creation operation. I.e., two names are equal only if they are the same name. Throughout the book, we will see other types that obey name equality, for example procedures, functions, functors, threads, cells, and chunks. Name equality is important when creating abstract data types (see Chapter 8).

Unification (the = operation)

So far we have seen only simple examples of the equality statement, such as:

\[ W = \text{tree(I Y LT LR)} \]

where \( W \) is a variable. This is a simple example since we are just binding an unbound variable. But what happens when two unbound variables are equated (e.g., \( X=Y \)), or when two large data structures are equated (e.g., \( a(X \ c(Z) Z) = a(b(Y) Y d(X)) \))? Intuitively, the meaning is simple: just enough bindings are added to the store to make the variables or data structures equal.

Since the store is single-assignment, it might be impossible to make the two data structures equal. For example, the equality \( a(b \ X \ X) = a(Y \ Y \ c) \) will try to bind \( b=Y, X=Y, \) and \( Y=c \). This means that it tries to bind \( b=c \). This situation is called a unification failure. When it happens, an exception is raised. Another way to get a unification failure is by executing the statement \texttt{fail}.

Here is an example of two successful unifications:

```
local X Y Z in
  f(1:X 2:b)=f(a Y)
  f(Z a)=Z
  {Browse [X Y Z]}
end
```

This shows the list \{a b R14=f(R14 a)\} in the browser, if the browser is set up to show the Graph representation. The term \( R14=f(R14 \ a) \) is the textual representation of a cyclic graph. The variable name \( R14 \) is introduced by the browser; different versions of Mozart might introduce different variable names.

As a second example, feed the following unification when the browser is set up for Graph, as before:

```
declare X Y Z in
  a(X \ c(Z) Z)=a(b(Y) Y d(X))
  {Browse X#Y#Z}
```

Now set up the browser for the Minimal Graph representation and display the term again. How do you explain the difference?
Here is an example that shows what happens when variables with incompatible values are equated:

\[\text{declare } X Y Z \text{ in}\]
\[X = f(c \ a)\]
\[Y = f(Z \ b)\]
\[X = Y\]

Doing the equality \(X=Y\) binds \(Z\) to \(c\) but also raises an exception while trying to equate \(a\) and \(b\).\(^8\) The exception is caught by the system. The program can catch the exception by using a \text{try} statement:

\[\text{try } X Y Z \text{ in}\]
\[X = f(c \ a)\]
\[Y = f(Z \ b)\]
\[X = Y\]
\[\text{catch } E \text{ then } \{\text{Browse } E\} \text{ end}\]

This displays the actual exception that is caught.

Let us give a more precise definition of unification. Think of the store as a dynamically expanding array of memory words called nodes. Each node is labeled by a variable. When a new variable \(x\) is introduced then a new node is created in the store, labeled by \(x\), having the value \textit{unknown}. At this point, the node has no value; it is an empty container that may be filled later.

Nodes can contain any arbitrary value. For example, the operation:

\[W = \text{tree}(1:I 2:Y 3:LT 4:LR)\]

stores the record in the node associated with \(w\). The node \(w\) contains a record with four fields. The fields contain arcs pointing to the nodes labeled by \(I\), \(Y\), \(LT\), and \(LR\) respectively. Each arc, in turn, is labeled by the corresponding feature of the record. With each new unification we are incrementally building a directed graph.

Given two variables \(x\) and \(y\), the unification \(x = y\) will try to \textit{merge} their respective nodes, i.e., to make them equal. The “merge” operation never suspends; it either succeeds in its attempt or fails immediately. It is defined as follows:

- If \(x\) and \(y\) label the same node, then the operation is completed without doing anything.

- If \(x\) is unbound then merge the node of \(x\) with the node of \(y\). If \(y\) is unbound, then do the same in the other order. If both are unbound, then pick one arbitrarily. Merging an unbound node \(x\) means replacing all references to \(x\) by a reference to \(y\).\(^9\) Conceptually the original node of \(x\) has been discarded.

\(^8\)To get the exception in Mozart we have to outsmart the compiler, which also detects the incompatibility. Replace \(X=Y\) by the call \{Equal \(X\ Y\)\} to procedure \text{proc} \{Equal \(X\ Y\)\} \(X=Y\ \text{end}\).

\(^9\)This can be done in various ways. One way is to change \(x\) to be a new kind of node called \textit{reference node}, which points to \(y\). Reference nodes are always traversed before performing any unification operation. This is called \textit{dereferencing} the node.
• If \( X \) and \( Y \) label different nodes containing the records \( R_X \) and \( R_Y \) respectively:
  
  - If \( R_X \) and \( R_Y \) have different labels, arities, or both, then complete the operation by raising an exception.
  
  - Otherwise, the fields of \( R_X \) and \( R_Y \) with the same feature are merged pairwise in arbitrary order.

In general the two graphs to be merged could have cycles. This is a perfectly normal situation and does not lead to infinite loops. A correct implementation of the merge operation will remember the node pairs for which an attempt to merge has been made earlier, and consider the operation to be successfully performed without trying to merge them again. A more formal description of the unification operation is found in [39].

**Entailment and disentailment checks (the == and \( \not= \) operations)**

The entailment check \( X == Y \) is a boolean function that tests whether \( X \) and \( Y \) are equal or not. The opposite check, \( X \not= Y \), is called a disentailment check. Both checks use essentially the same algorithm.\(^{10}\) The entailment check returns \textbf{true} if the store implies the information \( X = Y \) (the store “entails” \( X = Y \)) and \textbf{false} if the store will never imply \( X = Y \) (the store “disentails” \( X = Y \)). The check suspends if it cannot determine whether \( X \) and \( Y \) are equal or will never be equal. It is defined as follows:

- It returns the value \textbf{true} if the graphs starting from the nodes of \( X \) and \( Y \) have the same structure, i.e., all pairwise corresponding nodes have identical values or are the same node.

- It returns the value \textbf{false} if the graphs have different structure, or some pairwise corresponding nodes have different values.

- It suspends when it arrives at pairwise corresponding nodes that are different, but at least one of them is unbound.

Here is an example:

```plaintext
declare L1 L2 L3 Head Tail in
L1=Head|Tail
Head=1
Tail=2|nil

L2=[1 2]
{Browse L1==L2}
```

\(^{10}\)Strictly speaking, there is a single algorithm that does both the entailment and disentailment checks simultaneously. It returns \textbf{true} or \textbf{false} depending on which check calls it.
All three lists L1, L2, and L3 are identical. Here is an example where the entailment check cannot decide:

```lisp
declare L1 L2 X in
L1=[1]
L2=[X]
(Browse L1==L2)
```

Feeding this example will not display anything, since the entailment check cannot decide whether L1 and L2 are equal or not. In fact, both are possible: if X is bound to 1 then they are equal and if X is bound to 2 then they are not. Try feeding X=1 or X=2 to see what happens. What about the following example:

```lisp
declare L1 L2 X in
L1=[X]
L2=[X]
(Browse L1==L2)
```

Both lists contain the same unbound variable X. What will happen? Think about it before reading the answer in the footnote. Here is a final example:

```lisp
declare L1 L2 X in
L1=[1 a]
L2=[X b]
(Browse L1==L2)
```

This will display false. While the comparison 1==X suspends, further inspection of the two graphs shows that there is a definite difference, so the full check returns false.

## 4.5 Exercises

1. Section 4.4.5 explains how a procedure call is executed through the example call (MulByN A B). The environment before this call is [..., A=10, B=Unb1]. Step 3 of the execution says that the external frame, i.e., the mapping N=3, is added to the environment. Why is this a necessary step? In particular, would not N=3 already exist somewhere in the invisible part of the environment (“...”), and would not this be enough to ensure that the identifier N already maps to 3? Give an example where N does not exist in the invisible part. Then give a second example where N does exist there, but is bound to a different value than 3.

---

11The browser will display true, since L1 and L2 are equal no matter what X might be bound to.
2. If a function body has an \texttt{if} statement with a missing \texttt{else} case, then an exception is raised if the \texttt{if} condition is false. Explain why this behavior is correct. This situation does not occur for procedures. Explain why not.

3. Section 4.4.10 defines the \texttt{case} statement as a new linguistic abstraction. Translate the statement of Figure 4.14 into the declarative kernel language, using the record operations of Appendix B when necessary.

4. Section 4.4.10 defines the function \texttt{SMerge} that merges two sorted lists. Expand \texttt{SMerge} into procedural notation. The resulting procedure should be tail recursive, if the rules of Section 4.4.8 are followed correctly.
Chapter 5

Declarative Programming Techniques

“S’il vous plaît... dessine-moi un arbre!”
“*If you please – draw me a tree*”
– freely adapted from Le Petit Prince, Antoine de Saint-Exupéry (1900–1944)

“The nice thing about declarative programming is that you can write a specification and run it as a program. The nasty thing about declarative programming is that some clear specifications make incredibly bad programs. The hope of declarative programming is that you can move from a specification to a reasonable program without leaving the language.”
– The Craft of Prolog, Richard O’Keefe (?–)

Consider any computational “operation” in a general sense, i.e., a program fragment that has input arguments and results. We say the operation is declarative if, whenever called with the same arguments, it returns the same results independent of any other computation state. Figure 5.3 illustrates the concept. A declarative operation is independent (does not depend on any execution state outside of itself), stateless\(^1\) (has no internal execution state that is remembered between calls), and deterministic (always gives the same results when given the same arguments). Most of the basic operations on data types are declarative in this sense, e.g., arithmetic, list, and record operations. It is possible to combine declarative operations to make new declarative operations, if certain rules are followed. For example, combining declarative operations according to the operations of the declarative model will result in a declarative operation. This is explained in Section 5.1.2. It is also possible to build declarative operations out of non-declarative components. This is explained in later chapters.

The standard rule in algebra that equals can be replaced by equals is an example of a declarative combination. For example, given the function \( f(a) = a^2 \),

\(^1\)The concept of “stateless” is sometimes called “immutable”.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
then we can replace \( f(a) \) by \( a^2 \) in any other equation. The equation \( b = 7f(a)^2 \) then becomes \( b = 7a^4 \). This is possible because \( f(a) \) is declarative: it depends only on its arguments and not on any other computation state.

Declarative programming is important because it decreases the complexity of writing programs. The program can be partitioned into components (defined in Section 5.2). Each component can be written, tested, and proved correct completely independent of the context in which it is used. The interaction between components is well-defined; it is determined solely by their arguments and results. Because of this independence, the complexity of proving correct a program with a declarative component is the sum of the complexities of proving correct the declarative component and proving correct the rest. If there would be a more intimate interaction, the complexity of both would be much bigger, e.g., the product of the complexities of each. For a program with many components that interact intimately, this very quickly explodes, making proofs completely impractical. An example of such an intimate interaction is a concurrent program with shared state, as explained in Chapter 10.

This chapter explains how to write practical declarative programs. Figure 5.1 shows the four parts into which the chapter is organized. The first part defines “declarativeness” and situates declarative programming with respect to other computation models. The second part gives an overview of programming tech-
5.1 What is declarativeness?

We have defined declarativeness in one particular way, so that reasoning about programs is simplified. But this is not the only way to make precise the intuitive

Figure 5.2: A classification of declarative programming

niques. The third part explains abstraction, both procedural and data, and shows its power for encapsulating and making generic. The fourth part shows how declarative programming interacts with the rest of the computing environment.

The declarative model is one of the simplest practical computation models that exist. It is close to strict functional programming, although the existence of partial values makes it more expressive. Functional programming is a style of programming where the program is a set of function definitions. Each function takes as arguments the results of evaluating other functions. In a strict function, all arguments are evaluated before the function itself is evaluated.

Section 5.2 introduces some more expressive models, gives a road map on how they are presented in the book, and gives a rule of thumb when they should be used. It turns out that, despite its small size, the declarative model is surprisingly expressive. Many important concepts and algorithms in computing can be expressed there.

The basic technique for writing declarative programs is to formulate the program as a set of recursive functions, using higher-orderness to simplify the program structure. A recursive function is one whose definition body refers to the function itself, either directly or indirectly. Direct recursion means that the function itself is used in the body. Indirect recursion means that the function refers to another function that directly or indirectly refers to the original function. Higher-orderness means that functions can have other functions as arguments. This ability underlies all the techniques for building abstractions that we will show in the book. Higher-orderness can compensate somewhat for the lack of expressiveness of the declarative model, i.e., it makes it easy to code limited forms of concurrency and state in the declarative model.

5.1 What is declarativeness?

We have defined declarativeness in one particular way, so that reasoning about programs is simplified. But this is not the only way to make precise the intuitive

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Table 5.1: A descriptive declarative kernel language

| (statement) ::= |
| skip | Empty statement |
| (statement) (statement) | Statement sequence |
| local X in (statement) end | Variable declaration |
| X=Y | Bind to variable |
| X=f(l1:X1 ... ln:Xn) | Bind to record |

notion of declarative programming. There are many different ways. Figure 5.2 summarizes the classification used in the book. The first distinction is based on the expressiveness. There are two levels:

- The descriptive level. This is the weakest level. The definition just gives a summary of “what” without explaining the “how”. Table 5.1 defines a language at this level. This language can only define records! It contains just the first five statements of the kernel language in Table 4.1. Another example is a formatting language like HTML, which gives the structure of the document without telling how to do the formatting. Chapter 12 gives a third example that is useful for defining graphic user interfaces. The descriptive level is too weak to write general programs. So why is it interesting? Because it consists of data structures that are easy to manipulate formally. The records of Table 5.1, HTML documents, and the declarative user interfaces of Chapter 12 can all be created and transformed easily.

- The programmable level. Arbitrary programs can be written in this level. For example, Table 4.1 defines a language at this level. See the introduction to Chapter 8 for more on the relationship between the descriptive and programmable levels.

Within the programmable level, there are two fundamentally different approaches. Declarativeness can be a definitional concept, i.e., one that depends on how the component is defined. Or it can be an observational concept, i.e., one that depends on the observed behavior of a component.

In the definitional approach, two styles have become particularly popular: the functional and the logical. In the functional style, we say that a component defined as a mathematical function is declarative. Functional languages such as Haskell and ML follow this approach. In the logical style, we say that a component defined as a logical relation is declarative. Logic languages such as Prolog and Mercury follow this approach. It is harder to formally manipulate functional or logical programs than descriptive programs, but they still follow simple algebraic laws. The declarative model of this chapter is an instance of both the functional and logic styles.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
5.1 What is declarativeness?

The definitional approach rejects components that not defined in the right way. Often this is too strong a limitation. We would like to be able to use declarative components in a declarative program even if they are written in a nondeclarative model. For example, a database interface can be a valuable addition to a declarative language. Yet, the implementation of this interface is almost certainly not going to be logical or functional. The observational approach lets us consider this package as declarative, if it behaves declaratively. That is, it behaves as if it could have been defined declaratively.

The observational approach respects the principle of abstraction: that it is enough to know the specification of a component to use it, without knowing its implementation. This book uses both the definitional and observational approaches. When we are interested in manipulating the structure of the component, we will use the definitional approach. When we are interested in the component in terms of how it is used, we will use the observational approach. Sometimes a declarative component will be written in a functional or logical style, and sometimes it will not be. We will not be dogmatic about the matter; we will consider the component to be declarative if it behaves declaratively.

In this chapter we will use declarative techniques to build declarative components. Later chapters introduce nondeclarative techniques. There we will show how to build both declarative and nondeclarative components using these techniques.

5.1.1 Specification languages

Proponents of declarative programming sometimes claim that it allows to dispense with the implementation, since the specification is all there is. That is, the specification is the program. This is true in a formal sense, but not in a practical sense. Practically, declarative programs are very much like other programs: they require algorithms, data structures, structuring, and reasoning about the order of operations. This is because declarative languages are limited in what kinds of mathematics they can use: they are limited to what can be implemented efficiently. Declarative programs can be long, whereas specifications should ideally be short. So the distinction between specification and implementation still makes sense, even for declarative programs.

There is a trade-off between expressiveness and efficiency of declarative languages. It is possible to define a declarative language that is much more expressive than the ones we see in this book: its “programs” are much shorter. Such a language is called a specification language. It is usually impossible to implement specification languages efficiently. This does not mean that they are impractical; on the contrary. They are an important tool for thinking about programs. They can be used together with a theorem prover, i.e., a program that can do certain kinds of mathematical reasoning. Practical theorem provers are not completely automatic; they need human help. But they can take over much of the drudgery of reasoning about programs, i.e., the tedious manipulation of mathematical for-
mulas. With the aid of the theorem prover, a developer can often prove very strong properties about his or her program. Using a theorem prover in this way is called proof engineering. Up to now, proof engineering is only practical for small programs. But this is enough for it to be used successfully when safety is of critical importance, e.g., when lives are at stake, such as in medical apparatus or public transportation.

Specification languages are outside the scope of this book.

5.1.2 Implementing components in the declarative model

We explain why combining declarative operations according to the operations of the declarative model results in a declarative operation. An operation is declarative if it always gives the same results for the same input arguments and if it does not depend on anything other than the input arguments (there is no “context”). We will reason informally as follows:

- All basic operations in the declarative model are declarative. This includes all operations on basic types, which are explained in Chapter 4.

- Combining declarative operations with the constructs of the declarative model gives a declarative operation. The following five compound statements exist in the declarative model:

  - The statement sequence.
  - The local statement.
  - The if statement.
  - The try statement.
  - The proc statement.

All these ways of combining statements are deterministic (if their component statements are deterministic, then so are they) and they do not depend on any context.

5.1.3 The richness of declarative programming

Despite the simplicity of its computation model, declarative programming is a surprisingly rich area. The declarative model of this chapter is a sufficient base for practical programming, even though it lacks important concepts such as state and concurrency. This chapter shows the basic techniques of programming in this model. The next two chapters show how to extend the model by adding search and concurrency, while staying declarative. We summarize the declarative techniques as follows:

- Functional and logic programming. When all data are values, reasoning about programs is simple. As we will show, both logical and functional reasoning can be done.
• **Recursive programming.** This includes using recursion to simulate execution with state, with difference lists and accumulating parameters. Difference lists are an important case of “almost” having state in a stateless model. Recursion is important for divide-and-conquer programming.

• **Higher-order programming.** This uses procedural abstraction to modularize programs in a natural way. We extract useful patterns from typical recursive programs: map, forall, fold, etc. We show how to build generic reusable components.

• **Abstract data types.** We show how to design abstract data types with encapsulation. There are two approaches to encapsulation: procedures and names. We show the advantages and disadvantages of procedures versus names for encapsulation. Names allow us to do encapsulation while separating operations from data. This is a precursor to secure programming and object-oriented programming.

• **Programming for efficiency.** How much time and memory do programs use? We explain the last call optimization and why efficient declarative programs need it. We estimate the performance of programs. We explain memory management and its problems, and show how to avoid memory bugs, i.e., dangling pointers and memory leaks.

• **Declarative concurrency** (in Chapter 6). It is a little-known fact that much of the power of concurrent programming can be obtained in a declarative model.

• **Relational programming** (in Chapter 7). We add one statement, choice, that chooses among a list of alternatives. This allows to program with nondeterminism and search. This is a conceptually very powerful tool that can be practical, if correctly used.

### 5.2 Computation models and program structure

A *component* is defined as a precisely delimited program fragment with well-defined *inputs* and *outputs*. A component itself can be defined in terms a set of simpler components. For example, in the declarative model a procedure is a component. An application program is the topmost component in a hierarchy of components. The hierarchy bottoms out in primitive components which are provided by the system.

Any component, at any level in the hierarchy, is written in a particular computation model. There exist many computation models that differ in how expressive they are and how hard it is to reason on programs written in them. Programming in the declarative model, as explained in this chapter, is among the simplest of
all. However, not all programs can be written in the declarative model. There are two very different issues:

- The declarative model always gives the same outputs when given the same inputs. Nondeterministic behavior, as occurs in concurrent programs, cannot be expressed in the declarative model.

- The declarative model always calculates new values from old values, because values cannot be changed. This means that some algorithms cannot be expressed conveniently or efficiently. In particular, this includes algorithms that depend on destructive modification (state).

More expressive computation models are needed to alleviate these limitations. The major computation models presented in the book are the following:

- **Declarative** model (see Chapter 4 and this chapter; defined in Section 4.2). This model embraces strict functional programming and deterministic logic programming. It extends the former with partial values (using “logic variables”, called “dataflow variables” in this book) and the latter with higher-order procedures. Reasoning with this model is based on algebraic calculations with values. Equals can be substituted for equals and algebraic identities can be applied. A component’s behavior is independent of when it is executed or of what happens in the rest of the computation. Because this still holds, components that raise exceptions are also declarative.

- **Concurrent declarative** model (see Chapter 6; defined in Section 6.1). This is the declarative model extended with explicit threads and by-need computation. This model embraces lazy functional programming and deterministic concurrent logic programming. A component can interact with the
rest of the computation by sharing dataflow variables. Thread execution is interleaved. Reasoning with this model requires reasoning on interleaving computations. If it does not raise any exceptions, a component is still declarative. If it raises exceptions, then it is no longer declarative—with exceptions it becomes similar to the concurrent stateful model (see below).

- **Relational** model (see Chapter 7; defined in Section 7.1). This is the declarative model extended with explicit nondeterministic choice (“don’t know” nondeterminism). The choice is repeated until it gives a satisfactory result, which leads to programs that do “search”. This model allows to program with relations. It embraces nondeterministic logic programming in the Prolog style.

- **Stateful** model (see Chapters 8–9; defined in Section 8.3). This is the declarative model extended with explicit state. This model subsumes sequential object-oriented programming. State is a sequence of values that is extended as the computation proceeds. Having explicit state means that a component does not always give the same result when called with the same arguments. The component can “remember” information from one call to the next. This allows the component to have a “history”, which lets it interact more meaningfully with its environment by adapting and learning from its past. Reasoning with this model requires reasoning on the history.

- **Concurrent stateful** or **Stateful concurrent** model (see Chapter 10; defined in Section 10.1). This is the declarative model extended with both explicit state and threads. This model subsumes concurrent object-oriented programming. The concurrency is more expressive than in Chapter 6 since it can use explicit state to wait simultaneously on one of several events occurring (this is called *nondeterministic choice*). Unlike in Chapter 7, the choice is implicit and irrevocable and cannot be used to search (“don’t care” nondeterminism). This model corresponds most closely to the real world, which is both stateful and parallel.\(^2\) Reasoning with this model is the most complex since there can be multiple histories interacting in unpredictable ways.

### 5.2.1 Picking the right model

For a given program, some models are better than others. For example, concurrency is often needed when interacting with the external world. It should be used, instead of trying to get by with just the declarative model. But the more

\(^2\)We distinguish between *concurrency*, a language concept that allows to express logically independent computations, and *parallelism*, an implementation concept that expresses computations that happen at the same time due to special hardware. Concurrency is related to language expressiveness; parallelism is related to implementation performance. Concurrency and parallelism are independent properties.
expressive models are not “better” than the others, since they do not always give simpler programs and reasoning in them is usually harder. In our experience, all models have their place and can be used together to good effect in the same program. For example, in a program with concurrent state, many components can be declarative. Conversely, in a declarative program, some components (e.g., graph algorithms) need state to be implemented well. We summarize this experience in the following rule:

**Rule of least expressiveness**

When programming a component, the right computation model for the component is the least expressive model that results in a simple program.

The idea is that each component should be programmed in its “natural” model. Using a less expressive model would give a more complex program and using a more expressive model would not give a simpler program but would make reasoning about it harder.

The problem with this rule is that we have not defined “simple”. This is because to some degree, simplicity is a subjective property. Different people may find different models easier to use, because of their differing knowledge and background. The issue is not the precise definition of “simple”, but the fact that such a definition exists for each person, even though it might be different for different people.

Here is an example to make this clear. Assume that we are using the declarative model to implement a large declarative component and that we would like to use state. We can encode the state by adding two arguments to each procedure. Then the component looks something like this:

```plaintext
proc {Component ...}
    proc {P ... S1 S0}
        {P1 ... S1 S2}
        {P2 ... S2 S3}
        {P3 ... S3 S4}
        {UseState S4 S5}
        ...
        {Pn ... Sn S0}
    end
end

% Other procedure definitions come here

in
    {P ... InitialState LastState}
end
```

Each procedure defined inside the component has two extra arguments. The first inputs the current state and the second outputs the new state. These arguments

---

3Another reason why they are not better has to do with distributed programming and network-awareness, which is explained in Chapter 13.
are threaded between the procedure calls. This technique requires a bit of bookkeeping, but it works quite well. We can make things simpler with a preprocessor that adds the extra arguments so that we do not have to write them all down.

We can also write the component in the *stateful* model. This model is defined in Chapter 8; for now assume that we have a new language entity, called “cell”, that we can assign and access (with the := and @ operators), similar to a variable in imperative programming languages. Then the component looks something like this:

```plaintext
proc {Component ...}
   C={NewCell InitialState}

proc {P ...}
   {P1 ...}
   {P2 ...}
   {P3 ...}
   C:={UseState @C}
   ...
   {Pn ...}
end
%

in
   {P ...}
end
```

There are two main differences between these two implementations:

- The first implementation is not modular, because *every* procedure definition and call needs two extra arguments. A component is *modular* if changing one of its subcomponents does not require changing other subcomponents. Modularity is discussed further in Section 8.9.2. The second implementation is modular because the cell is mentioned only where it is needed, not everywhere.

- The first implementation is slower because it does much extra argument passing.

Which implementation is simpler: the first or the second? The first has a simpler model but a more complex program. The second has a slightly more complex model but a simpler program. In our view, the second implementation is clearly the simplest overall.

---

4To be precise, the two extra arguments are needed if the procedure uses state directly (in its body) or indirectly (in the body of another procedure, which is directly or indirectly called by this one). In practice, this means that essentially all procedures need the extra arguments.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
5.2.2 Using different models together

There are many ways to use different models together in the same program. Typically, any well-structured program of reasonable size has different parts written in different models. This section gives an example of a particularly useful structuring technique, sometimes called *impedance matching*, that naturally leads to using different models together in the same program.

Impedance matching is one of the most powerful and practical ways to implement the general principle of separation of concerns. Consider two computation models A and B, such that model A is more expressive than B, but harder to reason in. For example, model A can be stateful and model B declarative. With impedance matching, we can write a program in model B that can “live” in the computational environment of model A.

Impedance matching works by building an abstraction in model B that is parameterized with a program in model A. The heart of impedance matching is finding and implementing the right abstraction. This hard work only needs to be done once; afterwards there is only the easy work of using the abstraction. Perhaps surprisingly, it turns out that it is almost always possible to find and implement an appropriate abstraction. Here are some typical cases of impedance matching:

- A concurrent model can be reduced to a sequential model. For example, the abstraction can be a *serializer* that accepts concurrent requests and passes them sequentially.

- A stateful model can be reduced to a declarative model. For example, the abstraction can be a *storage manager* that passes its content to the declarative program and stores the result as its new content.

- A distributed model can be reduced to a centralized model. For example, the abstraction can be a *collector* that accepts requests from any site and pass them to a single site.

- An insecure model can be reduced to a secure model. A insecure model is one that assumes the existence of malicious entities that can disturb programs in well-defined ways. A secure model assumes that no such entities exist. The abstraction can be a *protector* that insulates a computation by verifying all requests from other computations. The abstraction handles all the details of coping with the presence of malicious adversaries. The protector is a kind of firewall.

- An open model can be reduced to a closed model. An open model is one that lets independent computations find each other and interact. The abstraction can be a *connector* that accepts requests from one computation to connect to another, using an open addressing scheme. The ClientServer abstraction of Chapter 13 is both a serializer, a collector, and a connector.
5.3 Iterative computation

We will now look at how to program in the declarative model. We start by looking at a very simple kind of program, the iterative computation. An iterative computation is one whose environment size is bounded by a constant, independent of recursion depth.

- A model with partial failure can be reduced to a reliable model. For example, the abstraction can be a replicator that implements active replication between several sites and manages recovery when one of the sites fails. This is what the RedundantServer abstraction of Chapter 13 does.

These cases are orthogonal. As the examples show, it is often a good idea to implement several cases in one abstraction. This book has abstractions that illustrate all these cases and more. Usually, the abstraction puts a minor condition on the program written in model A. For example, the RedundantServer assumes that the server is deterministic.

Impedance matching is extensively used in the Erlang project at Ericsson [4]. They have developed a whole library of abstractions for it. A typical Erlang abstraction takes a declarative program (written in a functional language) and makes it stateful, concurrent, and fault tolerant.

Figure 5.4: Finding roots using Newton’s method (first version)

```
fun {Sqrt X}
  Guess=1.0
  in
    {SqrtIter Guess X}
  end
fun {SqrtIter Guess X}
  if {GoodEnough Guess X} then Guess
  else
    {SqrtIter {Improve Guess X} X}
  end
end
fun {Improve Guess X}
  (Guess + X/Guess) / 2.0
end
fun {GoodEnough Guess X}
  {Abs X-Guess*Guess}/X < 0.00001
end
fun {Abs X} if X<0.0 then ~X else X end end
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
5.3.1 Iteration with numbers

A good example of iterative computation is Newton’s method for calculating the square root of a positive real number $x$. The idea is to start with a guess $g$ of the square root, and to improve this guess iteratively until it is accurate enough. The improved guess $g'$ is the average of $g$ and $x/g$:

$$g' = \frac{(g + x/g)}{2}.$$

To see that the improved guess is better, let us study the difference between the guess and $\sqrt{x}$:

$$\epsilon = g - \sqrt{x}$$

Then the difference between $g'$ and $\sqrt{x}$ is:

$$\epsilon' = g' - \sqrt{x} = \frac{(g + x/g)}{2} - \sqrt{x} = \frac{\epsilon^2}{2g}$$

For convergence, $\epsilon'$ should be smaller than $\epsilon$. (We can assume that $\epsilon > 0$, since if it is not, we start with $\epsilon'$, which is always greater than 0.) This implies $\epsilon^2/2g < \epsilon$, which implies $\epsilon < 2g$. Substituting the definition of $\epsilon$, we get the final condition $\sqrt{x} + g > 0$. If $x > 0$ and the initial guess $g > 0$, then this is always true. The algorithm therefore always converges.

Figure 5.4 shows one way of defining Newton’s method as an iterative computation. The function $\{\text{SqrtIter Guess X}\}$ calls $\{\text{SqrtIter \{Improve Guess X\} X}\}$ until $\text{Guess}$ satisfies the condition $\{\text{GoodEnough Guess X}\}$. The improved guess is calculated according to the formula given above. The “good enough” check is $|x - g^2|/x < 0.00001$, i.e., the square root has to be accurate to five decimal places. This check is relative, i.e., the error is divided by $x$. Why does the check have to be relative?

5.3.2 Using local procedures

In the Newton’s method program of Figure 5.4, several “helper” procedures are defined: $\text{SqrtIter}$, $\text{Improve}$, $\text{GoodEnough}$, and $\text{Abs}$. These procedures are used as building blocks for the main procedure $\text{Sqrt}$. In this section, we will discuss where to define helper procedures. The basic principle is that a helper procedure defined only as an aid to define another procedure should not be visible elsewhere. (Note that we use the word “procedure” for both functions and procedures.) In the Newton example, $\text{SqrtIter}$ is only needed inside $\text{Sqrt}$, $\text{Improve}$ and $\text{GoodEnough}$ are only needed inside $\text{SqrtIter}$, and $\text{Abs}$ is a utility function that could be used elsewhere. There are two basic ways to express this visibility, with somewhat different semantics. The first way is shown in Figure 5.5: the helper procedures are defined outside of $\text{Sqrt}$ in a local statement. The second way is shown in Figure 5.6: each helper procedure is defined inside of the procedure that needs it.
local

fun {Improve Guess X}
  (Guess + X/Guess) / 2.0
end

fun {GoodEnough Guess X}
  (Abs X-Guess*Guess)/X < 0.00001
end

fun {SqrtIter Guess X}
  if {GoodEnough Guess X} then Guess
  else
    {SqrtIter {Improve Guess X} X}
  end
end in

fun {Sqrt X}
  Guess=1.0
in
  {SqrtIter Guess X}
end end

fun {Abs X} if X<0.0 then ~X else X end end

Figure 5.5: Finding roots using Newton's method (second version)
fun {Sqrt X}
  fun {SqrtIter Guess X}
    fun {Improve Guess X}
      (Guess + X/Guess) / 2.0
    end
    fun {GoodEnough Guess X}
      {Abs X-Guess*Guess}/X < 0.00001
    end
  end
  in
    if {GoodEnough Guess X} then Guess
    else
      {SqrtIter {Improve Guess X} X}
    end
  end
  Guess=1.0
in
  {SqrtIter Guess X}
end
fun {Abs X} if X<0.0 then ~X else X end end

Figure 5.6: Finding roots using Newton’s method (third version)

In Figure 5.5, there is a trade-off between readability and visibility: Improve and GoodEnough could be defined local to SqrtIter only. This would result in two levels of local declarations, which is harder to read. We have decided to put all three helper procedures in the same local declaration.

In Figure 5.6, each helper procedure sees the arguments of its enclosing procedure. These arguments do not change during a call to Sqrt. This means we could simplify the definition by removing these arguments from the helper procedures. This gives Figure 5.7.

There is a trade-off between putting the helper definitions outside the procedure that needs them or putting them inside:

- Putting them inside (Figures 5.6 and 5.7) lets them see the arguments of the main procedures, according to the lexical scoping rule (see Section 4.4.2). But each time the main procedure is invoked, new helper procedures are created. This means that new closures are created.

- Putting them outside (Figures 5.4 and 5.5) means that the closures are created once and for all, for all calls to the main procedure. But then the helper procedures need arguments so that the main procedure can pass information to them.

In Figure 5.7, new definitions of Improve and GoodEnough are created on each iteration of SqrtIter, whereas SqrtIter itself is only created once. This suggests a good trade-off, where SqrtIter is local to Sqrt and both Improve and
fun {Sqrt X}
  fun {SqrtIter Guess}
    fun {Improve}
      (Guess + X/Guess) / 2.0
    end
    fun {GoodEnough}
      {Abs X-Guess*Guess}/X < 0.00001
    end
    in
      if {GoodEnough} then Guess
      else
        {SqrtIter {Improve}}
      end
    end
  end
in
  Guess=1.0
in
  {SqrtIter Guess}
end
fun {Abs X} if X<0.0 then X else X end end

Figure 5.7: Finding roots using Newton’s method (fourth version)

GoodEnough are outside SqrtIter. This gives the final definition of Figure 5.8, which we consider the best in terms of both efficiency and visibility.

5.3.3 General pattern of iteration

In the Newton example, we gave the recursive function SqrtIter and claimed that it defines an iterative computation, i.e., one that has a bounded environment size. This section substantiates this claim: we present a general form of iterative computation, of which the Newton example is a special case, and we give a step-by-step execution to show what happens with the environment. This will give us insight into exactly what it means for a computation to be iterative.

We represent a general iteration by means of the function Iter:

fun {Iter State Transform IsDone}
  if {IsDone State} then State
  else Statel in
    Statel={Transform State}
    {Iter Statel Transform IsDone}
  end
end

The calculation performed by Iter depends on IsDone and Transform. To use it in a particular case, the variables IsDone and Transform have to be bound to one-argument functions. For example, we can make Iter behave exactly like the
fun {Sqrt X}
  fun {Improve Guess}
    (Guess + X/Guess) / 2.0
  end
  fun {GoodEnough Guess}
    {Abs X-Guess*Guess}/X < 0.00001
  end
  fun {SqrtIter Guess}
  in
    if {GoodEnough Guess} then Guess
    else
      {SqrtIter {Improve Guess}}
    end
  end
  Guess=1.0
in
  {SqrtIter Guess}
end
fun {Abs X} if X<0.0 then ~X else x end end

Figure 5.8: Finding roots using Newton’s method (fifth version)

Newton example’s SqrtIter by binding IsDone to GoodEnough and Transform to Improve.

As an aside, note that the arguments Transform and IsDone are themselves procedures. Passing procedures as arguments to procedures is known as higher-order programming. This is a powerful technique that is further explained in Section 5.8.

Let us follow a typical execution of Iter step by step, to see what happens to the environment. At each step of the execution, there is a statement to execute and its current environment. Executing one step of the statement results in a new statement and a new environment. The execution looks like this:

\[
(S,E) \rightarrow (S',E') \rightarrow (S'',E'') \rightarrow ... \]

where \((S)\) is a statement and \(E\) is an environment. An environment \(E\) is an ordered list of variable identifiers each with its entity in the store: \(E=[\text{Var1=Ref1, Var2=Ref2, ...}]\).

Initial statement

Now assume the initial statement is:

local
R
S0=s0
Assume the environment contains just the `Browse` and `Iter` identifiers before executing this statement, i.e., \( E = [ \text{Browse} = \text{proc} \{ \$ \ldots \} \ldots \text{end}, \text{Iter} = \text{fun} \{ \$ \ldots \} \ldots \text{end} ] \). To avoid clutter in subsequent execution steps, we will abbreviate the `Browse` and `Iter` entries by “...”, giving \( E = [...]. \)

**Next execution step: set up arguments to Iter**

Now execute the `local` statement and the variable initializations up to just before the call to `Iter`. This defines the initial state \( S_0 \) and the functions \( T \) and \( D \). The new statement to be executed is:

\[
\begin{align*}
\text{if} \{ \text{IsDone State} \} & \text{ then State} \\
\text{else State1} \text{ in} \\
\text{State1} & = \{ \text{Transform State} \} \\
\{ \text{Iter State1 Transform IsDone} \} \\
\text{end} \\
\{ \text{Browse R} \}
\end{align*}
\]

The new environment just before executing this statement is

\[
E = [..., R, S_0 = s_0, T = \text{fun} \{ \$ \ldots \} \ldots \text{end}, D = \text{fun} \{ \$ \ldots \} \ldots \text{end}] 
\]

Abbreviate the definition of \( T \) as \( tf \) and the definition of \( D \) as \( df \). Then we have:

\[
E = [..., R, S_0 = s_0, T = tf, D = df] 
\]

**Next execution step: first call to Iter**

Now call `Iter` for the first time. The new statement contains the body of `Iter`:

\[
\begin{align*}
R & = \text{if} \{ \text{IsDone State} \} \text{ then State} \\
\text{else State1} \text{ in} \\
\text{State1} & = \{ \text{Transform State} \} \\
\{ \text{Iter State1 Transform IsDone} \} \\
\text{end} \\
\{ \text{Browse R} \}
\end{align*}
\]

We say that we have entered the function `Iter`. The environment is extended with the formal parameters:

\[
E = [..., R, S_0 = s_0, T = tf, D = df, \text{State} = s_0, \text{Transform} = tf, \text{IsDone} = df] 
\]

Because \( S_0, T, \) and \( D \) no longer occur in the statement, they can be safely removed from the environment (see Section 4.4.12). This gives the following “trimmed” environment:

\[
E = [..., R, \text{State} = s_0, \text{Transform} = tf, \text{IsDone} = df] 
\]
Next execution step: execute condition \texttt{IsDone}

Now execute \texttt{(IsDone State)}. Assume this returns \texttt{false} and that the \texttt{else} clause is chosen. This gives the new statement:

\begin{verbatim}
R=local State1 in
  State1={Transform State}
  {Iter State1 Transform IsDone}
end
{Browse R}
\end{verbatim}

The environment is unchanged:

\[ E=\{..., R, State=s0, Transform=tf, IsDone=df\} \]

Next execution step: execute transformation \texttt{Transform}

Now the \texttt{local} statement declares \texttt{State1} and the \texttt{(Transform State)} calculates its value \( s1 \). The new statement is:

\begin{verbatim}
R={Iter State1 Transform IsDone}
{Browse R}
\end{verbatim}

The environment has been extended with \texttt{State1}, which is bound to \( s1 \):

\[ E=\{..., R, State1=s1, Transform=tf, IsDone=df\} \]

Because \texttt{State} no longer occurs in the statement it can safely be removed from the environment. This gives the following trimmed environment:

\[ E=\{..., R, State1=s1, Transform=tf, IsDone=df\} \]

For clarity, we show \texttt{State1} in the place formerly occupied by \texttt{State}.

Summary

Let us examine the environments just before the calls to \texttt{Iter}. First, before the call \texttt{(Iter S0 T D)}, we find the environment \( E_0 \):

\[ E_0=\{..., R, S0=s0, T=tf, D=df\} \]

Then, before the call \texttt{(Iter State1 Transform IsDone)}, we find the environment \( E_1 \):

\[ E_1=\{..., R, State1=s1, Transform=tf, IsDone=df\} \]

If we continue execution, then before the recursive calls to \texttt{(Iter State1 Transform IsDone)} we will find environments of the form:

\[ E_2=\{..., R, State1=s2, Transform=tf, IsDone=df\} \]
\[ E_3=\{..., R, State1=s3, Transform=tf, IsDone=df\} \]
\[ E_4=\{..., R, State1=s4, Transform=tf, IsDone=df\} \]
\[ ... \]

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
5.3 Iterative computation

We see that the computation transforms one state into another (i.e., the binding of State1), without changing anything else. In particular, the environments contain exactly the same identifiers. Continuing execution, we eventually come to a call \{Iter State1 Transform IsDone\} with environment:

\[ E_n = [..., R, State1=sn, Transform=tf, IsDone=df] \]

such that \{IsDone sn\} = true. Executing this call will immediately return sn and bind R to it, giving the environment:

\[ E_n = [..., R=sn] \]

What happens for procedures

We have shown the execution of the function Iter. For procedures, the execution unfolds in exactly the same way. If Iter were a procedure, then R would be an argument, and there would be a sequence of calls of the form:

\{Iter s0 T D R\}
\{Iter s1 Transform IsDone R\}
\{Iter s2 Transform IsDone R\}
\{Iter s3 Transform IsDone R\}
...
\{Iter sn Transform IsDone R\}
R=sn

At the end, R is bound to sn just like before.

5.3.4 Last call optimization

During the execution of Iter, the environment does not grow. What is so special about Iter that makes this so? It is simple: it is the “trimming” of the environment that can be done when variable identifiers are no longer needed. When entering a function, the environment is extended with the formal parameters until the function completes. But it is possible to restore the original environment earlier. When, exactly, is this possible? The recursive function call \{Iter Start1 Transform IsDone\} in the body of Iter is the last call:

```plaintext
fun {Iter State Transform IsDone}
  if {IsDone State} then State
  else Statel in
    Statel={Transform State}
    {Iter Statel Transform IsDone} % Last call
  end
end
```

As soon as the last call is started, then the caller’s environment is no longer needed. This is easy to see: when the last call returns, then there is nothing more to do in the caller, which itself can return immediately.
Last call optimization means to remove the caller’s environment at the moment one enters the last call in the function body. This optimization is always possible and is easy to implement. Last call optimization makes iterative computations possible in a declarative program. All reasonable implementations of declarative programming must support last call optimization, since otherwise they would not be able to do iterative computations as efficiently as imperative programming, i.e., programming with explicit state. Iterative computation can be expressed with recursion where the recursive function calls are last calls. This means that at any point in time during a series of recursive calls, only one set of identifiers is needed, namely for the currently active one. All the others have started their last call.

If the function is recursive, then the last call optimization is often called the tail recursion optimization. A common misuse of terminology is to talk about tail recursion optimization when one means last call optimization.

There is a generalization of last call optimization called environment trimming. The idea is to reduce the environment size at each call, by keeping in the environment only the identifiers needed after the call.

5.4 Recursive computation

A recursive computation is one whose environment size is a linear function of the recursion depth during execution. A typical example is the factorial function. Its definition is:

\[
\begin{align*}
0! &= 1 \\
n! &= n \cdot (n - 1)! \text{ if } n > 0
\end{align*}
\]

This is a recurrence equation, i.e., the factorial \( n! \) is defined in terms of a factorial with a smaller argument, namely \((n - 1)!\). The procedure definition follows this mathematical definition. To calculate \( \text{Fact N} \) there are two possibilities, namely \( N=0 \) or \( N>0 \). In the first case, return 1. In the second case, calculate \( \text{Fact N-1} \), multiply by \( N \), and return the result. This gives the following program:

```plaintext
fun \{Fact N\}
  if N==0 then 1
  elseif N>0 then N*{Fact N-1}
  else raise domainError end
end
```

Note that factorial is a partial function. It is not defined for negative \( N \). The program reflects that by raising an exception in that case. We can see there is no infinite loop since the factorial of a big number is defined in terms of the factorial of a smaller number. Since the numbers are all nonnegative, so they bottom out at zero.
Notice that we have done two things. First, we have followed the mathematical
definition to get a correct implementation. Second, we have reasoned about
termination, i.e., we show that the program terminates for all legal arguments,
i.e., arguments in the function’s domain.

This definition of factorial gives a recursive computation, not an iterative
computation. This is easy to see by following the same reasoning as for the Iter
function in the previous section. We will skip the details and directly give the
summary. For the initial statement

```
local
    R={Fact 5}
in
    {Browse R}
end
```

we get the following statements just before each call to Fact:

```
R={Fact 5}
R=(5*{Fact 4})
R=(5*(4*{Fact 3}))
R=(5*(4*(3*{Fact 2})))
R=(5*(4*(3*(2*{Fact 1}))))
R=(5*(4*(3*(2*(1*{Fact 0})))))
```

Notice that the expression gets bigger and bigger. This is because each multipli-
cation depends on the result of a Fact call. The multiplication cannot be done
until the Fact call returns. Now let us return from these calls one by one and
calculate the final result:

```
R=(5*(4*(3*(2*(1*1)))))
R=(5*(4*(3*(2*1))))
R=(5*(4*(3*2)))
R=(5*(4*6))
R=(5*24)
R=120
```

Why does the recursion depth increase with the argument? Essentially, it is
because there is an operation (i.e., multiplication) that has to be done after the
recursive call. The environment of each call therefore cannot be trimmed.

Let us rearrange the computation of factorial to become an iterative compu-
tation. Later on, we will give a systematic way of making iterative computations.
For now, we just give a hint. In the previous calculation:

```
R=(5*(4*(3*(2*(1*1)))))
```

it is enough to rearrange the numbers:

```
R=(((1*5)*4)*3)*2)*1)
```

Then the calculation can be done incrementally, starting with 1*5. This gives 5,
then 20, then 60, then 120, and finally 120. The iterative definition of factorial
that does things this way is:
fun {FactIter N A}
   if N==0 then A
   elseif N>0 then {FactIter N-1 A*N}
   else raise domainError end
end

fun {Fact N}
   {FactIter N 1}
end

Observe that the function that does the work, FactIter, has an extra argument A. This extra argument is crucial; without it an iterative factorial is impossible.

## 5.5 Thinking inductively

When writing declarative programs, the most important technique is thinking inductively. Declarative programs very often manipulate structures that are defined inductively, i.e., in terms of smaller versions of themselves. The programs reflect this inductive structure. In this section, we will look at techniques for writing programs that manipulate lists. In later sections, we will extend these techniques to other inductive data structures such as trees or graphs.

A list has an inductive structure: it is either nil or it is a pair of an element and a smaller list. In the usual case, a program that manipulates lists has the following structure:

- **Base case.** For sufficiently small lists, it computes the answer directly.

- **Inductive case.** For bigger lists, it computes the result in terms of the results of one or more smaller lists.

Here is a simple recursive function that calculates the length of a list according to this technique:

```
fun {Length Ls}
   case Ls
      of nil then 0
      [] _ | Lr then 1+{Length Lr}
   end
end

{Browse {Length [a b c]}} % displays 3
```

The base case is the empty list nil, for which the function returns 0. The inductive case is any other list. If the list has length $n$, then its tail has length $n-1$. The tail is smaller than the original list, so the program will terminate.

Our second example is a function that appends two lists $Ls$ and $Ms$ together to make a third list. The question is, on which list do we use induction? Is it the first or the second? We claim that the induction has to be done on the first list. Here is the function:

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
fun {Append Ls Ms}
  case Ls
  of nil then Ms
      [] X|Lr then X|{Append Lr Ms}
  end
end

This function follows exactly the following two properties of append:

- append(nil, m) = m
- append(x|l, m) = x | append(l, m)

The inductive case always calls Append with a smaller first argument, so the program terminates.

5.6 Thinking about efficiency

Declarative programming is still programming; even though it has strong mathematical properties it still results in real programs that run on real computers. Therefore, it is important to think about computational efficiency: the execution time and the memory space consumed by the program. In this section we give a simple example of how to calculate the execution time of a program. We calculate the approximate time by counting the number of “basic operations” that the program does, under the assumption that basic operations take a fixed time. Later sections will often use similar kinds of reasoning.

This kind of calculation is necessarily approximate, but it is usually sufficient to find the asymptotic time complexity of a program. The asymptotic time complexity is the best upper bound on the execution time as a function of the program’s input size, up to a constant factor. For example, say a sorting program takes $34n^2$ nanoseconds to sort an input list of length $n$. Using “big-oh” notation, we say that the program’s asymptotic time complexity is $O(n^2)$.

The “big-oh” notation $O(f(n))$ represents a positive real number that is bounded above by $c.f(n)$ where $c$ is a constant that does not depend on $n$. To be precise, there exists positive constants $c$ and $n_0$ such that $\forall n \geq n_0$, the number is bounded above by $c.f(n)$. Big-oh notation has to be handled with care (see, e.g., Knuth [56], for more details). Saying $g(n) = O(f(n))$ is really an abuse of the “=” symbol, since there is no equality, but just an upper bound. But tradition has kept it this way. There is an analogous “little-oh” notation $o(f(n))$, that represents a number that becomes vanishingly small in relation to $f(n)$. That is, the number divided by $f(n)$ has limit zero as $n$ approaches infinity.

To find the exact “wall-clock” time needed by a program, counting the basic operations is not good enough. There is only one technique that really works well: execute the program and measure the time it takes. It is extremely hard, well-nigh impossible in fact, to calculate this time. Why is this? It is because
modern computers have a complex structure that introduces much unpredictability in the execution time: they have complex memory systems (with caches and virtual memory), they have complex pipelined architectures (many instructions are simultaneously in various stages of execution; an instruction’s execution time depends on the other instructions present), and the operating system does context switches at unpredictable times. For more information on measuring performance and its pitfalls, we recommend [49].

We take a small program and calculate how many basic operations it takes. We will use the results of this calculation to improve the program. The program will append three lists together, \( L_s, M_s, \) and \( N_s \). The simplest way is to use the two-list append as a primitive. Then there are two ways to define a three-list append. The first way starts by appending \( M_s \) and \( N_s \):

\[
\text{fun} \ \{\text{Append3a} \ L_s \ M_s \ N_s\} \\
{\text{Append} \ L_s \ \{\text{Append} \ M_s \ N_s\}}
\]

end

The second way starts by appending \( L_s \) and \( M_s \):

\[
\text{fun} \ \{\text{Append3b} \ L_s \ M_s \ N_s\} \\
{\text{Append} \ \{\text{Append} \ L_s \ M_s\} \ N_s}
\]

end

Which way is best? To answer this question, we introduce the notion of operation complexity of the program, defined as the number of operations it does. The execution time of the program is proportional to the number of operations. To make things precise, assume that lists \( L_s, M_s, \) and \( N_s \) have lengths \( l, m, \) and \( n \) respectively. Then the function call \( \{\text{Append} \ L_s \ M_s\} \) does \( l \) case statements, \( l \) list pair creations (e.g., \( X|... \)), and \( l \) recursive calls, for a total of \( 3l+2 \) operations (which includes the original call and case). We can derive this formula easily by mathematical induction. Let \( T(l) \) denote the number of operations for a list of length \( l \). Then there are two cases:

- **Base case.** If \( L_s \) is \( \text{nil} \), then two operations are done (the original call and one \text{case} statement): \( T(0) = 2 \).

- **Inductive case.** If \( L_s \) is not \( \text{nil} \), then three operations are done (one case, one list pair, and one recursive call), and then we have a list with one less element: \( T(l) = 3 + T(l-1) \).

Solving these two equations gives us \( T(l) = 3l+2 \). Now we can calculate the number of operations for \( \text{Append3a} \) and \( \text{Append3b} \). We can define the two functions \( T_{\text{Append3a}}(l,m) \) and \( T_{\text{Append3b}}(l,m) \)

- \( T_{\text{Append3a}}(l,m) = 1 + T(m) + T(l) = 3m + 3l + 5 \)
- \( T_{\text{Append3b}}(l,m) = 1 + T(l) + T(l + m) = 3m + 6l + 5 \)
Solving recurrence equations

Here is a simple three-step technique that almost always works in practice. First, get exact numbers for some small inputs (in our case: \( T(0) = 1, T(1) = 4, T(2) = 7 \)). Second, guess the form of the result (in our case: \( T(n) = an + b \), for some as yet unknown \( a \) and \( b \)). Third, plug in the guessed form into the equations. In our case this gives \( b = 1 \) and \( (an + b) = 3 + (a(n-1) + b) \). This gives \( a = 3 \), for a final result of \( T(n) = 3n + 1 \). The three-step technique works if the guessed form is correct. There is another, much more powerful technique, called generating functions, that gives closed-form or asymptotic results in a wide variety of cases without having to guess the form. It requires some technical knowledge of infinite series and calculus, but not more than is seen in a first university-level course on these subjects. See Knuth [56] and Wilf [107] for good introductions to generating functions.

This shows that Append3a is the fastest: it always does \( 3l \) less operations than Append3b. Recurrence equations like these often show up when calculating the number of operations that programs take. Solving them is not always easy. The box explains some of the important techniques:

**BOX: give table of some common recurrence equations and their solutions.**
- \( T(x) = a + bT(x-1) \)
- \( T(x) = a + bT(x-1) + cT(x-2) \)
- \( T(x) = a + bx + cT(x/2) \)

Cite Foundations of Computer Science, Aho and Ullman.

### 5.7 Records, lists, and trees

Three basic compound data structures are records, lists, and trees. Records are the most basic one. We construct lists and trees by means of records.

#### 5.7.1 Programming with lists

Lists are one of the most basic recursive data structures. List values are very concise to write and patterns let us take them apart so easily. This makes declarative programming with lists very easy and powerful.

**Recursive functions and their domains**

Let us define the function \texttt{Nth} to get the \( n \)th element of a list. This function is usually provided as a primitive. In Mozart, the Base module \texttt{List} defines it. But...
it’s nice to see how it works:

```haskell
fun {Nth Xs N}     
  if N==1 then Xs.1
  elseif N>1 then {Nth Xs.2 N-1}
  end
end
```

Remember that a list Xs is either nil or a tuple X|Y with two arguments. Xs.1 gives X and Xs.2 gives Y. What happens when we feed the following:

```haskell
{Browse {Nth [a b c d] 5}}
```

The list has four elements. Trying to ask for the fifth element means trying to do Xs.1 or Xs.2 when Xs=nil. This will raise an exception. This is an example of a general technique to define functions: always use statements that raise exceptions when values are given outside their domains. This will ensure that a function will raise an exception when called with an input outside the domain.

We can define a function that sums all the elements of a list of integers:

```haskell
fun {SumList Xs}     
  case Xs
    of nil then 0
    [] X|Xr then X+{SumList Xr}
  end
end
{Browse {SumList [1 2 3]}}
```

This displays 6. Since Xs can be one of two values, namely nil or X|Xr, it is natural to use a `case` statement. As in the `Nth` example, not using an `else` in the case will raise an exception if the argument is outside the domain of the function. For example:

```haskell
{Browse {SumList 1|foo}}
```

raises an exception because 1|foo is not a list, and the definition of `SumList` assumes that its input is a list.

**Naive definitions are often slow**

We define a function to reverse the elements of a list. Start with an inductive definition of list reversal:

- Reverse of nil is nil.
- Reverse of X|Xs is Z, where reverse of Xs is Ys, and append Ys and [X] to get Z.

This works because X is moved from the front to the back. Following this inductive definition, we can immediately write a function:
fun {Reverse Xs}
  case Xs
  of nil then nil
  [] X|Xr then
      {Append {Reverse Xr} [X]}
      end
  end
end

Is this function efficient? To see, we will calculate its time complexity, given an input list of length \(n\). We use the three-step technique explained before. Assume \(T(n)\) is the number of operations. Then \(T(0) = 2\) (one call and one case). We know that Append takes \(3k + 2\) operations for an input list of length \(k\) (see before). This means that \(T(n) = 3 + (3(n-1) + 2) + T(n-1)\) (one call, one case, and one list pair construction \([x]\)). Assume that \(T(n) = an^2 + bn + c\). Solving this recurrence gives \(T(n) = (3n^2 + 7n + 4)/2\). Usually we do not have to be this precise; it is enough to know the most-significant term. This means \(T(n) = O(n^2)\), which is quite slow. We would expect that reversing a list would take time proportional to the input length, not to its square. There is another defect of this program: it defines a recursive computation, not an iterative one. Naively following the inductive definition has given us a rather inefficient result! Luckily, there are simple techniques for getting around both these inefficiencies. They will let us define linear-time iterative computations whenever possible.

Converting recursive into iterative computations

Let us see how to convert recursive computations into iterative ones. Instead of using Reverse, we take a simpler function that calculates the length of a list:

fun {Length Xs}
  case Xs of nil then 0
  [] _ |Xr then 1+{Length Xr}
  end
end

Note that the SumList function has the same structure. This function is linear-time but it defines a recursive computation: the environment size is proportional to the recursion depth, which is equal to the length of \(Xs\). Why does this problem occur? It is because the addition \(1+{Length Lr}\) happens after the recursive call. The recursive call is not last, so the function’s environment cannot be recovered before it.

How can we calculate the list length with an iterative computation, which has bounded environment size? To do this, we have to formulate the problem as a sequence of state transformations. That is, we start with a state \(s_0\) and we transform it successively, \(s_1, s_2, \ldots\), until we reach the final state \(s_n\), which is the answer. For the list length, we can take the length of the part of the list already seen as being the state. Each time we look at one more element, we add 1 to this state. This gives us the function:
fun {IterLength N Xs}
  case Xs
  of nil then N
     [] _|Xr then {IterLength N+1 Xr}
   end
end

Note the difference with the previous definition. Here, the addition N+1 is done before the recursive call to IterLength, which is the last call. We have defined an iterative computation.

A minor problem is that in the initial call, the initial value of N has to be given. This initial value is 0 since we have not yet looked at any part of the list. This initialization can be hidden by defining IterLength as a local procedure. The final result is therefore:

local
  fun {IterLength N Xs}
    case Xs
    of nil then N
       [] _|Xr then {IterLength N+1 Xr}
    end
  end
in
  fun {Length Xs}
    {IterLength 0 Xs}
  end
end

This defines an iterative computation to calculate the list length. There is no advantage to defining IterLength local to Length, since it does not use Length’s argument Xs. We therefore define it in a separate local declaration. This avoids creating a new closure each time Length is called.

We can use the same technique on Reverse as we used for Length. In the case of Reverse, the state is the reverse of the part of the list already seen. Updating the state is easy: we just put a new list element in front. The initial state is nil. This gives the following version of Reverse:

local
  fun {IterReverse Rs Xs}
    case Xs
    of nil then Rs
       [] X|Xr then {IterReverse X|Rs Xr}
    end
  end
in
  fun {Reverse Xs}
    {IterReverse nil Xs}
  end
end

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
This version of \texttt{Reverse} is both linear-time and an iterative computation.

\textbf{The case of the iterative append}

How can we append two lists together? The naive way is the following function:

\begin{verbatim}
fun {Append As Bs}
  case As of nil then Bs
     [] A|As2 then A|{Append As2 Bs}
  end
end
\end{verbatim}

The call \(\{\text{Append}\ [a\ b]\ [c\ d]\}\) returns \([a\ b\ c\ d]\). This seems simple enough. But there is more here than meets the eye. First of all, the above definition of \texttt{Append} uses functional nesting (see Section 4.4.8). The expanded definition is:

\begin{verbatim}
proc {Append As Bs Cs}
  case As of nil then Bs=Cs
     [] A|As2 then Cs2 in
     Cs=A|Cs2
        {Append As2 Bs Cs2}
  end
end
\end{verbatim}

This shows that \texttt{Append} is an iterative computation, because it can do last call optimization. What’s more, the argument \(Cs2\) is an unbound dataflow variable. Last call optimization is only possible here because of dataflow variables. If the language did not have dataflow variables, then the straightforward definition of \texttt{Append} would have to be as follows:

\begin{verbatim}
fun {Append As Bs}
  case As of nil then Bs
     [] A|Ar then Cr in
        Cr={Append Ar Bs}
        A|Cr
  end
end
\end{verbatim}

This definition is recursive, not iterative. This is because the list pairing operation \(A|Cr\) requires that \(A\) and \(Cr\) both be values. The pairing operation must therefore be done \textit{after} the recursive call.

It is possible to write an iterative \texttt{Append} without dataflow variables, but it is more complicated. We strongly suggest that you put down the book right now and give it a try. The effort will pay dividends later on. Here is a solution:

\begin{verbatim}
local
  fun {Prepend Rs Bs}
    case Rs
\end{verbatim}
This uses the iterative Reverse defined previously. The function \{Prepend Rs Bs\} appends the reverse of Rs to Bs. We conclude that the existence of dataflow variables makes it easier to write iterative list operations.

Performing an operation on all list elements

Let us write a function that scales the elements of a list, i.e., that multiplies all of them by the same number, giving a new list:

\[
\text{fun } \{\text{Scale Xs Factor}\} \\
\text{case Xs of} \\
\text{nil then nil} \\
[] X|Xr then (X*Factor)|\{Scale Xr Factor\} \\
end \\
end
\]

This function is an example of a pattern that we might want to use with other operations than multiplication with a constant. For example, we might want to add a constant to all elements of a list. We can make the pattern generic by letting the operation be an argument. This gives the following definition:

\[
\text{fun } \{\text{Map Xs F}\} \\
\text{case Xs of} \\
\text{nil then nil} \\
[] X|Xr then \{F X\}|\{Map Xr F\} \\
end \\
end
\]

This replaces the scaling by a function F that is passed as an argument. We call the new function Map since it maps list elements according to any function. We say Map is a generic mapping function. Now we can write Scale in terms of Map:

\[
\text{fun } \{\text{Scale Xs Factor}\} \\
\{\text{Map Xs fun } \{X\} \ X*Factor \end\}
\]

Notice the use of an anonymous procedure to get the correct number of arguments: we define a one-argument function that multiplies by a constant.
5.7 Records, lists, and trees

Nested lists

Lists can be nested, e.g., \([1 2 4 [5 10]]\). Here is a function that counts the elements of a nested list. We assume that anything that is not a list pair is a leaf to be counted.

\[
\text{fun} \ \{\text{Leaf} \ X\} \ X\!\!=\!(\_|\_) \ \text{end}
\]

\[
\text{fun} \ \{\text{LengthL} \ Ls\}
\begin{align*}
\text{case} & \ Ls \\
\text{of} & \ \text{nil} \ \text{then} \ 0 \\
[& \ [] \ L|Lr \ \text{then} \\
\quad & \text{if} \ \{\text{Leaf} \ L\} \ \text{then} \ 1+\{\text{LengthL} \ Lr\} \\
\quad & \ \text{else} \ \{\text{LengthL} \ Lr\}+\{\text{LengthL} \ L\} \ \text{end}
\end{align*}
\end{align*}
\]

What do these two calls display:

\[
\text{declare} \\
X=[[1 \ 2 \ 4 \ [5 \ 10]]] \\
XX=[X \ X] \\
\{\text{Browse} \ \{\text{LengthL} \ X\}\} \\
\{\text{Browse} \ \{\text{LengthL} \ XX\}\}
\]

One way to represent a matrix is as a list of lists. We can use the \text{Map} function to sum the elements of a matrix represented as a list of lists:

\[
\text{declare} \\
\text{fun} \ \{\text{SumMatrix} \ XXs\}
\begin{align*}
\ & \{\text{SumList} \ \{\text{Map} \ XXs \ \text{SumList}\}\}
\end{align*}
\]

\[
\text{XXs}=[[1 \ 2 \ 3 \ 1] \\
[1 \ 1 \ 2 \ 3] \\
[3 \ 1 \ 1 \ 2] \\
[2 \ 3 \ 1 \ 1]] \\
\{\text{Browse} \ \{\text{SumMatrix} \ XXs\}\}
\]

Sorting with mergesort

We define a function that takes a list of numbers or atoms and returns a new list sorted in ascending order. It uses the comparison operator \(<\), so the numbers all have to be of the same type (all integers, all reals, or all atoms). We use the mergesort algorithm, which is efficient and can be programmed easily in a declarative model. The mergesort algorithm uses a simple divide-and-conquer strategy:

- Split the list into two smaller lists of approximately equal length.
- Use mergesort recursively to sort the two smaller lists.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Figure 5.9: Sorting with mergesort

- Merge the two sorted lists together to get the final result.

Figure 5.9 shows the recursive structure. Mergesort is efficient because the split and merge operations are both linear-time iterative computations. We first define the merge and split operations and then mergesort itself:

```plaintext
fun {Merge Xs Ys}
case Xs # Ys
  of nil # Ys then Ys
  [] Xs # nil then Xs
  [] (X|Xr) # (Y|Yr) then
    if X<Y then X|{Merge Xr Ys}
    else Y|{Merge Xs Yr}
  end
end
end
```

We define split as a procedure because it has two outputs. It could also be defined as a function returning a pair as a single output.

```plaintext
proc {Split Xs Ys Zs}
case Xs
  of nil then Ys=Zs=nil
  [] [X] then Ys=[X] Zs=nil
  [] X1|X2|Xr then Yr Zr in
    Ys=X1|Yr
    Zs=X2|Zr
    {Split Xr Yr Zr}
  end
end
```

Here is the definition of mergesort itself:

```plaintext
fun {MergeSort Xs}
case Xs
  of nil then nil
end
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
[] [X] then Xs  % This is better than [X]
else Ys Zs in
   {Split Xs Ys Zs}
   {Merge {MergeSort Ys} {MergeSort Zs}}
end
end

The splitting up of the input list bottoms out at lists of length zero and one, which can be sorted immediately. A small question for the attentive reader: when the input is [X], why is it better to return Xs instead of [X]? Think carefully before reading the answer in the footnote.5

How efficient is mergesort? Let us calculate its time complexity. Define \( T(n) \) and \( S(n) \) as the time complexities of \( \text{MergeSort} \) and \( \text{Split} \) in number of basic operations, where \( n \) is the length of the input list. We know that \( S(n) \) is a linear function. We know also that \( \text{Split} \) outputs two lists of lengths \( \lceil n/2 \rceil \) and \( \lfloor n/2 \rfloor \). From the definition of \( \text{MergeSort} \), we can define the following recurrence equations:

- \( T(0) = 2 \)
- \( T(1) = 2 \)
- \( T(n) = 2 + S(n) + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) \) if \( n \geq 2 \)

This uses the ceiling and floor functions, which are a bit tricky. To get rid of them, assume that \( n \) is a power of 2, i.e., \( n = 2^k \) for some \( k \). Also write \( 2 + S(n) = an + b \) for some constants \( a \) and \( b \). Then the equations become:

- \( T(0) = 2 \)
- \( T(1) = 2 \)
- \( T(n) = an + b + 2T(n/2) \) if \( n \geq 2 \)

Expanding the last equation gives:

- \( T(n) = \sum_{i=0}^{k} \left( \frac{n}{2^i} \right)(an + b) + 2T(1) \)

Replacing \( L(n) \) and \( T(1) \) by their values gives:

- \( T(n) = \sum_{i=0}^{k} \left( \frac{n}{2^i} \right)(an + b) + 4 \)

Doing the sum gives:

\[ T(n) = \sum_{i=0}^{k} \left( \frac{n}{2^i} \right)(an + b) + 4 \]

---

5Returning [X] creates a new list pair. Returning Xs reuses an existing list pair, which uses less operations and less memory. A good compiler will detect that [X] is the same as Xs, and replace the former by the latter.
\[ T(n) = an + (n - 1)b + 4 \]

We conclude that \( T(n) = O(n \log n) \). For values of \( n \) that are not powers of 2, we use the easily-proved fact that \( n \leq m \Rightarrow T(n) \leq T(m) \) to show that the big-oh bound still holds. The bound is independent of the content of the input list. This means that the \( O(n \log n) \) bound is also a \textit{worst-case} bound.

A generic mergesort

The above definition of mergesort is hardwired to use the ‘<’ comparison function. In the same way we went from Scale to Map, we can make mergesort generic by passing the comparison function as an argument. We only have to change the \textit{Merge} and \textit{MergeSort} functions; \textit{Split} is unaffected:

```plaintext
fun {GenericMergeSort F Xs} fun {Merge Xs Ys} case Xs # Ys of nil # Ys then Ys [] Xs # nil then Xs [] (X|Xr) # (Y|Yr) then if {FXY} then X|{Merge Xr Ys} else Y|{Merge Xs Yr} end end fun {MergeSort Xs} case Xs of nil then nil [] [X] then Xs else Ys Zs in {Split Xs Ys Zs} {Merge {MergeSort Ys} {MergeSort Zs}} end end in {MergeSort Xs} end
```

We put the definitions of \textit{Merge} and \textit{MergeSort} inside a new function \textit{GenericMergeSort}. This avoids passing the function \( F \) on each recursive call to \textit{Merge} and \textit{MergeSort}. It works by creating two new closures when \textit{GenericMergeSort} is called: \textit{Merge}, which references \( F \), and \textit{MergeSort}, which references the new \textit{Merge}. We can define the original mergesort in terms of \textit{GenericMergeSort}:

```plaintext
fun {MergeSort Xs} {GenericMergeSort Number.‘<’ Xs} end
```

We write \textit{Number.‘<’} because the comparison ‘<’ is in the module \textit{Number}.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Accumulating parameters

An accumulating parameter (also known as accumulator) is a programming technique that uses procedure arguments to carry a state, as shown in the previous example. Each procedure calculates the next state from the previous state. This means that one accumulating parameter requires two extra arguments in the procedure: one to input the previous state and one to output the next state. In IterLength and IterReverse this structure is not so clear, because they are functions. What is happening is that the input state is passed to the function and the output state is what the function returns.

Consider the following procedure, which takes an expression containing identifiers, integers, and addition operations (using label plus). It calculates two results: it translates the expression into machine code for a simple stack machine and it calculates the number of instructions in the resulting code.

```
proc {ExprCode E C1 C0 S1 S0}
  case E
    of plus(A B) then C2 C3 S2 S3 in
      {ExprCode A C1 C2 S1 S2}
      {ExprCode B C2 C3 S2 S3}
      C3=plus|C0
      S0=S3+1
    [] I then
      C1=push(I)|C0
      S0=S1+1
    end
  end
end
```

This procedure has two accumulating parameters: one to build the list of machine instructions and another to hold the number of instructions. Here is a sample execution:

```
{ExprCode plus(plus(a 3) b)) Code nil 0 Size}
{Browse Size#Code}
```

This displays:

```
5#[push(a) push(3) plus push(b) plus]
```

More complicated programs need more accumulating parameters. When writing large declarative programs, we have typically used around half a dozen accumulating parameters simultaneously. The Aquarius Prolog compiler was written in this style [101, 100]. Some of its procedures have as many as 12 accumulating parameters. This means 24 additional arguments! This is difficult to do without mechanical aid. We used a preprocessor that takes declarations of accumulating parameters and adds the arguments automatically. We no longer program in this style; we find that programming with state is simpler and more efficient (see Chapter 8).

It’s reasonable to use a few accumulating parameters in a declarative program; it’s actually quite rare that a declarative program does not need a few. On the
other hand, if you need many, then it’s a sign that some of them would be better written with state.

**Sorting with mergesort (revisited)**

In the previous definition of mergesort, we first called the function Split to divide the input list into two halves. There is a simpler way to do the mergesort, by using an accumulating parameter. The parameter represents “the part of the list still to be sorted”. The specification of MergeSort then becomes:

- \{MergeSort S1 S0 N\} takes an input list S1 and an integer N. It returns the sorted list of the first N elements of S1. It binds S0 to the remainder of S1.

The accumulating parameter is defined by the two arguments S1 and S0. This gives the following definition:

```plaintext
fun {MergeSort Xs}
  fun {MergeSort S1 S0 N}
    if N==0 then
      S1=S0 nil
    elseif N==1 then X in
      S1=X|S0 [X]
    elseif N>1 then
      NL=N div 2
      NR=N-NL
      in
        {Merge {MergeSort S1 S2 NL} {MergeSort S2 S0 NR}}
      end
    end
  end
  {MergeSort Xs nil {Length Xs}}
end
```

The Merge function is unchanged. We can make this definition generic by changing Merge in the same way as before. Remark that this mergesort does a different split than the previous one. Here, the split separates the first half and the second half of the input list. In the previous version, split separates the odd-numbered list elements from the even-numbered elements.

This version has the same time complexity as the previous version. It uses less memory because it does not create the two split lists. They are defined implicitly by the combination of the accumulating parameter and the number of elements.

**Declarative dictionaries**

We introduce a very useful data type, the *dictionary*. A dictionary is a mapping from a set of atoms to a set of language entities. Each atom is mapped to a language entity. We would like to be able to create the mapping dynamically,
i.e., by adding to it during the execution. This gives the following set of basic operations:

- `{NewDictionary}` returns a new empty dictionary.
- `{Put D Key Value}` returns a new dictionary which is identical to D except that it adds the mapping Key→Value. If Key already exists in D, then the new dictionary replaces it.
- `{Get D Key}` returns the value corresponding to Key.
- `{Domain D}` returns a list of the keys in D.

Figure 5.10 shows a declarative implementation of this specification. The dictionary is represented as a list of pairs Key#Value that are sorted on the key. Instead of Get, we define a slightly more general access operation, CondGet:

- `{CondGet D Key Default}` returns the value corresponding to Key. If Key is not present, then returns Default.

This is almost as easy to implement as Get and is very useful, as we will see in the next example.

This implementation is extremely slow for large dictionaries. Put and CondGet need to look at half the list on average, if the key is present, and the whole list if it is not. The number of operations is \(O(n)\) for dictionaries with \(n\) keys. We say that the implementation does a \textit{linear search}. A more efficient declarative implementation is possible by using a binary tree. With careful programming, this reduces the \textit{worst-case} number of operations to \(O(\log n)\). This is the best we can do in the declarative model. Section 5.7.4 gives a simple version using binary trees that has \textit{average} performance of \(O(\log n)\).

**Word frequencies**

Now we have defined enough functions to do substantial calculations. Let us write a program to count word frequencies in a string. Later on, we will see how to use this to count words in a file. Figure 5.11 defines the function `WordFreq`, which is given a list of characters \(Cs\) and returns a list of pairs \(W#N\), where \(W\) is a word (a maximal sequence of letters and digits) and \(N\) is the number of times the word occurs in \(Cs\). The function `WordFreq` is defined in terms of the following functions:

- `{WordChar C}` returns true iff \(C\) is a letter or digit.
- `{WordToAtom PW}` converts a reversed list of word characters into an atom containing those characters. The function `VirtualString.toAtom` is used to create the atom.
fun {NewDictionary} nil end
fun {Put Ds Key Value}
  case Ds
  of nil then [Key=Value]
    [] (K#V)|Dr andthen Key==K then
     (Key#Value) | Dr
    [] (K#V)|Dr andthen K>Key then
     (Key#Value)|(K#V)|Dr
    [] (K#V)|Dr andthen K<Key then
     (K#V) |{Put Dr Key Value}
  end
end
fun {CondGet Ds Key Default}
  case Ds
  of nil then Default
    [] (K#V)|Dr andthen Key==K then
     V
    [] (K#V)|Dr andthen K>Key then
     Default
    [] (K#V)|Dr andthen K<Key then
     {CondGet Dr Key Default}
  end
end
fun {Domain Ds}
  {Map Ds fun {$ K#__} K end}
end

Figure 5.10: Declarative dictionary (with linear list)
fun \{WordChar C\} \\
\text{\((&a=C \text{ andthen } C=\&z) \text{ orelse (}&A=C \text{ andthen } C=\&Z) \text{ orelse (}&0=C \text{ andthen } C=\&9)\) end

fun \{WordToAtom PW\} \\
\text{(VirtualString.toAtom \{Reverse PW\}) end}

fun \{IncWord D W\} \\
\text{(Put D W \{CondGet D W 0\}+1) end}

fun \{CharsToWords PW Cs\} \\
\text{case Cs of nil andthen PW==nil then nil \[
\text{ nil then \{WordToAtom PW\}\] C|Cr andthen \{WordChar C\} then CharsToWords \{Char.toLower C\}|PW Cr\[\] C|Cr andthen PW==nil then CharsToWords nil Cr\[\] C|Cr then WordToAtom PW|CharsToWords nil Cr\end\end}

fun \{CountWords D Ws\} \\
\text{case Ws of W|Wr then \{CountWords \{IncWord D W\} Wr\} \[
\text{ nil then D\end\end}

fun \{WordFreq Cs\} \\
\text{(CountWords \{NewDictionary\} \{CharsToWords nil Cs\}) end}

Figure 5.11: Word frequencies (with declarative dictionary)
{IncWord D W} takes a dictionary \( D \) and an atom \( W \). Returns a new dictionary in which the \( W \) field is incremented by 1. Remark how easy this is to write with CondGet, which takes care of the case when \( W \) is not yet in the dictionary.

{CharsToWords \( \text{nil} \) Cs} takes a list of characters Cs and returns a list of atoms, where the characters in each atom’s print name form a word in Cs. The function Char.toLowerCase is used to convert uppercase letters to lowercase, so that “The” and “the” are considered the same word.

{CountWords D Ws} takes an empty dictionary and the output of CharsToWords. It returns a dictionary in which each key maps to the number of times the word occurs.

Here is a sample execution. The following input:

```declare```
```
T="Oh my darling, oh my darling, oh my darling Clementine. She is lost and gone forever, oh my darling Clementine."
{Browse \{WordFreq T\}}
```

displays this word frequency count:

```[she#1 is#1 clementine#2 lost#1 my#4 darling#4 gone#1 and#1 oh#4 forever#1]```

We have run \text{WordFreq} on a more substantial text, namely an early draft of the book. The text contains 712626 characters, giving a total of 110457 words of which 5561 are different. We have run \text{WordFreq} with three implementations of dictionaries: using lists (see previous example), using binary trees (see Section 5.7.4), and using state (the built-in implementation of dictionaries; see Section 8.10.1). Figure 5.12 shows part of the internal structure of the binary tree.
5.7 Records, lists, and trees

dictionary, drawn with the algorithm of Section 5.7.5. The code we measured is in Section 5.12.3. Running it gives the following times (accurate to 10%).

<table>
<thead>
<tr>
<th>Dictionary implementation</th>
<th>Execution time</th>
<th>Operation complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using lists</td>
<td>620 seconds</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Using binary trees</td>
<td>8 seconds</td>
<td>$O(log n)$</td>
</tr>
<tr>
<td>Using state</td>
<td>2 seconds</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

The time is the wall-clock time to do everything, i.e., read the text file, run WordFreq, and write a file containing the word counts. The difference between the three times is due completely to the different dictionary implementations. Comparing the times gives a good example of the practical effect of using different implementations of an important data type. The complexity shows how the time to insert or look up one item depends on the size of the dictionary.

5.7.2 Parsing with an accumulating parameter

We now give a more realistic example of declarative programming. We will write a parser for a small Pascal-like language. A parser is part of a compiler. A compiler translates a sequence of characters, which represents a program, into a sequence of low-level instructions that can be executed on a machine. In its most basic form, a compiler consists of three parts:

- **Tokenizer.** The tokenizer reads a sequence of characters and outputs a sequence of tokens.
- **Parser.** The parser reads a sequence of tokens and outputs an abstract syntax tree.
- **Code generator.** The code generator traverses the syntax tree and generates low-level instructions.

Usually this structure is extended by optimizers that try to improve the generated code. In this section, we will just write the parser. We first define the input and output formats of the parser. The parser accepts a sequence of tokens according to the grammar given in Table 5.2 and outputs an abstract syntax tree. The grammar is carefully designed to be right recursive and deterministic. This means that the choice of grammar rule is completely determined by the next token. For example, in the rule for $\langle$Term$\rangle$ the absence of a $\langle$TOP$\rangle$ token means that the next token has to be in the set $\{ \langle$EOP$\rangle, \langle$COP$\rangle, ;, end, then, do, else, $\rangle \}$. This makes it possible to write a top down, left to right parser with only one token lookahead.

There are two kinds of symbols in Table 5.2: nonterminals and terminals. A **nonterminal** symbol is one that is further expanded according to a grammar.
Table 5.2: The parser’s input language (which is a token sequence)

rule. A terminal symbol corresponds directly to a token in the input. It is not expanded. The nonterminal symbols are \( \langle \text{Prog} \rangle \) (complete program), \( \langle \text{Stat} \rangle \) (statement), \( \langle \text{Comp} \rangle \) (comparison), \( \langle \text{Expr} \rangle \) (expression), \( \langle \text{Term} \rangle \) (term), \( \langle \text{Fact} \rangle \) (factor), \( \langle \text{COP} \rangle \) (comparison operator), \( \langle \text{EOP} \rangle \) (expression operator), and \( \langle \text{TOP} \rangle \) (term operator). To parse a program, start with \( \langle \text{Prog} \rangle \) and expand until finding a sequence of tokens that matches the input.

The parser output is a nested record with syntax given in Table 5.3. Superficially, Tables 5.2 and 5.3 have very similar content, but they are actually quite different: the first defines a sequence of tokens and the second defines a nested record. The first does not show the structure of the input program—we say it is flat. The second exposes this structure—we say it is nested. Because it exposes the program’s structure, we call the nested record an abstract syntax tree. The parser’s role is to extract the structure from the flat input. Without this structure, it is extremely difficult to write the code generator and code optimizers.

The main parser call is the function \{Prog S1 S0\}, where S1 is an input list of tokens and S0 is the rest of the list after parsing. This call returns the parsed output. For example:

```latex
\textbf{declare} \textbf{A} \textbf{S0 in}
\{Prog \textbf{A}
  \langle \text{Prog} \rangle \quad \langle \text{Stat} \rangle \quad \langle \text{Comp} \rangle \quad \langle \text{Expr} \rangle \quad \langle \text{Term} \rangle \quad \langle \text{Fact} \rangle \quad \langle \text{COP} \rangle \quad \langle \text{EOP} \rangle \quad \langle \text{TOP} \rangle \quad \langle \text{Integer} \rangle \quad \langle \text{Id} \rangle
\langle \text{program foo ; ; while a + 3 < b do b := b + 1 end} \rangle \quad \textbf{S0}
\{Browse \textbf{A}\}
```

displays:

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
\[ \text{Prog} ::= \text{prog}(\text{Id} \text{Stat}) \]
\[ \text{Stat} ::= \text{'};\text{'}(\text{Stat} \text{Stat}) \]
\[ \text{assign}(\text{Id} \text{Expr}) \]
\[ \text{'if'}(\text{Comp} \text{Stat} \text{Stat}) \]
\[ \text{while}(\text{Comp} \text{Stat}) \]
\[ \text{read}(\text{Id}) \]
\[ \text{write}(\text{Expr}) \]

\[ \text{Comp} ::= \text{COP}(\text{Expr} \text{Expr}) \]
\[ \text{Expr} ::= \text{Id} | \text{Integer} | \text{OP}(\text{Expr} \text{Expr}) \]
\[ \text{COP} ::= \text{'=='} | \text{'}!=\text{'} | \text{'}>' | \text{'}<\text{'} | \text{'}='=\text{'} | \text{'}>=\text{'} \]
\[ \text{OP} ::= \text{'+'} | \text{'}-' | \text{'}*\text{'} | \text{'}/\text{'} \]
\[ \text{Integer} ::= \text{(integer)} \]
\[ \text{Id} ::= \text{(atom)} \]

Table 5.3: The parser’s output language (which is a value)

\[ \text{prog}(\text{foo while}(\text{'}<\text{'}(\text{'}+\text{'}(\text{a 3}) \text{b}) \text{assign}(\text{b} \text{'}+\text{'}(\text{b 1})))) \]

We give commented program code for the complete parser. \text{Prog} is written as follows:

\begin{verbatim}
fun {Prog S1 S0}
  Y Z S2 S3 S4
in
  S1=program|S2
  Y={Id S2 S3}
  S3=\text{'};\text{'}|S4
  Z={Stat S4 S0}
  prog(Y Z)
end
\end{verbatim}

The accumulating parameter is passed through all terminal and nonterminal symbols. Each nonterminal symbol has a procedure to parse it. Statements are parsed with \text{Stat}, which is written as follows:

\begin{verbatim}
fun {Stat S1 S0}
  T|S2=S1
in
  case T of begin then
    {Sequence Stat fun {\$ X} X\text{'}==\text{'};\text{'} end S2 \text{'}end\text{'}|S0}
[] \text{'if'} then C X1 X2 S3 S4 S5 S6 in
  {Comp C S2 S3}
  S3=\text{'then'}|S4
  X1={Stat S4 S5}
  S5=\text{'else'}|S6
  X2={Stat S6 S0}
end
\end{verbatim}
The one-token lookahead is put in \( T \). With a \texttt{case} statement, the correct branch of the \texttt{Stat} grammar rule is found. Statement sequences (surrounded by \texttt{begin} – \texttt{end}) are parsed by the procedure \texttt{Sequence}. This is a generic procedure that also handles comparison sequences, expression sequences, and term sequences. It is written as follows:

\[
\text{fun} \ \{\text{Sequence NonTerm Sep S1 S0}\} \\
\quad \text{X1 S2 T S3} \\
\text{in} \\
\quad \text{X1}=\{\text{NonTerm S1 S2}\} \\
\quad \text{S2}=\text{T}|\text{S3} \\
\quad \text{if} \ \{\text{Sep T}\} \ \text{then} \ \text{X2 in} \\
\quad \quad \{\text{Sequence NonTerm Sep X2 S3 S0}\} \\
\quad \quad \text{T(X1 X2)} \quad \text{% Dynamic record creation} \\
\quad \text{else} \\
\quad \quad \text{S2}=\text{S0} \\
\quad \quad \text{X1} \\
\text{end} \\
\text{end}
\]

This takes two input functions, \texttt{NonTerm}, which is parses any nonterminal, and \texttt{Sep}, which detects the separator symbol in a sequence. Comparisons, expressions, and terms are parsed as follows with \texttt{Sequence}:

\[
\begin{align*}
\text{fun} \ \{\text{Comp S1 S0}\} \ \{\text{Sequence Expr COP S1 S0}\} \ \text{end} \\
\text{fun} \ \{\text{Expr S1 S0}\} \ \{\text{Sequence Term EOP S1 S0}\} \ \text{end} \\
\text{fun} \ \{\text{Term S1 S0}\} \ \{\text{Sequence Fact TOP S1 S0}\} \ \text{end}
\end{align*}
\]
Each of these three functions has its corresponding function for detecting separators:

\[
\text{fun } \{\text{COP } Y\} \\
Y==`<` \text{ orelse } Y==`>` \text{ orelse } Y==`=<` \text{ orelse } Y==`=>` \text{ orelse } Y==`=!` \\
end
\]

\[
\text{fun } \{\text{EOP } Y\} \ Y==`+` \text{ orelse } Y==`-` \text{ end}
\]

\[
\text{fun } \{\text{TOP } Y\} \ Y==`*` \text{ orelse } Y==`/` \text{ end}
\]

Finally, factors and identifiers are parsed as follows:

\[
\text{fun } \{\text{Fact } S1 \ S0\} \\
T \mid S2=S1 \\
in \\
\quad \text{if } \{\text{IsInt } T\} \text{ orelse } \{\text{IsIdent } T\} \text{ then} \\
\quad \quad \S2=S0 \\
\quad \quad \ T \\
\quad \text{else } E \ S2 \ S3 \text{ in} \\
\quad \quad \S1=`('|S2 \\
\quad \quad \ E=\{\text{Expr } S2 \ S3\} \\
\quad \quad \S3=`('|S0 \\
\quad \quad \ E \\
\quad \text{end} \\
\text{end}
\]

\[
\text{fun } \{\text{Id } S1 \ S0\} \ S1=X|S0 \text{ true=}\{\text{IsIdent } X\} \ X \text{ end}
\]

\[
\text{fun } \{\text{IsIdent } X\} \ \{\text{IsAtom } X\} \text{ end}
\]

Integers are represented as built-in integer values and detected using the built-in \text{IsInt} function.

This parsing technique works for grammars where one-token lookahead is enough. Some grammars, called ambiguous grammars, require to look at more than one token to decide which grammar rule is needed. A simple way to parse them is with nondeterministic choice, as explained in Chapter 7.

- What is the error behavior of this parser?

5.7.3 Programming with difference lists

A difference list is a pair of two lists, each of which might have an unbound tail. The two lists have a special relationship: it must be possible to get the second list from the first by removing zero or more elements from the front.

Difference lists are a special case of difference structures. A difference structure is a pair of two partial values. The second value should be “embedded” in the first. The difference structure represents a value that is the first structure minus the second structure. Using difference structures makes it easy to construct iterative computations on many recursive datatypes, e.g., lists or trees. It also allows to use simple inductive reasoning which results in iterative computation.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
If you have not heard of it before, the concept of difference list might seem completely strange and abstract. Some examples will show that it is actually quite natural:

\[
\begin{align*}
& X\#X \quad \text{\% Represents the empty list} \\
& \text{nil}\#\text{nil} \quad \text{\% idem} \\
& [a]\#[a] \quad \text{\% idem} \\
& (a|b|c|X)\#X \quad \text{\% Represents \{a b c\}} \\
& (a|b|c|d|X)\#(d|X) \quad \text{\% idem} \\
& [a b c d]\#[d] \quad \text{\% idem}
\end{align*}
\]

The advantage of using difference lists is that when the second list is unbound, another difference list can be appended to it in constant time. To append \((a|X)\#X\) and \((b|c|Y)\#Y\), just bind \(X\) and \((b|c|Y)\). This creates the difference list \((a|b|c|Y)\#Y\). Here is a function that appends two difference lists:

```
fun {AppendD D1 D2}
    S1#E1=D1
    S2#E2=D2
in
    E1=S2
    S1#E2
end
```

It can be used like a list append:

```
local X Y in {Browse {AppendD (1|2|3|X)#X (4|5|Y)#Y}} end
```

This displays \((1|2|3|4|5|Y)#Y\). The standard list append function iterates on its first argument, and therefore takes time proportional to the length of the first argument. Here is its definition:

```
fun {Append L1 L2}
    case L1 of
        X|T then X|{Append T L2}
        [] nil then L2
    end
end
```

The disadvantage of using difference lists is that they can be appended only once. This property means that difference lists can only be used in special circumstances. For example, they are a natural way to write programs that construct big lists in terms of lots of little lists that must be appended together.

Difference lists are a data type that lives in the “no-man’s land” between values and state. They are not values, since they contain unbound variables. They are not stateful, since they cannot be assigned to more than one value. However, this value can start out by being unknown. Binding can be seen as a way of making the value “become known”. As a result, difference lists have the advantage of being declarative while partaking of some of the power of destructive assignment.
Flattening a list

Consider the problem of flattening a list. We first give a solution using lists and then we show that a much better solution is possible with difference lists. Assume we have a list whose elements are either non-list terms or lists. These lists can also have members that are non-list terms or lists, and so forth. We would like to calculate one list that contains all the non-list elements, no matter at what level they were in the input list. With inductive reasoning on lists we can construct a program in a very natural way:

- Flatten of nil is nil.
- Flatten of \( X | Xs \) is \( Z \), where
  - flatten of \( X \) is \( Y \),
  - flatten of \( Xs \) is \( Ys \), and
  - append \( Y \) and \( Ys \) to get \( Z \).
- Flatten of anything else, call it \( X \), is \([X]\).

Following this inductive reasoning, we get the following definition:

```plaintext
fun \{Flatten Xs\}
  case Xs
    of nil then nil
    [] X|Xr then
      {Append \{Flatten X\} \{Flatten Xr\}}
    else [Xs]
  end
end
```

Calling:

```
{Browse \{Flatten \[[a b] [[c] [d]] nil [e [f]]]\}}
```

displays \([a b c d e f]\). This program is very inefficient because it needs to do many append operations (see Exercises). Now we do the same inductive reasoning as before, but using difference lists:

- Flatten of \( \text{nil} \) is \( X#X \) (empty difference list).
- Flatten of \( X | Xs \) is \( Y1#Y4 \), where
  - flatten of \( X \) is \( Y1#Y2 \),
  - flatten of \( Xs \) is \( Y3#Y4 \), and
  - equate \( Y2 \) and \( Y3 \) to append the difference lists.
- Flatten of anything else, call it \( X \), is \( (X|Y)#Y \) (one element different list).

We can write the second case as follows:

- Flatten of \( X | Xs \) is \( Y1#Y4 \), where
  - flatten of \( X \) is \( Y1#Y2 \) and
  - flatten of \( Xs \) is \( Y2#Y4 \).

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
This gives the following program:

```
fun {Flatten Xs}
  proc {FlattenD Xs Ds}
    case Xs
    of nil then Y in Ds=Y#Y
        [] X|Xr then Y0 Y1 Y2 in
            Ds=Y0#Y2
            {FlattenD X Y0#Y1}
            {FlattenD Xr Y1#Y2}
        [] X then Y in Ds=(X|Y)#Y
    end
    end
    in
    {FlattenD Xs $#nil}
  end
end
```

This program is efficient: its number of operations is proportional to the total number of atoms and lists in the input. We convert the difference list returned by FlattenD into a regular list by binding its second argument to nil. We write FlattenD as a procedure because its output is part of its last argument, not the whole argument (see Section 4.4.9). It is common style to write a difference list in two arguments:

```
fun {Flatten Xs}
  proc {FlattenD Xs S E}
    case Xs
    of nil then S=E
        [] X|Xr then S1 in
            Ds=(X|S1)
            {FlattenD X S S1}
            {FlattenD Xr S1 E}
        [] X then S=X|E
    end
    end
    in
    {FlattenD Xs $ nil}
  end
end
```

As a further simplification, we can write FlattenD as a function. To do this, we use the fact that S is the output:

```
fun {Flatten Xs}
  fun (FlattenD Xs E)
    case Xs
    of nil then E
        [] X|Xr then
            Ds=(X|{FlattenD X {FlattenD Xr E}})
        [] X then X|E
    end
    end
end
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
What is the role of $E$? It gives the “rest” of the output, i.e., when the FlattenD call exhausts its own contribution to the output.

### Reversing a list

We give some other procedures using difference lists. For example, look again at the list reversal function that we studied in the last section. We can define it as follows with difference lists:

- Reverse of $\text{nil}$ is $X\#X$ (empty difference list).
- Reverse of $X|Xs$ is $Z$, where
  reverse of $Xs$ is $Y1\#Y2$ and
  append $Y1\#Y2$ and $(X|Y)\#Y$ together to get $Z$.

Rewrite the last case as follows, by doing the append:

- Reverse of $X|Xs$ is $Y1\#Y$, where
  reverse of $Xs$ is $Y1\#Y2$ and
  equate $Y2$ and $X|Y$.

It is perfectly allowable to move the equate before the reverse (why?). This gives:

- Reverse of $X|Xs$ is $Y1\#Y$, where
  reverse of $Xs$ is $Y1\#(X|Y)$.

Here is the final definition:

```haskell
fun {Reverse Xs}
proc {ReverseD Xs Y1 Y}
case Xs
  of nil then Y1=Y
[] X|Xr then
  {ReverseD Xr Y1 X|Y}
end
end
in
  {ReverseD Xs $ nil}
end
```

Look carefully and you will see that this is almost exactly the same solution as in the last section. The only difference between IterReverse and ReverseD is the argument order: the output of IterReverse is the second argument of ReverseD. So what’s the advantage of using difference lists? We derived ReverseD without thinking, whereas to derive IterReverse we had to guess an intermediate state that could be updated.
A FIFO queue

An interesting application of difference lists is to define FIFO queues. A FIFO queue is a sequence of elements with an insert and a delete operation. The insert operation adds an element to one end of the queue and the delete operation removes an element from the other end. With difference lists we can implement FIFO queues with constant-time insert and delete operations. Here is the definition:

\[
\begin{align*}
\text{fun } \{\text{NewQueue}\} X \in & \\
& q(0 \ X \ X) \\
\text{end} \\
\text{fun } \{\text{Insert } q(N \ S \ E) \ X\} \ E1 \ in & \\
& E=X\!\!|E1 \\
& q(N+1 \ S \ E1) \\
\text{end} \\
\text{proc } \{\text{Delete } q(N \ S \ E) \ Q \ X\} \ S1 \ in & \\
& \text{in} \\
& S=X\!\!|S1 \\
& Q=q(N-1 \ S1 \ E) \\
\text{end} \\
\text{fun } \{\text{EmptyQueue } q(N \ S \ E)\} \ N==0 \ \text{end}
\end{align*}
\]

This uses the triple \(q(\ N \ S \ E)\) to represent the queue’s state. \(N\) is the number of elements in the queue. The difference list \(S\#E\) is the queue’s content. Here is an example how to use the queue:

\[
\begin{align*}
\text{declare } Q1 \ Q2 \ Q3 \ Q4 \ Q5 \ Q6 & \\
& \{\text{Browse } \{\text{NewQueue}\}=Q1\} \\
& \{\text{Browse } \{\text{Insert } Q1 \ 1\}=Q2\} \\
& \{\text{Browse } \{\text{Insert } Q2 \ 2\}=Q3\} \\
& \{\text{Browse } \{\text{Delete } Q3 \ Q4 \ $\}\} \\
& \{\text{Browse } \{\text{Delete } Q4 \ Q5 \ $\}\} \\
& \{\text{Browse } \{\text{Delete } Q5 \ Q6 \ $\}\} \\
& \{\text{Browse } \{\text{Insert } Q6 \ \text{seif}\}\}
\end{align*}
\]

We insert three elements and delete them.

5.7.4 Programming with trees

Next to lists, trees are the most important recursive data structure in a programmer’s repertory.
Declarative dictionaries with binary trees

In Section 5.7.1 we saw a very useful data type, the dictionary, implemented in a very slow way, using lists. Now we show how to improve this implementation by using binary trees. A dictionary is defined as a tree with the following syntax:

\[
\langle \text{tree} \rangle ::= \text{tree}(\text{key}:K, \text{val}:V, \text{left}: \langle \text{tree} \rangle, \text{right}: \langle \text{tree} \rangle) \\
| \text{leaf}
\]

We assume the following invariant: for all nodes, the node’s key is greater than all keys in its left subtree and less than all keys in its right subtree. With this invariant, we can rewrite the dictionary operations as given in Figure 5.13. In this implementation, the \textit{Put} and \textit{CondGet} operations take \(O(\log n)\) time and space for a tree with \(n\) nodes, given that the tree is “reasonably balanced”. That is, for each node, the sizes of the left and right subtrees should not be “too different”.

We can do even better than this implementation by leaving the declarative model behind and using explicit state (see Section 8.5.2). Using state can reduce the number of operations to \(O(1)\), i.e., bounded by a constant.

5.7.5 Drawing trees

Given a binary tree in abstract syntax, we would like to draw the tree in an aesthetically pleasing way. The problem is to calculate the coordinates of each node. We define the abstract syntax as follows:

\[
\langle \text{tree} \rangle ::= \text{tree}(\text{key}:K, \text{val}:V, \text{left}: \langle \text{tree} \rangle, \text{right}: \langle \text{tree} \rangle) \\
| \text{leaf}
\]

Each node has zero, one, or two children. Assume that we have the following constraints on how the tree is drawn:

1. There is a minimum horizontal spacing between both subtrees of every node. To be precise, the rightmost node of the left subtree is at a minimal horizontal distance from the leftmost node of the right subtree.
2. If a node has two child nodes, then its horizontal position is the arithmetic average of their horizontal positions.
3. If a node has only one child node, then the child is directly underneath it.
4. The vertical position of a node is proportional to its level in the tree.

Figure 5.14 shows these constraints graphically in terms of the coordinates of each node. The example tree of Figure 5.15 is drawn as shown in Figure 5.17. The initial arguments are \texttt{Level=1} and \texttt{LeftLim=Scale}, where \texttt{Scale} gives the minimum distance between nodes.

The algorithm to calculate the node’s positions is based on traversing the tree, passing information between nodes, and calculating values at each node.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
fun \{NewDictionary\} leaf end
fun \{Put Ds Key Value\}
case Ds
  of leaf then
      tree(key:Key value:Value left:leaf right:leaf)
      \[] tree(key:K left:L right:R ...) andthen K==K then
          tree(key:K value:Value left:L right:R)
      \[] tree(key:K value:V left:L right:R) andthen K>Key then
          tree(key:K value:V left:{Put L Key Value} right:R)
      \[] tree(key:K value:V left:L right:R) andthen K<Key then
          tree(key:K value:V left:L right:{Put R Key Value})
  end
end
fun \{CondGet Ds Key Default\}
case Ds
  of leaf then
      Default
      \[] tree(key:K value:V ...) andthen K==K then
          V
      \[] tree(key:K left:L ...) andthen K>Key then
          \{CondGet L Key Default\}
      \[] tree(key:K right:R ...) andthen K<Key then
          \{CondGet R Key Default\}
  end
end
fun \{Domain Ds\}
proc \{DomainD Ds S1 S0\}
case Ds
  of leaf then
      S1=S0
      \[] tree(key:K left:L right:R ...) then S2 S3 in
          S1=K|S2
          \{DomainD L S2 S3\}
          \{DomainD R S3 S0\}
  end
end
in \{DomainD Ds $ nil\} end

Figure 5.13: Declarative dictionary (with binary tree)
1. Distance \( d \) between subtrees has minimum value
2. If two children exist, \( a \) is average of \( b \) and \( c \)
3. If only one child exists, it is directly below parent
4. Vertical position \( y \) is proportional to level in the tree

![Tree drawing constraints diagram](image)

Figure 5.14: Tree drawing constraints

```
tree(key:a val:111
    left:tree(key:b val:55
        left:tree(key:x val:100
            left:tree(key:z val:56 left:leaf right:leaf)
            right:tree(key:w val:23 left:leaf right:leaf))
        right:tree(key:y val:105 left:leaf
            right:tree(key:r val:77 left:leaf right:leaf))))
    right:tree(key:c val:123
        left:tree(key:d val:119
            left:tree(key:g val:44 left:leaf right:leaf)
            right:tree(key:h val:50
                left:tree(key:i val:5 left:leaf right:leaf)
                right:tree(key:e val:133 left:leaf right:leaf))
    end
end
```

Figure 5.15: An example tree

The traversal has to be done carefully so that all the information is available at the right time. Exactly what traversal is the right one depends on what the constraints are. For the above four constraints, it is sufficient to traverse the tree in a depth-first order. In this order, each left subtree of a node is visited before the right subtree. A basic depth-first traversal looks like this:

```
proc {DepthFirst Tree}
    case Tree
        of tree(left:L right:R ...) then
            {DepthFirst L}
            {DepthFirst R}
            [] leaf then
                skip
            end
        end
end
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
The tree drawing algorithm does a depth-first traversal and calculates the \((x,y)\) coordinates of each node during the traversal. As a first step to implementing the algorithm, we extend the tree’s abstract syntax with the fields \(x\) and \(y\) at each node:

\[
\text{fun } \{\text{AddXY Tree}\}
\text{ case Tree}
\text{ of } \text{tree(left:L right:R ...)} \text{ then}
\{\text{Adjoin Tree tree(x:_ y:_ left:{AddXY L} right:{AddXY R})}\}
\[\] \text{leaf then}
\text{leaf}
\text{ end}
\text{ end}
\]

The function \text{AddXY} returns a new tree with the two fields \(x\) and \(y\) added to all nodes. It uses the \text{Adjoin} function, which is explained in Section B.3. The function \text{AddXY} illustrates the general principle of software decomposition, in which independent parts of a program are kept independent as much as possible. Here the independent parts are the tree-drawing module and the program that uses the tree. Two programming techniques are used to realize the principle:

- The function adds two unbound variables to each node. These unbound variables are filled in later.

- The function adds two fields without making any assumptions about the other fields in the \text{tree} record. This means that the new tree keeps all the information of the old tree. If later on, the tree is modified to have other fields than \text{key} and \text{val}, then the tree drawing algorithm need not be changed at all.

Actually, to guarantee that there is no conflict between the added fields and the original field, \text{names} should be used. The correct definition of \text{AddXY} is then:

\[
\text{local}
\text{ X=\{NewName\}}
\text{ Y=\{NewName\}}
\text{ in}
\text{ fun } \{\text{AddXY Tree}\}
\text{ case Tree}
\text{ of } \text{tree(left:L right:R ...)} \text{ then}
\{\text{Adjoin Tree}
\text{ tree(X:_ Y:_ left:{AddXY L} right:{AddXY R})}\}
\[\] \text{leaf then}
\text{leaf}
\text{ end}
\text{ % All code that needs X and Y comes here}
\text{ end}
\]

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Each identifier \( x \) and \( y \) references a name. Because names are globally unique and unforgeable (see Section B.2), there will never be any conflicts with fields in the original tree.

To implement the tree drawing algorithm, we extend the depth-first traversal by passing two arguments \textit{down} (namely, level in the tree and limit on leftmost position of subtree) and two arguments \textit{up} (namely, horizontal position of the subtree’s root and rightmost position of subtree). Downward-passed arguments are sometimes called \textit{inherited} arguments. Upward-passed arguments are sometimes called \textit{synthesized} arguments. With these extra arguments, we have enough information to calculate the positions of all nodes. Figure 5.16 gives the complete tree drawing algorithm. The \texttt{Scale} parameter gives the basic size unit of the drawn tree. There are four cases, depending on whether a node has two subtrees, one subtree (left or right), or zero subtrees. The pattern matching tests in the \texttt{case} statement pick the right case. This takes advantage of the fact that the tests are done in sequential order.
5.7.6 The declarative for loop

As the examples in the previous sections show, loops in the declarative model tend to be verbose because they need explicit recursive calls. We define a linguistic abstraction, the declarative for loop, to improve this situation. We give examples to show the different possibilities.

The for loop is not the only way to write declarative loops. Another way is to use higher-order programming with lists, as explained in Section 5.8. Many of the most useful higher-order operations are provided as built-in operations in the \texttt{List} module. Using for loops is easier than higher-order programming; when writing loops we recommend to try them first.

\textbf{Iterating over integers}

A common operation is iterating for successive integers from a lower bound \( I \) to a higher bound \( J \). Without loop syntax, the standard declarative way to do this needs two steps. First define a looping function:

\begin{verbatim}
fun {For I J P}
  if I>J then nil else {P I} {For I+1 J P} end
end
\end{verbatim}

Then call the looping function:

\begin{verbatim}
{For I J proc {$ P} (Stmt) end}
\end{verbatim}

Both steps can be replaced by a single \texttt{for} loop:

\begin{verbatim}
for X in I..J do (Stmt) end
\end{verbatim}
The Mozart system accepts this simpler syntax and translates it internally into a simple recursive function:

```plaintext
local
  fun {For X J P}
    if X>J then nil else (Stmt) {For X+1 J P} end
  end
in
  {For I J}
end
```

This translation is hidden from the programmer.

### Iterating over lists

The `for` loop can be extended to iterate over lists as well as over integer intervals. For example, the call:

```plaintext
{ForAll Ls proc ${ X} (Stmt) end}
```

can be written as:

```plaintext
for X in Ls do (Stmt) end
```

The list can be a stream, i.e., potentially unbounded.

### Patterns

The `for` loop can be extended to contain patterns that implicitly declare variables. For example, if the elements of `Ls` are triplets of the form `obj(name:N price:P location:L)`, then we can loop over them as follows:

```plaintext
for obj(name:N price:P location:L) in Ls do
  if P<1000 then {Show N} end
end
```

This declares and binds the new variables `N`, `P`, and `L` for each iteration. Their scope ranges over the loop body.

### Collecting results

A useful extension of the `for` loop is to collect results. For example, let us make a list of all integers from 1 to 1000 that not multiples of either 2 or 3:

```plaintext
for I in 1..1000 collect:C do
  if I mod 2 \= 0 andthen I mod 3 \= 0 then {C I} end
end
```

The “`collect:C`” declaration defines a collection procedure `C` whose scope ranges over the loop body. The collection procedure can be seen as a way to add elements to an internal accumulating parameter. Without `for` this can be done in two steps. First define the following function:
fun {Collect I J}
  if I>J then nil
  else
  if I mod 2 \= 0 andthen I mod 3 \= 0 then
    I|{Collect I+1 J}
  else
    {Collect I J}
  end
end
end
Then call the function as {Collect 1 1000}.

Other useful extensions

The above examples give some of the most-used looping idioms in a declarative
loop syntax. Many more looping idioms are possible. For example: immediately
exiting the loop (break), immediately exiting and returning an explicit result
(return), immediately continuing with the next iteration (continue), multiple
iterators that advance in lockstep, and other collection procedures (e.g., append
and prepend for lists and sum and maximize for integers). The complete loop
support in Mozart is documented in [26]. For other example designs of declarative
loops we recommend studying the loop macro of Common LISP [86] and the state
threads package of SICStus Prolog [54].

5.8 Higher-order programming

Higher-order programming means to pass procedures as arguments to other pro-
cedures. This is the most powerful technique available in the declarative model.
We can define the concept of order of a procedure. A procedure all of whose
arguments are not procedures is of order zero. A procedure that has at least one
zero-order procedure in an argument is of order one. And so forth: a procedure
is of order \( n + 1 \) if it has at least one argument of order \( n \) and none of higher
order. Higher-order programming means simply that procedures can be of any
order, not just order zero.

Higher-order programming is especially useful together with procedural ab-
straction. Remember that procedural abstraction allows to take any statement
and turn it into a procedure (see Section 4.4.6). This means that a procedure
represents a “packet” of work, in a very general sense. There are two operations
on a procedure: defining it and calling it. A procedure is defined once and can be
called any number of times. There is complete freedom to choose what happens
to the statement’s free variables. They can either be bound at definition time
(when the procedure is declared; happens at a particular time and program point
at run time) or at call time (when the procedure is called; its formal parameters
are bound at call time).

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Here are some of the things we can do with higher-order programming:

- We can make *generic* operations. For example, a sorting routine can be based on a boolean function, which determines the order between two elements. Different order functions can be passed to the same sorting routine.

- We can build *modules* by grouping together related operations in records. This leads naturally to the concept of *software component*, which is a module specification that defines the modules it needs (see Section 8.9).

- By bringing together the previous two items, we can build *frameworks* that can be parameterized for particular tasks. For example, a numerical analysis package can have routines to do numerical integration, differentiation, and solving differential equations. These routines can take functions as arguments.

- We can define new *control structures* as higher-order procedures. This is possible because a procedure is a packet of work whose execution is delayed. It is the control structure that decides when to execute it.

- We can do *lazy evaluation*, that is, evaluation on demand. The consumer of a data structure is given a function that calculates a pair: part of the data structure and a new function. This means the consumer can control explicitly how much of the data structure must be evaluated.

- We can do *formal manipulations* with functions. For example, we can define a general function that takes any two one-argument functions and returns their functional composition.

In the following sections, we examine these possibilities.

### 5.8.1 Defining generic procedures

We have already seen three examples of higher-order programming in the previous sections. They were introduced so gently that perhaps you have not noticed them. The general iteration function $\text{Iter}$ of Section 5.3.3 uses two procedure arguments, $\text{Transform}$ and $\text{IsDone}$. The $\text{Map}$ function of Section 5.7.1 uses one procedure argument, $F$. The $\text{GenericMergeSort}$ function takes a comparison function as argument.

All three examples use higher-orderness to make functions *generic*. To make a function generic is to replace something specific (any entity, operation, or value) by a parameter of the function. We say the thing is *abstracted out* of the function body. The specific thing is given when the function is called. Each time the function is called another thing can be given. Consider the function $\text{SumList}$:

```plaintext
fun {SumList Xs}
   case Xs
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
This function has two specific entities: the number zero (0) and the operation plus (+). The zero is a neutral element for the plus operation. These two entities can be abstracted out. Any neutral element and any operation are possible. We give them as parameters. For historical reasons, the generic function is called FoldR:

```plaintext
fun {FoldR Xs F U}
  case Xs
    of nil then U
    [] X|Xr then {F X {FoldR Xr F U}}
  end
end
```

Now we can define SumList as a special case of FoldR:

```plaintext
fun {Add X Y} X+Y end
fun {SumList Xs} {FoldR Xs Add 0} end
```

FoldR takes its name from the fact that it associates to the right. For the list Xs=[X1 X2 ... Xn], calling {FoldR Xs F U} gives:

```
{F X1 {F X2 {.....{F Xn U}}}}
```

This is like using F as an operator that is grouped from the right. If f is an infix operator and u is the neutral element, then this looks like:

```
(x_1f(x_2f(...(x_nf u)))
```

We can use FoldR to define other functions on lists. Here is a simple one:

```plaintext
fun {Multiply X Y} X*Y end
fun {ProductList Xs} {FoldR Xs Multiply 1} end
```

Here is one that returns true if there is a true in the list:

```plaintext
fun {Or X Y} X orelse Y end
fun {Some Xs} {FoldR Xs Or false} end
```

A problem with the above definition of FoldR is that it defines a recursive computation. The environment size can grow very big. Can you define a version of FoldR that is iterative? The way to do it is to define an intermediate state and a state transformation function. Look at the nested formula given above—what is the intermediate state? How do you get to the next state? Before peeking at the answer, we suggest you put down the book and try to define an iterative FoldR. Here is the answer:

```plaintext
fun {FoldR Xs F U}
  fun {Loop R Xs}
    case Xs
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
of X|Xr then
  {Loop {F X R} Xr}
[] nil then
  R
end
end
in
{Loop U {Reverse Xs}}
end

FoldR has a cousin FoldL, which associates to the left:

fun {FoldL Xs F U}
  fun {Loop R Xs}
    case Xs
    of X|Xr then
      {Loop {F R X} Xr}
    [] nil then
      R
    end
  end
in
  {Loop U Xs}
end

This definition is very similar to FoldR. The resulting calculation is:

{F ... (F {F U X1} X2) ... Xn}

5.8.2 Traversing and folding trees

It is very common to traverse trees, i.e., to visit all tree nodes in a particular order and do certain operations while visiting the nodes. For example, the compiler mentioned in Section 5.7.2 has to traverse the nodes of the abstract syntax tree to generate machine code. The tree drawing program of Section 5.7.5, after it calculates the node’s positions, has to traverse the nodes in order to draw them.

First we will traverse a tree in a generic way. Consider an n-ary tree with the following structure:

```
(tree) ::= tree(node:N sons:<treelist>)
(treelist) ::= nil | {<tree>}+ |
```

Each node can have any number of sons. One of the simplest forms of traversal is known as depth-first traversal. The skeleton of this is given in Section 5.7.5 for binary trees. Here is a simple depth-first traversal for n-ary trees:

```
proc (DepthFirst Tree)
  tree(sons:Sons ...) = Tree
in
  for T in Sons do (DepthFirst T) end
end
```
We can “decorate” this routine to do something at each node it visits. For example, we call \{P T\} for each node \(T\). This is done as follows:

\[
\begin{align*}
\text{proc} & \quad \{\text{VisitNodes Tree P}\}
\quad \text{tree}(\text{sons:Sons ...})=\text{Tree}
\quad \text{in}
\quad \{\text{P Tree}\}
\quad \text{for } T \text{ in Sons do } \{\text{VisitNodes } T\} \text{ end}
\text{end}
\end{align*}
\]

An only slightly more involved traversal is to call \{P Tree T\} for each father-son link of a father node \(Tree\) and one of its sons \(T\):

\[
\begin{align*}
\text{proc} & \quad \{\text{VisitLinks Tree P}\}
\quad \text{tree}(\text{sons:Sons ...})=\text{Tree}
\quad \text{in}
\quad \text{for } T \text{ in Sons do }
\quad \{\text{P Tree } T\}
\quad \{\text{VisitLinks } T\}
\quad \text{end}
\text{end}
\end{align*}
\]

These two routines were used to draw the trees of Section 5.7.5 after the node positions were calculated. VisitLinks drew the lines between nodes and VisitNodes drew the nodes themselves.

Another way to traverse a tree is to accumulate a result when visiting its nodes. This is similar to a fold operation with lists. Because the tree’s structure is more complicated than a list’s, there are more possibilities. For example, assume we calculate a folded value for each node. Then the folded value for a father is a function of the father’s node and the values for the sons. There are two functions: \(LF\) to fold together all sons of a given father, and \(TF\) to fold their result together with the father. This gives the following:

\[
\begin{align*}
\text{local} & \quad \text{fun } \{\text{FoldTreeR Sons TF LF U}\}
\quad \text{case Sons}
\quad \text{of } \text{nil } \text{then } U
\quad \{[] \text{ S}\mid \text{Sons2 } \text{then }
\quad \{\text{LF } \{\text{FoldTree } S \text{ TF LF U}\} \{\text{FoldTreeR } \text{Sons2 } \text{ TF LF U}\}\}
\quad \text{end}
\quad \text{end}
\quad \text{in}
\quad \text{fun } \{\text{FoldTree } Tree TF LF U\}
\quad \text{tree}(\text{node:N sons:Sons ...})=\text{Tree}
\quad \text{in}
\quad \{\text{TF N } \{\text{FoldTreeR Sons TF LF U}\}\}
\quad \text{end}
\quad \text{end}
\end{align*}
\]

For example:
local
proc {HelpOp Op C To Step P}
  if {Op C To} then {P C} {HelpOp Op C+Step To Step P}
end
end
in
proc {For From To Step P}
  {HelpOp Value. (if Step>0 then ´=<´ else ´>=´ end)
               From To Step P}
end
end

Figure 5.18: The For iterator

declare
T=tree(node:1
  [tree(node:2 nil)
   tree(node:3 [tree(node:4 nil)])])
{Browse {FoldTree T Add Add 0}}

This displays 10.

5.8.3 New control structures

Higher-order procedures can be used to define new control structures. Many are
declared in the Base modules Control and List, as well as in other modules.
Here are some examples. The procedure {For From To Step P} is an iterator
that applies the unary procedure P to integers from From to To proceeding in
steps Step. If P is of order n, then For is of order n + 1. Executing {For
1 10 1 Browse} displays the integers 1 2 ... 10. Executing {For 10 1 ~1
Browse} displays 10 9 8 ... 1. The For iterator is defined in Figure 5.18. It
picks the right comparison operator, ´=<´ or ´>=´, by record selection from the
module Value.

Another often-used control structure is the ForAll iterator defined in the
List module (see Section B.5). ForAll applies a unary procedure on all the
elements of a list, in the order defined by the list. It is defined as follows:

proc {ForAll Xs P}
  case Xs
  of nil then skip
  [] X|Xr then
      {P X}
      {ForAll Xr P}
  end
end

What happens if the list is produced incrementally by another concurrent thread?
local
  fun {And B1 B2}
    if B1 then B2 else false end
  end
in
  fun {BinaryTree T}
    case T
    of leaf then true
      [] tree(K V T1 T2) then
        {And {BinaryTree T1} {BinaryTree T2}}
      else false end
    end
  end
end

Figure 5.19: Checking a binary tree

In this case the ForAll iteration will synchronize on the availability of data in
the list. The list behaves as a stream of elements and we automatically get stream
communication between threads. This behavior is used in Chapter 6.

5.8.4 New control structures (with delayed evaluation)

Since we have been programming with binary trees, let us define a program that
checks if is a data structure is actually a binary tree. The function BinaryTree
shown in Figure 5.19 checks a structure to verify whether it is a binary tree or
not, and accordingly returns true or false. This program defines the auxiliary
local procedure And.

Consider the call {And {BinaryTree T1} {BinaryTree T2} B}. It can
sometimes do unnecessary work. According to our nesting rules, it evaluates its
second argument even if the first is false. We can fix this problem by making
a new function AndThen that takes as its first two arguments two functions. It
does not call the second function if the first returns false. This evaluates the
argument only if it is really needed. Figure 5.20 shows the binary tree checker
written with AndThen.

The ability to do delayed evaluation in And is so useful that we provide it as
a primitive operation, called andthen. Figure 5.21 shows the binary tree checker
written with andthen. In a complementary way to andthen, there also exists
an operation orelse that evaluates its second argument only if the first returns
false.

5.8.5 Lazy execution with programmed triggers

Modern functional languages have a flow control mechanism called lazy evaluation
or lazy execution. Here we show how to program lazy execution explicitly with
local
  fun {AndThen BP1 BP2}
    if {BP1} then {BP2} else false end
  end
in
  fun {BinaryTree T}
    case T
    of leaf then true
      [] tree(K V T1 T2) then
        {AndThen
          fun {$_} {BinaryTree T1} end
          fun {$_} {BinaryTree T2} end
        } else false end
    end
  end
end

Figure 5.20: Checking a binary tree (with delayed evaluation)

fun {BinaryTree T}
  case T of leaf then true
    [] tree(K V T1 T2) then
      (BinaryTree T1) andthen (BinaryTree T2)
    else false end
end

Figure 5.21: Checking a binary tree lazily (using andthen)
higher-order programming. This is a powerful programming technique, but it is cumbersome to program explicitly. Section 6.6 shows how to make lazy execution *internal*, i.e., where the mechanics of triggering the execution are handled by the system. As we shall see in Section 6.1, lazy execution is closely connected to concurrent execution. Internal lazy execution is usually preferable to programmed lazy execution because it is much easier to write programs in.

Lazy execution is an example of *demand-driven* execution. A data structure (such as a list) is constructed incrementally. The consumer of the list structure asks for new list elements when they are needed. This is an important technique to modularize programs. It is very different from the usual, *supply-driven* evaluation, where the list is completely calculated independent of whether the elements are needed or not.

To implement lazy evaluation, the consumer should have a mechanism to ask for new elements to be constructed when they are needed. We call such a mechanism a *trigger*. A simple way to implement a trigger is as a function. The consumer calls this function when it needs a new list element. The function call returns a *pair*: the list element and also a new function. The new function is the new trigger: calling it returns the next data item and another new function. And so forth.

- Give a code example of a programmed trigger.

### 5.8.6 Currying

Currying is a technique that sometimes simplifies programs that heavily use higher-order programming. The idea is to write functions of *n* arguments as *n* nested functions of one argument. For example, the maximum function:

```plaintext
fun {Max X Y}
  if X>=Y then X else Y end
end
```

is rewritten as follows:

```plaintext
fun {Max X}
  fun { Y}
    if X>=Y then X else Y end
  end
end
```

This keeps the same function body. It is called as \{Max 10 20\}, giving 20. The advantage of using currying is that the intermediate functions can be useful in themselves. For example, the function \{Max 10\} bounds its result to be never less than 10. We can give it the name *LowerBound10*:

```plaintext
declare
LowerBound10={Max 10}
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
In functional programming languages such as ML and Haskell, all functions are implicitly curried. In these languages, if the function call `max 10 20` is possible, then `max 10` is also possible.

To use currying to maximum advantage, it should have a simple syntax and an efficient implementation. Functional languages support both of these accommodations. They define the syntax so that curried functions can be written without parentheses. They use compilers that avoid constructing and calling the intermediate functions whenever possible.

The declarative computation model does not have any particular support for currying. The Mozart system has no syntactic support nor implementation support for it. However, currying is still possible. There are two basic ways to do it. The first is to define functions in nested fashion, as in the `Max` example. This is rather expensive, so we do not recommend it. The second is to define the intermediate functions explicitly, as they are needed. For example, the `LowerBound10` function can be defined as:

```
fun {LowerBound10 Y} {Max 10 Y} end
```

It is defined only if it is needed. We find that the second solution works very well in practice.

### 5.8.7 More higher-order programming

First property: Procedures are values.

Procedures can be passed as arguments to other procedures. This allows operations to be parameterized in sophisticated ways. For example, a sort operation can be parameterized according to a boolean function that checks the order of two elements.

Second property: Compositionality.

A procedure definition, being an executable and nestable statement, can be put anywhere that a statement or expression is allowed.

Third property: Visibility based on lexical scoping.

This allows separating an interface from an implementation, which allows building abstractions.

Examples:

- Simple higher order with `ForAll` and `Map`.
- Implementing control structures: `andthen`, `orelse`, `?`
- A general iterator based on $\text{Iter}$ procedure given earlier, with examples:

$$
\begin{align*}
\text{fun } \{ & \text{Iter State Transform IsDone} \} \\
& \text{if } \{ \text{IsDone State} \} \text{ then State} \\
& \text{else State1 in} \\
& \quad \text{State1=}\{ \text{Transform State} \} \\
& \quad \{ \text{Iter State1 Transform IsDone} \} \quad \% \text{Last call} \\
& \end{align*}
$$

Redo Newton example with the iterator:

$$
\begin{align*}
\text{fun } \{ \text{Sqrt X} \} & \\
& \{ \text{Iter} \\
& \quad 1.0 \\
& \quad \text{fun } \{ \$ \text{ Guess} \} \ (\text{Guess} + X/\text{Guess}) / 2.0 \ \text{end} \\
& \quad \text{fun } \{ \$ \text{ Guess} \} \ {\{ \text{Abs X-}\text{Guess*}\text{Guess}/X < 0.00001 \text{end} \}} \\
& \end{align*}
$$

- Tree insertion with order parameter
- Checking whether something is a tree
- $\text{Map Filter FoldL FoldR}$
- $\text{Sort with order parameter: can sort anything according}$
  to any order function.
- A procedure of infinite order.

## 5.9 User-defined data types

We explain how to define your own data types, both abstract and concrete.

- Compare with what exists in ML and Haskell

### 5.9.1 Concrete data types

- Define the concept of built-in versus user-defined data types. Essential
difference: the latter represent something else than themselves. Similar
to type constructors in ML, Haskell.
- Up to now, we have been quick in defining our user-defined data types
in terms of pre-existing ones (esp. records and lists).
- Example of built-in data type at different levels of abstraction:
  virtual string -> string -> list
- We have seen a few examples of user-defined types: FIFO queue,
dictionary, tree. Up to now, these are concrete, not abstract.
Section \ref{coreadt} shows how to make true ADTs that are encapsulated.

- More examples:
  - Polynomial: a record structure, maybe has same shape as list but is not a list (Calculate polynomial from points (Van Roy Solver)).
  - Generating fractals from simple grammar rules (generation, inverse of parsing).

**Principles of defining data types**

1. Fixed-length groups are records, variable-length groups are lists.
2. Avoid possibility of conflict. Put names that could conflict in different “name spaces”, e.g., different arguments of lists.
   Use names, if no print representation is needed.

**Documenting data types**

In a dynamically-typed language, types do not need to be declared at compilation time. But types are important as documentation and for readability. We strongly recommend that you document a program’s types. For now, we will show you how to give an informal specification as a program comment. This specification will turn out to be a legal program in the sense of relational programming (see Chapter 7) and constraint inferencing (see Chapter 14).

We explain the informal specification technique by means of an example. Let us define a “geometric shape” type that can be a rectangle, an ellipse, a right-angled triangle, or a polygon with any number of sides. This type is redundant because triangles and rectangles can be represented as polygons, but it makes computation easier. Squares and circles can be defined in terms of rectangles and ellipses, so it is reasonable to keep these more general shapes.

The new type is a subset of the Record data type. We represent a rectangle as the record \texttt{rectangle(S1 S2)}, where S1 and S2 are the lengths of its sides; an ellipse as the record \texttt{ellipse(S1 S2)}, where S1 and S2 are the ellipse’s two axes; a right-angled triangle as the record \texttt{rightTriangle(S1 S2)}, where S1 and S2 are the two perpendicular sides; and a polygon as the record \texttt{polygon(Vs)}, where Vs is the list of vertices, represented by pairs \texttt{X#Y} giving the coordinates of the vertex. All lengths and coordinates are floating point numbers. The new type is specified as follows:

```plaintext
fun \{Shape\}
    choice rectangle({Side} {Side})
    [] ellipse({Radius} {Radius})
    [] rightTriangle({Side} {Side})
    [] polygon({ListP Vertex})
end
end

fun \{Radius\} \{FloatP\} end
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
fun {Side} {FloatP} end
fun {Vertex} {FloatP}#{FloatP} end

Each type is defined as a nullary function. The choice statement is a disjunction, i.e., an “or” between alternative possibilities. The functions can be read either purely declaratively or as runnable programs. As runnable programs, they can check whether a value is correctly typed (see Section 7.3.5). For now, we will consider them purely as documentation. Here are the definitions of the basic types FloatP and ListP:

```plaintext
proc {FloatP X} {IsFloat X}=true end
fun {ListP T}
  choice nil
  [] {T}|{ListP T}
end
```

Metaprogramming with types

Using the dynamic record operations together with higher-order to inspect types and to create new types from existing ones. For example, replace or add attributes to trees or records.

5.9.2 Abstract data types

An abstract data type (ADT) is a set of values together with a set of operations on those values. The only possible operations are those defined as part of the abstract data type. To implement an abstract data type we need to be able to protect the implementation, so that only the predefined operations can access the type’s representation. This means that we need two things:

- A way of encapsulating values. The declarative model has two ways of encapsulating values: chunks and procedures.

- A way of creating new encapsulated values from previously-existing ones, without compromising the encapsulation. The declarative model has one way to do this: use names to access the old value in a noncompromising way, and then create the new value. Chapter 8 gives another way, namely by using encapsulated state. A procedure can modify the state inside the encapsulation, without compromising it.

ADTs are especially interesting when used together with explicit state. The most visible example of this is object-oriented programming, which is a widely-used technique for structuring programs. But there are other kinds of ADTs that use state. Chapter 8 explains all these ADTs in detail.

The queue and dictionary data types that we introduced before are completely unprotected. That is, a program can inspect the implementation of a data type,
and what’s worse, can construct a “fake” value of the type. We show how to protect the queue and dictionary so that this is not possible.

An abstract queue

We show how to convert the queue data type of Section 5.7.3 into an ADT. The definition of Section 5.7.3 is completely unprotected: any program can access the implementation and create “fake” values of the queue. To protect the queue, we use a chunk with one feature. If the feature is a name then it can only be accessed by those who know the name. Here is the new definition:

```plaintext
local
  Key={NewName}
in
  fun {NewQueue} X in
    {Chunk.new wrap(Key:q(0 X X))}
  end

  fun {Insert C X} q(N S X|E1)=C.Key
  in
    {Chunk.new wrap(Key:q(N+1 S E1))}
  end

proc {Delete C1 ?C2 ?X} q(N S E)=C1.Key
  H|S1=S
  in
    X=H
    C2={Chunk.new wrap(Key:q(N-1 S1 E))}
  end

  fun {EmptyQueue C} q(N S E)=C.Key
  in
    N==0 end
end
```

To see the inside of the chunk, it is necessary to know the name `Key`. The name is protected by lexical scoping. It is visible only inside the four operations of the queue.

An abstract dictionary

We show how to convert the declarative dictionary data types of Section 5.7.4 into an ADT. To protect a dictionary, we wrap it in a chunk with one feature, which is a name. Then unwrap the dictionary before accessing it and wrap it before outputting it. Here is the code:

```plaintext
Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
```
Because the name Key is known only inside the scope of the local, the wrapped dictionary cannot be unwrapped by anyone outside of this scope. This technique works for both the list and binary tree implementations.

It is easy to extend this technique for all data types. To make an ADT it’s enough to know where the unprotected data type is input and output. Inside the ADT, the unprotected version should be accessible. Outside the ADT, only the protected version should be accessible. At each input, “unwrap” the protected version. At each output, “wrap” to make a new protected version.

Read-only variables (the !! operation)

Sometimes it is useful for an ADT to have a variable as an output. But all the variables we have seen so far can be bound by anyone who has a reference to them. With ADTs this is too much freedom since it can break the abstraction. For the ADT to work correctly, it should be impossible for anyone outside the abstraction to bind the variable. Read-only variables are designed to achieve this. They are restricted versions of dataflow variables that can only be tested, not bound.

Read-only variables are actually read-only views of variables. That is, they come in pairs: one variable that can be bound and another is equal to the first but that can only be read. Sometimes read-only variables are called futures, by analogy with futures in MultiLisp, but they are not exactly the same thing (see Section 6.4.2). The operation:

\[ Y = !!X \]
creates a read-only variable \( Y \) for the variable \( X \). An attempt to use the value of \( Y \) will suspend until \( Y \) is bound. Binding \( X \) also binds the read-only variable \( Y \).

Read-only variables are also important because of security. This extends their use in abstract data types to more general management of protected information. For example, we may want to pass a variable to a routine, but not want the routine to bind the variable. To ensure this, we pass a read-only variable.

### 5.9.3 Declarative game boards

As a more substantial example, we define an ADT for a game board with the typical operations needed for games. The game board is represented as a value. We then show how to program the Game of Life with the game board. The minesweeper game introduced in Chapter 2 is programmed in Chapter 14 using the game board.

**Example: the Game of Life**

With the GameBoard component we can easily program Conway’s Game of Life.

- Define rules of Game of Life.
- Define the game as a function: it accepts a game board as input and returns a game board with the next game state.
- Show how it runs.

### 5.10 Security

It is very important to distinguish language security from implementation security. For example, names are easily made globally unique and unforgeable when restricted to language security. To extend this to implementation security, must accommodate tampering with internals of processes and with network. Especially with network: often the processes will be in trusted domains but the network will not be.

Add figure + text from NGC98 paper to give context.

Unforgeable globally unique constants are very useful when programming security within the language. Of course, the implementation must guarantee their properties. (Explain how.) This is not simple, but it can be done, and the implementation is efficient. In particular, no more network messages are needed than if they were just regular (insecure) constants. (Explain why.)
We say a computation is secure if it has well-defined and controllable properties, independent of the existence of other (possibly malicious) entities (either computations or humans) in the system. We call these entities “adversaries”. Because of the Internet, which encourages sharing of network resources among many entities, security is becoming more and more important in computing. Security is not just a property of the computer system. It is a global property of any organization that includes the human beings part of it, its physical housing, and all parts of it including its computer systems. This book shows how to program secure computations; the rest of the organization is outside of its scope. We refer interested readers to [?] for more information on the aspects of security not treated in this book.

A security weakness in any part of the organization may be enough to compromise the security of a computation. For a computer to be secure, many conditions must be fulfilled: its hardware must be secure, its operating system must be secure, its programming language and language implementation must be secure, and its applications must be written in a secure manner. Securing all these levels is a tall order. Perfect security does not exist in practice. In most cases, compromises are made, to get security that is “good enough”. For example, we can assume that putting the computer inside a building with card key access is enough to guarantee hardware security.

In this section, we will talk about language security and application security. We will present the language properties that are necessary to build secure applications, and we will show how to build secure applications. An adversary staying within the language will not be able to break our techniques. However, there is a major caveat to our presentation: we do not discuss implementation security. That is, an adversary can succeed if he is willing to dig in to the implementation, e.g., examine the representation used by the implementation by reading a process image or file formats as a sequence of bits.

A secure language allows to write programs that insulate computations and data structures from undesired interactions with other computations. Undesired interactions can come both from malicious computations and from innocent (but erroneous) computations. In the latter, these interactions are usually called “bugs”. We see that security allows to protect computations both from adversaries and from bugs in other computations. That is, even if there are no malicious computations, security is useful.

Section 5.10.1 defines the basic concepts of security, including the basic four properties of integrity, authentication, confidentiality, and guaranteed service. Section 5.10.2 introduces the language concepts used to implement the four basic security properties. Section 5.10.3 shows how to implement the three of these properties that are relevant for declarative programs. This section introduces the concept of encapsulation, which is the basic mechanism for implementing secure programs. Section 5.10.4 introduces capabilities, which are a powerful technique of which the previous section has given examples. Section 5.10.5 shows how to program secure applications using the language concepts already introduced. Sec-
5.10 Security

5.10.1 Basic concepts

There are four basic security properties:

- **Integrity.** This is the guarantee that an adversary cannot corrupt the data given to it by a secure computation.

- **Authentication.** This is the guarantee that the data came from a given secure computation. An adversary cannot forge data to make a computation believe that it came from a given source.

- **Confidentiality.** This is the guarantee that an adversary cannot read the data meaningfully. A computation cannot read the data unless the secure computation has given it the right to do so.

- **Guaranteed service.** This is the guarantee that a secure service will provide its service within a given time period. An adversary can not interfere with the service, e.g., by flooding the network with irrelevant messages.

Guaranteed service is not addressed in this chapter. For sequential declarative programs it has no meaning. We will address guaranteed service later, when we introduce concurrent and distributed programming.

5.10.2 Language properties

What properties must a language possess in order to build secure programs by simply using the language, without any external support? This question is at the heart of E, a programming language and system for writing secure applications [66, 88]. The model we give has much in common with E. We begin by presenting two basic properties that must always hold:

- **Unforgeable references.** It must not be possible for a computation to “forge” a reference, i.e., create a reference that behaves exactly like a given reference \(X\) even though it has not been given \(X\). This implies among other things that unrestricted type casts should not be possible. Unforgeability of references is a prerequisite to unforgeability of certain values in the language (e.g., names), but can allow other values to be forgeable (e.g., numbers). A number is forgeable because any computation can create the number without being passed it explicitly. A name is unforgeable because this is not possible.
<table>
<thead>
<tr>
<th>Type</th>
<th>Possible operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Records</td>
<td>structure equality, field extraction, many other operations ...</td>
</tr>
<tr>
<td>Names</td>
<td>name equality</td>
</tr>
<tr>
<td>Procedures</td>
<td>name equality, invocation</td>
</tr>
<tr>
<td>Chunks</td>
<td>name equality, field extraction</td>
</tr>
<tr>
<td>Read-only variable</td>
<td>read</td>
</tr>
</tbody>
</table>

Table 5.4: Types relevant to security

- **Controlled visibility.** A computation should only get access to a reference by a simple language mechanism. This mechanism should allow computations to prevent references from being known by other computations.

Controlled visibility is provided by lexical scoping and the “inheritance property”. Lexical scoping guarantees that a reference is initially visible in a scope that is known from visual inspection of the program text. Section 4.4.2 defines this precisely. The *inheritance property* states that a computation can pass references to other computations, and that a computation can obtain references in only two ways: by creating them through lexically-scoped declarations or by being passed them from another computation.

These properties are not enough for security, because they do not sufficiently delimit what an adversary can do with information he collects from a computation. For example, an adversary can receive the record `person(name: "George")`, examine this record, and create a new record `person(name: "Bill")`. It is not possible to determine, without further mechanisms, whether the new record came from the original computation or not. That is, authentication is impossible.

To implement these three concepts on complete values, we now show that it is sufficient to have immutable data types, first-class procedures with lexical scoping, names, and chunks. For dataflow variables, read-only views are needed as well, as we will see in Chapter 13. Table 5.4 summarizes the types that are particularly relevant to security. The common characteristic of all (except records) is that they have only a *limited* set of operations. For example, names are limited versions of atoms and chunks are limited versions of records. Unlike atoms, names have no print representation and no conversion operations to and from a string. Unlike records, chunks have no label and arity operations and unification with them is not allowed.

### 5.10.3 Implementing the basic concepts

For each of the four security properties defined in Section 5.10.1, we show how to express them in the language:
5.10 Security

- **Integrity.** A sufficient condition is that values be immutable, i.e., that once created they cannot be changed. This condition is satisfied by all values in the language. This condition is relevant to declarative programs because they manipulate only values. Declarative computations can therefore satisfy the integrity property. For example, assume a computation creates the record \( X = \text{person}(\text{name: "George"}) \) and passes \( X \) both to a friendly computation and to an adversary. The computation can assume that the adversary cannot make the friendly computation read the data as something else.

- **Confidentiality.** A data structure must be “wrappable” in such a way that only a properly authorized computation can remove the wrapper and read the data. Wrapping the data structure in a record is not good enough, since it can be extracted by any computation. For example, \( X \) can be put in a new record \( \text{foo}(X) \), but any computation can get the record’s arity and extract \( X \). This is also true if a name is used as a feature, e.g., \( \text{foo}({\text{NewName}}:X) \), since the arity operation will return the name. Perhaps it is good enough to have a limited form of record, for which no arity operation is possible? This intuition is correct, as we will see below when using chunks.

- **Authentication.** A data structure must be signable by its creator, in such a way that the signature is usable only for that data structure. For example, to sign \( \text{person}(\text{name: "George"}) \), it is not enough to add a field \( \text{owner} \), giving \( \text{person}(\text{name: "George"} \text{ owner:X}) \) for some \( X \) identifying the owner. An adversary could extract \( X \) and use it in a new record, e.g., \( \text{person}(\text{name: "Bill"} \text{ owner:X}) \) that was never created by the original computation. Perhaps a name can be used as a signature, if it is hidden when the signature is checked? This intuition leads to a good solution, as explained below.

- **Guaranteed service.** This is a matter of managing the computational resources (time and memory).

  This is related to thread priorities, cooperative, and competitive concurrency. It is related to all resources, not just computational ones. It can only be explained properly after we have introduced concurrency and resources.

The name and chunk data types are used to implement confidentiality and authentication. A name is an unforgeable constant, as explained in Section B.2. A chunk is a record with a limited set of operations, as explained in Section B.4.

To implement confidentiality, the data can be wrapped in a chunk and accessed with a name. That is, to make \( X \) confidential, do the following:
\begin{verbatim}
declar\ Key={NewName}
Xwrap={Chunk.new foo(Key:X)}
\end{verbatim}

This creates the chunk \texttt{Xwrap}, which contains \texttt{X}. To access \texttt{X}, you have to know \texttt{Key}:

\begin{verbatim}
declar\ X=Xwrap.Key
\end{verbatim}

Since \texttt{Key} is unforgeable, only computations that have been given it can access \texttt{X}. We call these \textit{authorized} computations.

Note that the above code is not sufficient to implement authentication. Indeed, any computation that knows \texttt{Key} can create a chunk in the same way that the authorized computation does it.

To implement authentication, the key should be hidden from all but the original computation, but still usable for doing operations. In this way, anyone can check whether the data structure is correctly signed, without being able to use the key for other purposes. Here is one way to implement this idea. The procedure \texttt{NewWrapper} creates pairs of functions \texttt{Wrap/Unwrap}. \texttt{Wrap} hides the data in a chunk and \texttt{Unwrap} extracts it from the chunk. For each new pair, an internal key is created. Lexical scoping hides the key from all but the pair. Here is the definition of \texttt{NewWrapper}:

\begin{verbatim}
proc {NewWrapper ?Wrap ?Unwrap}
    Key={NewName}
    in
        fun {Wrap X}
            {Chunk.new foo(Key:X)}
        end
        fun {Unwrap C}
            C.Key
        end
    end
\end{verbatim}

\texttt{Unwrap} will only correctly extract data wrapped by \texttt{Wrap}. It raises an exception if given any other input.

If you just want to check whether the data has been created by a given computation, but do not want to actually access the data, then the following boolean function will do the trick. The function is a variant of \texttt{Unwrap} and should be defined inside \texttt{NewWrapper} alongside \texttt{Unwrap}:

\begin{verbatim}
fun {TestOwner C}
    try
        _=C.Key
        true
    catch _ then
        false
    end
end
\end{verbatim}

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
This function returns \texttt{true} if \( C \) is created by \texttt{Wrap}, and \texttt{false} otherwise. This allows to do authentication without compromising confidentiality.

### 5.10.4 Capabilities

A \textit{capability} is an unforgeable data structure that gives its owner the irrevocable right to perform some action. The functions \texttt{Wrap}, \texttt{Unwrap}, and \texttt{TestOwner} are all capabilities. In the declarative model, capabilities can be represented by procedures. The procedure cannot be forged and it gives its owner the right to \textit{call} it, i.e., to perform whatever it was designed for. The procedure can hide internally all the information it needs to perform these actions, without the caller being able to access them. Indeed, there is no operation that lets one access the external references of procedures, just like there is no arity operation on chunks.

### 5.10.5 Programming secure applications

The two-level view: mechanism vs. policy.

We explain how the two views and four properties are supported by the language.

Open vs. closed application. This is an application property.
Program developer decides it; system supports whatever he decides.
It’s a matter of degree.
Completely open = literally any computation can ‘‘join’’.
Completely closed = no computation can join.
Usual case: some way of controlling who can join.
Security means control should be what you want + after computation joins, interaction should be what you want.
For example, in a completely closed system, security has two roles: prevent anyone from joining and protect against bugs in independent parts of the application.

\subsection{Basic language support}

Language concepts:
- visibility based on lexical scoping: allows
- unforgeability of references
- unforgeability of names & chunks & procedures, etc.
- there exists a certain number of 'chunk-based' data structures:
  proc, fun, functor, (more later) ...
- forgeability of certain, ‘‘open’’ data structures:
  records, numbers, lists, etc.
Numbers must be creatable by anyone. Trade-off efficiency/security:
than 'integrity' security (i.e., no 'hiding' security).

Note: abstract store, procedures, and lexical scoping already provide mechanisms for security. But they do not allow \(\text{em separating}\) the data from the operations while maintaining security. Procedures encapsulate both the data and the operations, and themselves play the role of the key as well as the data as well as the operation.

Separating data, key, operation, allows any to be stored and passed around independently of the others. It is lightweight, e.g., the key is just a small constant (a ‘name’). In order to do an operation, you need the data and the operation of course, but also the key. Without the key nothing is possible.

Language data types:
- chunks: secure wrappers, for encapsulation. Basically a record with limited operations (no arity) -- one cannot ‘guess’ the fields. The label is a name -- chunk cannot be forged.
- names: ‘key’ to open the wrapper
Names can be passed around the program in a dynamic way completely devised by programmer -- very flexible mechanism that can be used to implement many policies
- read-only variable: secure dataflow variable
- procedures (with name equality): provide default way to do secure encapsulation where the data and its operation are always together. With chunks, data is separate from operation, whichever operation has the key and the data can do the operation.

To be explained later:
- threads & processes: threads give 'cooperative concurrency', i.e., no quality of service guarantee. Processes can support 'competitive concurrency', i.e., with a quality of service guarantee.
- network behavior: in current release, no support for any of the security properties. Later releases will add such support.
Role of cryptography: will need crypto for this.

\subsection{Unencapsulated data types}

What security properties do standard record- and list-based data types, as used in Section~\ref{userdefinedtypes}, give us? Essentially, they give us \(\text{em integrity}\), because they cannot be changed.

To be precise, complete values cannot be altered,
and partial values that may contain read-only variables but no variables cannot be altered.

The standard user-defined types are not protected, i.e., any computation that references a data structure can read it, and not authenticated, i.e., we have no guarantee as to who is the creator of any particular data structure.

\subsection{Encapsulated data types}

The goal of encapsulation is to add a "wrapper" around a data structure, in such a way that only "authorized" computations can look inside. There exist two techniques for doing encapsulation: using procedural abstraction (wrapper based on lexical scoping) or using chunks and names (wrapper with unforgeable keys). Using procedures means the operations and the data are always together. This may be less efficient (more memory space needed). Using chunks allows the data to exist separately from the "key" needed to look inside. No procedures are needed. This uses less memory. Using chunks allows a computation to possess the data without being able to read inside. Only when the name (the "key", in the physical sense) is passed to the computation, can it read the data.

\subsection{Programming with modules and functors}

Functor: defines a new module by specifying how to create the module and by specifying the other modules needed.

Functors and modules are very lightweight -- can create many (in fact, of similar weight to classes and objects, so do not hesitate to use them if you need them).

\subsection{Resources}

Resources are modules that can only be used in a particular process. A process’s resources can be protected by that process from unauthorized access. This provides encapsulation at the process level. In general, only open, distributed programs

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
can take advantage of process-level encapsulation. This is explained in Chapter~\ref{distchapter}.

\section{Relation to other declarative models}

At this point we have seen the most important programming techniques for the declarative computation model. We said that the declarative model of this chapter embraces both strict functional programming and deterministic logic programming. Let us now examine how and under what conditions it does so.

This section comes with a caveat: to understand well what it says requires some understanding of logic and functional programming.

\subsection{Relation to logic programming}

A logic program is a statement of logic that is given an operational semantics, i.e., it is executed. If the operational semantics is well-designed, then the execution has two properties: it respects the logical semantics and it allows to write efficient programs. One of the simplest ways to define logic programs is to use first-order predicate logic as the underlying logic. Then a program is simply a set of predicate definitions. The logical semantics is given by a model of these definitions, where the term “model” is used in the logical sense. We explain what this means. For each predicate, we give a set of tuples for which it is true. That is, each predicate defines a relation. If all the relations taken together satisfy the predicate definitions, then we say that they form a model of the definitions.

Many programs in the declarative model of Chapter 4 have a logical semantics as well as an operational semantics. It is straightforward to translate a declarative program into a logical formula. If the program terminates correctly, i.e., it does not suspend, go into an infinite loop, or raise an exception, then its result will be in accord with the logical formula. That is, the results of all predicates are valid tuples in the predicates’ relations. We call this deterministic logic programming.

Table 5.5 defines the translation scheme $T$, which translates a declarative statement $(S)$ into a logical formula $T((S))$. All statements of the declarative kernel language of Section 4.2 are translated, except for exception handling (\texttt{try} and \texttt{raise}). Procedure definitions are translated into predicate definitions. Exceptions are a special case. Raising an exception signals that the normal, logical execution is no longer valid. The logical formula therefore does not hold.

It is important to keep the logical and operational semantics clearly separate. For example, the following three statements:

1. $x=y\ (S)$
2. $(S)\ x=y$
3. $\text{if } x==y\ \text{then } (S)\ \text{else fail end}$
### Table 5.5: Translating a declarative statement to logic

<table>
<thead>
<tr>
<th>Statement</th>
<th>Logical formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>true</td>
</tr>
<tr>
<td>fail</td>
<td>false</td>
</tr>
<tr>
<td>( &lt;S1&gt; \langle S \rangle ) end</td>
<td>( T(&lt;S1&gt;) \land T(&lt;S2&gt;) )</td>
</tr>
<tr>
<td>local ( x ) in ( &lt;S&gt; ) end</td>
<td>( \exists x.T(&lt;S&gt;) )</td>
</tr>
<tr>
<td>( X=Y )</td>
<td>( x=y )</td>
</tr>
<tr>
<td>( X=f(l1:X1 ... ln:Xn) )</td>
<td>( x=f(l1:x_1, ..., l_n:x_n) )</td>
</tr>
<tr>
<td>if ( X==Y ) then ( &lt;S1&gt; ) else ( &lt;S2&gt; ) end</td>
<td>( (x = y \land T(&lt;S1&gt;)) \lor (x \neq y \land T(&lt;S2&gt;)) )</td>
</tr>
<tr>
<td>case ( X ) of ( f(l1:X1 ... ln:Xn) ) then ( &lt;S1&gt; ) else ( &lt;S2&gt; ) end</td>
<td>( \exists x_1, ..., x_n.x = f(l1:x_1, ..., l_n:x_n) \land T(&lt;S1&gt;) ) \lor ( \neg \exists x_1, ..., x_n.x = f(l1:x_1, ..., l_n:x_n) \land T(&lt;S2&gt;) )</td>
</tr>
<tr>
<td>proc ( {P \ X1 ... Xn} \langle S \rangle ) end</td>
<td>( \forall x_1, ..., x_n.p(x_1, ..., x_n) \leftrightarrow T(&lt;S&gt;) \quad \text{(define p)} )</td>
</tr>
<tr>
<td>( {P Y1 ... Yn} )</td>
<td>( T(&lt;S&gt;)[y_1/x_1, ..., y_n/x_n] ) (where ( T(&lt;S&gt;) ) defines p)</td>
</tr>
</tbody>
</table>

all have the exactly same logical semantics, namely:

\[ x = y \land T(<S>) \]

But their operational semantics is very different! The first statement binds \( X \) and \( Y \) and then executes \( <S> \). The second statement executes \( <S> \) and then binds \( X \) and \( Y \). The third statement waits until it can determine whether or not \( X \) and \( Y \) are equal. It then executes \( <S> \), if it determines that they are equal.

Since functions are shorthand notation for procedures, they are translated as procedures. For example, the Append function:

```haskell
fun \{Append A B\}
  case A
    of nil then B
    [\] X|As then X|{Append As B}
  end
end
```

is shorthand for the following procedure:

```haskell
proc \{Append A B C\}
  case A
    of nil then C=B
    [\] X|As then Cs in
      C=X|Cs
      {Append As B Cs}
  end
end
```

It has the following logical semantics:

\[
\forall a, b, c.\ append(a, b, c) \leftrightarrow (a = \text{nil} \land c = b) \lor (\exists x, a', c'.a = x \mid a' \land c = x \mid c' \land append(a', b, c'))
\]
The call:

{Append [1 2 3] [4 5] X}

is translated to the relation tuple \( \text{append}([1,2,3],[4,5],x) \). The call executes successfully and returns \( X=[1\ 2\ 3\ 4\ 5] \). The tuple then becomes \( \text{append}([1,2,3],[4,5],[1,2,3,4,5]) \).

We saw that there are three ways that a procedure can be incorrect: it can suspend, loop indefinitely, or raise an exception. Let us look closer at suspension. It means that the program did not have enough information to continue. There are two possibilities:

- **Programmer bug.** That is, it was not the programmer’s intention that the program suspend. For example, the programmer expects the call \{Append X [3] [1 2 3]\} to return \( X=[1\ 2] \), but the program cannot because it assumes \( X \) is bound to a value on input. This kind of problem can be solved by changing the program. To calculate \( X \) from the last two arguments, the definition of Append must be changed:

  \[
  \text{proc} \{\text{Append} \ A \ B \ C\} \\
  \quad \text{if} \ B==C \text{ then} \ A=\text{nil} \\
  \quad \text{else} \\
  \quad \quad \text{case} \ C \text{ of} \ X|Cs \text{ then} \ As \text{ in} \\
  \quad \quad \quad A=X|As \\
  \quad \quad \quad \{\text{Append} As \ B \ Cs\} \\
  \quad \quad \text{end} \\
  \quad \text{end} \\
  \]

  This version of Append expects its last two arguments to be inputs and its first argument to be an output. It has a different operational semantics, but the same logical semantics as the previous version. To be precise, its logical semantics according to Table 5.5 is:

  \[
  \forall a, b, c. \text{append}(a,b,c) \leftrightarrow \\
  (b = c \land a = \text{nil}) \lor (\exists x, c', a'.c = x \mid c' \land a = x \mid a' \land \text{append}(a',b,c'))
  \]

  This sentence is logically equivalent to the previous one.

- **Incomplete execution.** That is, the programmer intended the program to choose between alternatives in a disjunction. Since this kind of choice is not possible in the declarative computation model, execution simply suspends. For example, the call \{Append X Y [1 2 3]\} suspends because there are four logical solutions. It is possible to write an Append that calculates just one of these solutions, e.g., \( X=[1] \) and \( Y=[2\ 3] \). But often we do not know which solution is the right one. We would like the program to find the right one. The process of trying multiple solutions until the right one is found is called search. In the context of logic programming, we call it nondeterministic logic programming. There are many ways to do
search. For example, it can be programmed explicitly, by calculating a list of solutions. Another way to add search is to use relational programming, which has a \textbf{choice} statement for search. Section 7.3.1 shows how to do this for \texttt{Append}. A more general way is to use constraint programming (see Chapter 14).

In conclusion, if our declarative program corresponds to a logical formula, then we can use logic to reason about its correctness. We need to reason \textit{operationally} only for termination and efficiency. If we would like our program to \textit{choose} between alternatives, i.e., to do search, then we need to use either relational programming or constraint programming.

\subsection*{5.11.2 Relation to functional programming}

A \textit{functional program} is a set of function definitions. Executing the program consists of evaluating the functions. A declarative procedure \{\texttt{P X1 ... Xn}\} can be considered as a function definition because the procedure always has a fixed set of \textit{input} and \textit{output} arguments. When calling the procedure, the inputs are values and the outputs are different unbound variables. The procedure binds the outputs to its results. Which arguments are inputs and which are outputs depends on the procedure definition. The inputs are those variables used as tests in \texttt{if} and \texttt{case} statements. The outputs are usually bound to values.

For simplicity, assume that \texttt{X1, ..., Xi} are inputs and \texttt{Xj, ..., Xn} are outputs, where \(j = i + 1\). Then the procedure \texttt{P} is a function taking \(x_1, ..., x_i\) and returning \(x_j, ..., x_n\). For example, the \{\texttt{Append A B C}\} procedure in the previous section is a function from a list pair \((a, b)\) to a list \(c\). We call this \textit{strict} functional programming, since all the inputs are evaluated before the function is called. The \texttt{Append} procedure can be defined in a completely functional way, as follows:

\begin{verbatim}
fun {Append A B}
  case A
  of nil then B
   [] _|_ then A.1|{Append A.2 B}
  end
end
\end{verbatim}

This is equivalent to the first definition in the previous section. It has the following functional semantics:

\[
\begin{align*}
\text{append}(a,b) &= \\
\text{if } a = \text{nil} \text{ then } b \\
\text{else if } a = _|_ \text{ then } a.1|\text{append}(a.2,b)
\end{align*}
\]

We assume that the list pairing constructor \(|\) has two arguments, numbered 1 and 2 (see Section B.5).

Any procedure execution that completes without raising an exception corresponds to a successful execution of a strict functional program. If an exception is
raised then the arguments are outside the domain of the function. If the execution
suspends, then the following problem has occurred:

- **Programmer bug.** An input variable is mistakenly used as an output. That is, the procedure expects it to have a value when it does not. For example, the programmer calls \{Append X [2 3] [1 2 3]\} and expects X to be an output, when according to the definition it is an input. One way to solve this problem is by changing the definition of Append so that X is indeed an output. The same change can be done as in the previous section. This gives the following definition (the same as before, but written as a function):

```plaintext
fun \{EndAppend B C\}
    if B==C then nil
    else
        case C of _|_ then
            C.1|\{EndAppend B C.2\}
        end
    end
end
```

This definition has the following functional semantics:

```
endAppend(b, c) =
    if b = c then nil
    else if c = _|_ then c.1|endAppend(b, c.2)
```

Because functional calculations are directional (clear distinction between inputs and outputs), the equivalence of the variants \(c = append(a, b)\) and \(a = endAppend(b, c)\) is less easy to see compared to the analogous variants in the logic case.

Using the above ideas, we can reason about mathematical functions to reason about many declarative programs. We can use any of the techniques used to reason about functions, from elementary algebra to denotational semantics.

The version of functional programming explained here is called *strict* because a function’s arguments are always evaluated before calling the function. There is a more general version of functional programming in which the arguments are not evaluated before calling the function. They are evaluated only if needed. This is called *lazy* functional programming. Lazy evaluation is a powerful flow control technique in functional programming [45]. It allows to program with potentially infinite data structures without giving loop bounds. Chapter 6 explains lazy execution, which is a general version of lazy evaluation for a concurrent language, and shows how to use it.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
5.11.3 Discussion

It is clear that there is a strong connection between deterministic logic programming and strict functional programming, as we define them here. We can make the following two identities:

- Declarative model = deterministic logic programming + higher-order procedures.
- Declarative model = strict functional programming + dataflow variables.

The two paradigms coincide for declarative programs consisting only of zero-order functions on complete values. A zero-order function is one whose arguments are never procedure values.

5.12 Nondeclarative needs

- Use a complete example that interacts with the real world.
  For example: a program that reads an XML file and parses it into a record, does something to the record, and then does the reverse.

Declarative programming, because of its “pure functional” view of programming, is somewhat detached from the real world, in which entities have memories (state) and can evolve independently and proactively (concurrency). To connect a declarative program to the real world, some nondeclarative operations are needed. This section talks about four classes of such operations: exceptions, standalone compilation, interactive user interfaces, and file input/output. Exceptions are a mechanism to manage what happens when a component is used outside its specification. Standalone compilation is used to build applications that do not depend on the interactive interface. Interactive user interfaces and file input/output indirectly introduce state into the calculation, since they usually read the current state of a stateful entity (a file or a keyboard).

It turns out that the operations given in this section fit into more general computation models than the declarative one, in particular stateful and concurrent models. We will see that the “nondeclarative operations” of this section are just operations in these more general models. Some of these operations manipulate state that is external to the program; this is just a special case of the system decomposition principle explained in Section 8.9.2. In a general sense, this section ties in with the discussion on limits of declarative programming in Section 5.14.

5.12.1 Programming with exceptions

A routine will usually behave correctly, i.e., according to its specification. But in spite of all precautions, this is almost impossible to guarantee. There are two main ways that a routine can deviate from its specification:
Because it itself calls another routine outside of its specification. When one routine calls another, it’s possible for the caller to use the other in ways that were not intended. For example, what happens if a function is called outside its domain? Since a function’s domain can have a complicated structure (a real function can have poles, for example), this is not always obvious.

Because it relies on a particular behavior of the external world which may not always happen. For example, when doing file input/output, what do we do if the file does not exist?

In general, there are just too many possible things that can go wrong. We would like to write our program so that it is simple in the common case that nothing goes wrong (no particular precautions needed), but that it has the appropriate reaction when something goes wrong. This is an example of the principle of separation of concerns. As we will see, exceptions are a mechanism to help us write our program to satisfy this requirement.

It is important to be precise with terminology. We say a routine succeeds if it terminates its execution and its results satisfy its specification. In the other case, we say the routine fails. When a routine executes, it will either succeed or fail. In general, we prefer for a failing routine to signal the failure by raising an exception. An exception is a run-time event within the computation model that can be signaled and caught by routines. Exceptions add a semblance of order to the messy area of faulty behavior. They allow routines to detect wrong behavior and take the necessary action. A routine can fail in the following ways:

- It terminates with results that do not satisfy the specification. This means the routine is written incorrectly, i.e., is “buggy”. There are four ways to proceed:
  - The preferable way is to rewrite the routine so that it does not behave incorrectly. That is, to “fix the bug”. If this cannot be done, then go on to the next case.
  - If this is not possible, then the routine should be rewritten to raise an exception when its result could be incorrect. This gives the routine a predictable behavior that can be used by the caller.
  - Another possibility is to “enlarge” the specification to take into account the peculiar behavior, so that it becomes part of the specification. The “bug” becomes a feature. For example, take the case of a square root function that returns a floating point number. We can define the output to be either ans(X) when X is the real square root of a nonnegative number, or err(Reason) in the case of an erroneous call, where Reason is the reason for the error. This approach can be reasonable in some special cases (such as floating point calculations).
  - Finally, as a last resort, the caller should always check to make sure the results are what it expects.
The last two approaches are both laborious because extra operations have to be done with the routine’s results on each call: either examine their structure or test their validity. This does not satisfy separation of concerns.

- It does not terminate. Again, this means the routine was written incorrectly and should be corrected. The preferable solution is to eliminate this case by rewriting the routine. If this cannot be done then the caller should time the execution of the routine and terminate it forcibly if it suspects that it will not terminate. The caller can then cause the routine to signal an exception, as in the following case.

- It terminates abruptly, possibly leaving an inconsistent execution state, and signals an exception. In this case, the following approaches are possible:
  - If possible, it is preferable to rewrite the routine to fix the problem. This can be done by rewriting the routine to catch the exception internally, fixing the execution state, and retrying the calculation again, with the same or a different algorithm.
  - If this cannot be done, then it is preferable to leave a consistent execution state before signaling the exception. This can be done by rewriting the routine to catch the exception internally, fixing the execution state, and raising the exception again for the caller.
  - If even this cannot be done, then the caller should be conscious of the possible mess and be ready to handle the problem itself.

- If exceptions occur in a model, they actually make the program behave as if written in a higher model. Which?
  Exceptions can be ‘‘caught’’ in a component and hidden to higher-level components.
- declarative modulo constant external references.
  The computation does depend on external references, but during the computation(s) they are constant.
- Explain the two error-propagating approaches in the context of floating point and the IEEE standard. One approach: a number becomes a NaN, which propagates through the calculation. Other approach: exceptions.

5.12.2 Modules and standalone compilation

A module is a record that groups together related language entities (usually procedures, but anything is allowed including classes, objects, etc.). A module typically has a number of private procedures, i.e., that are not visible outside the module, and a number of interface procedures, i.e., that provide the external services.
of the module. Lexical scoping is used to hide the private procedures from the outside.

A module specification is called a functor, or sometimes loosely a software component. A functor specifies what other modules it needs (the import part), what module interface it provides (the export part), and its initialization code (the define part). Table 5.6 gives the syntax of functors as statements. Installing a functor creates a new module in two steps. First, the new module is linked to the other modules it needs. Second, the initialization code is executed. In this context, the set of already-linked modules is called the environment. The entity that handles the installing and that manages the environment is called a module manager. The System module Module has the ability to create multiple module managers within one process (although this ability is not often used).

A functor is a kind of software component. Like a component, a functor is a unit of independent deployment, a unit of third-party development, and has no persistent state (following the definition given in [91]). With this terminology, a module is a component instance; it is the result of installing a functor in a particular environment. The environment consists of a set of modules, each of which may have an execution state. Functors are used by Oz developers to exchange software. For example, the Mozart system has a library, called MOGUL (for Mozart Global User Library), in which third-party developers can put any kind of information. Usually, they put in functors and applications.

In addition to the three component properties mentioned above, functors in Oz are values just like procedures and classes. They share the following properties with procedures and classes:

- A functor definition can be evaluated at run time, giving a functor.
- A functor can have external references to other language entities. For example, it is easy to make a functor that contains data calculated at run time. This is useful, for example, to include large tables or image data in source form.

Table 5.6: Functors

7With the nesting notation they can also be used as expressions.
5.12 Nondeclarative needs

A functor can be stored in a file or copied between processes without changing its meaning in any way. This makes it easy to create libraries of third-party functors, such as MOGUL. It is a consequence of the network-transparent distribution of values (see Chapter 13).

A functor is lightweight; it can be used to encapsulate a single entity such as one object or class, in order to make explicit the modules needed by the entity.

Because functors are values, it is possible to manipulate them in sophisticated ways within the language. For example, a software component can be built that implements connection-oriented programming, in which components determine at run time which components they need and when to link them. Even more flexibility is possible when dynamic typing is used. A component can link any arbitrary functionality at run time, by installing any functors and calling them according to their needs.

Examples

Let us first see a real example of a module, and then we will declare a functor that specifies that module. In general a module is a record, and its interface is accessed through the record’s fields. We construct a module called MyList that provides interface procedures for appending, sorting and testing membership of lists. This can be written as follows:

```
declare MyList in
local
    proc {Append ... } ... end
    proc {MergeSort ...} ... end
    proc {Sort ... } ... {MergeSort ...} ... end
    proc {Member ...} ... end
in
    MyList=‘export´(append: Append
      sort: Sort
      member: Member
    ...)
end
```

The procedure MergeSort is inaccessible outside of the `local` statement. The other procedures cannot be accessed directly, but only through the fields of the MyList module, which is a record. For example, Append is accessible as MyList.append. Most of the library modules of Mozart, i.e., the Base and System modules, follow this structure. The MyList module can be created from the following functor:

```
functor
  export
    append: Append
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
sort:Sort
member:Member
...
define
proc {Append ... } ... end
proc {MergeSort ...} ... end
proc {Sort ... } ... {MergeSort ...} ... end
proc {Member ...} ... end
end

Note that the statement between `define` and `end` does implicit variable declaration, exactly like the statement between `local` and `in`. Assume that this functor has been compiled and stored in the file `/home/xxx/list.ozf` (we will see below how to compile a functor). Then the module can be created as follows in the interactive interface:

define {List} = {Module.link['/home/xxx/list.ozf']}

The function `Module.link` is defined in the System module `Module`. It takes a list of functors, links them together (i.e., evaluates them in the same environment), and returns a corresponding list of modules.

Functors can have `import` declarations. To import a library module it is enough to give the name of its functor. On the other hand, to import a user-defined module requires stating the URL of the file where the functor is stored. This is logical, since the system knows where the library modules are stored, but does not know where you have stored your own functors. The URL gives a convenient global address for files since it is widely supported through the World-Wide Web. Consider the following functor:

functor
import
    Browser
    FO at 'file:///home/mydir/FileOperations.ozf'
define
    {Browser.browse {FO.countLines '/etc/passwd'}}
end

The `import` declaration imports the System module `Browser` and the user-defined module `FO` specified by the functor stored in the file `/home/mydir/FileOperations.ozf`. When this functor is linked, the statement between `define ... end` is executed. This calls the function `FO.countLines`, which counts the number of lines in a given file, and then calls the procedure `Browser.browse` to display the result. This particular functor is defined for its `effect`, not for the module that it creates. It therefore does not export any interface.

Library modules

The library modules available in the Mozart system consist of Base modules and System modules. The Base modules are available immediately upon startup.
They are part of the language definition, providing basic operations on the language data types. The number, list, and record operations given in this chapter are in the Base modules. The System modules are not available immediately upon startup but are loaded dynamically on need. They provide additional important functionality such as distributed programming, logic and constraint programming, operating system access, and so forth.

The Mozart interactive interface can give a full list of the library modules in Mozart. In the interactive Oz menu, open the Compiler Panel and click on the Environment tab. This shows all the defined variables including the modules; the modules are red or purple.

**Standalone compilation**

A functor can be compiled in two ways: as a compiled functor (which is importable by other functors) or as a standalone program (which can be directly executed from the command line). Any functor can be compiled to make a standalone program. In that case, no export part is necessary and the initialization part defines the program’s effect. Given a file \texttt{x.oz} defining a functor, the compiled functor \texttt{x.ozf} is created by typing the following command from a shell interface:

\begin{verbatim}
  ozc -c x.oz
\end{verbatim}

The standalone executable \texttt{x} is created by typing the following:

\begin{verbatim}
  ozc -x x.oz
\end{verbatim}

Many other operations are possible together with compiling a functor. For example, it is possible to link all imported modules directly in the compiled program, so that no dynamic linking is needed.

**5.12.3 Text input/output with files**

A simple way to interface declarative programming with the real world is by using files. A \textit{file} is a sequence of values that is stored external to the program on a permanent storage medium such as a hard disk. A \textit{text file} is a sequence of characters. In this section, we show how to read and write text files. This is enough for using declarative programs in a practical way. The basic pattern of access is simple:

\begin{center}
  \text{Input file \textit{read} \rightarrow compute function \textit{write} \rightarrow output file}
\end{center}

Later on we will do more sophisticated file operations, but this is enough for now.

External to the Oz environment, files are named by strings of characters. From within Oz, a file name is specified as a virtual string.
Reading a file

The first step is to open the file. Then the file contents can be read. Finally, the file must be closed. Here is how to read all the contents in one step:

```
declare
  F={New Open.file init(name:`foo.txt`)}
  L={F read(list:$ size:all)}
  {F close}
```

This binds \( L \) to a string containing all the file's characters. In this code, the entity \( F \) is an object and \( \text{Open.file} \) is a class. Objects and classes are explained in Chapter 8. For now, we just show how to use them to do file operations. Objects are created with \( \text{New} \). Objects are called as \( \{ F \ \text{M} \} \) where \( M \) is a message passed to the object. Objects are not declarative since they have an internal state.

Files can also be referred to by URL (see Section ??). It is just as easy to read a file through its URL as through its file name:

```
F={New Open.file
    init(url:`http://www.mozart-oz.org/features.html`)}
```

That's all there is to it. Files can only be read, not written, through a URL. There are many other operations on files, but we will introduce them later. In particular, there are operations to read the contents of a file incrementally. This is useful if the file is extremely large.

Writing a file

The first step is to open file for writing. Then any number of virtual strings can be written to the file. Finally, the file must be closed. Here is how to write a file:

```
declare
  F={New Open.file init(name:`foo.txt` flags:`[write create]`)}
  {F write(vs:`This comes in the file
          `)}
  {F write(vs:`The result of 43*43 is `#43*43`
          `)}
  {F write(vs:`Strings are ok too`)}
  {F close}
```

Example execution

In Section 5.7.1 we defined the function \( \text{WordFreq} \) that calculates the word frequencies in a string. We can use this function to calculate word frequencies in a file:

```
declare
  F={New Open.file init(name:`book.raw`)}
  L={F read(list:$ size:all)}
  {F close}
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
D={WordFreq L}
% Write output file
G={New Open.file init(name:´word.freq´ flags:[create write])}
for X in (Domain D) do
    {G write(vs:{CondGet D X ˜1}#occurrences of word ´#X#´\n})
end
{G close}

If the file is very big, then the dictionary implementation of Section 5.7.1 will be very slow. Some timing figures are given there for different dictionary implementations.

5.12.4 Text input/output with interactive windows

The most interactive way to interface declarative programming with a human user is through a windowing interface. In this section we show how to input and output textual data to a window. Later on we will do more sophisticated graphical user interfaces, but this limited set of operations is enough for many declarative programs. We give a brief introduction to a subset of QTk, a toolkit that allows to build user interfaces by means of concise, mostly declarative specifications. QTk combines a declarative base together with the judicious use of objects and threads. Objects and threads are presented in later chapters; they are particularly nice for the windowing interface. We give the briefest possible introduction, just enough to use the toolkit. Chapter 12 gives a fuller discussion of declarative user interface programming, as exemplified by the QTk toolkit, and the principles underlying its design.

A window on the screen consists of a set of widgets. A widget is a rectangular area in the window that has a particular interactive behavior. For example, some widgets can display text or graphic information, other widgets can accept user interaction such as keyboard input and mouse clicks. Here are the widgets we will use now:

- The **label** widget can display a text. The widget is specified by the record:

  label(text:VS)

  where VS is a virtual string.

- The **text** widget is used to display and enter large quantities of text. With a vertical scrollbar, the widget is specified by the record:

  text(handle:H tdscrollbar:true)

  The handle h will be bound to an object used to control the widget. Consider h as a one-argument procedure: {h set(VS)} displays a text and {h get(VS)} reads the text.
The button widget creates a button and executes an action when the button is pressed. The widget is specified by the record:

\[
\text{button(text:VS action:P)}
\]

where VS is a virtual string and P is a zero-argument procedure. For each window, all its actions are executed sequentially.

The td (top-down) and lr (left-right) widgets can arrange other widgets in top-down or left-right order. The widgets are specified by the records:

\[
\text{lr(W1 W2 \ldots Wn)}
\]
\[
\text{td(W1 W2 \ldots Wn)}
\]

where W1, W2, ..., Wn are other widget specifications.

Each widget can have a glue feature that indicates whether the widget's borders should be glued to its surrounding enclosure or not. The glue feature's argument is an atom consisting of any combination of the four characters n (north), s (south), w (west), e (east), indicating for each direction whether the border should be glued or not. Here are some examples:

- **No glue.** The widget keeps its natural size and is centered in the space allotted to it, both horizontally and vertically.
- **glue:nswe** glues to all four borders, stretching to fit both horizontally and vertically.
- **glue:we** glues horizontally left and right, stretching to fit. Vertically, the widget is not stretched but centered in the space allotted to it.
- **glue:w** glues to the left edge and does not stretch.
- **glue:wns** glues vertically top and bottom, stretching to fit vertically, and glues to the left edge, not stretching horizontally.
The \texttt{QTk} module can take a record built out of these widgets and create the window. For example, here is a simple text I/O interface:

\begin{verbatim}
\scriptsize
\texttt{\textbf{declare}}
\texttt{In Out}
\texttt{D=td(title:"Simple text I/O interface"
    \texttt{td(lr(label(text:"Input:")
      \texttt{text(handle:In tdscrollbar:true glue:nswe)
      \texttt{glue:nswe})
    \texttt{lr(label(text:"Output:")
      \texttt{text(handle:Out tdscrollbar:true glue:nswe)
      \texttt{glue:nswe})
    \texttt{lr(button(text:"Do It" glue:nswe
      \texttt{action: proc \$\{} \texttt{Out set({In get(\$)})\}} end
    \texttt{button(text:"Quit" glue:nswe
      \texttt{action: toplevel#close) glue:we})
    \texttt{glue:nswe))
  \texttt{W={Build D}} % Build window from description
  \texttt{\{W show(wait:true)\}} % Show window, block until it closes
\end{verbatim}

At first glance, this may seem complicated, but look again: there are just six widgets (two label, two text, two button) arranged with \texttt{td} and \texttt{lr} widgets. Compare the record \texttt{D} with Figure 5.22 to see how they correspond. Look at the action of the “Do It” button: it gets text from the \texttt{In} handle and gives it to the \texttt{Out} handle.

The \texttt{Build} function takes the description \texttt{D} to build the window of Figure 5.22 and create the handler objects \texttt{In} and \texttt{Out}. The \texttt{toplevel#close} idiom means to close the window. Putting \texttt{nswe} glue almost everywhere allows the window to behave properly when resized. The \texttt{lr} widget with the buttons has \texttt{we} glue only, so that the buttons do not expand vertically. The \texttt{label} widgets have no glue, so they have fixed sizes.

\subsection{Stateless data input/output with files}

Input/output of a string is simple, since a string consists of characters that can be stored directly in a file. What about other values? It would be a great help to the programmer if it would be possible to save any value to a file and to load it back later. The System module \texttt{Pickle} provides exactly this ability. It can save and load any complete value:

\begin{verbatim}
\{Pickle.save X FN\} % Save X in file FN
\{Pickle.load FNURL X\} % Load X from file (or URL) FNURL
\end{verbatim}

All data structures used in declarative programming can be saved and loaded except for those containing unbound variables. For example, consider this program fragment:

\begin{verbatim}
\texttt{\textbf{declare}}
\texttt{fun \{Fact N\}}
\end{verbatim}
if \( N=0 \) then 1 else \( N \times \text{Fact } N-1 \) end 

\[
\begin{align*}
F100 &= \{\text{Fact } 100\} \\
F100Gen1 &= \text{fun } \{$} \ F100 \ \text{end} \\
F100Gen2 &= \text{fun } \{$} \ \{\text{Fact } 100\} \ \text{end} \\
FNGen1 &= \text{fun } \{$} \ N F=\{\text{Fact } N\} \ \text{fun } \{$} \ F \ \text{end} \ \text{end} \\
FNGen2 &= \text{fun } \{$} \ N \ \text{fun } \{$} \ \{\text{Fact } N\} \ \text{end} \ \text{end}
\end{align*}
\]

\( F100 \) is a (rather big) integer; the four other entities are functions. The following operation saves the four functions to a file:

\[
\{\text{Pickle.save } [F100Gen1 F100Gen2 FNGen1 FNGen2] \ \text{´} \text{factfile}´\}
\]

To be precise, this saves a value consisting of list of four elements in the file \texttt{factfile}. In this example, all elements are functions. The functions have been chosen to illustrate various degrees of delayed calculation. The first two return the result of calculating 100!.\(^8\) The first, \( F100Gen1 \), knows the integer and returns it directly, and the second, \( F100Gen2 \), calculates the value each time it is called. The third and fourth, when called with an integer argument \( n \), return a function that when itself called, returns \( n! \). The third, \( FNGen1 \), calculates \( n! \) when called, so the returned function just returns a known integer. The fourth, \( FNGen2 \), does no calculation but lets the returned function calculate \( n! \) when called.

To use the contents of \texttt{factfile}, it must first be loaded:

\[
declare
[F1 F2 F3 F4]=\{\text{Pickle.load } ´\text{factfile}´\}
\]

This displays 100! four times. Of course, the following is also possible:

\[
declare \ F1 \ F2 \ F3 \ F4 \ in
[F1 F2 F3 F4]=\{\text{Pickle.load } ´\text{factfile}´\}
\]

After the file is loaded, this displays exactly the same as before. This illustrates yet again how dataflow makes it possible to use a variable before binding it.

We emphasize that the loaded value is \emph{exactly} the same as the one that was saved. There is no difference at all between them. This is true for all possible

\(^8\)To be precise, the first two both return the integer 93 326 215 443 944 152 681 699 238 856 266 700 490 715 968 264 138 621 468 592 963 895 217 599 993 229 915 608 941 463 976 156 518 286 253 697 920 827 223 758 251 185 210 916 864 000 000 000 000 000 000 000 000.
values: numbers, records, procedures, chunks, names, atoms, lists, functors, and so on, including other values that we will see later on in the book. Executing this on one process:

... % First statement (defines X)
{Pickle.save X `myfile´}

and then this on a second process:

X={Pickle.load `myfile´}
... % Second statement (uses X)

is rigorously identical to executing the following on a third process:

... % First statement (defines X)
{Pickle.save X `myfile´}
_={Pickle.load `myfile´}
... % Second statement (uses X)

If the calls to Pickle are removed, like this:

... % First statement (defines X)
... % Second statement (uses X)

then there are two minor differences:

- The first case creates and reads the file `myfile`. The second case does not.
- The first case raises an exception if there was a problem in creating or reading the file.

### 5.13 More on efficiency

In Section 5.6 we introduced the concepts of time complexity and space complexity of programs. We showed how to calculate them, up to a constant factor. In this section we will go further into the topic of efficiency. We will talk about optimization, the art of improving the efficiency of a program, and memory management, the techniques for allocating and reclaiming program memory.

#### 5.13.1 Reflections on optimization

Ever since the beginning of the computer era in the 1940’s, both space and time have been becoming cheaper at an exponential rate (a constant factor improvement each year). They are currently very cheap, both in absolute terms and in perceived terms: a low-cost personal computer of the year 2000 typically has at least 64MB of random-access memory and 4 GB of persistent storage on disk, with a performance of several hundred million instructions per second, where each instruction can perform a full 64-bit operation including floating point. It is comparable to or faster than a Cray-1, the world’s fastest supercomputer in 1975 [83].

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Because of this situation, performance is usually not a critical issue. Given good techniques in program and algorithm design, such as are presented in this book, time and space can be disregarded for most parts of most applications. In other words, given a reasonable asymptotic complexity of a program, the constant factor is almost never critical. This is even true for most multimedia applications (which use video and audio) because of the excellent graphics libraries that exist.\footnote{The exception to this rule is games with realistic graphics, which by definition are always at the edge of what is possible.}

In some cases, performance is not sufficient. Rewriting a program to improve some characteristic is called optimizing it, although it is never “optimal” in any mathematical sense. Usually, the program can easily be improved up to a point, after which diminishing returns set in and the program rapidly becomes more complex for ever smaller improvements. Optimization should therefore not be done unless necessary. The bane of computing is premature optimization.

Optimization has a good side and a bad side. The good side is that the overall execution time of most applications is largely determined by a very small part of the program text. Therefore performance optimization, if necessary, can almost always be done by rewriting just this small part (sometimes a few lines suffice). The bad side is that it is usually not obvious, even to experienced programmers, where this part is \textit{a priori}. Therefore, this part should be identified \textit{after} the application is running and \textit{only if} a performance problem is noticed. If no such problem exists, then no performance optimization should be done. The best technique to identify the “hotspots” is \textit{profiling}, which instruments the application to measure its run-time characteristics.

Reducing a program’s space use is easier than reducing its execution time. The overall space use of a program depends on the data representation chosen. If space is a critical issue, then a good technique is to use a compression algorithm on the data when it is not part of an immediate computation. This trades time for space.

\subsection{Memory management}

From the viewpoint of memory use, a running program does two things: it creates new data structures when it needs them and it forgets old data structures when it no longer needs them. “Forgetting” a data structure means that it no longer has a reference to it. The total amount of memory available to the program is finite. At all times, this memory is separated into two parts: the active memory and the free memory. The \textit{active memory} is all the memory the program needs for the data structures that it is currently using. The \textit{free memory} is all the rest; it is not used by the program. When a program creates a data structure, it takes some of the free memory to store it. When a program forgets a data structure, then the data structure’s memory can be put back in the free list. This seems quite simple; yet it is not so! There are two principal difficulties:
When can a program “forget” a data structure? When it no longer needs it—that is, when it no longer has a reference to it. But programs can reference a data structure from many points, directly or indirectly through other data structures. How can we be sure that the data structure is really no longer needed?

Data structures are of all sizes. The active memory and free memory are not simple contiguous blocks; on the contrary, they are completely intertwined and fragmented. How can we create a data structure that is bigger than the biggest contiguous block in the free memory?

There are two fundamental approaches to memory management:

- **Manual memory management.** The application (i.e., the programmer) is responsible for knowing when a data structure is no longer needed. This is quite tricky. There are two possible errors. First, deciding a structure is no longer needed when it is still needed. This error leads to the situation called **dangling pointer**: all references to the data structure become invalid, but since they can still be used by the program they are “dangling”. Second, deciding a structure is still needed when it no longer is. If this lasts forever, then this situation gives rise to a **memory leak**: part of the memory has become permanently inaccessible. This is a common error in very long running programs.

- **Automatic memory management.** The run-time system is responsible for knowing when a data structure is no longer needed. This is tricky to implement correctly, but once implemented, it is much easier to use for the programmer. With automatic memory management, dangling pointers are impossible. Memory leaks are rarer but still possible. There is a leak if the program keeps a reference to a data structure that is no longer needed. There is a run-time overhead because the system runs a specialized program, called **garbage collector**, to determine when data structures are no longer needed and to put their memory back in the free memory. Modern garbage collectors are quite efficient: typically they add less than 10% overhead to overall running time. In some cases, they are even more efficient than manual memory management. We recommend [53] to learn more about these collectors and their algorithms. More serious than this overhead is the fact that many garbage collectors cannot run concurrently with the rest of the program. The program must stop while the garbage collector runs. Much effort has been put in designing garbage collectors for which this time is short. If there is a guaranteed upper bound to this time, then the garbage collector is called **real time**.

Almost all systems, even those with manual memory management, do some automatic memory management in those cases where it is easy. For example, almost
all systems have a *stack* to manage procedure environments. A procedure’s environment exists only while the procedure call is active; when it finishes the environment can be recovered. Because procedure calls are *nested*, a stack can be used to store the environments and recover their memory in an efficient way. Just before the procedure executes its last procedure call, the environment can be removed from the stack. Section 5.3 explains why this is possible *before* the last call, and not just after it.

**A typical garbage collector**

Typically, a garbage collector has two phases. First, it determines what the active memory is. It does this by following all pointers to all data structures, starting from the root set. The *root set* is the set of pointers that by definition are needed by the application. It includes all pointers in ready threads and all pointers in operating system data structures. Second, the garbage collector compacts the memory. That is, it makes all the active memory into one contiguous block and the free memory into one contiguous block.

**The Mozart garbage collector**

The Mozart system does automatic memory management. It has both a local garbage collector and a distributed garbage collector. The latter is used for distributed programming and is explained in Chapter 13. The local garbage collector uses a *copying dual-space* algorithm. Memory is divided into two spaces, each taking up half. At any instant, the program executes in one half. While following the root pointers, it copies all active data structures to the other half. Since they are copied to a contiguous block in the other half, this also does compaction. The main advantage of this kind of collector is that its execution time is proportional to the amount of active memory, not to the total amount. The two main disadvantages are that half the memory is unusable at any given time and that long-lived data structures (like system tables) have to be copied at each garbage collection. The algorithm can be modified to avoid this second disadvantage.

**5.13.3 Garbage collection is not magic**

Having garbage collection lightens the burden of memory management for the developer, but it does not eliminate it completely. There are two cases that remain the developer’s responsibility: avoiding memory leaks and managing external resources.

**Avoiding memory leaks**

It is the programmer’s responsibility to avoid memory leaks. If the program continues to reference a data structure that it no longer needs, then that data...
structure’s memory will never be recovered. The program should be careful to lose all references to data structures no longer needed.

For example, take a recursive function that traverses a list. If the list’s head is passed to the recursive call, then list memory will not be recovered during the function’s execution. Here is an example:

\[
L = [1 \ 2 \ 3 \ \ldots \ \ 1000000]
\]

\[
\text{fun } \{\text{Sum } X \ L1 \ L\}
\text{ case } L1 \ \text{of} \ Y | L2 \ \text{then} \ \{\text{Sum } X + Y \ L2 \ L\}
\text{ else } X \ \text{end}
end
\]

\[
\{\text{Browse } \{\text{Sum } 0 \ L \ L\}\}
\]

Sum sums the elements of a list. But it also keeps a reference to L, the original list, even though it does not need L. This means L will stay in memory during the whole execution of Sum. A better definition is as follows:

\[
\text{fun } \{\text{Sum } X \ L1\}
\text{ case } L1 \ \text{of} \ Y | L2 \ \text{then} \ \{\text{Sum } X + Y \ L2\}
\text{ else } X \ \text{end}
end
\]

Here the reference to L is lost immediately. This example is trivial. But things can be more subtle. For example, consider an active data structure S that contains a list of other data structures D1, D2, ..., Dn. If one of these, say D1, is no longer needed by the program, then it should be removed from the list. Otherwise, its memory will never be recovered.

A well-written program therefore has to do some “cleanup” after itself: making sure that it no longer references data structures that it no longer needs. This cleanup can be done in the declarative model, but it is cumbersome. It is most efficiently done with encapsulated state, as explained in Chapter 8.

Managing external resources

A Mozart program often needs data structures that are external to its process. We call such a data structure an external resource. It is the programmer’s responsibility to manage these data structures. In most cases, there is a data structure inside the Mozart process that corresponds to the one outside. For example, a record could correspond to a graphic entity in a graphics display or to an open file in a file system. If the record is no longer needed, then the graphic entity has to be removed or the file has to be closed. Otherwise, the graphics display or the file system will have a memory leak.

If the program recovers the memory of the internal data structure, then the external resource has to be recovered as well. This is done with a technique called finalization. Finalization is explained in Section 10.6.
5.14 Limitations of declarative programming

This chapter shows that declarative programming is a rich area with many programming techniques. A natural question to ask is, how far can declarative programming go? Can everything be programmed in a declarative way, such that the programs are both natural and efficient? We say a program is natural if very little code is needed just for technical reasons unrelated to the problem at hand. We say a program is efficient if its performance differs by just a constant factor from the performance of an assembly language program to solve the same problem.

The answer to this question depends on the sophistication of the compiler. Paraphrasing science fiction author and futurologist Arthur C. Clarke, we can say that “any sufficiently advanced compiler is indistinguishable from magic” [20]. That is, it is unrealistic to expect the compiler to rewrite your program. Even after several decades of research, no such compiler exists for general-purpose programming. The farthest we have come is compilers that can rewrite the program in particular cases. Paul Hudak calls them “smart-aleck” compilers. Because of their unpredictable optimizations, they are hard to use. Therefore, for the rest of the discussion, we assume that the compiler does a straightforward mapping from the source program to the target, in the sense that time and space complexities are the same for the language semantics and the compiled code. Now we can answer the question.

Given a straightforward compiler, the answer to the question is no. We find that there are five trade-offs in using declarative programming:

1. It leads to modularity problems when state or observable nondeterminism are needed. The state has to be threaded and the nondeterminism has to be fed from the outside, which means that the program’s components are tied together too strongly. The Model Independence Principle of Section 8.9.2 is lost. Here are a few examples:

   - Consider a declarative program with an external database interface. Assume this interface is visible as a declarative component. This component certainly has encapsulated state. To model within the system both the program and its interface requires encapsulated state.

   - Consider a declarative program that we wish to instrument. During execution, we want to know how many times that some of its subcomponents are invoked. We would like to add counters to these subcomponents, preferably without changing the subcomponent interfaces. This is impossible if the subcomponents are declarative. Section 8.1 gives an example.

   - Consider a client-server program with two clients and one server. To model faithfully the clients’ independence, we need to add true nondeterminism to the model. Otherwise, we need to feed an external source
of nondeterminism, e.g., a stream of decisions, to the clients and the server [11]. This is an overspecification.

- Consider a declarative program that behaves like a function. To improve performance, we would like to add memoization to the function. That is, we add an internal cache of previously-calculated results, indexed by the function arguments. When calling the function, we first check the cache to see if the result is already there. We would like to add the memoization cache without changing the component interface. Section ?? gives an example.

2. It often leads to intricate code. That is, even though the program can be written declaratively, doing so makes it more complex. This follows because declarative programs impose more constraints on how they are written. Section 8.8.4 shows an example: a transitive closure algorithm written in both declarative and stateful models. The stateful algorithm is easier to understand.

3. It leads to inefficient code for programs that do incremental modifications of large data structures, e.g., for simulations (see Section 8.10.3). Even after decades of research, there is no straightforward compiler that can take such a program and implement it efficiently.10

4. It leads to interfacing problems when used together with non-declarative components. Such components are omnipresent, e.g., operating systems and user interfaces are both inherently concurrent and stateful (see Section 5.12). These non-declarative properties have to be either masked or encoded somehow.

5. If the program’s specification mentions state, then declarative programming has to encode this in some way. For example, a specification for a collaborative tool may require that each user lock what they are working on to prevent conflicts. In the implementation, the locks have to be encoded in some way. Simply using locks directly leads to an implementation that is closer to the specification.

We recommend to use the declarative model (or the concurrent extension of Chapter 6) as often as possible. This simplifies reasoning about the program. But keep the above trade-offs in mind: ease of reasoning has to be seen in context. For a given problem, if a declarative program is more complex than a stateful program, then it might be simpler to reason about the stateful program.

### 5.15 Exercises

10There are a few special cases that can be done by a simple trick. This is explained in Section 8.3.2.
1. **Absolute value of real numbers.** We would like to define a function \texttt{Abs} that calculates the absolute value of a \textit{real} number. The following definition does not work:

```
fun \{Abs X\} if X<0 then ~X else X end end
```

Why not? How would you correct it? Hint: the problem is trivial.

2. **Cube roots.** This chapter uses Newton’s method to calculate square roots. The method can be extended to calculate roots of any degree. For example, the following method calculates cube roots. Given a guess \texttt{g} for the cube root of \texttt{x}, an improved guess is given by \((x/g^2 + 2g)/3\). Write a declarative program to calculate cube roots using Newton’s method.

3. **The half-interval method.** The half-interval method is a simple but powerful technique for finding roots of the equation \(f(x) = 0\), where \(f\) is a continuous function. The idea is that, if we are given points \(a\) and \(b\) such that \(f(a) < 0 < f(b)\), then \(f\) must have at least one root between \(a\) and \(b\). To locate a root, let \(x = (a + b)/2\) and compute \(f(x)\). If \(f(x) > 0\) then \(f\) must have a root between \(a\) and \(x\). If \(f(x) < 0\) then \(f\) must have a root between \(x\) and \(b\). Repeating this process will define smaller and smaller intervals that converge on a root. Write a declarative program to solve this problem using the techniques of iterative computation.

4. **Iterative factorial.** This chapter gives a definition of factorial that produces a recursive computation. Give another definition of factorial which results in an iterative computation. Hint: do state transformations starting from an initial state, like in the \texttt{Iter} example.

5. **An iterative SumList.** Rewrite the function \texttt{SumList} of Section 5.7.1 to be iterative using the techniques developed for \texttt{Length}.

6. **Another append function.** This chapter defines the \texttt{Append} function by doing recursion on the first argument. What happens if we try to do recursion on the second argument? Here is a possible solution:

```
fun \{Append Ls Ms\}
  case Ms
  of nil then Ls
    [] X|Mr then \{Append \{Append Ls [X]\} Mr\}
  end
end
```

Is this program correct? Does it terminate? Why or why not?

---

11 This example is taken from Abelson & Sussman [1].
7. **Iterative computations and dataflow variables.** This chapter gives an example to show that using dataflow variables makes it simpler to write iterative list operations. This leads to the following question. For any iterative operation defined with dataflow variables, is it possible to give another iterative definition of the same operation that does not use dataflow variables?

8. **Limitations of difference lists.** What goes wrong when trying to append the same difference list more than once?

9. **Complexity of list flattening.** Calculate the number of operations needed by the two versions of the `flatten` function given in Section 5.7.3. With \( n \) elements and maximal nesting depth \( k \), what is the worst-case complexity of each version?

10. **FIFO queues.** Consider the FIFO queue defined in Section 5.7.3. Answer the following two questions:

- What happens if you delete an element from an empty queue?
- Why is it wrong to define \texttt{EmptyQueue} as follows?

\[
\text{fun}\ \{\text{EmptyQueue } q(N\ S\ E)\}\ S\Rightarrow E\ \text{end}
\]

11. **Quicksort.** The following is a possible algorithm for sorting lists. Its inventor, C.A.R. Hoare, called it \textit{quicksort}, because it was the fastest known general-purpose sorting algorithm at the time it was invented. It uses a divide and conquer strategy to give an average time complexity of \( O(n \log n) \). Here is an informal description of the algorithm for the declarative model. Given an input list \( L \). Then do the following operations:

- Pick \( L \)'s first element, \( x \), to use as a pivot.
- Partition \( L \) into two lists, \( L1 \) and \( L2 \), such that all elements in \( L1 \) are less than \( x \) and all elements in \( L2 \) are greater or equal than \( x \).
- Use quicksort to sort \( L1 \) giving \( S1 \) and to sort \( L2 \) giving \( S2 \).
- Append the lists \( S1 \) and \( S2 \) to get the answer.

Write this program with difference lists to avoid the linear cost of append.

12. **Infinite order procedure.**

- Give an example of a procedure of infinite order.
- Ask what it does.

*Copyright c⃝ 2001 by P. Van Roy and S. Haridi. All rights reserved.*
5.16 Perspectives

The computation model of this chapter, called the declarative model, is the smallest practical subset of the full computation model of the book. This model subsumes strict functional programming and deterministic logic programming. In strict functional programming, all calculation is function evaluation on complete values. A program is just a set of function definitions. Strict means that all a function’s arguments are always evaluated before the function itself is called.

In deterministic logic programming, all calculation is logical deduction. A program defines a set of logical predicates. Each predicate is defined as a logical formula that can contain other predicates. Calculation is a set of deduction steps, where a deduction step replaces a predicate by part of its definition. Deterministic means that in each deduction step there is exactly one possibility.

By introducing one new statement (choice) and search strategies that use it (the search module), it is possible to do a basic form of nondeterministic logic programming. This basic form was first popularized by the Prolog language.

In imperative languages like C and Java, a variable can be assigned multiple times. In contrast, dataflow variables can be assigned to only one value. This notion is not a new one; variants of it have been used in many languages including dataflow languages (e.g., Id) and concurrent logic programming languages (e.g., Concurrent Prolog). Typically, in other dataflow languages, a variable can be assigned only once. In the declarative model, a variable can be assigned to only one value. This is a subtle difference, but it marks a major difference in philosophy. A dataflow variable has two phases in its lifetime. Initially it is introduced with unknown value, and later it might be assigned a value, in which case the variable becomes bound. Once a variable is bound, it cannot be changed. A dataflow variable, or logic variable, is a single-assignment variable that can also be equated with another variable.

Difference lists as explained in this chapter come from Prolog. A difference list is not the same as an accumulating parameter, despite what some texts claim [97].
Chapter 6

Declarative Concurrency

“Only then did Atreyu notice that the monster was not a single, solid body, but was made up of innumerable small steel-blue insects which buzzed like angry hornets. It was their compact swarm that kept taking different shapes.”

“Twenty years ago, parallel skiing was thought to be a skill attainable only after many years of training and practice. Today, it is routinely achieved during the course of a single skiing season. [...] All the goals of the parents are achieved by the children: [...] But the movements they make in order to produce these results are quite different.”
– Mindstorms: Children, Computers, and Powerful Ideas, Seymour Papert (?–)

The declarative model of Chapter 4 lets us write many programs and use powerful reasoning techniques on them. But, as Section 5.14 explains, there are many useful programs that cannot be written easily or efficiently in it. In particular, concurrent programs cannot be written easily, i.e., programs with multiple independently-evolving computations. Some examples are producer/consumer, client/server, and pipelined programs. In this chapter we remove this limitation. We show how to extend the declarative model of Chapter 4 with concurrency while still being declarative. The extended model is called the concurrent declarative model.

Surprising as it may seem, a program written in the concurrent declarative model is truly concurrent as well as being declarative. There is just one simple condition that the program has to satisfy: there should be no observable nondeterminism. Given this, all the programming and reasoning techniques for declarative programming still apply.

The condition of no observable nondeterminism is always satisfied in the concurrent declarative model if no unification exceptions are raised. This is explained fully in Section 6.1. Briefly, whenever a dataflow variable is bound, there is only
one possible value it is bound to. The execution does not have to “choose” between different values. There are no attempts to bind it to a second value. One consequence of this condition is that there are no race conditions or contention problems in a concurrent declarative program.

In a traditional computer science curriculum, concurrency is taught by extending a stateful model, just as Chapter 10 extends Chapter 8. This is rightly considered to be very complex and difficult to program with. However, it is less widely known that there are much simpler forms of concurrency that can almost always be used in place of stateful concurrency (see ski quote above). The declarative concurrency of this chapter is one of the simplest forms, yet in a practical sense it has almost all the power of the general form. Another simple form, message passing between active objects, is explained in Chapter 10.

This chapter is structured as follows. Section 6.1 defines the computation model. Then, Section 6.2 gives the basics of threads and programming with threads. We introduce the concept of thread priority and show how it affects computation. Section 6.3 explains the technique of order-determining concurrency, which uses concurrency to help write a sequential program. Section 6.4 defines the various forms of synchronization, which is one of the basic operations needed in a concurrent program. Section 6.5 explains stream communication between active entities, which is a basic way to organize concurrency in a declarative model. We show how state shows up in the structure of concurrent programs, even in the declarative model, when doing calculations with streams. We distinguish between eager and lazy stream communication and explain their respective advantages and disadvantages. Section 6.6 shows how to program with laziness. We introduce list comprehensions, which are a powerful abstraction built on top of lazy streams. Section 6.7 gives more advanced programming techniques such as concurrent composition. Finally, Section 6.8 shows how to do soft real time computations. Later chapters extend the use of concurrency. Chapter 10 explains how to use concurrency together with state, in particular for concurrent object-oriented programming.

6.1 The declarative model with concurrency

In Chapter 4 we presented the declarative computation model, which consists of one statement, i.e., activity, that executes over a single-assignment store. How do we generalize this model to express several activities that execute simultaneously. A system with this property is called concurrent. Concurrency is essential for programs that interact with their environment, e.g., for agents, GUI programming, OS interaction, and so forth. Concurrency also lets a program be organized into parts that execute independently and interact only when needed. This is an important software engineering property.

How can we express concurrent execution in the declarative model? Without changing the model, the only way is to encode it. That is, to program the exe-
6.1 The declarative model with concurrency

Multiple statements (threads)

W=atom
Z=person(age: Y)
X
Y=42
U

Figure 6.1: The concurrent declarative model

\[
\begin{align*}
\text{(statement)} & ::= \\
\text{skip} & \quad \text{Empty statement} \\
| (\text{statement}) (\text{statement}) & \quad \text{Statement sequence} \\
\text{local } X \text{ in } (\text{statement}) \text{ end} & \quad \text{Variable declaration} \\
| X & \equiv Y & \quad \text{Bind to variable} \\
| X & \equiv f(l1:X1 \ldots l:n:Xn) & \quad \text{Bind to record} \\
\text{if } X \text{ then } (\text{statement}) \text{ else } (\text{statement}) \text{ end} & \quad \text{Test for boolean} \\
\text{proc } (P \ X1 \ldots Xn) \text{ (statement) end} & \quad \text{Procedure declaration} \\
| (P \ X1 \ldots Xn) & \quad \text{Procedure call} \\
\text{try } (\text{statement}) \text{ catch } X \text{ then } (\text{statement}) \text{ end} & \quad \text{Exception handler} \\
| \text{raise } X \text{ end} & \quad \text{Raise exception} \\
| \text{thread } (\text{statement}) \text{ end} & \quad \text{Thread creation} \\
| \text{(ByNeed P X)} & \quad \text{Trigger creation}
\end{align*}
\]

Table 6.1: The concurrent declarative kernel language

cution of the activities explicitly, using the declarative model as the underlying system. For example, a program can keep track of each activity and do one step of each in round-robin fashion. This is complicated and error-prone.

A better way is to extend the declarative model. We extend the model with two concepts, threads and triggers. We call the resulting model the concurrent declarative model. Table 6.1 gives its kernel language. Here we briefly explain and justify threads and triggers. We also answer the interesting question why we need two extra concepts. Why is it not enough to add just threads? Threads and triggers are explained more fully in respectively Sections 6.2 and 6.6.
6.1.1 Threads

A simple approach to add concurrency is to allow more than one statement to reference the store. This gives the model of Figure 6.1, which extends Figure 4.4. Roughly, all these statements are executing “at the same time”. We call an executing statement a *thread*. Now we have to be precise about what “at the same time” means. We distinguish between the language viewpoint and the implementation viewpoint:

- The language viewpoint is the semantics of the language, as seen by the programmer. From this viewpoint, the simplest assumption is to let the threads do an *interleaving* execution: there is one, totally ordered sequence of computation steps and threads take turns doing one or more computation steps in this sequence. Computation steps do not overlap, or in other words, each computation step is atomic. This makes reasoning about programs easier.

- The implementation viewpoint is how the multiple threads are actually implemented on a real machine. If the system is implemented on a single processor, then we can let the implementation also do interleaving. However, the system can also be implemented on multiple processors, taking advantage of parallelism to improve performance. Either way, the language has the same semantics.

The two viewpoints are quite independent of each other. An interleaving execution can easily be implemented in a parallel way. For example, it can be done on a shared-memory multiprocessor. Different threads are allocated to different processors. Because the threads see the same memory, there is still one, totally ordered sequence of computation steps.

Another way to see the difference between sequential and concurrent execution is in terms of an order defined among all execution states of a given program:

<table>
<thead>
<tr>
<th>Causal order of computation steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>For a given program, all computation steps form a partial order, called the <em>causal order</em>. An execution state occurs <em>before</em> another state, if in all possible executions of the program, it happens before the other state. Similarly for an execution state that occurs <em>after</em> another state. Sometimes a state is neither before nor after another state. In that case, we say that the two states are <em>concurrent</em>.</td>
</tr>
</tbody>
</table>

In a sequential program, all execution states are totally ordered. There are no concurrent states. In a concurrent program, all execution states of a *given thread* are totally ordered. The execution states of the program form a partial order. Figure 6.2 shows the difference between sequential and concurrent execution.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
A thread is ready, also called runnable, if its statement has all the information it needs to execute. A ready thread can be executed at any time. A thread that is not ready is called suspended. It is up to the system to choose which ready thread to execute. This is called scheduling. We say the system is fair if it does not let any ready thread “starve”, i.e., it does not ignore any ready thread indefinitely. In the rest of the book, we will assume that threads are scheduled fairly.

### 6.1.2 Nondeterminism

We say an execution is nondeterministic if in some configuration it has a choice of which statement to execute. If the choice is irrevocable, we call this don’t care nondeterminism. In this case, the program should be written so that it is correct no matter which choice is taken. Hence the adjective “don’t care”. If the choice can be repeated until some condition is satisfied, we call this don’t know nondeterminism. In this case, the program should be written so that it searches possible choices until the right one is found. The program does not know the right choice at first, but keeps on trying until it finds it. Hence the adjective “don’t know”.

The terms “don’t care” and “don’t know” come from logic programming, where they were originally introduced to distinguish search-based logic programming (Prolog-style) from concurrent logic programming.

#### Don’t care nondeterminism

In the concurrent declarative model, more than one thread can be simultaneously ready. This means that more than one statement can be ready at a given instant. When doing a computation step, the system therefore has to choose one of these statements to execute first. The choice is done at a low level in the system’s implementation and is irrevocable. It is an example of don’t care nondeterminism.
Observable nondeterminism

A nondeterministic choice by the system may or may not be observable in the program. To be observable, the choice must satisfy two conditions: (1) different store contents are possible when different choices are made, and (2) the store contents can affect the program’s execution. For example, calling the following procedure satisfies (1) but not (2):

```plaintext
class HiddenNondet
in
    thread try x=1 catch _ then skip end end
    thread try x=2 catch _ then skip end end
end
```

One of two possible bindings is done and the other raises an exception. The exception is caught and has no effect. The binding is encapsulated by lexical scoping. Therefore different store contents are possible, but they are not observable to the caller. To be observable, the nondeterminism has to be visible to the caller:

```plaintext
class VisibleNondet x

thread try x=1 catch _ then skip end end
    thread try x=2 catch _ then skip end end
end
```

This procedure returns either 1 or 2, so the nondeterminism is observable. If a program in the concurrent declarative model has no observable nondeterminism, then it is declarative. As we will see below, a sufficient condition for this is that the program does not raise any exceptions.

Don’t know nondeterminism

Relational programming, as explained in Chapter 7, also has nondeterminism. It extends the concurrent declarative model with a `choice` statement that allows the execution to choose one of a set of alternative statements. The choice is not irrevocable; on the contrary, the program can search all possible choices. It is an example of don’t know nondeterminism. This nondeterminism differs in two ways from the nondeterminism introduced by concurrency:

- It is explicit. It is introduced by a new statement, `choice`, that was designed for this purpose. Because `choice` just picks an alternative, it is not concurrent—it can be used even in purely sequential programs. On the other hand, the nondeterminism introduced by concurrency is implicit: it is not introduced specifically, but only as a consequence of the decision to introduce concurrency.

- It is controllable. It is possible to control how choices are made. The choices define a search space and the system has operations to explore this space.
This is not possible with implicit nondeterminism, which makes irrevocable choices.

The relational model of Chapter 7 can also be declarative. The same condition holds as for the concurrent declarative model: the nondeterminism should not be observable.

### 6.1.3 Exceptions

If a component raises an exception in the concurrent declarative model then it is no longer declarative. We can understand why by investigating why exceptions occur. First of all, to be declarative, a component has to be deterministic. For example, if the statements $x=1$ and $x=2$ are executed concurrently, then execution is no longer deterministic: one of them will succeed and the other will raise an exception. In the store, $x$ will be bound either to 1 or to 2; both cases are possible. This is a clear case of nondeterminism. Because the store has one of two possible values, the nondeterminism is observable. The exception is a witness to this; it is raised on unification failure, which means that potentially there is nondeterminism. The exception is not a guarantee of nondeterminism; for example executing $x=1$ and $x=2$ in order in the same thread will raise an exception, yet $x$ is always bound to 1. But if there are no exceptions, then execution is surely deterministic and hence declarative.

Secondly, an exception is also raised when an operation cannot provide its service. This can be due to internal reasons, e.g., the arguments are outside the operation’s domain (such as dividing by zero), or external reasons, e.g., the external environment has a problem (such as trying to open a file that does not exist). In both cases, the exception indicates that an operation was attempted outside of its specification. When this happens, all bets are off, so to speak. From the viewpoint of semantics, there is no guarantee on what the operation has done; it could have done anything. The operation has potentially become nondeterministic.

To summarize, when an exception is raised, this is an indication either of nondeterministic execution or of an execution outside specification. In either case, the component is no longer declarative. We say that the concurrent declarative model is declarative *modulo exceptions*. It turns out that the concurrent declarative model with exceptions is similar to the concurrent stateful model of Chapter 10. See Section 10.1 for more information.

So what do we do when an exception occurs? Are we completely powerless to write a declarative program? Not at all. In some cases, the component can “fix things” so that it is still declarative when viewed from the outside. The basic problem is to make the component deterministic. All sources of nondeterminism have to be hidden from the outside. For example, if a component executes $x=1$ and $x=2$ concurrently, then the minimum it has to do is (1) catch the exception by putting a `try` around each binding, and (2) encapsulate $x$ so its value is not
observable from the outside. It may have to do more to guarantee determinism, but these two things at least have to be done.

6.1.4 Triggers

Is it enough to extend the declarative model by adding threads? It may seem so at first glance, but in fact, something fundamental is missing. Threads as introduced above always execute in supply-driven fashion, i.e., they execute as soon as they are ready, regardless of whether the execution’s result is needed anywhere else in the program. There is another basic way to execute threads, namely in demand-driven fashion. In this second way, a statement is only executed when its result is needed. Demand-driven execution is very important—it is still declarative and it often allows to modularize programs in interesting ways. It is very close to the concept of lazy evaluation, as explained in Section 6.6.

How do we add demand-driven execution to the model? It turns out that a practical way to integrate both supply-driven and demand-driven execution in the same model is to make supply-driven execution the default and to add an extra operation to introduce a demand-driven part. This is because most practical execution is supply-driven: applications are written to do primarily useful calculations i.e., calculations whose results will be needed, and supply-driven execution can be implemented more efficiently than demand-driven execution. This does not mean that the demand-driven part can be left out completely. On the contrary, often the best way to structure an application is to build it in supply-driven fashion around a central demand-driven part. Section 6.6 gives examples of this structure.

To add demand-driven execution we introduce the concept of trigger. A trigger is a pair consisting of a boolean expression, called the condition, and a procedure, called the action. When the condition becomes true, then the action is executed once. We distinguish between programmed triggers, which are written explicitly by the programmer, and internal triggers, which are provided by the language implementation. For demand-driven execution we add internal triggers to the computation model. The internal trigger is a link between a dataflow variable and a zero-argument function. When the variable’s binding is needed, the trigger is activated, which causes the function to be evaluated in its own thread and its result to be bound to the variable. We therefore call it a by-need trigger. We say a value is needed by an operation if the operation cannot complete unless the value is present. For example, the addition \( X+Y=Z \) needs \( X \) and \( Y \) and gives the result \( Z \). The basic operations on internal triggers are explained in Section 6.6.7.

6.1.5 The model’s “systolic” character

In the concurrent declarative model, threads communicate through shared dataflow variables. There is a close correspondence between operations on dataflow variables (binding, waiting until bound) and operations on a communications channel
Threads

6.2 Threads

6.2.1 The thread statement

The \texttt{thread} statement creates a new thread:

\begin{verbatim}
thread
  proc {Count N} if N>0 then {Count N-1} end end
in
  {Count 1000000}
\end{verbatim}
This creates a new thread that runs concurrently with the main thread. It can also be used as an expression (see Table 6.3):

```plaintext
declare X in
X = thread 10*10 end + 100*100
{Browse X}
end
```

This is just syntactic sugar for:

```plaintext
declare X in
local Y in
  thread Y=10*10 end
  X=Y+100*100
end
```

A new dataflow variable, Y, is created to communicate between the main thread and the new thread. The addition suspends until the calculation 10*10 is finished.

When a thread has no more statements to execute then it terminates. Each nonterminated thread that is not suspended will eventually be run. We say that threads are executed fairly. Thread execution is implemented with preemptive scheduling. That is, if more than one thread is ready to execute, then each thread will get processor time in discrete intervals called time slices. It is not possible for one thread to take over all the processor time.

### 6.2.2 Threads and the browser

The browser is a good example of a program that works well in a concurrent environment. For example:

```plaintext
thread {Browse 111} end
(Browse 222)
```

In what order are the values 111 and 222 displayed? The answer is, either order is possible! Is it possible that something like 112122 will be displayed, or worse, that the browser will behave erroneously? At first glance, it might seem so, since the browser has to execute many statements to display each value 111 and 222. If no special precautions are taken, then these statements can indeed be executed in almost any order. But the browser is designed for a concurrent environment. It will never display strange interleavings. Each browser call is given its own part of the browser window to display its argument. If the argument contains an
unbound variable that is bound later, then the display will be updated when the variable is bound. In this way, the browser will correctly display even multiple streams that grow concurrently, for example:

\begin{verbatim}
declare X1 X2 Y1 Y2 in
thread (Browse X1) end
thread (Browse Y1) end
thread X1=all|roads|X2 end
thread Y1=all|roams|Y2 end
thread X2=lead|to|rome|_ end
thread Y2=lead|to|rhodes|_ end
\end{verbatim}

This correctly displays all|roads|lead|to|rome|_ and all|roams|lead|to|rhodes|_ in separate parts of the browser window. In this chapter and later chapters we will see how to write concurrent programs that behave correctly, like the browser.

### 6.2.3 Dataflow computation with threads

Let us see what we can do with threads. First, remember that each thread is a dataflow thread, i.e., it blocks on availability of data. Consider the following program:

\begin{verbatim}
declare X0 X1 X2 X3 in
thread
  Y0 Y1 Y2 Y3 in
  {Browse [Y0 Y1 Y2 Y3]}
  Y0=X0+1
  Y1=X1+Y0
  Y2=X2+Y1
  Y3=X3+Y2
  {Browse completed}
end
{Browse [X0 X1 X2 X3]}
\end{verbatim}

If you feed this program then the browser will display all the variables as being unbound. Observe what happens when you input the following statements one at a time:

\begin{verbatim}
X0=0
X1=1
X2=2
X3=3
\end{verbatim}

With each statement, the thread resumes, executes one addition, and then suspends again. That is, when \(X0\) is bound the thread can execute \(Y0=X0+1\). It suspends again because it needs the value of \(X1\) while executing \(Y1 =X1+Y0\), and so on.
Figure 6.3: A concurrent map function

```haskell
fun {Map Xs F}
case Xs
  of nil then nil
      [] X | Xr then thread \{F X\} end \{Map Xr F\}
  end
end
```

Figure 6.4: A concurrent Fibonacci function

```haskell
fun {Fib X}
  if X≤2 then 1
  else
    thread \{Fib X-1\} end + \{Fib X-2\}
  end
end
```

Concurrent map function

Figure 6.3 gives a concurrent version of the Map function defined in Section 5.7.1. The thread statement is used here as an expression. Let us discuss the behavior of this program. If we enter the following statements:

```haskell
define
F Xs Ys Zs
\{Browse thread \{Map Xs F\} end\}
```

then a new thread executing \{Map Xs F\} is created. It will suspend immediately in the case statement because Xs is unbound. If we enter the following statements (without a define!):

```haskell
Xs=1|2|Ys
fun \{F X\} X*X end
```

then the main thread will traverse the list, creating two threads for the first two arguments of the list, thread \{F 1\} end and thread \{F 2\} end, and then it will suspend again on the tail of the list Y. Finally, doing

```haskell
Ys=3|Zs
Zs=nil
```

will create a third thread with thread \{F 3\} end and terminate the computation of the main thread. The three threads will also terminate, resulting in the final list [1 4 9].

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Concurrent Fibonacci function

Figure 6.4 defines a concurrent divide-and-conquer program to calculate the Fibonacci function. This program is based on the sequential recursive Fibonacci function; the only difference is that the first recursive call is done in its own thread. This program creates an exponential number of threads! Figure 6.5 shows all the thread creations and synchronizations for the call \( \{\text{Fib 6}\} \). A total of eight threads are involved in this calculation. You can use this program to test how many threads your Mozart installation can create. Feed:

\[
\text{(Browse \{Fib 25\})}
\]

while observing the Panel to see the threads. The Panel, shown in Figure 6.6, is a tool that gives information on system behavior (runtime, memory usage, threads, etc.). To start the Panel, select the Panel entry of the Oz menu in the interactive interface. If \( \{\text{Fib 25}\} \) completes successfully, try a larger argument.

Any declarative program of Chapter 5 can be made concurrent by putting \textbf{thread ... end} around some of its statements and expressions. Because each dataflow variable will be bound to the same value as before, the final result of the concurrent version will be exactly the same as the original sequential version. The next section explains how to use this technique to good advantage.

What can we learn from this example? The idea of explicit thread creation is to enable the programmer to structure his or her application in a modular way. In this respect the Mozart system is excellent. Threads are so cheap that one can afford to create them in large numbers. At the time of this writing, an entry-level personal computer with at least 64 MB of active memory can support more than 100000 simultaneous active threads.
If using concurrency lets your program have a simpler structure, then use it without hesitation. But keep in mind that even though threads are cheap, sequential programs are even cheaper. Sequential programs are always faster than concurrent programs having the same structure. The \texttt{Fib} program in Figure 6.4 is faster if the \texttt{thread} statement is removed. You should create threads only when the application needs them. On the other hand, you should not hesitate to create a thread if it improves application structure.

### 6.2.4 Thread scheduling and priority

In many applications, it is not enough to guarantee that threads are executed fairly. Some more control is needed over how processor time is shared between threads. For example, during the course of a computation, an event may happen that requires urgent treatment, bypassing the “normal” computation. On the other hand, it should not be possible for urgent computations to starve normal computations, i.e., to cause them to slow down inordinately.

#### Priority levels

A compromise that seems to work well in practice is to have priority levels for threads. Each priority level is given a minimum percentage of the processor time. Within each priority level, threads share the processor time fairly. The Mozart system uses this technique. It has three priority levels, \texttt{high}, \texttt{medium}, and \texttt{low}. By default, processor time is divided among the priorities in the ratios 100 : 10 : 1 for \texttt{high} : \texttt{medium} : \texttt{low} priorities. This is implemented in a very simple way: every tenth time slice of a high priority thread, a medium priority thread
is given one slice. Similarly, every tenth time slice of a medium priority thread, a low priority thread is given one slice. This means that high priority threads, if there are any, divide at least 100/111 (about 90%) of processor time amongst themselves. Similarly, medium priority threads, if there are any, divide at least 10/111 (about 9%) of processor time amongst themselves. And last of all, low priority threads, if there are any, divide at least 1/111 (about 1%) of processor time amongst themselves. These percentages are guaranteed lower bounds. If there are fewer threads, then they might be higher. For example, if there are no high priority threads, then a medium priority thread can get up to 10/11 of the processor time. The two ratios high : medium and medium : low are both 10 by default, but they can be changed with the Property module.

Priority inheritance

When a thread creates a child thread, then the child is given the same priority as the parent. This is particularly important for high priority threads. In an application, these threads are used for “urgency management”, i.e., to do work that must be handled in advance of the normal work. The part of the application doing urgency management can be concurrent. If the child of a high priority thread would have, say, medium priority, then there is a short “window” of time during which the child thread is medium priority, until the parent or child can change the thread’s priority. The existence of this window would be enough to keep the child thread from being scheduled for many time slices, because the thread is put in the medium priority queue. This could result in hard-to-trace timing bugs. Therefore a child thread should never get a lower priority than its parent.

Cooperative and competitive concurrency

Threads are intended for cooperative concurrency, not for competitive concurrency. That is, they are intended for applications that run in an environment where all parts trust one another. Therefore, any thread may use the Property module to change the time ratios between the three priorities. On the other hand, competitive concurrency consists of computations that are interested only in their own performance, not in the global performance. Competitive concurrency is usually managed by the operating system. In Mozart, it is supported by the distributed computation model of Chapter 13 through the module Remote. This module can create separate operating system processes each with its own computational resources. Each competitive computation can then be put in a separate process. This is easy to program because the distributed model is network-transparent; see Chapter 13 for examples and more information.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Time slice duration

What is the effect of the time slice’s duration? A short slice gives very “fine-grained” concurrency: threads react quickly to external events. But if the slice is too short, then the overhead of switching between threads becomes significant. Another question is how to implement preemption: does the thread itself keep track of how long it has run, or is it done externally? Both solutions are viable, but the second is much easier to implement. Modern multitasking operating systems, such as Unix or Windows NT, have timer interrupts that can be used to trigger preemption. These interrupts arrive at a fairly low frequency, 60 or 100 per second. The Mozart system uses this technique.

A time slice of 10 ms may seem short enough, but for some applications it is too long. For example, assume an application with 100000 active threads. Then each thread gets one time slice every 1000 seconds. In practice, we find that this is not a problem. In applications with many threads, such as large constraint programs (see Chapter 14), the threads usually depend strongly on each other and not on the external world. Each thread only uses a small part of its time slice before yielding to another thread.

On the other hand, it is possible to imagine an application with many threads, each of which interacts with the external world independently of the other threads. For such an application, it is clear that Mozart, and even Unix or Windows NT, are unsatisfactory. The hardware itself of a personal computer is unsatisfactory. What is needed is a hard real time computing system, which uses a special kind of hardware together with a special kind of operating system. Hard real time is outside the scope of the book.

6.2.5 Operations on threads

The modules Thread and Property provide a number of operations pertinent to threads. Some of these operations are summarized in Figure 6.7. The priority \( P \) can have three values, the atoms low, medium, and high. Each thread has a unique name, which refers to the thread when doing operations on it. The thread name is a value of Name type. The only way to get a thread’s name is for the thread itself to call Thread.this. It is not possible for another thread to get the name without cooperation from the original thread. This makes it possible to rigorously control access to thread names. The system procedure:

\[
\{\text{Property.put priorities(high:X medium:Y)}\}
\]
sets the processor time ratio to \( X:1 \) between high priority and medium priority and to \( Y:1 \) between medium priority and low-priority. \( X \) and \( Y \) are integers. If we execute:

\[
\{\text{Property.put priorities(high:10 medium:10)}\}
\]
then for each 10 time slices allocated to runnable high priority threads, the system will allocate one time slice to medium priority threads, and similarly between
medium and low priority threads. This is the default. Within the same priority level, scheduling is fair and round-robin.

6.3 Order-determining concurrency

“In whichever order these twenty-four cards are laid side by side, the result will be a perfectly harmonious landscape.”

– From “The Endless Landscape”: 24-piece Myriorama, Leipzig (1830s).

A simple use of concurrency in a declarative program is to find the order of calculations. That is, we know what calculations have to be done, but because of data dependencies, we do not know their order. What’s more, the order may depend on the values of the data, i.e., there is no one static order that is always right. This is a very restrictive way to use concurrency. It is part of the concurrent declarative model.

We give an example of order-determining concurrency using the tree drawing algorithm of Chapter 5. This algorithm is given a tree and calculates the positions of all the tree’s nodes so that the tree can be drawn in an aesthetically pleasing way. The algorithm traverses the tree in two directions: first from the root to the leaves and then from the leaves back up to the root. During the traversals, all the node positions are calculated. One of the tricky details in this algorithm is the order in which the node positions are calculated. Consider the algorithm definition given in Section 5.7.5. In this definition, Level and LeftLim are inputs (they propagate down towards the leaves), RootX and RightLim are outputs (they propagate up towards the root), and the calculations have to be done in the correct order to avoid deadlock. There are two ways to find the correct order:

- The first way is for the programmer to deduce the order and to program
Figure 6.8: Tree drawing algorithm with order-determining concurrency

accordingly. This is what Section 5.7.5 does. This gives the most efficient code, but if the programmer makes an error then the program blocks without giving a result.

- The second way is for the system to deduce the order dynamically. The simplest way to do this is to put each calculation in a different thread. Dataflow execution then finds the correct order at run time.

Figure 6.8 gives a version of the tree drawing algorithm that uses order-determining concurrency to find the correct calculation order at run time. Each calculation that might block is done in a thread of its own. The algorithm’s result is the same as before. This is true because the concurrency is used only to change the calculation order, not to change which calculations are done. This is an example of how to use concurrency in declarative programming and remain declarative. In the above code, the threads are created before the recursive calls. In fact, the

---

1Variable-variable binding is not put in its own thread because it never blocks.
6.4 Synchronization

In the previous section, we have seen that threads can communicate through shared dataflow variables. When a thread needs the result of a calculation done by another thread then it waits until this result is available. We say that it synchronizes on the availability of the result. Synchronization is one of the fundamental concepts in concurrent programming. Let us now investigate this concept more closely.

We first define precisely the basic concept, called a synchronization point. Consider threads T1 and T2, each doing a sequence of computation steps. T1 does $\alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots$ and T2 does $\beta_0 \rightarrow \beta_1 \rightarrow \beta_2 \rightarrow \ldots$. The threads actually execute together in one global computation. This means that there is one global sequence of computation steps that contains the steps of each thread, interleaved: $\alpha_0 \rightarrow \beta_0 \rightarrow \beta_1 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots$. There are many ways that the two computations can be interleaved. But not all interleavings can occur in real computations:

- Because of fairness, it is not possible to have an infinite sequence of $\alpha$ steps without some $\beta$ steps. Fairness is a global property that is enforced by the system.

- If the threads depend on each other’s results in some way, then there are additional constraints called synchronization points. A synchronization point links two computation steps $\beta_i$ and $\alpha_j$. We say that $\beta_i$ synchronizes on $\alpha_j$ if in every interleaving that can occur in a real computation, $\beta_i$ occurs after $\alpha_j$. Synchronization is a local property that is enforced by operations happening in the threads.

How does the program specify when to synchronize? There are two broad approaches:

- Implicit synchronization. In this approach, the synchronization operations are not visible in the program text; they are part of the operational
semantics of the language. For example, using a dataflow variable will synchronize on the variable being bound to a value.

- **Explicit synchronization.** In this approach, the synchronization operations are visible in the program text; they consist of explicit operations put there by the programmer. For example, the programmer can define a critical section in the program text, namely a part of the program inside of which only one thread can execute at a time. A second thread that tries to enter the critical section will synchronize on the first thread leaving.

There are two directions of synchronization:

- **Supply-driven synchronization (eager execution).** Attempting to execute an operation causes the operation to wait until its arguments are available. In other words, the operation synchronizes on the availability of its arguments. This waiting has no effect on whether or not the arguments will be calculated; if some other thread does not calculate them then the operation will wait indefinitely.

- **Demand-driven synchronization (lazy execution).** Attempting to execute an operation causes the calculation of its arguments. In other words, the calculation of the arguments synchronizes on the operation needing them.

Table 6.4 shows the four possibilities that result. All four are practical and exist in real systems. Explicit synchronization is the primary mechanism in most languages that are based on a stateful model, e.g., Java, Smalltalk, and C++. This mechanism is presented in Chapter 10 and there compared with implicit synchronization. Implicit synchronization is the primary mechanism in most languages that are based on a declarative model, e.g., functional languages such as Haskell use lazy evaluation and logic languages such as Prolog and concurrent logic languages use dataflow execution. This mechanism is presented in this chapter.

All four possibilities can be used efficiently in the computation models of this book. This lets us compare their expressiveness and ease of use. We find that concurrent programming is simpler with implicit synchronization than with explicit synchronization. In particular, we find that programming with dataflow execution makes concurrent programs simpler. Even in a stateful model, like the one in Chapter 10, dataflow execution is advantageous. After comparing languages with

<table>
<thead>
<tr>
<th>Implicit</th>
<th>Supply-driven execution</th>
<th>Demand-driven execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit</td>
<td>locks, monitors, etc.</td>
<td>programmed trigger</td>
</tr>
</tbody>
</table>

Table 6.4: Classifying synchronization
explicit and implicit synchronization, Bal et al come to the same conclusion: that
dataflow variables are “spectacularly expressive” in concurrent programming as
compared to explicit synchronization, even without explicit state [8]. This ex-
pressiveness is one of the reasons why we emphasize implicit synchronization in
the book. Let us now examine more closely the usefulness of dataflow execution.
In Section 6.6 we will look at lazy execution.

6.4.1 Usefulness of dataflow execution

Section 6.2.3 shows how dataflow execution is used for synchronization in the
concurrent declarative model. There are many other uses for dataflow execution.
This section summarizes these uses. We give pointers to examples throughout
the book to illustrate them. Dataflow execution is useful because:

- It is a powerful primitive for concurrent programming (see this chapter and
  Chapter 10). It can be used for synchronizing and communicating between
  concurrent computations. Many concurrent programming techniques be-
  come simplified and new techniques become possible when using dataflow
  variables.

- It removes order dependencies between parts of a program (see this chapter
  and Chapter 10). To be precise, it replaces static dependencies (decided by
  the programmer) by dynamic dependencies (decided by the data). This is
  the basic reason why dataflow computation is useful for parallel program-
  ming. The output of one part can be passed directly as input to the next
  part, independent of the order in which the two parts are executed. When
  the parts execute, the second one will block only if necessary, i.e., only if it
  needs the result of the first and it is not yet available.

- It is a powerful primitive for distributed programming (see Chapter 13). It
  improves latency tolerance and third-party independence. A dataflow vari-
  able can be passed among sites arbitrarily. At all times, it “remembers its
  origins,” i.e., when the value becomes known then the variable will receive
  it. The communication needed to bind the variable is part of the variable
  and not part of the program manipulating the variable.

- It makes it possible to do declarative calculations with partial information.
  This was exploited in Chapter 5 with difference lists. One way to look at
  partial values is as complete values that are only partially known. This is
  a powerful idea that is further exploited in constraint programming (see
  Chapter 14).

- It allows the declarative model to support logic programming (see Sec-
  tion 5.11.1). That is, it is possible to give a logical semantics to many
  declarative programs. This allows reasoning about these programs at a very
  high level of abstraction. From a historical viewpoint, dataflow variables
were originally discovered in the context of concurrent logic programming, where they are called logic variables.

An insightful way to understand dataflow variables is to see them as a middle ground between having no state and having state:

- A dataflow variable is stateful, because it can change state (i.e., be bound to a value), but it can be bound to just one value in its lifetime. The stateful aspect can be used to get some of the advantages of programming with state (as explained in Chapter 8) while staying within a declarative model. For example, difference lists can be appended in constant time, which is not possible for lists in a pure functional model.

- A dataflow variable is stateless, because binding is monotonic. By monotonic we mean that more information can be added to the binding, but no information can be changed or removed. Assume the variable is bound to a partial value. Later on, more and more of the partial value can be bound, which amounts to binding the unbound variables inside the partial value. But these bindings cannot be changed or undone.

The stateless aspect can be used to get some of the advantages of declarative programming within a non-declarative model. For example, it is possible to add concurrency to the declarative model, giving the concurrent declarative model of this chapter, precisely because threads communicate through shared dataflow variables.

### 6.4.2 Futures and I-structures

The dataflow variables used in this book are but one technique to implement dataflow execution. Another, quite popular technique is based on a slightly different concept, the single-assignment variable. Two of the best-known instances of this concept are futures and I-structures. The purpose of futures and I-structures is to increase the potential parallelism of a program by removing inessential dependencies between calculations. They allow concurrency between a computation that calculates a value and one that uses the value. This concurrency can be exploited on a parallel machine. We define futures and I-structures and compare them with dataflow variables.

Futures were first introduced in Multilisp, a language intended for writing parallel programs [37]. Multilisp introduces the function call (future $E$) (in Lisp syntax), where $E$ is any expression. This does two things: it immediately returns a placeholder for the result of $E$ and it initiates a concurrent evaluation of $E$. When the value of $E$ is needed, i.e., a computation tries to access the placeholder, then the computation blocks until the value is available. We model this as follows in the concurrent declarative model (where $E$ is a zero-argument function):
fun {Future E} 
X in 
  thread X={E} end 
!!X 
end

A future can only be bound by the concurrent computation that is created along with it. That is why the above code returns a read-only variable. Multilisp also has a delay construct that does not initiate any evaluation but uses by-need synchronization. It causes evaluation of its argument only when the result is needed.

An I-structure (for incomplete structure) is a single-assignment array whose elements can be accessed before all the elements are computed. I-structures were introduced as a language construct for writing parallel programs on dataflow machines [6, 102, 46]. An I-structure permits concurrency between a computation that calculates the array elements and a computation that uses their values. When the value of an element is needed, then the computation blocks until it is available. Like a future and a read-only variable, an element of an I-structure can only be bound by the computation that calculates it.

There is a fundamental difference between dataflow variables on one side and futures and I-structures on the other side. The latter can be bound only once, whereas dataflow variables can be bound more than once, as long as the bindings are consistent with each other. Two partial values are consistent if they are unifiable. A dataflow variable can be bound many times to different partial values, as long as the partial values are unifiable. Section 6.5.1 gives an example when doing stream communication with multiple readers. Multiple readers are each allowed to bind the list’s tail, since they bind it in a consistent way.

### 6.5 Streams

A basic technique for concurrent programming in the concurrent declarative model is to use streams to communicate between threads. A stream is a potentially unbounded list of messages. Threads communicating through streams are a kind of “active object”. No locking or mutual exclusion is necessary since each thread has its own data structures, accessed only by itself.

Streams extend the “channel” intuition of Section 6.1.5. Any number of messages can be sent to and received from the stream. A stream can be implemented with dataflow variables. The idea is to wrap a message in a pair of the message and a new variable. Sending is binding the pair to the old variable. The new variable then takes over the role of the old. In this way, a stream can have a potentially unbounded length.

There are two primitive operations on streams: an asynchronous ordered send and a synchronous receive. The messages are received in the same order as they are sent. As we saw in Section 6.1.5, all this is a consequence of the semantics of
dataflow. Sending is binding a dataflow variable and receiving is waiting until a dataflow variable is bound.

### 6.5.1 Eager stream communication

In this section we explain how streams work and we show how to program an asynchronous producer-consumer with streams. In the concurrent declarative model, a stream is represented by a list whose tail is an unbound variable:

```
declare Xs Xs2 in
Xs=1|2|3|4|Xs2
```

A stream is created incrementally by binding the tail to a new list pair and a new tail:

```
declare Xs3 in
Xs2=5|6|7|Xs3
```

One thread, called the producer, creates the stream in this way, and other threads, called the consumers, read the stream. Because the stream’s tail is a dataflow variable, the consumers will read the stream as it is created. The following program asynchronously generates a stream of integers and sums them:

```
fun {Generate N Limit}
  if N<Limit then
    N|{Generate N+1 Limit}
  else nil end
end

fun {Sum Xs A}
  case Xs of
    X|Xr then {Sum Xr A+X}
    [] nil then A end
end

local Xs S in
  thread Xs={Generate 0 150000} end % Producer thread
  thread S={Sum Xs 0} end % Consumer thread
  {Browse S}
end
```

This displays 11249925000. The producer, Generate, and the consumer, Sum, run in their own threads. They communicate through the shared variable Xs, which is bound to a stream of integers. The case statement in Sum suspends when Xs is unbound (no more elements), and resumes when Xs is bound (new elements arrive).

In the consumer, dataflow behavior of the case statement suspends execution until the arrival of the next stream element. This synchronizes the consumer thread with the producer thread. Waiting on for a dataflow variable to be bound
is the basic mechanism for synchronization and communication in the concurrent declarative model.

Using a higher-order iterator

The recursive call to \texttt{Sum} has an argument \(A\) that is the sum of all elements seen so far. This argument, also called \textit{accumulating parameter}, is a kind of \textit{state}. We see that state appears in the program even though the declarative model has no explicit state. We can avoid the need for \texttt{Sum} and the accumulating parameter by using a higher-order iterator:

\begin{verbatim}
local Xs S in
  thread Xs=(Generate 0 150000 Xs) end
  thread S={FoldL Xs Number.‘+’ 0} end
  {Browse S}
end
\end{verbatim}

The \texttt{FoldL} operation is part of the Base module \texttt{List}. \texttt{FoldL} takes a list of elements \(\{A \ B \ C \ldots\}\), a two-argument function \(F\), and its neutral element \(N\). It then evaluates \(\{F \ (F \ (F \ A \ N) \ B) \ C\} \ldots\). In our case, \(F\) is the addition operation, which is part of the Base module \texttt{Number} and is accessed as \texttt{Number.‘+’}. The \texttt{FoldL} operation can be defined as follows:

\begin{verbatim}
fun \{FoldL Xs F N\}
  fun \{FoldL2 Xs Prev\}
    case Xs
      of nil then Prev
      [] X\|Xr then
        \{FoldL2 Xr \{F X Prev\}\}
    end
  end
in
  \{FoldL2 Xs N\}
end
\end{verbatim}

The recursive function \texttt{FoldL2} has the accumulating parameter \(\texttt{Prev}\), which keeps track of the intermediate result. Note that \texttt{FoldL2} has the same dataflow behavior as \texttt{Sum}. The technique of getting rid of an accumulating parameter by using a generic iterator is a general one. In fact, the accumulating parameter is not really gone, it is just hidden inside the iterator. But development is simplified since the programmer no longer has to reason in terms of state. The \texttt{List} module has many iterators that can be used to implement many different kinds of recursive functions.

It is not always practical to get rid of state in this way. An important case in which the state stubbornly refuses to go away is when it is accessed by a changing number of threads, e.g., a server that is accessed by a changing number of clients. Let the server and each client be implemented by one thread. If clients can come
and go during execution, then this is complicated to program in the declarative model with concurrency. With explicit state it is easy, as Chapter 10 shows.

Multiple readers

We can introduce multiple consumers without changing the program in any way. For example, here are three consumers, reading the same stream:

```plaintext
local Xs S1 S2 S3 in
  thread Xs={Generate 0 150000} end
  thread S1={Sum Xs 0} end
  thread S2={Sum Xs 0} end
  thread S3={Sum Xs 0} end
end
```

Each consumer thread will receive stream elements independently of the others. The consumers do not interfere with each other because the stream $Xs$ is stateless.

6.5.2 Using thread priorities to tune resource consumption

Changing the relative execution times of threads can make a big difference in the memory behavior of a program. Try running the above producer-consumer example with 15000000 elements instead of 150000. Switch on the panel and observe the memory behavior of the program. You will quickly notice that the program does not behave well at all. It may even exhaust system memory before completing. The reason has to do with the asynchronous message passing. If the producer sends messages, i.e., creates new elements in the stream, at a faster rate than the consumer can consume them, then increasingly more buffering will be needed until the system starts to break down.

The best way to solve this problem is to introduce some kind of flow control between producers and consumers, for example by doing lazy execution (as in the next section) or by adding a bounded buffer between the producer and consumer. Doing flow control is best because it works for all thread speeds without twiddling with any “magic numbers”. A different and inferior way is to change the relative priorities between the producer and consumer threads, so for example consumers get more time slices than producers. This way is inferior because it is fragile: its success depends on the amount of work needed for an element to be produced $w_p$ and consumed $w_c$. It succeeds only if the speed ratio $s_c/s_w$ between the consumer thread and the producer thread is greater than $w_c/w_p$. The latter depends not only on thread priorities but also on how many other threads exist.

---

Ironically, in the distributed computation model of Chapter 13, stream communication across sites works better because of a designed flow control mechanism that suspends producers when the network buffers are full.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
That said, let us fix the producer-consumer program of the previous section. We give the producer low priority and the consumer high priority. We also set the priority ratios to 10:1 and 10:1:

```{Property.put threads priorities(high:10 medium:10)}
local Xs S in
thread
  {Thread.setThisPriority low}
  Xs={Generate 0 150000}
end
thread
  {Thread.setThisPriority high}
  S={Sum Xs 0}
end
{Browse S}
end```

In our case, this will work since the time to consume an element is not 100 times greater than the time to produce an element. In the next section we provide another good solution, namely demand-driven (lazy) execution. However, lazy execution has a fairly large constant factor overhead. Later on in the book we will see how to do buffering, which is a good compromise between lazy and eager execution.

The general lesson is that changing thread priorities should never be used to get a program to work correctly. The program should already work correctly with reasonable resources, no matter what the priorities are. Changing thread priorities is a performance optimization; it should only be used to improve the performance of a program that is already working.

### 6.5.3 Managing resources and improving throughput

The throughput of a stream connection is the number of messages passed per unit of time. How can we increase the throughput of the producer-consumer connection, while keeping system resource consumption reasonable? The eager stream communication of Section 6.5.1 has a very high throughput, but it may use too much memory and computational resources. There are two goals:

- The first goal is to limit consumption of system resources. Solving this problem requires a mechanism for flow control, i.e., for decreasing or increasing the rate of message sending to satisfy some global criterium. For example, demand-driven execution is a type of flow control. Adding flow control requires that some information be sent back from the consumer to the producer, to tell the producer when to decrease or increase its sending rate. There are many ways to send the information back. One way that works well is to use a trigger, either a programmed trigger or an implicit trigger. Lazy programming, as explained in Section 6.6, is implemented with implicit triggers. In this section we will use programmed triggers.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Once limited resource consumption is achieved, the second goal is to maximize the throughput. This is usually done by adding a buffer. In a producer-consumer situation, a buffer is a third entity placed between the producer and the consumer. The producer generates several more elements than are immediately needed by the consumer. These elements are temporarily stored in the buffer until the consumer needs them. If the buffer has a maximum size then it is called a bounded buffer.

In the rest of this section, we show how to implement a bounded buffer in the case of producer-consumer communication with flow control. We do this in two steps:

- The first step is to modify the program of Section 6.5.1 by adding flow control. We will modify it to do demand-driven execution with a programmed trigger.

- The second step is to add a buffer between the producer and the consumer.

### Producer-consumer with flow control

We modify the producer and consumer so that the consumer can ask the producer whenever it needs an additional list element. The simplest way to do this is to use dataflow variables. The consumer binds the stream’s end to \( X | X_r \). The producer waits for this and binds \( X \) to the next list element. This is an example of a programmed trigger. With this technique, the producer and consumer are coded as follows:

```plaintext
proc {Generate N Xs}  
  case Xs of X|Xr then  
    X=N  
    {Generate N+1 Xr}  
  end  
end

fun {Sum Xs A Limit}  
  if Limit>0 then  
    X|Xr=Xs  
  in  
    {Sum Xr A+X Limit-1}  
  else A end  
end

local Xs S in  
  thread {Generate 0 Xs} end  
  thread S={Sum Xs 0 150000} end  
  {Browse S}
end
```

See the Exercises for an alternative way to implement programmed triggers.
6.5 Streams

Increasing throughput with a bounded buffer

Now that we have a producer-consumer connection with flow control, we can insert a bounded buffer to increase the throughput.

- write a bounded buffer with programmed triggers

6.5.4 Skipping over stream elements

In the previous examples, the consumer consumes all the elements that the producer produces. Sometimes the consumer needs to skip over stream elements that are already produced, to take only the latest one. For example, the stream might be a sequence of video frames. The consumer might want to skip quickly to the latest frame that was sent. This kind of stream management can be done with the IsDet operation introduced in Section 6.1.5. The test \( \{ \text{IsDet} \ \text{Xs} \} \) checks immediately whether Xs is already bound or not, without waiting when it is not bound. Using IsDet, we can define the function Skip that takes a stream and returns its unbound tail:

\[
\text{fun} \ \{ \text{Skip} \ \text{Xs} \} \\
\quad \text{if} \ \{ \text{IsDet} \ \text{Xs} \} \ \text{then} \\
\quad\quad \text{case} \ \text{Xs} \ \text{of} \ _|\text{Xr} \ \text{then} \ \{ \text{Skip} \ \text{Xr} \} \ [\] \ \text{nil} \ \text{then} \ \text{nil} \ \text{end} \\
\quad\quad \text{else} \ \text{Xs} \ \text{end} \ \text{end}
\]

This iterates down the stream until it finds an unbound tail. Here is a slightly different version that always waits until there is at least one element:

\[
\text{fun} \ \{ \text{Skip1} \ \text{Xs} \} \\
\quad \text{case} \ \text{Xs} \ \text{of} \ \text{X}|\text{Xr} \ \text{then} \\
\quad\quad \text{if} \ \{ \text{IsDet} \ \text{Xr} \} \ \text{then} \ \{ \text{Skip1} \ \text{Xr} \} \ \text{else} \ \text{Xs} \ \text{end} \\
\quad\quad \ [\] \ \text{nil} \ \text{then} \ \text{nil} \ \text{end} \ \text{end}
\]

With Skip1, the function Sum can be rewritten so that after it consumes an element, the next one it consumes is not the next one in the stream, but the most-recently calculated one:

\[
\text{fun} \ \{ \text{Sum} \ \text{Xs} \ \text{A} \} \\
\quad \text{case} \ \{ \text{Skip1} \ \text{Xs} \} \\
\quad\quad \text{of} \ \text{X}|\text{Xr} \ \text{then} \ \{ \text{Sum} \ \text{Xr} \ \text{A}+\text{X} \} \\
\quad\quad \ [\] \ \text{nil} \ \text{then} \ \text{A} \ \text{end} \ \text{end}
\]

Running this version of Sum in the example of Section 6.5.1 will return an integer smaller than 11249925000. The particular value depends on how the threads are scheduled. It can fluctuate greatly from one call to the next. Running it a few times, we get values from a low of 8372 to a high of 533252.
6.5.5 “Active objects” with streams

With stream communication we can write concurrent programs as networks of communicating “active objects”. In the concurrent declarative model, an active object is a recursive procedure that executes in its own thread. It communicates with other processes through stream arguments and keeps its internal state through accumulating parameters:

```
proc {ActiveObject S1 X1 T1}
  case S1
    of M|S2 then N T2 in
      X2={NextState M X1}
      T1=N|T2
      {ActiveObject S2 X2 T2}
    else skip end
  end
declare S0 X0 T0 in
thread
  {ActiveObject S0 X0 T0}
end
```

In this example, the procedure ActiveObject is a “template” for creating an active object. Executing the procedure in a thread creates the object with an input stream, S0, an output stream, T0, and an internal state, X0. The object reads messages from the input stream, does internal calculations, and sends messages on the output stream. In general, an object can have any fixed number of input and output streams.

Objects can be linked together in a graph, where each object receives messages from one or more other objects and sends messages to one or more other objects. For example, here is a pipeline of three active objects:

```
declare S0 T0 U0 V0 in
thread {ActiveObject S0 0 T0} end
thread {ActiveObject T0 0 U0} end
thread {ActiveObject U0 0 V0} end
```

The first object receives from S0 and sends on T0, which is received by the second object, and so forth.

The following example is a pipeline of active objects that implements the prime number sieve of Eratosthenes (see Figure 6.9). The first active object generates a stream of integers. Subsequent active objects take an input stream and remove the multiples of the stream’s first element. Procedure \{GenInt N Max S0\} generates a finite stream of integers S0 from N up to Max. Procedure \{FilterFirst N S0 S1\} takes the first element off its input stream S0, returns it in N, removes all multiples of this element, and outputs the resulting stream in S1. Procedure \{MakePipe K S0 S1 Out1\} makes a pipeline of K invocations

\[3\]This should not be confused with objects in the sense of object-oriented programming, which are introduced in Chapter 9.
of \texttt{FilterFirst} in their own threads, each taking its input from the preceding element in the pipeline. The pipeline input is $S_0$ and the output is $S_1$. The second output $Out_1$ includes the elements that were peeled off by the \texttt{FilterFirst} calls.

\begin{verbatim}
proc \{GenInt N Max S0\}
  if N=<Max then
    S0=N\{GenInt N+1 Max\}
  else S0=nil end
end

proc \{FilterFirst N S0 S1\}
  case S0 of M|T0 then
    N=M
    \{Filter T0 fun \{X\} X mod M \= 0 end S1\}
  else skip end
end

proc \{MakePipe K S0 S1 Out1\}
  if K>0 then Sm in
    \textbf{thread \{FilterFirst N S0 Sm\}} end
    Out1=N\{Out2\}
    \{MakePipe K-1 Sm S1 Out2\}
  else S1=S0 Out1=S0 end
end
\end{verbatim}

Using the higher-order procedure \texttt{Filter} makes it easy to write \texttt{FilterFirst}. Here is a sample execution with 11 active objects linked together:

\begin{verbatim}
declare S0 S1 Out in
\textbf{thread \{GenInt 2 1000 S0\}} end
\{MakePipe 10 S0 S1 Out\}
\{Browse Out\}
\end{verbatim}

This displays the first one thousand prime numbers. If the input list is not too large, then this example can be sped up by making it sequential, executing each pipe element to completion before starting the next one. This can be done by removing all thread creation from the code: replace each \texttt{thread \{S\}} end by the statement \{S\}. In this case, no deadlock is possible since each pipe element
depends only on the preceding one. In other words, the dependency graph has no cycles.

6.5.6 Limitations of stream communication

The active object model introduced in the preceding section seems to be quite powerful. Within a declarative framework, it seems to let us write networks of communicating active objects. However, the model has limitations that make it unusable for most networks of active objects, except in special cases. Let us see why by means of an example: a simple client/server application. The server has an input stream, from which it reads commands and performs them. Assume there is one client, which sends commands to the server. How can a second client connect to the server? The second client has to obtain a reference to a stream that it can bind and that is read by the server. The problem is that such a stream does not exist! There is only one stream, between the first client and the server. The second client cannot bind that stream, since it would conflict with the first client.

What is the source of this problem? It is due to an observable nondeterminism. Declarative programming is deterministic. If a program has observable nondeterminism, then it is not declarative. Let us consider the client/server application in this light. Consider a server with two clients. If the clients are independent, then the server does not know which client will be the next one to send it a command. This is an observable nondeterminism, so the program is not declarative. On the other hand, if the clients are dependent on each other, so that the server always knows which client will send the next command, then the program is declarative. This is because the nondeterminism inherent in the concurrency is not observable.

Figure 6.10 illustrates the problem: $\text{InS}$ is the server’s input stream and $\text{OutS}_1$ and $\text{OutS}_2$ are the two client’s output streams. How can the messages appearing on both client streams be given to the server? The simple answer is that in the concurrent declarative model they cannot! In the concurrent declarative model, an active object always has to know which stream it will read from next.
Chapter 10 explains how to extend the model to remove this limitation, at the
price of making the model no longer declarative. The limitation is removed by
adding a nondeterministic operation to the model.

6.6 Lazy execution

“All things spring up without a word spoken,
and grow without a claim for their production.”
– Tao-te Ching, Lao-tzu (6th century BC)

“Necessity is the mother of invention.”
“But who is the father?”
“Laziness!”
– freely adapted from a traditional proverb.

Up to now, we have seen statement sequences that execute in a well-defined or-
der, starting from the first statement and continuing in the order of the sequence.
(The order may be determined statically by textual sequence or dynamically by
dataflow synchronization.) When a statement becomes the first in the sequence,
it will execute whether or not its result is needed later on in the program. This
is called *eager execution* or *supply-driven execution*.

There is another execution order, in which a statement is only executed if its
result is needed somewhere else in the program. In the context of a functional
language, this is called *lazy evaluation* or *demand-driven evaluation*. For example,
take the following calculation:

\[
\begin{align*}
B &= \{F_1 \ X\} \\
C &= \{F_2 \ Y\} \\
A &= B + C
\end{align*}
\]

Assume that \(F_1\) and \(F_2\) are lazy functions. In lazy evaluation, as done by lazy
functional languages such as Haskell, the statements \(B = \{F_1 \ X\}\) and \(C = \{F_2 \ Y\}\)
are only executed when their results are needed, i.e., when the addition \(A = B + C\)
is invoked. At this point, the function calls are executed one after the other, in
sequential order.

In lazy evaluation, the calls are executed sequentially and the requesting cal-
culation continues when they have both completed. This semantics fits in a se-
quential computation model. In more general computation models, other forms
of laziness are possible. For example, the calls could be executed concurrently.
We use the general term *lazy execution* for all these forms, to distinguish them
from the special case of lazy evaluation.

Lazy execution is a powerful concept that can simplify many programming
makes a good case for the importance of higher-order programming and lazy
evaluation, as two important kinds of “glue” that can be used to modularize
programs [45]. He states that lazy evaluation “allows termination conditions
to be separated from loop bodies – a powerful modularisation”. He also states that lazy evaluation is “perhaps the most powerful glue functional programmers possess”. As we will see throughout the book, both higher-order programming and lazy evaluation are useful in many computation models, not just in functional programming.

This section presents the programming techniques that are made possible by lazy execution. We start by showing how to do lazy stream communication. We present some more applications, including list comprehensions, which gives a high level of abstraction to stream programming. We end with a discussion how to implement lazy execution.

**Laziness and dataflow**

We have seen another kind of glue in this chapter, namely dataflow execution. Both do synchronization, as explained in Section 6.4. It is interesting to compare them. Assume we invoke an operation. Then the two concepts seem quite similar:

- Dataflow execution suspends the operation until all arguments it needs are available.
- Lazy execution does the same thing, and in addition it triggers an execution to evaluate these arguments.

But their uses are very different. Dataflow is used to remove static ordering constraints: execute two statements concurrently, and dataflow will order them dynamically so that all is well. Laziness is used to reduce the amount of calculation: execute a statement giving a final result, and laziness will execute all those and only those statements that are needed for it. Laziness and dataflow can be used together. For example, a statement whose result is needed can be executed in its own thread, so that its ordering constraints with the rest are minimized. This is how lazy execution is implemented in the concurrent declarative model.

### 6.6.1 Lazy stream communication

In the producer-consumer example of Section 6.5.1, it is the producer that decides how many list elements to generate, i.e., execution is eager. This is a reasonable technique if the total amount of work is finite and does not use many system resources (e.g., memory or processor time). On the other hand, if the total work potentially uses many resources, then it may be better to use lazy execution. With lazy execution, the consumer decides how many list elements to generate. If an extremely large or a potentially unbounded number of list elements are needed, then lazy execution will use many fewer system resources at any given point in time. Problems that are impractical with eager execution can become practical with lazy execution. On the other hand, lazy execution may use many more total resources, because of the cost of its implementation. The need for laziness must take both of these factors into account.
Lazy execution can be implemented in two ways in the concurrent declarative model: with *programmed* triggers or with *internal* triggers. Section 6.5.3 gives an example with programmed triggers. Programmed triggers require explicit communications from the consumer to the producer. A simpler way is to use internal triggers, i.e., for the language to support laziness directly. In that case the language semantics ensures that a function is evaluated only if its result is needed. This makes the function definition simpler because it does not have to do the “bookkeeping” of the trigger messages. The concurrent declarative model has syntactic support for this technique: the function can be annotated as “lazy”. Here is how to do the previous example with a lazy function that generates a potentially infinite list:

```plaintext
fun lazy {Generate N}
    N|{Generate N+1}
end

fun {Sum Xs A Limit}
    if Limit>0 then
        case Xs of X|Xr then
            {Sum Xr A+X Limit-1}
        end
    else A end
end

local Xs S in
    thread Xs={Generate 0} end  % Producer thread
    thread S={Sum Xs 0 150000} end % Consumer thread
{Browse S}
end
```

As before, this displays 11249925000. With lazy execution it is the consumer *Sum* that decides how many list elements should be generated. The addition *A+X* implicitly triggers the generation of a new list element *X*. To see the difference in resource consumption between this version and the preceding version, try both with 150000 and then with 15000000 elements. With 150000 elements, there are no memory problems (on a personal computer with 64MB memory) and the eager version is faster. This is because of the overhead of the lazy version’s implicit triggering mechanism. With 15000000 elements, the lazy version needs only a very small memory space during execution, while the eager version needs a huge memory space. Lazy execution in the concurrent declarative model is implemented with the *ByNeed* function (see Section 6.6.7).

### Declaring lazy functions

In lazy functional languages such as Haskell, *all* functions are lazy by default. In contrast to this, the concurrent declarative model requires laziness to be declared explicitly, with the *lazy* annotation. We find that this makes things simpler both for the programmer and the compiler. Eager evaluation is several times
more efficient than lazy evaluation because there is no triggering mechanism. To get good performance in Haskell, this implies that the compiler has to determine which functions can safely be implemented with eager evaluation. This is called strictness analysis.

Multiple readers

The multiple reader example of Section 6.5.1 will also work with lazy execution. For example, here are three lazy consumers using the \texttt{Generate} and \texttt{Sum} functions defined in the previous section:

\begin{verbatim}
local Xs S1 S2 S3 in
  thread Xs={Generate 0} end
  thread S1={Sum Xs 0 150000} end
  thread S2={Sum Xs 0 100000} end
  thread S3={Sum Xs 0 50000} end
end
\end{verbatim}

Each consumer thread asks for stream elements independently of the others. If one consumer is faster than the others, then the others may not have to ask for the stream elements, if they have already been calculated.

6.6.2 Improving throughput

Just as in the eager case, we can also ask how to increase the throughput of the producer-consumer connection. Unlike in the eager case, lazy execution does not need an extra mechanism to manage resources; it already does flow control. To this we add a lazy buffer to improve throughput. A lazy buffer is a function that takes a lazy input stream \texttt{In} and an integer \texttt{N}, and returns a lazy output stream \texttt{Out}. Adding a lazy buffer to a lazy producer-consumer program can be done without changing the producer and consumer definitions in any way.

Defining a lazy buffer is a good exercise in lazy programming because it shows how laziness and declarative concurrency interact. Let us do the design in stages. We first specify the behavior of the buffer. When the buffer is first called, it first fills itself with \texttt{N} elements by asking the producer. Afterwards, whenever the consumer asks for an element, the buffer in its turn asks the producer for another element. In this way, the buffer always contains \texttt{N} elements. Figure 6.11 shows the resulting definition. The call \texttt{(List.drop In N)} drops \texttt{N} elements from the list \texttt{In}, giving \texttt{End}. This means that \texttt{End} always “looks ahead” \texttt{N} elements with respect to \texttt{In}. The lazy function \texttt{Loop} is executed whenever a list element is needed. This returns the next list element (\texttt{In.1}), but also asks the producer for one more element (by calling \texttt{End.2}). In this way, the buffer always contains \texttt{N} elements.

The buffer of Figure 6.11 is incomplete. It has two major problems. First, when the buffer is first called, then it cannot serve any consumer requests until the producer generates \texttt{N} elements. Second, when the buffer serves a consumer
request, then the answer cannot come until the producer has generated the next element. Because of these two problems, this buffer is not usable. A usable buffer should be able to serve requests as quickly as possible, i.e., as soon as it has at least one element.

To fix these problems, we have to determine why the buffer waits on the producer. In the definition of Buffer1, where are the producer requests generated? There are two places: (1) the call to List.drop, which can only complete when \( n \) elements are calculated, and (2) the call \( \text{End.2} \), which calculates the next element, starting from the \( n \)-th element. Putting a thread ... end around these calculations solves both problems. Figure 6.12 shows the fixed definition. This is a correct implementation of a lazy buffer that serves consumer requests as soon as the elements are calculated by the producer.

The buffer of Figure 6.12 works fine if the list \( \text{In} \) never ends. If it ends (i.e., terminates in nil), then the buffer raises an exception since \( \text{In.1} \) no longer exists. How do we modify the buffer to work with finite lists? Since \( \text{End} \) looks ahead, it is the first to see the end of the list. We can therefore do a test on \( \text{End} \). If the list is done then we just return what’s left. Figure 6.13 shows the resulting definition. This is a correct implementation of a lazy buffer that serves requests as soon as possible and that correctly handles finite input lists.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
fun {Buffer3 In N}
    End=thread {List.drop In N} end
fun lazy {Loop In End}
    case End
    of | | then
        In.1|{Loop In.2 thread End.2 end}
    [] nil then
        In
    end
in
    {Loop In End}
end

Figure 6.13: Lazy buffer (correct definition for finite lists)

Example execution

Let us see how this buffer works. We define a producer that generates an infinite list of successive integers, but only one integer per second:

fun lazy {Ints N}
    {Delay 1000}
    N|(Ints N+1)
end

Now let us create this list and add a buffer of 5 elements:

declare
    In={Ints 1}
    Out={Buffer3 In 5}
    {Browse Out}
    {Browse Out.1}

The call Out.1 requests one element. Calculating this element takes one second. Therefore, the browser first displays Out<Future> and one second later adds the first element, which updates the display to 1<Future>. The notation "<Future>" denotes a read-only variable. In the case of lazy execution, this variable has an internal trigger attached to it. Now wait at least 5 seconds, to let the buffer fill up. Then enter:

    {Browse Out.2.2.2.2.2.2.2.2.2.2}

This requests 10 elements. Because the buffer only has 5 elements, it is immediately emptied, displaying:

1|2|3|4|5|6|_<Future>

One more element is added each second for four seconds. The final result is:

1|2|3|4|5|6|7|8|9|10|<Future>

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
At this point, all consumer requests are satisfied and the buffer will start filling up again at the rate of one element per second.

### 6.6.3 Lazy reading of a file

There are two ways to read a large file. Either it can be read in all at once, which returns a list of characters, or it can be read incrementally in blocks, which returns a sequence of blocks. If the file is very large, then the second way uses much less active memory. However, it is cumbersome to program, since we have to manage the blocks. We would like to have the best of both worlds: read the file as if it were one very long list of characters, yet have it be read incrementally. We can achieve both goals with lazy evaluation. The following function returns a lazy list of the characters in a file:

```plaintext
fun {LazyRead FN}
    InFile={New Open.file init(name:FN)}
    fun lazy {LR} L T N in
        {InFile read(list:L tail:T size:1024 len:N)}
        if N==0 then T=nil {InFile close} else T={LR} end
        L
    end
    in
        {LR}
    end
end
```

The `LazyRead` function reads a file lazily in blocks of 1024 characters. Whenever a block is exhausted then another block is read. Reading blocks is much more efficient than reading single characters. The cost of laziness is spread over a whole block. This means that `LazyRead` is practically speaking just as efficient as doing the reads explicitly. The initial `LazyRead` call opens the file and calls LR. The latter is a lazy function. When part of the file is needed, LR is called and reads up to 1024 characters of the file. When the end of file is reached then the tail is bound to nil and the file is closed. Otherwise, LR is called recursively to bind the tail.

The `LazyRead` function is acceptable if the program reads all of the file, but if it only reads part of the file, then it is not good enough. Can you see why not? Think carefully before reading the answer in the footnote!\(^4\) Section 10.6.2 shows the right way to use laziness together with external resources.

### 6.6.4 The Hamming problem

The Hamming problem is a classic problem in demand-driven execution. The problem is to generate the first \(n\) integers of the form \(2^a3^b5^c\) with \(a, b, c \geq 0\). The idea is to generate the integers in increasing order in a potentially infinite stream.

\(^4\)It is because the file stays open during the whole execution—this consumes valuable system resources including a file descriptor and a read buffer.
At all times, a finite part $h$ of this stream is known. To generate the next element of $h$, we take the least element $x$ of $h$ such that $2x$ is bigger than the last element of $h$. We do the same for 3 and 5, giving $y$ and $z$. Then the next element of $h$ is $\min(2x, 3y, 5z)$. We start the process by initializing $h$ to have the single element 1. We can program this algorithm with two lazy functions. The first function multiplies all elements of a list by a constant:

\[
\textbf{fun} \ \text{lazy} \ \{\text{Mul} \ N \ H\}
\]

\[
\text{case} \ H
\]

\[
of \ X|H2 \ \text{then} \ N*X|\{\text{Mul} \ N \ H2\}
\]

\[
[] \ nil \ \text{then} \ nil
\]

\[
\end
\]

\[
\text{end}
\]

The second function takes two lists of integers in increasing order and merges them into a single list:

\[
\textbf{fun} \ \text{lazy} \ \{\text{Merge} \ Xs \ Ys\}
\]

\[
\text{case} \ Xs\#Ys
\]

\[
of \ nil\#Ys \ \text{then} \ Ys
\]

\[
[] \ Xs\#nil \ \text{then} \ Xs
\]

\[
[] \ (X|Xr)\#(Y|Yr) \ \text{then}
\]

\[
\text{if} \ X<Y \ \text{then} \ X|\{\text{Merge} \ Xr \ Ys\}
\]

\[
\text{elseif} \ X>Y \ \text{then} \ Y|\{\text{Merge} \ Xs \ Yr\}
\]

\[
\text{else} \ X|\{\text{Merge} \ Xr \ Yr\}
\]

\[
\end
\]

\[
\text{end}
\]

With these two functions, it is easy to solve the Hamming problem:

\[
\textbf{declare}
\]

\[
H=1|\{\text{Merge} \ \{\text{Mul} \ 2 \ H\} \ \{\text{Merge} \ \{\text{Mul} \ 3 \ H\} \ \{\text{Mul} \ 5 \ H\}\}\}
\]

\[
\{\text{Browse} \ H\}
\]

If we execute this as is, then it displays very little:

\[
1|<\text{Future}>
\]

No elements are calculated. To get the first $n$ elements of $H$, we need to \textit{ask} that they be calculated. For example, we can define the procedure \textit{Touch}:

\[
\textbf{proc} \ \{\text{Touch} \ N \ H\}
\]

\[
\text{if} \ N>0 \ \text{then} \ \{\text{Touch} \ N-1 \ H.2\}
\]

\[
\text{else} \ \text{skip} \ \text{end}
\]

This traverses $N$ elements of $H$, which causes them to be calculated. Now we can calculate 20 elements by calling \textit{Touch}:

\[
\{\text{Touch} \ 20 \ H\}
\]

This displays:

\[
1|2|3|4|5|6|8|9|10|12|15|16|18|20|24|25|27|30|32|36|<\text{Future}>
\]
6.6 Lazy execution

6.6.5 List operations

All the list functions of Section 5.7.1 can be made lazy. For example, here is a lazy version of Append:

```plaintext
fun lazy {LAppend As Bs}
    case As
    of nil then Bs
        [] A|Ar then A|(LAppend Ar Bs)
    end
end
```

The only difference with the eager version is the "lazy" annotation. The lazy definition works because it is recursive: it calculates part of the answer and then calls itself. Calling LAppend with two lists will append them lazily:

```plaintext
L={LAppend "foo" "bar"}
{Browse L}
```

If we "touch" L this will successively show f, o, o, one character at a time, and then it will show "bar" all at once. Why? Think before reading the answer in the footnote. How do we make a list append that returns a completely lazy result? One way is to give LAppend a lazy list as second argument. First define a function that takes any list and returns a lazy version:

```plaintext
fun lazy {MakeLazy Ls}
    case Ls
    of X|Lr then X|{MakeLazy Lr}
    else nil end
end
```

MakeLazy works by iterating over its input list, i.e., like LAppend, it calculates part of the answer and then calls itself. This only changes the control flow; considered as a function between lists, MakeLazy is an identity. Now call LAppend as follows:

```plaintext
L={LAppend "foo" {MakeLazy "bar"}}
{Browse L}
```

This will lazily enumerate both lists, i.e., it successively returns the characters f, o, o, b, a, and r.

Lazy mapping

We have seen Map in Section 5.8; it evaluates a function on all elements of a list. It is easy to define a lazy version of this function:

```plaintext
fun lazy {LMap Xs F}
    case Xs
    of nil then nil
        [] X|Xr then {F X}|{LMap Xr F}
    end
```

5It is because LAppend iterates over "foo" but not over "bar".

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
This version takes any list or lazy list $Xs$ and returns a lazy list.

Lazy integer lists

We define the function $\{LFrom \ I \ J\}$ that generates a lazy list of integers from $I$ to $J$:

$$\text{fun} \ \{LFrom \ I \ J\}$$
$$\text{fun} \ \text{lazy} \ \{LFromLoop \ I\}$$
$$\text{if} \ I>J \ \text{then} \ \text{nil} \ \text{else} \ I|\{LFromLoop \ I+1\} \ \text{end}$$
$$\text{end}$$
$$\text{fun} \ \text{lazy} \ \{LFromInf \ I\} \ I|\{LFromInf \ I+1\} \ \text{end}$$
$$\text{in}$$
$$\text{if} \ J=\text{inf} \ \text{then} \ \{LFromInf \ I\} \ \text{else} \ \{LFromLoop \ I\} \ \text{end}$$
$$\text{end}$$

Why is $LFrom$ itself not annotated as lazy? This definition allows $J=\text{inf}$, in which case an infinite lazy stream of integers is generated.

Lazy flatten

This definition shows that lazy difference lists are as easy to generate as lazy lists. As with the other lazy functions, it suffices to annotate as lazy all recursive functions that calculate part of the solution on each iteration.

$$\text{fun} \ \{Flatten \ Xs\}$$
$$\text{fun} \ \text{lazy} \ \{FlattenD \ Xs \ E\}$$
$$\text{case} \ Xs$$
$$\text{of} \ \text{nil} \ \text{then} \ E$$
$$[] \ X\r \ \text{then}$$
$$\{FlattenD \ X \ \{FlattenD \ Xr \ E\}\}$$
$$[] \ X \ \text{then} \ X|E$$
$$\text{end}$$
$$\text{end}$$
$$\text{in}$$
$$\{FlattenD \ Xs \ \text{nil}\}$$
$$\text{end}$$

We remark that this definition has the same asymptotic efficiency as the eager definition, i.e., it takes advantage of the constant-time append property of difference lists.

---

6Only recursive functions need to be controlled, since they would otherwise do a potentially unbounded calculation.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
6.6 Lazy execution

Lazy filter

fun lazy {LFilter L F}
  case L
     of nil then nil
     [X|L2] then
       if {F X} then X|{LFilter L2 F} else {LFilter L2 F} end
     end end

6.6.6 List comprehensions

List comprehensions are a powerful tool for calculating with lazy streams. They allow to specify lazy streams in a way that closely resembles the mathematical notation of set comprehension. For example, the mathematical notation \( \{ x \times y \mid 1 \leq x \leq 10, \; 1 \leq y \leq x \} \) specifies the set \( \{1 \times 1, 2 \times 1, 2 \times 2, 3 \times 1, 3 \times 2, 3 \times 3, \ldots 10 \times 10\} \), i.e. \( \{1, 2, 3, 4, 5, \ldots, 100\} \). We turn this notation into a practical programming tool by modifying it to specify not sets, but lazy streams. This makes the notation very efficient to implement, while keeping it at a high level of abstraction. For example, the list comprehension \([x \times y \mid 1 \leq x \leq 10, \; 1 \leq y \leq x]\) (notice the square list brackets!) specifies the list \([1 \times 1, 2 \times 1, 2 \times 2, 3 \times 1, 3 \times 2, 3 \times 3, \ldots, 10 \times 10]\) (in this order), i.e., the list \([1, 2, 4, 3, 6, 9, \ldots, 100]\). The list is calculated lazily. Because of laziness the list comprehension can generate a potentially unbounded stream, not just a finite list.

List comprehensions have the following basic form:

\[
[f(x) \mid x \leftarrow \text{generator}(a_1, \ldots, a_n), \; \text{guard}(x, a_1, \ldots, a_n)]
\]

The generator \(x \leftarrow \text{generator}(a_1, \ldots, a_n)\) calculates a lazy list whose elements are successively assigned to \(x\). The guard \(\text{guard}(x, a_1, \ldots, a_n)\) is a boolean function. The list comprehension specifies a lazy list containing the elements \(f(x)\), where \(f\) is any integer function and \(x\) takes on values from the generator for which the guard is true. In the general case, there can be any number of variables, generators, and guards. A typical generator is \textit{from}:

\(x \leftarrow \text{from}(a, b)\)

Here, \(x\) takes on the integer values \(a, \; a + 1, \; \ldots, \; b\), in that order. Calculation is done from left to right. The generators, when taken from left to right, are considered as nested loops: the rightmost generator is the innermost loop.

There is a close connection between list comprehensions and the relational programming of Section 7. Both provide lazy interfaces to infinitely long sequences and make it easy to write “generate-and-test” programs. Both allow to specify the sequences in a declarative way. The main difference between the two is that relational programs use nondeterminism, whereas list comprehensions are completely deterministic.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
While list comprehensions are usually considered to be lazy, they can in fact be programmed in both eager and lazy versions. For example, the list comprehension:

\[ z = [x\#x \mid x \leftarrow \text{from}(1,10)] \]

can be programmed in two ways. An eager version is:

\[
Z = \{\text{Map} \ \{\text{From} \ 1 \ 10\} \ \text{fun} \ \{\ X\} \ X\#X \ \text{end}\}
\]

For the eager version, the declarative model of Chapter 4 is good enough. It uses the \texttt{Map} and \texttt{From} functions of Section 5.7. A lazy version is:

\[
Z = \{\text{LMap} \ \{\text{LFrom} \ 1 \ 10\} \ \text{fun} \ \{\ X\} \ X\#X \ \text{end}\}
\]

The lazy version uses the \texttt{LMap} and \texttt{LFrom} functions of the previous section. This example and most examples of this section can be done with either a lazy or eager version. Using the lazy version is always correct. Using the eager version is a speed optimization. It is several times faster if the cost of calculating the list elements is not counted. The optimization is only possible if the whole list fits in memory. In the rest of this section, we always use the lazy version. In lazy functional languages, such as Haskell, all lists are lazy. These languages are usually implemented with strictness analysis to avoid paying the execution cost of lazy functions when the laziness is not needed.

Here is a list comprehension with two variables:

\[ z = [x\#y \mid x \leftarrow \text{from}(1,10), \ y \leftarrow \text{from}(1,x)] \]

This can be programmed as:

\[
Z = \{\text{LFlatten} \ \{\text{LMap} \ \{\text{LFrom} \ 1 \ 10\} \ \text{fun} \ \{\ X\} \ \{\text{LMap} \ \{\text{LFrom} \ 1 \ X\} \ \text{fun} \ \{\ Y\} \ X\#Y \ \text{end}\} \ \text{end}\}
\]

We have seen \texttt{LFlatten} in the previous section; it converts a list of lists to a “flat” lazy list, i.e., a lazy list that contains all the elements, but no lists. We need \texttt{LFlatten} because otherwise we have a list of lists. We can put \texttt{LFlatten} inside \texttt{LMap}:

\[
\text{fun} \ \{\text{FMap} \ L \ F\} \\
\{\text{LFlatten} \ \{\text{LMap} \ L \ F\}\}
\]

This simplifies the program:

\[
Z = \{\text{FMap} \ \{\text{LFrom} \ 1 \ 10\} \ \text{fun} \ \{\ X\} \\
\{\text{LMap} \ \{\text{LFrom} \ 1 \ X\} \ \text{fun} \ \{\ Y\} \\
X\#Y \ \text{end}\} \\
\text{end}\}
\]
Here is an example with two variables and a guard:

\[ z = [x\#y \mid x \leftarrow \text{from}(1,10), \ y \leftarrow \text{from}(1,10), \ x + y \leq 10] \]

This gives the list of all pairs \(x\#y\) such that the sum \(x + y\) is at most 10. It can be programmed as:

\[
Z = \{\text{LFilter} \\
\quad \{\text{FMap} \ {\text{LFrom} \ 1 \ 10} \ \text{fun} \ \{\$ \ X\} \\
\quad \quad \{\text{LMap} \ {\text{LFrom} \ 1 \ 10} \ \text{fun} \ \{\$ \ Y\} \\
\quad \quad \quad X\#Y \\
\quad \quad \end{end} \\
\quad \text{end} \\
\text{fun} \ \{\$ \ X\#Y \ X+Y<10 \ \text{end} \}
\]

This uses the function \text{LFilter} defined in the previous section. We can reformulate this example to be more efficient. The idea is to generate as few elements as possible. In the above example, 100 (=10*10) elements are generated. From \(2 \leq x + y \leq 10\) and \(1 \leq y \leq 10\), we derive that \(1 \leq y \leq 10 - x\). This gives the following solution:

\[ z = [x\#y \mid x \leftarrow \text{from}(1,10), \ y \leftarrow \text{from}(1,10-x)] \]

The program then becomes:

\[
Z = \{\text{FMap} \ {\text{LFrom} \ 1 \ 10} \ \text{fun} \ \{\$ \ X\} \\
\quad \{\text{LMap} \ {\text{LFrom} \ 1 \ 10-X} \ \text{fun} \ \{\$ \ Y\} \\
\quad \quad X\#Y \\
\quad \end{end} \\
\quad \text{end} \\
\text{fun} \ \{\$ \ X\#Y \ X+Y<10 \ \text{end} \}
\]

This gives the same list as before, but only generates about half as many elements.

### 6.6.7 By-need synchronization

Lazy execution, read-only variables, and dynamic linking are all implemented with a general mechanism called \textit{by-need synchronization} [64]. When a thread needs a variable binding to continue, then this mechanism initiates a calculation of the binding and synchronizes on the result. By-need synchronization is implemented with a by-need trigger, which is installed on \(X\) by calling the procedure \(\text{ByNeed} F X\). This installation completes immediately without waiting.

When \(X\) is needed, the by-need trigger is activated, which causes \(F\) to be executed in its own thread. When \(F\) completes, its result is bound to \(X\). Figure 6.14 shows the by-need protocol when one variable \(X\) is needed. During the evaluation of \(F\), the protocol blocks any other attempt to bind \(X\). This ensures that \(F\) is the first to bind \(X\). If more than one variable is needed, then the protocol calculates them concurrently. E.g., in the addition \(X+Y\) each of the two variables \(X\) and \(Y\) is calculated in its own thread. Chapter 15 defines the protocol formally.

A trigger can be activated only once. Once activated, it is removed from \(X\). For example, feed the following:
1. X is needed
2. Disallow binding X except by T2
3. Create T2
4. Evaluate Y={F}
5. Bind X=Y
6. Terminate T2
7. Allow binding X

Figure 6.14: The by-need synchronization protocol

```
declare
X={ByNeed fun (111*111) end}
{Browse X}
```

This will display X without calculating its value, since the browser does not need the value of X. To trigger the calculation of X, it is enough to do `{Wait X}`. This causes 12321 to be displayed.

By-need synchronization has nothing to do with a variable being determined, i.e., being bound to a value. It is simply a way to synchronize two calculations based on the condition of “need”. The by-need calculation is triggered when X is needed by an operation. A variable is needed by an operation in two ways: (1) if the operation cannot complete unless the variable is bound to a value, or (2) if the operation would bind the variable to something else than an unbound variable. The by-need calculation may bind X to any partial value, including an unbound variable. If X is bound to an unbound variable, then an operation in case (1) will still suspend after the by-need calculation is done.

**Implementing lazy evaluation with by-need**

The lazy functions of the previous sections, e.g.:

```
fun lazy {Generate N} N|{Generate N+1} end
```

can all be defined with ByNeed, with the following schema:

```
fun {Generate N}
    fun {F} N|{Generate N+1} end
in
    {ByNeed F}
end
```
Thanks to ByNeed, \( F \) is only called when the value of \{Generate N\} is needed. In the example of Generate, it is needed in the Sum function, in a `case` statement. Threads are cheap enough in Mozart that this semantic definition of lazy evaluation is practical.

### Implementing synchronous send with by-need

The following procedure implements a synchronous send operation, i.e., a binding that blocks until it is needed.

```mozart
proc {SynchSend X Y} Z in
  X={ByNeed fun {$} Z=unit Y end}
  {Wait Z}
end
```

Section 6.1.5 explains the role of the synchronous send operation.

### Implementing read-only variables with by-need

Read-only variables, as defined in Section 5.9.2, block any attempt to bind them until they are determined. This can be implemented with ByNeed and Wait. Creating a read-only variable can then be defined as follows:

```mozart
fun {ReadOnlyView X}
  {ByNeed fun {$} {Wait X} X end}
end
```

The call \( Y={ReadOnlyView Y} \) has the same behavior as \( Y=!!X \). If \( Y \) is a read-only view of \( X \), then trying to bind \( Y \) will do a \{Wait X\}, i.e., it will wait indefinitely until \( X \) is determined and then continue. If during this wait, another thread tries to bind \( Y \), then this thread will suspend as well, because the by-need calculation is still in progress.

If there is a by-need trigger installed on \( X \), then the \{Wait X\} will initiate the calculation of \( X \). That is, the read-only view does allow some information to be passed from \( Y \) to \( X \), namely whether the variable is needed.

### Implementing dynamic linking with by-need

Dynamic linking is a special case of lazy execution. We briefly explain what dynamic linking is all about and the role played by lazy execution. Dynamic linking is used to implement a general approach to structuring applications called component-based programming. More information can be found in Chapter 8 and [28]. Briefly, an application's source code consists of a set of component specifications, called `functors`. A running application consists of instantiated components, called `modules`. A module is represented by a record that groups together the module's operations. Each record field references one operation. Components are linked when they are needed, i.e., their functors are loaded into memory and
instantiated. As long as the module is not needed, then the component is not linked. When a program attempts to access a module field, then the component is needed and by-need synchronization is used to link the component.

### 6.7 Other programming techniques

#### 6.7.1 Coroutines

A coroutine is a nonpreemptive thread. A coroutine has its own sequence of instructions, like a thread, but it is the program that explicitly switches execution between coroutines. With threads, it is the system that automatically switches execution between threads. With coroutines, this is the program’s responsibility. Figure 6.15 compares coroutines with procedure calls and threads. Procedure calls are under program control, just like coroutines, but there is only one sequence of instructions. A procedure call transfers control once to the procedure body (the call), and then back (the return). At all times, there is only one sequence of instructions being executed.

Since coroutines introduce no nondeterminism, programs using them are still declarative. Coroutines have two operations, Spawn and Resume. The Spawn operation creates a new coroutine, which is like creating a new thread. The Resume operation transfers control from the current coroutine to another. Each coroutine has the responsibility to transfer control often enough so that the others have a chance to execute. If a coroutine does not do this, then the others have...
no chance to execute. This is called starvation and is usually due to programmer error. Starvation is not possible with threads if they are scheduled fairly.

Coroutines cannot be implemented in the concurrent declarative model because their implementation needs explicit shared state. This needs the concurrent stateful model of Chapter 10. We give the implementation of \texttt{Spawn} and \texttt{Resume} in Section 10.2.2.

Thread scheduling is often made controllable in ways that resemble coroutines. For example, we can introduce an operation similar to \texttt{Resume}, call it \texttt{Thread.preempt}, that immediately preempts a thread, i.e., switches execution to another runnable thread (if one exists). We can introduce operations to control whether a thread is allowed to execute or not. We call these operations \texttt{Thread.suspend} and \texttt{Thread.resume}.

### 6.7.2 Termination detection

We have seen how threads are forked using the \texttt{thread} statement. A natural question that arises is how to join back a forked thread into the original thread of control. That is, how can the original thread wait until the forked thread has terminated? This is a special case of detecting termination of multiple threads, and making another thread wait on that event. The general scheme is quite easy when using dataflow execution. Assume that we have \( n \) statements \( \langle S_1 \rangle, \ldots, \langle S_n \rangle \). Then the following code will execute each statement in a different thread and wait until they have all completed:

```plaintext
local X1 X2 X3 ... Xn1 Xn in
  thread \( \langle S_1 \rangle \) X1=unit end
  thread \( \langle S_2 \rangle \) X2=X1 end
  thread \( \langle S_3 \rangle \) X3=X2 end
  ...
  thread \( \langle S_n \rangle \) Xn=Xn1 end
  {Wait Xn}
end
```

This works by using the unification operation of dataflow variables (see Section 4.4.13). When all threads have terminated then the variables \( X_1, X_2, \ldots, X_n \) will be unified ("merged together") and bound to \texttt{unit}. The operation \{Wait \( X_n \)} suspends until \( X_n \) is bound.

A different way to detect termination does not depend on unification, but uses an auxiliary thread:

```plaintext
local X1 X2 X3 ... Xn1 Xn Done in
  thread \( \langle S_1 \rangle \) X1=unit end
  thread \( \langle S_2 \rangle \) X2=unit end
  thread \( \langle S_3 \rangle \) X3=unit end
  ...
  thread \( \langle S_n \rangle \) Xn=unit end
  thread
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
\begin{verbatim}
proc {Barrier Ps}
    proc {BarrierLoop Ps L R}
    case Ps of P|Pr then M in
        thread {P} L=M end
        {BarrierLoop Pr M R}
    |[] nil then L=R
    end
end

in
    {Wait {BarrierLoop Ps unit $}}
end
\end{verbatim}

Figure 6.16: Concurrent composition

\{Wait X1\} \{Wait X2\} \{Wait X3\} \ldots \{Wait Xn\}

Done=unit

end

{Wait Done}
end

If n is very large, the unification solution is faster.

\section*{6.7.3 Concurrent composition}

Figure 6.16 defines the combinator \texttt{Barrier} that implements concurrent composition. A \emph{combinator} is a procedure that defines a desired control structure in a generic way. \texttt{Barrier} takes a list of zero-argument procedures, starts each procedure in its own thread, and terminates after all these threads terminate. This does termination detection using the unification scheme of the previous section.

\section*{6.8 Soft real-time programming}

\subsection*{6.8.1 Basic operations}

The module \texttt{Time} contains a number of useful soft real-time operations. A \emph{real-time} operation has a set of deadlines (particular times) at which certain calculations must be completed. A \emph{soft real-time} operation requires only that the real-time deadlines be respected most of the time. This is opposed to \emph{hard real-time}, which has hard deadlines, i.e., that must be respected all the time, without any exception. Hard real time is needed when lives are at stake, e.g., in medical equipment and air traffic control. Soft real time is used in other cases, e.g., for telephony and consumer electronics. Hard real time requires special techniques for both hardware and software. Standard personal computers cannot do hard real time because they have unpredictable hardware delays (e.g., virtual memory,
6.8 Soft real-time programming

local

proc {Ping N}
    if N==0 then {Browse ‘ping terminated’}
    else {Delay 500} {Browse ping} {Ping N-1} end
end

proc {Pong N}
    {For 1 N 1
        proc {$ I} {Delay 600} {Browse pong} end
    }
    {Browse ‘pong terminated’}
end

in

{Browse ‘game started’}

thread {Ping 50} end

thread {Pong 50} end

end

---

Figure 6.17: A simple ’Ping Pong’ program

caching, process scheduling). Soft real time is much easier to implement and is often sufficient. Two soft real-time operations provided by Time are:

- {Delay I}: suspends the executing thread for at least I milliseconds and then continues.

- {Alarm I U}: creates a new thread that binds U to unit after at least I milliseconds. Alarm can be implemented with Delay.

We illustrate the use of Delay by means of a simple example that shows the interleaving execution of two threads. The program is called ’Ping Pong’ and is defined in Figure 6.17. It starts two threads. One displays ping periodically each 500 milliseconds and the other displays pong each 600 milliseconds. Because pongs come out slower than pings, it is possible for two pings to be displayed without any pongs in between. Can the same thing happen with two pongs? That is, can two pongs ever be displayed with no pings in between? Assume that the Ping thread has not yet terminated, otherwise the question would be too easy. Think carefully before reading the answer in the footnote.7

A simple standalone application

Section 5.12.2 in Chapter 4 has shown how to make standalone applications in Oz. To make the ’Ping Pong’ program standalone, the first step is to make a functor

---

7The language does indeed allow two pongs to be displayed with no intervening pings because the definition of Delay only gives the minimum suspension time. The thread suspending for 500 milliseconds can occasionally suspend for a longer time, for example for 700 milliseconds. But this is a rare occurrence in practice because it depends on external events in the operating system or in other threads.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
functor
import
  Browser(browse:Browse)
define
  proc {Ping N}
  if N==0 then {Browse 'ping terminated'}
  else {Delay 500} {Browse ping} {Ping N-1} end
  end
  proc {Pong N}
  {For 1 N 1
  proc {$ I} {Delay 600} {Browse pong} end }
  {Browse 'pong terminated'}
  end
in
  {Browse 'game started'}
thread {Ping 50} end
thread {Pong 50} end
end

Figure 6.18: A standalone 'Ping Pong' program

of it, as shown in Figure 6.18. If the source code is stored in file PingPong.oz, then the program can be compiled with the following command:

    ozc -x PingPong.oz

Type PingPong in your shell to start the program. To terminate this program in a Unix shell you have to type CONTROL-C.

The program of Figure 6.18 does not terminate properly when the Ping and the Pong threads terminated, because there was no mechanism for detecting when the threads terminate. We can fix this problem using the techniques of Section 6.7.2. Figure 6.19 adds a termination detection that terminates the main thread only when both the Ping and the Pong threads terminate. Then we use the call {Application.exit 0} to cleanly exit the application (which terminates any remaining threads in the system).

6.8.2 Ticking

We would like to invoke an action (e.g., send a message on a port, call a procedure, etc.) exactly once per second, giving it the local time as argument. We have three operations at our disposal: {Delay D}, which delays for at least D milliseconds, {OS.localTime}, which returns a record giving local time accurate to one second, and {Time.time}, which returns an integer giving the number of seconds since the current year started. How does the following function measure up:

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
functor

import

  Browser(browse:Browse)
  Application

define

  ...
  X1 X2

in

{Browse `'game started´}
thread {Ping 50} X1=unit end
thread {Pong 50} X2=unit end
{Wait X1} {Wait X2}
{Application.exit 0}

end

Figure 6.19: A standalone ‘Ping Pong’ program that exits cleanly

fun {NewTicker}
  TickStream
  proc {Loop S}
    {OS.localTime}|S1=S
  in
    {Delay 1000}
    {Loop S1}
  end
  in
    thread {Loop TickStream} end
  TickStream
end

This function creates a stream that grows by one element per second. To execute an action once every second, create a thread that reads the stream and performs the action:

    thread for X in {NewTicker} do {Browse X} end end

Any number of threads can read the same stream. The problem is, this solution is not quite right. It sends a message \textit{almost exactly} once per second. The problem is the “almost”. Every once in a while, one second is lost, i.e., successive elements on the stream show a difference of two seconds. However, there is one good point: the same second cannot be sent twice, since {Delay 1000} guarantees a delay of at least 1000 milliseconds, to which is added the execution of the instructions in Loop. This gives a total delay of at least 1000 + $\epsilon$ milliseconds, where $\epsilon$ is a fraction of a microsecond.

How can we correct this problem? A simple way is to compare the current result of OS.localTime with the previous result, and to add an element to the stream only when the local time changes. This gives:
fun {NewTicker}
    TickStream
proc {Loop S T}
    S1 T1={OS.localTime}
    in
        if T1\=T then S=T1|S1 else S=S1 end
        {Delay 900}
        {Loop S1 T1}
    end
    in
    thread {Loop TickStream {OS.localTime}} end
end

This version guarantees that exactly one tick will be sent per second, if {Delay 900} always delays for less than one second. The latter condition holds if there are not too many active threads and garbage collection does not take too long. One way to guarantee the first condition is to give the Loop thread high priority and all other threads medium or low priority. To guarantee the second condition, the program must ensure that there is not too much active data, since garbage collection time is proportional to the amount of active data.

This version has the minor problem that it "hesitates" every 9 seconds. That is, it can happen that {OS.localTime} gives the same result twice in a row, since the two calls are separated by just slightly more than 900 milliseconds. This means that the stream will not be updated for 1800 milliseconds. Another way to see this problem is that 10 intervals of 900 milliseconds are needed to cover 9 seconds, which means that nothing happens during one of the intervals. How can we avoid this hesitation? A simple way is to make the delay smaller. With a delay of 100 milliseconds, the hesitation will never be greater than 100 milliseconds plus the garbage collection time.

A better way to avoid the hesitation is to use synchronized clocks. That is, we create a free-running counter that runs at approximately one second per tick, and we adjust its speed so that it remains synchronized with the operating system time. Here is how it is done:

fun {NewTicker}
    TickStream
proc {Loop N S}
    S1 T={Time.time}
    in
        if T>N then {Delay 900}
        elseif T<N then {Delay 1100}
        else {Delay 1000} end
        S=N|S1
        {Delay 1000} end
        {Loop N+1 S1}
    end
    in

The loop has a counter, \( n \), that is always incremented by one. We compare the counter value to the result of \{Time.time\}. If the counter is slower (\( T>n \)), we speed it up. Likewise, if the counter is faster (\( T<n \)), we slow it down. The speedup and slowdown factors are small (10% in the example), which makes the hesitation unnoticeable.

### 6.9 Exercises

1. **Concurrent Fibonacci.** Consider the following recursive definition of the Fibonacci function:

   ```haskell
   fun {Fib X}
   if X=<2 then 1
   else
     {Fib X-1}+{Fib X-2}
   end
   end
   ```

   and compare it with the concurrent definition given in Figure 6.4. Run both on the Mozart system. How much faster is the sequential definition? How many threads are created by the concurrent call \{Fib N\} as a function of \( N \)?

2. **Order-determining concurrency.** Explain what happens when executing the following:

   ```haskell
   declare A B C D in
   thread D=C+1 end
   thread C=B+1 end
   thread A=1 end
   thread B=A+1 end
   {Browse D}
   ```

   In what order are the threads created? In what order are the additions done? What is the final result? Compare with the following:

   ```haskell
   declare A B C D in
   A=1
   B=A+1
   C=B+1
   D=C+1
   {Browse D}
   ```
Here there is only one thread. In what order are the additions done? What is the final result? What do you conclude?

3. **The Wait operation.** Explain why the \{Wait X\} operation could be defined as:

```plaintext
proc {Wait X}
    if X==unit then skip else skip end
end
```

Use your understanding of the dataflow behavior of the if statement.

4. **Thread scheduling.** Section 6.5.4 shows how to skip over already-calculated elements of a stream. If we use this technique to sum the elements of the integer stream in Section 6.5.1, the result is **much** smaller than 11249925000, which is the sum of the integers in the stream. Why is it so much smaller? Explain this result in terms of thread scheduling.

5. **Programmed triggers using higher-order programming.** Programmed triggers can be implemented by using higher-order programming instead of concurrency and dataflow variables. The producer passes a zero-argument function \(F\) to the consumer. Whenever the consumer needs an element, it calls the function. This returns a pair \(X\#F2\) where \(X\) is the next stream element and \(F2\) is a function that has the same behavior as \(F\). Modify the example of Section 6.5.3 to use this technique.

6. **Dataflow behavior in a concurrent setting.** Consider the function \{Filter In F\}, which returns the elements of In for which the boolean function \(F\) returns **true**. Here is a possible definition of Filter:

```plaintext
fun {Filter In F}
    case In
    of X|In2 then
        if \(F\ X\) then
            X|{Filter In2 F} else {Filter In2 F} end
        else
            nil
        end
    end
end
```

Executing the following:

```plaintext
{Show {Filter [5 1 2 4 0] fun \$ X\ X>2 end}}
```

displays:

```
[5 4]
```
So \texttt{Filter} works as expected in the case of a sequential execution when all the input values are available. Let us now explore the dataflow behavior of \texttt{Filter}.

- What happens when we execute the following:

\begin{verbatim}
declare A
{(Show {Filter [5 1 A 4 0] fun ($ X) X>2 end})}
\end{verbatim}

One of the list elements is a variable \texttt{A} that is not yet bound to a value. Remember that the \texttt{case} and \texttt{if} statements will block the thread in which they execute, until they can decide which alternative path to take.

- What happens when we execute the following:

\begin{verbatim}
declare Out A
thread Out={Filter [5 1 A 4 0] fun ($ X) X>2 end} end
{(Show Out)}
\end{verbatim}

Remember that calling \texttt{Show} displays its argument as it exists at the instant of the call. Several possible results can be displayed; which and why? Is the \texttt{Filter} function deterministic? Why or why not?

- What happens when we execute the following:

\begin{verbatim}
declare Out A
thread Out={Filter [5 1 A 4 0] fun ($ X) X>2 end} end
{Delay 1000}
{(Show Out)}
\end{verbatim}

Remember that the call \{Delay \texttt{N}\} blocks its thread for at least \texttt{N} milliseconds. During this time, other ready threads can be executed.

- What happens when we execute the following:

\begin{verbatim}
declare Out A
thread Out={Filter [5 1 A 4 0] fun ($ X) X>2 end} end
thread A=6 end
{Delay 1000}
{(Show Out)}
\end{verbatim}

What is displayed and why?

7. \textbf{Lazy execution.} This exercise looks closer at the concurrent behavior of lazy execution. Executing the following:

\begin{verbatim}
fun lazy {MakeX} {Browse x} {Delay 3000} 1 end
fun lazy {MakeY} {Browse y} {Delay 6000} 2 end
fun lazy {MakeZ} {Browse z} {Delay 9000} 3 end
\end{verbatim}

\begin{verbatim}
X={MakeX}
Y={MakeY}
\end{verbatim}
\[ Z = \{ \text{MakeZ} \} \]

\[ \{ \text{Browse } (X+Y)+Z \} \]
displays \(x\) and \(y\) immediately, \(z\) after 6 seconds, and the result 6 after 15 seconds. Explain this behavior. What happens if \((X+Y)+Z\) is replaced by \(X+(Y+Z)\) or by \textbf{thread} \(X+Y\) end \(+Z\)? Which form gives the final result the quickest? How would you program the addition of \(n\) integers to give the final result the quickest?

8. \textit{By-need synchronization}. Define an operation that requests the calculation of \(X\) but that does not wait.

9. \textit{List comprehensions}. Define a linguistic abstraction for list comprehensions (both lazy and eager) and add it to the Mozart system. Use the \texttt{gump} parser-generator tool documented in [57]. This exercise make take several days to complete.
Chapter 7

Relational Programming

Up to now, all our programs have been functional. That is, they had well-defined inputs and unique outputs. This was also true for procedures. We now generalize this functional programming to become relational. A relational computation does not necessarily have fixed inputs and outputs. When the computation starts, some of the relation’s arguments are known. The result of the computation is the other arguments. There can be more than one result for one input.

From the programmer’s point of view, relational programming is programming with an additional operation called “choice”. The “choice” operation nondeterministically picks one among a set of alternatives. During execution, the choice is implemented with search. We call this don’t know nondeterminism; see Section 6.1.2 for an explanation of this term and how it differs from the nondeterministic choice operation of Chapter 10. Introducing a choice statement is an old idea; Elcock [30] used it in 1967 in the Absys language and Floyd [31] explained it in the same year. The Prolog language used this idea at the core of its execution model in 1972 [22]. Floyd gives a lucid account of the execution model. He extends an Algol-like language with a function called \texttt{choice(n)}, which returns an integer from 1 to \textit{n}. He then shows how to implement a depth-first search strategy using flow charts to give the operational semantics.

Using the choice operation can make some programs much shorter and easier to understand. Deductive database algorithms and parsers, in particular, can use it fruitfully. It can also be used to explore complex combinatorial problems or to generate combinations. We have used it to automatically generate diagnostics for a RISC microprocessor, the VLSI-BAM [44, 99]. There exist many pure logic programs in Prolog that use it fruitfully. This includes DCG parsers for ambiguous grammars and David Warren’s WARPLAN planner [21].

Relational programming can easily lead to highly inefficient programs, if not used properly. This cannot be avoided in general since each new choice operation multiplies the search space by the number of alternatives. The size of the search space is exponential in the number of choice operations. However, relational programming is sometimes practical:

- **When the search space is small.** This is typically the case for database
applications. Another example is the above-mentioned VLSI-BAM diagnostics generator, which generated all combinations of instructions for register forwarding, condition bit forwarding, and branches in branch delay slots. This gave a total of about 70,000 lines of VLSI-BAM assembly language code. This was small enough to be used as input to the gate-level simulator.

- **As an exploratory tool.** If used on small examples, relational programming can give results even if it is impractical for bigger examples. The advantage is that the programs can be much shorter and easier to write: no algorithm has to be devised since search is a brute force technique that avoids the need for algorithms. This is an example of nonalgorithmic programming.

To use search in other cases, more sophisticated techniques are needed, e.g., powerful constraint-solving algorithms, optimizations based on the problem structure, and search heuristics. We leave these until Chapter 14. Here, we will study the use of nondeterministic programming as a tool for the two classes of problems for which it works well.

### 7.1 The declarative model with nondeterminism

To do relational programming, we add two concepts to the declarative model: the choice statement and the search module. These operations are not primitive; they are defined in terms of the constraint-based inferencing model of Chapter 14. But for now, let us consider them as primitive.

A relational program is defined by a one-argument procedure \( p \). The procedure's argument is bound to the answer. The procedure body is an arbitrary program that is allowed to use the choice statement. The choice statement gives a sequence of alternatives, where each alternative is any statement (see Table 7.1).

A relational program is executed with search. That is, the alternatives in the choice statements are selected one after the other until a "good" solution is found. Here, "good" means that some condition is satisfied. Choice combined with search is sometimes called don't know nondeterminism. An alternative is rejected when unification failure occurs, because this means that the alternative

---

Table 7.1: The choice statement

| (statement) ::= choice (inStatement) { \( \cdot \) (inStatement) } end | ...
| (expression) ::= choice (inExpression) { \( \cdot \) (inExpression) } end | ...

---

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
is inconsistent.\footnote{In the context of relational programs, unification failure does not raise an exception, but causes alternatives to be rejected.} A new alternative is then chosen. Unification failure occurs when trying to bind two different values, e.g., \(3=4\). This is an impossible operation, since values cannot change. Another way to get a unification failure directly is by executing the statement \texttt{fail}. See Section 4.4.13 for details.

To execute \(P\) in a relational way, we feed it to the \texttt{Search} module. This module implements \textit{search strategies}, that is, it decides how to pick the alternatives in the \texttt{choice} statements. The result is passed back to the calling program. We say that the \texttt{Search} module encapsulates relational programming and interfaces it with the declarative model. There are many possible search strategies. In this section, we will look at just a few:

- \{\texttt{Search.base.one }P\ Xs\} returns a singleton list \([X]\) containing the first solution \(X\) of the call \(\langle P\ X\rangle\) obtained by depth-first search. If there is no solution, \texttt{nil} is returned.

- \{\texttt{Search.base.all }P\ Xs\} returns the list of all solutions of \(P\) obtained by depth-first search.

- \{\texttt{Search.base.best }P\ \texttt{Ord }Xs\} returns a singleton list \([X]\) containing the \textit{best} solution of \(P\) with respect to the order function \texttt{Ord} (a one-argument function). A branch-and-bound search strategy is used. If there is no solution, \texttt{nil} is returned.

- \texttt{Search.object} is a class from which new search engines can be created. The following operations are possible:

  - \texttt{S=\{New Search.object script(P Ord)\}} creates a new search engine \(S\) for procedure \(P\). If the optional argument \texttt{Ord} (a one-argument function) is given, then a best solution search is done using a branch-and-bound strategy.

  - \{\texttt{S next(Xs)\}} returns a singleton list \([X]\) containing the next solution of \(P\). If no more solutions exist, \texttt{nil} is returned. If the search was stopped using a \texttt{stop} method, then \texttt{stopped} is returned.

  - \{\texttt{S last(Xs)\}} returns a singleton list \([X]\) containing the last solution of \(P\). If no more solutions exist, \texttt{nil} is returned. If the search was stopped using a \texttt{stop} method, then \texttt{stopped} is returned.

  - \{\texttt{S stop}\} stops the search engine at any time. This can be called while a search is executing. The search can be restarted by \texttt{next} and \texttt{last}.

The \texttt{choice} statement and \texttt{Search} module are compositional. That is, they can themselves do relational programming internally.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
These operations only scratch the surface of what the constraint-based computation model of Chapter 14 provides. Much more powerful choice and search possibilities exist; see Chapter 14 for explanations and examples. It is also possible to program one’s own choice and search abstractions using the primitives of the constraint-based computation model directly.

### 7.2 Simple examples

Here is a program that uses `choice` to count from 0 to 9:

```plaintext
fun {Digit}
    choice 0 [] 1 [] 2 [] 3 [] 4 [] 5 [] 6 [] 7 [] 8 [] 9 end
end
{Browse {Search.base.all Digit}}
```

This displays:

```
[0 1 2 3 4 5 6 7 8 9]
```

We can combine calls to `Digit` to count with more than one digit:

```plaintext
fun {TwoDigit}
    10*{Digit}+{Digit}
end
{Browse {Search.base.all TwoDigit}}
```

This displays:

```
[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 ... 98 99]
```

This shows what it means to do a depth-first search: when `two` choices are done, the program first makes the first choice and then makes the second. Here, the function chooses first the tens digit and then the ones digit. Changing the definition of `TwoDigit` to choose digits in the opposite order will give unusual results:
fun {StrangeTwoDigit}
  {Digit}+10*{Digit}
end

{Browse {Search.base.all StrangeTwoDigit}}

This displays:

\[0 10 20 30 40 50 60 70 80 90 1 11 21 31 41 \ldots 89 99\]

In this case, the tens digit is chosen second and therefore changes quicker than the
ones digit. Using \texttt{Digit}, we can already solve some interesting puzzles, like the
“palindrome product” problem. We would like to find all four-digit palindromes
that are products of two-digit numbers. A \emph{palindrome} is a number that reads
the same forwards and backwards, when written in decimal notation. The following
program solves the puzzle:

\begin{verbatim}
proc {Palindrome X}
  X=(10*{Digit}+{Digit})*(10*{Digit}+{Digit})  % Generate
  (X>0)=true
  (X>=1000)=true
  (X div 1000) mod 10 = (X div 1) mod 10  % Test 3
  (X div 100) mod 10 = (X div 10) mod 10  % Test 4
end
\end{verbatim}

{Browse {Search.base.all Palindrome}}

This displays all 118 palindrome products. See how much fun this is? In the next
sections, we give more serious applications of relational programming.

Palindrome product is an example of a \emph{generate-and-test} program: it generates
a set of possibilities and then it uses tests to filter out the bad ones. The tests use
unification failure to reject bad alternatives. Generate-and-test is a very naive
way to explore a search space. It generates \emph{all} the possibilities first and only
filters out the bad ones afterwards. In palindrome product, 10000 possibilities
are generated.

Chapter 14 introduces a much better way to explore a search space, called
\emph{propagate-and-search}. This approach does the filtering during the generation, so
that many fewer possibilities are generated. If we extend palindrome product
to 6-digit numbers then the naive solution takes 45 seconds.\footnote{On a 500 MHz Pentium III processor running Mozart 1.1.0.} The propagate-
and-search solution of Chapter 14 takes less than 0.4 second to solve the same
problem.

7.3 Programming techniques
7.3.1 Generate-and-test

Nondeterministic append and logic programming

In Section 5.11.1, we saw that the Append procedure in the declarative model has a logical semantics, but that the operational semantics is not able to realize this logical semantics for all patterns of inputs and outputs. In the declarative model, the operational semantics is deterministic (it gives just one solution) and directional (it works for only one pattern of input and output arguments). With relational programming, we can write programs with more flexible operational semantics. To see how, let us look again at the logical semantics of append:

\[
\forall a, b, c. \text{append}(a, b, c) \leftrightarrow (a = \text{nil} \land c = b) \lor (\exists x, a', c'. a = x \land a' \land c = x \land c' \land \text{append}(a', b, c'))
\]

How can we write a program that respects this logical semantics and is able to provide multiple solutions for the call \{Append X Y [1 2 3]\}? Look closely at the logical semantics. There is a disjunction (\lor) with a first alternative \((a = \text{nil} \land c = b)\) and a second alternative \((\exists x, a', c'. a = x \land a' \land c = x \land c' \land \text{append}(a', b, c'))\). To get multiple solutions, the program should be able to pick both alternatives. We can implement this by using a choice statement. This gives the following program:

```plaintext
proc {Append A B C}
  choice
    A=nil B=C
    [ ] As Cs X in
      A=X|As C=X|Cs {Append As B Cs}
  end
end
```

We can search for all solutions to the call \{Append X Y [1 2 3]\}:

```plaintext
{Browse {Search.base.all
  proc {$ S} X#Y=S in {Append X Y [1 2 3]} end}}
```

To get one output, we pair the solutions X and Y together. This displays all four solutions:

\[[\text{nil}#[1 2 3] [1]#2 3] [1 2]#3 [1 2 3]#\text{nil}\]

This program can also handle the directional cases, for example:

```plaintext
{Browse {Search.base.all
  proc {$ X} {Append [1 2] [3 4 5] X} end}}
```

displays \[[1 2 3 4 5]\] (a list of one solution). The program can even handle cases where no arguments are known at all, e.g., \{Append X Y Z\}. Since in that case there are an infinity of solutions, we create a search engine and ask for solutions lazily:

```plaintext
declare Engine={New Search.object
  script (proc {$ S} X#Y#Z=S in {Append X Y Z} end)}
```
We return a tuple containing all three arguments \((X\#Y\#Z)\). We can display solutions one by one by feeding:

\[
\{\text{Browse \{Engine next($)\}}\}
\]

This displays successive solutions:

- \([\text{nil}\#B\#B]\]
- \([[X1]\#B\#(X1|B)]\]
- \([[[X1\ X2]\#B\#(X1\|X2\|B)]\]
- \([[[X1\ X2\ X3]\#B\#(X1\|X2\|X3\|B)]\]

... 

This enumerates all possible solutions, in order of increasing length of the first argument. This can seem somewhat miraculous. It certainly seemed so to the first logic programmers, in the late 1960’s and early 1970’s. Yet it is a simple consequence of the semantics of the \texttt{choice} statement, which picks its two alternatives in order. Be warned that this style of programming, while it can sometimes perform miracles, is extremely dangerous. It is very easy to get into infinite loops, i.e., to generate solutions indefinitely, none of which is satisfactory. We advise you to write deterministic programs whenever possible and to use nondeterminism only in those cases when it is indispensable. And even then, verify that the solution you want is one of the enumerated solutions.

### 7.3.2 Elaborations of generate-and-test

### 7.3.3 Database queries

Relational programming is one of the most expressive techniques for querying deductive databases. It places no restrictions on the logical form of the query. If the query is highly disjunctive, then its execution can be extremely slow. But as an exploratory query tool, relational programming is hard to beat.

### 7.3.4 Parsing ambiguous grammars

- Be careful to distinguish the two kinds of ambiguity:
  - need for high lookahead to find the right parse
  - multiple correct parses for one input

Section 5.7.2 shows how to do parsing with a difference list. The grammar that is parses is a deterministic one with a lookahead of one token: it suffices to know the next token to know what grammar rule will apply. This is a very strong restriction in some cases. With relational programming, we can write a parser for extremely ambiguous grammars. In fact, relational programming is one of the most flexible grammar parsing formalisms that exists. It can parse grammars with absolutely no restriction on the form of the grammar. The disadvantage is that if the grammar is highly ambiguous, the parser can be extremely slow. But as an exploratory tool, it’s hard to beat.
7.3.5 Type checking

The type declarations of Section 5.9.1 are executable nondeterministic programs.

7.3.6 Lazy execution

It is easy to use Search.object as the heart of a lazy function. For example, here is how to use it to provide a lazy list of solutions:

```
fun {LazySearch P}
  S={New Search.object script(P)}
  fun lazy {Sols}
    case {S next($)}
    of [X] then X|{Sols}
    else nil end
  end
  in 
    {Sols}
  end
```

The search engine of Search.object corresponds to a programmed trigger, in the terminology of Section 6.1.4. The function LazySearch works like Search.base.all, except that it returns a lazy list. LazySearch does explicit calls to the search engine when it needs solutions. That is, it uses a programmed trigger to provide an implicit trigger.

7.4 Case studies

7.4.1 The $n$-queens problem

The $n$-queens problem consists of placing $n$ queens on an $n \times n$ chessboard so that no queen attacks another. There are many ways to solve this problem. The solution given in Figure 7.3 is noteworthy because it uses dataflow variables. We can get the first solution of an 8-queens problem as follows:

```
{Browse {Search.base.one fun {$} {Queens 8} end}}
```

This uses higher-order programming to define a zero-argument function from the one-argument function Queens. The answer displayed is:

```
[[1 7 5 8 2 4 6 3]]
```

How many solutions are there to the 8-queens problem (counting reflections and rotations as separate)? This is easy to calculate:

```
{Browse {Length {Search.base.all fun {$} {Queens 8} end}}}
```

This displays the number 92, which is the answer. Queens is not the best possible program for solving the $n$-queens problem. It is not practical for large $n$. Much
better programs can be gotten by using constraint programming or by designing specialized algorithms (which amounts almost to the same thing). But this program is simple and elegant.

How does this magical program work? We explain it by means of Figure 7.2. Each column, up diagonal, and down diagonal has one dataflow variable. The lists Cs, Us, and Ds contain all the column variables, up variables, and down variables, respectively. Each column variable “guards” a column, and similarly for the variables of the up and down diagonals. Placing a queen on a square binds the three variables to the queen’s number. Once the variables are bound, no other queen can bind the variable of the same column, up diagonal, or down diagonal. This is because a dataflow variable can only have one value. Trying to bind to another value gives a unification failure, which causes that alternative to be rejected.

The procedure PlaceQueens traverses a column from top to bottom. It keeps the same Cs, but “shifts” the Us one place to the right and the Ds one place to the left. At each iteration, PlaceQueens is at one row. It calls PlaceQueen, which tries to place a queen in one of the columns of that row, by binding one entry in Cs, Us, and Ds.

The Queens program was originally written in Prolog. We translated it into the relational computation model using the rules of Section 11.4. It has exactly the same logical and operational semantics as its Prolog equivalent.

Figure 7.2: The n-queens problem (when n = 4)
fun {Queens N}
  fun {MakeList N}
    if N==0 then nil else _|{MakeList N-1} end
  end

proc {PlaceQueens N Cs Us Ds}
  if N>0 then Ds2
    Us2=_|Us
    in
      Ds=_|Ds2
      {PlaceQueens N-1 Cs Us2 Ds2}
      {PlaceQueen N Cs Us Ds}
    else skip end
  end

proc {PlaceQueen N Cs Us Ds}
  choice
    Cs=N|_ Us=N|_ Ds=N|_
    [] |_Cs2=Cs _|Us2=Us _|Ds2=Ds in
      {PlaceQueen N Cs2 Us2 Ds2}
  end
in
  {PlaceQueens N Qs={MakeList N} _ _}
Qs
end

Figure 7.3: Solving the $n$-queens problem with relational programming

7.5 Exercises
Chapter 8

Encapsulated State

“L’état c’est moi.”
“I am the state.”
– Louis XIV (1638–1715)

“If functional programming is like a crystal, immutable and practically eternal, then stateful programming is organic: it grows and evolves as we watch.” – Inspired by On Growth and Form, D’Arcy Wentworth Thompson (1860–1948)

At first glance, encapsulated state is just a minor extension to declarative programming: in addition to depending on its arguments, the component’s result also depends on an internal parameter, which is called its “state”. This parameter gives the component a long-term memory, a “sense of history” if you will. Without state, a component has only short-term memory, one that exists during a particular invocation of the component. State adds a potentially infinite branch to a finitely running program. By this we mean the following. A component that runs for a finite time can only have gathered a finite amount of information. If the component has state, then to this finite information can be added the information stored by the state. This “history” can be indefinitely long, since the component can have a memory that reaches far into the past.

Oliver Sacks has described the case of people with brain damage who only have a short-term memory [78]. They live in a continuous “present” with no memory beyond a few seconds into the past. The mechanism to “fix” short-term memories into the brain’s long-term storage is broken. Strange it must be to live in this way. Perhaps these people use the external world as a kind of long-term memory? This analogy gives some idea of how important state can be for people. We will see that state is just as important for programming.

This chapter gives the basic ideas and techniques of using state in program design. Section 8.1 explains the difference between the general notion of state and the encapsulated state of this chapter. Section 8.2 explains the basic principles of system design and why state is an essential part of system design. It also gives
first definitions of component-based programming and object-oriented programming. Section 8.3 defines what we mean by state and introduces the stateful computation model. Section 8.4 explains how to build abstract data types with encapsulated state. The Mozart system provides a number of these types built-in, including ports, cells, dictionaries, and arrays, which are extremely useful. Section 8.5 introduces indexed collections, which are one of the most important stateful datatypes. Section 8.6 explains how to do iterative computation with state and how it differs from iterative computation in the declarative model. Section 8.7 explains how to reason about a stateful program. We explain a technique, the method of invariants, that can make this reasoning almost as simple as reasoning about declarative programs, when it can be applied. Section 8.8 shows a series of general programming techniques with state and compares the resulting programs with similar declarative programs. Section 8.9 explains component-based programming in depth. This is a basic program structuring technique that also comes in useful for object-oriented programming, which is a particular case of it.

Chapter 9 gives a case study of using encapsulated state to build a particularly rich and useful programming style, namely object-oriented programming. Because of the wide applicability of the object-oriented paradigm, we have devoted a full chapter to it.

A problem of terminology

Stateless and stateful programming are often called declarative and imperative programming, respectively. The latter terms are not quite right, but tradition has kept their use. Declarative programming, taken literally, means programming with declarations, i.e., saying what is to be done without saying how to achieve it. Taken at face value, this is close to the descriptive declarative model of Section 5.1. Imperative programming, taken literally, means to give commands, i.e., to say how to do something. In this sense, the declarative model of Chapter 4 is imperative too, because it defines sequences of commands. This book does not use these literal definitions but sticks to the traditional usage of declarative as stateless and and imperative as stateful.

There is an important sense in which declarative programming really is declarative according to the literal meaning. We explain this for logic and functional programming:

- A logic program can be “read” in two ways: either as a set of logical axioms (the what) or as a set of commands (the how). This is summarized by Kowalski’s famous equation Program = Logic + Control. The logical axioms, when supplemented by control flow information (either implicit or explicitly given by the programmer), give a program that can be run on a computer. Section 5.11.1 explains how this works for the declarative model.

- A functional program can also be “read” in two ways: either as a definition
8.1 What is state?

We have already programmed with state in the declarative model of Chapter 5. For example, the accumulating parameters of Section 5.7.1 are state. So why do we need a whole chapter devoted to state? To see why, let us look closely at what state really is. In its simplest form, we can define state as follows:

A state is a sequence of values in time that contains the intermediate results of a desired computation.

Let us examine the different ways that state can be present in a program.

8.1.1 Declarative state

The sequence need only exist in the mind of the programmer. It does not need any support at all from the computation model. This kind of state is called declarative state. As an example, look at the declarative function SumList:

```
fun {SumList Xs S}
  case Xs
    of nil then S
    [] X|Xr then {SumList Xr X+S}
  end
end
```

It is recursive. Each call has two arguments: Xs, the unexamined rest of the input list, and S, the sum of the examined part of the input list. While calculating the sum of a list, SumList calls itself many times. Taking the values of an argument at each call forms a sequence. This gives two quite natural sequences. For the call `SumList [1 2 3 4] 0` this gives

```
[1 2 3 4], [2 3 4], [3 4], [4], nil
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
for the first argument, and

\[ 0, 1, 3, 6, 10 \]

for the second argument. Each of these sequences is a state. When looked at in this way, \texttt{SumList} calculates with state. Yet neither the program nor the computation model “knows” this. The state is completely in the mind of the programmer.

### 8.1.2 Encapsulated state

It can be useful for a function to have a state that lives across function calls and that is hidden from the callers. For example, we can extend \texttt{SumList} to count how many times it is called. There is no reason why the function’s callers need to know about this extension. Even stronger: for modularity reasons the callers should not know about the extension. This cannot be programmed in the declarative model. The closest we can come is to add two arguments to \texttt{SumList} (an input and output count) and thread them across all the callers. To do it without additional arguments we need an encapsulated state:

An *encapsulated state* in a procedure is a state that exists over more than one procedure call without being present in the procedure’s arguments.

Encapsulated state cannot be expressed in the declarative model. To have it, we extend the model with a kind of container that we call a *cell*. A cell has a name, an indefinite lifetime, and a content that can be changed. If the procedure knows the name, it can change the content. If the name is known nowhere else in the program, then the successive contents form an encapsulated state. The declarative model extended with cells is called the *stateful model*. Unlike declarative state, encapsulated state is not just in the mind of the programmer. It is visible in both the program and the computation model.

By using a cell we can add a long-term memory to \texttt{SumList}. For example, let us keep track of how many times it is called:

```haskell
local
  C={NewCell 0}

in
  fun {SumList Xs S}
    {Assign C {Access C}+1}
    case Xs
    of nil then S
      [] X|Xr then {SumList Xr X+S}
    end
  end
end
```

This is the same definition as before, except that we define a cell and update its content in \texttt{SumList}. Briefly, \texttt{NewCell} creates a new cell with initial content 0.
The ability to have encapsulated state is very important. It removes the limits of declarative programming (see Section 5.14). It is at the heart of object-oriented programming, a powerful programming style that is presented in Chapter 9. The present chapter and Chapter 9 explore the ramifications of encapsulated state.

8.2 State and system building

The most successful system-building principle for entities with finite thinking abilities, such as human beings, is the principle of abstraction. Consider any system. It can be thought of as having two parts: a specification and an implementation. The specification is a contract, in a mathematical sense that is stronger than the legal sense. The contract defines how the rest of the world interacts with the system, as seen from the outside. The implementation is how the system is constructed, as seen from the inside. The miraculous property of the distinction specification/implementation is that the specification is usually much simpler to understand than the implementation. One does not have to know how to build a watch in order to read time on it. To paraphrase evolutionist Richard Dawkins, it does not matter whether the watchmaker is blind or not, as long as the watch works.

This means that it is possible to build a system as a concentric series of layers. One can proceed step by step, building layer upon layer. At each layer, build an implementation that takes the next lower specification and provides the next higher one. It is not necessary to understand everything at once.

How is this approach supported by declarative programming? With the declarative model of Chapter 4, all that the system “knows” is on the outside, except for the fixed set of knowledge that it was born with. To be precise, because a procedure is stateless, all its knowledge, its “smarts,” are in its arguments. The smarter the procedure gets, the “heavier” and more numerous the arguments get. Declarative programming is like an organism that keeps all its knowledge outside of itself, in its environment. Despite his claim to the contrary, this was exactly the situation of Louis XIV: the state was not in his person but all around him, in 17th century France.

How can we extend the declarative model so that new knowledge can be put inside the system? It turns out that it is sufficient to add encapsulated state to the model. This allows the system to add to the knowledge that it was born with. This opens up a world of new possibilities in system building. The smartness has

---

1 The only differences are a minor slowdown and a minor increase in memory use. In almost all cases, these differences are irrelevant in practice.

2To be fair to Louis, what he meant was that the decision-making power of the state was vested in his person.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
become independent of the arguments. They can be small in size and small in number, even for extremely smart and complex systems.

In this chapter, we add explicit state to a sequential computation model. This allows systems to grow and evolve. Because the model is sequential, though, there is a total order among all operations in the system. This cements a strong dependency between all parts of the system. Chapter 10 adds concurrency, which removes this dependency. This is the most general computation model introduced in the book. But it has problems of its own. Let us first investigate the sequential stateful model and not get ahead of ourselves.

8.2.1 System properties

What properties should a system have to best support the principle of abstraction? Here are three:

- **Encapsulation.** It should be possible to hide the internals of a part.
- **Compositionality.** It should be possible to combine parts to make new parts.
- **Instantiation/invocation.** It should be possible to create new instances of parts based on a single definition. These instances “plug” themselves into their environment (the rest of the system in which they will live) when they are created.

These properties do not require state; they can be used in declarative programming as well. For example, look at encapsulation. Encapsulation is orthogonal to state. On the one hand, it is possible to use encapsulation in declarative programs without state. We have already used it many times, for example in higher-order programming or in “active objects” in the concurrent declarative model. On the other hand, it is also possible to use state without encapsulation, by giving all components free access to the state.

It turns out that encapsulation and state are most useful when used together. Adding state to declarative programming makes reasoning about the program much harder. Bringing in encapsulation does much to make reasoning tractable again. This is because encapsulated stateful systems can be designed so that a particular property, called invariant, is always true when viewed from the outside. This makes reasoning about the system independent from reasoning about its environment. This gives us back one of the properties that makes declarative programming so attractive.

8.2.2 Component-based programming

The three properties of encapsulation, compositionality, and instantiation define component-based programming (see Section 8.9). A component specifies a program fragment with an inside and an outside, i.e., with a well-defined interface.
The inside is hidden from the outside, except for what the interface permits. Components can be combined to make new components. Components can be instantiated, making a new instance that is linked into its environment. Components are a ubiquitous concept. Here are two useful examples:

- We have seen a first example of components in the declarative computation model. The component is called a procedure definition and its instance is called a procedure invocation.

- A particularly useful kind of component is one that is a compilation unit, i.e., it can be compiled independently of other components. In this book, we call such components functors and their instances modules.

### 8.2.3 Object-oriented programming

A popular technique for stateful programming is called object-oriented programming (see Chapter 9). This essentially adds a fourth property to component-based programming:

- **Inheritance.** It is possible to build the system in incremental fashion, as a small extension or modification of another system.

Such components are called classes and their instances are called objects. Inheritance is a way of structuring programs so that a new implementation extends an existing one. But it is tricky: how can one extend an implementation by just fiddling with its interface and not changing the insides? This is what inheritance promises. In some cases this works. But extreme care is needed. Much of the literature on object-oriented design focuses on the correct use of inheritance.

Component-based programming also allows to extend an implementation. It consists of building a component that contains a subcomponent. The component offers a functionality and uses its subcomponent as part of its implementation. That’s it. We recommend to use component-based programming whenever possible and object-oriented programming in only those cases when inheritance is a good idea.

### 8.2.4 The importance of libraries

Now seems to be the right place to talk about libraries, just after we have introduced the two major abstractions used to program them: components and objects. Any programming language, to be practically useful, must be accompanied by a large set of useful abstractions. These are organized into libraries. A library is a coherent collection of one or more related abstractions that are useful in a particular problem domain. In Mozart, we consider a module to be a library. Depending on the language and the library, the library can be considered as part of the language or as being outside of the language. The dividing line can be quite vague: in almost all cases, many of a language’s basic operations are in fact
implemented in libraries. For example, higher functions on real numbers (sine, cosine, logarithm, etc.) are usually implemented in libraries.

In the future, the importance of libraries will continue to grow, fueled on the one side by the increasing speed and memory capacity of computers and on the other side by the increasing demands of users. A new language that does not come with a significant set of libraries, e.g., for network operations, graphic operations, database operations, etc., is either a toy, unsuited for real application development, or only useful in a narrow problem domain. Implementing libraries is a major effort. To alleviate this problem, new languages almost always come with an external language interface. This lets them communicate with programs written in other languages.

8.3 The declarative model with explicit state

One way to introduce state is to have components that run indefinitely and that can communicate with other components, like the “active objects” of Section 6.5.5. We call this the implicit state or encoded state approach. This approach has fundamental limits of expressiveness and efficiency, as explained in Sections 5.14 and 6.5.6. To overcome these limits, it is necessary to use another approach to introduce state, called explicit state or encapsulated state. This approach adds state directly to the computation model instead of “encoding” it. The result is called the stateful model. Explicit state is a mutable pair of two language entities. There exists an operation that when given the first entity returns the second. This operation defines a system-wide mapping between language entities. What makes it stateful is that the mapping can be modified. Interestingly, neither of the two language entities themselves needs to be modified.

8.3.1 Cells

We add explicit state as one new basic type to the computation model. We call the type a cell. A cell is a pair of a constant, which is a name value, and a reference into the single-assignment store. Because names are unforgeable, cells are a true abstract data type. The set of all cells lives in the mutable store. Figure 8.1 shows the resulting computation model. There are two stores: the immutable (single-assignment) store, which contains dataflow variables that can be bound to one value, and the mutable store, which contains pairs of names and references. Table 8.1 shows its kernel language. Compared to the declarative model, it adds just two new statements, the cell operations NewCell and Exchange. These operations are defined informally in Table 8.2. For convenience, this table adds two more operations, Access and Assign. These can be defined in terms of Exchange.

Amazingly, adding cells with their two operations is enough to build all the wonderful concepts that state can provide. All the sophisticated concepts of ob-
8.3 The declarative model with explicit state

U = Z.age  X = U + 1  if X < 2 then ...

W = atom
Z = person(age: Y)  X
Y = c1  U = c2
c1: W  c2: Z

Figure 8.1: The declarative model with explicit state

\[
\begin{array}{l}
\langle \text{statement} \rangle ::= \\
\quad \text{skip} \quad \text{Empty statement} \\
\quad \langle \text{statement} \rangle \langle \text{statement} \rangle \quad \text{Statement sequence} \\
\quad \text{local } X \text{ in } \langle \text{statement} \rangle \text{ end} \quad \text{Variable declaration} \\
\quad X = Y \quad \text{Bind to variable} \\
\quad X = f(l1:X1 ... ln:Xn) \quad \text{Bind to record} \\
\quad \text{if } X \text{ then } \langle \text{statement} \rangle \text{ else } \langle \text{statement} \rangle \text{ end} \quad \text{Test for boolean} \\
\quad \text{proc } \langle P X1 ... Xn \rangle \langle \text{statement} \rangle \text{ end} \quad \text{Procedure declaration} \\
\quad \text{try } \langle \text{statement} \rangle \text{ catch } X \text{ then } \langle \text{statement} \rangle \text{ end} \quad \text{Exception handler} \\
\quad \text{raise } X \text{ end} \quad \text{Raise exception} \\
\quad \{\text{NewCell } X \ C\} \quad \text{Cell declaration} \\
\quad \{\text{Exchange } C \ X \ Y\} \quad \text{Cell exchange}
\end{array}
\]

Table 8.1: The kernel language with explicit state

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{NewCell X C}</td>
<td>Create a new cell C with initial content X.</td>
</tr>
<tr>
<td>{Exchange C X Y}</td>
<td>Atomically unify X with the old content of</td>
</tr>
<tr>
<td></td>
<td>cell C and set Y to be the new content.</td>
</tr>
<tr>
<td>{Access C X}</td>
<td>Unify X with the content of cell C.</td>
</tr>
<tr>
<td>{Assign C X}</td>
<td>Set X to be the new content of cell C.</td>
</tr>
</tbody>
</table>

Table 8.2: Cell operations
objects, classes, and other abstract data types can be built with the declarative model extended with cells. Section 9.6.2 explains how to build classes and Section 9.6.3 explains how to build objects. In practice, their semantics are defined in this way, but the language has syntactic support to make them easy to use and the implementation has support to make them more efficient [41].

8.3.2 Relation to declarative programming

Sometimes it is possible to do stateful programming in a declarative model and vice versa, to do declarative programming in a stateful model. This is not always so; it depends on the structure of the program. This section investigates this relationship and the conditions imposed on programs to make it possible.

Stateful programming in a declarative model

It is possible to do stateful programming in a (sequential or concurrent) declarative model by encoding the state. The state is “threaded” as a pair of extra arguments to each procedure. The first argument is the input state and the second argument is the output state. In a concurrent setting, one thread at a time has exclusive access to the state, and this exclusive access right is explicitly passed from one thread to another. The declarative model cannot encode state that is shared between threads.

This encoding is cumbersome to write explicitly, because each instance of a stateful entity needs two extra arguments to be added to all procedures that use it, either directly or indirectly. It is possible to use an abstraction layer to hide the extra arguments (e.g., syntactic support with a preprocessor or library support). This technique is perfectly reasonable. It is used for writing large declarative Prolog programs [98, 54] and large functional programs (using monads [103]). If the abstraction layer can guarantee that old states are never used after new states are created, then it can be compiled efficiently with destructive assignment. However, this approach still has a major defect: it lacks the Model Independence Principle of Section 8.9.2, which is important for well-structured programs.

Encoding state in a declarative model has the limitation that the resulting model is still declarative. This limitation is not just a theoretical curiosity. There are many practical situations, such as client/server programs, when it makes a difference. A model that contains both explicit state and concurrency does not have this limitation, so it is strictly more expressive than a declarative model. As we will see in Chapter 10, such a model can have observable nondeterminism, which is not possible in the declarative model unless nondeterminism is added to the model [11].

Declarative programming in a stateful model (I)

It is possible to do declarative programming in a (sequential or concurrent) stateful model. The trivial way is to completely avoid using state. A more interesting
8.3 The declarative model with explicit state

approach is possible by using state in a restricted way. In a sequential stateful model there is one simple condition on how the state should be used to be declarative:

If after each state update operation the old state is never used again then we say that the program has threaded state, or is state threaded.
A sequential program that is state threaded is declarative.

This condition can often be verified by inspection. There is a simple way to transform a state threaded program into a program in the declarative model. It consists in transforming all the procedure definitions and procedure calls so that the state is explicitly threaded, like in the previous section. Consider a procedure call \{P X1 ... Xn\} and its definition:

```
proc \{P X1 ... Xn\}
...
{Exchange C A1 A2}
...
{Exchange C A3 A4}
...
{Exchange C A5 A6}
...
```

Assume that the procedure has an external reference to one cell C and that it does some exchange operations. Replace each call of P by the call \{Q X1 ... Xn In Out\}, which has two added arguments. In is the cell state when P starts and Out is the cell state when P finishes. Replace the definition of P by the following definition of Q:

```
proc \{Q X1 ... Xn In Out\}
...
In=A1
...
A2=A3
...
A4=A5 A6=Output
...
```

Q’s body is identical to P’s except that the Exchange calls are replaced by equalities as shown. At each exchange, the old content is bound with the input state and the new content becomes the output state.

We can extend this condition to a concurrent setting. Assume we have a state threaded program that is concurrent. At every computation step, if there is never a choice of more than one operation to change a given cell’s content, then the program is declarative. This is essentially the same condition as given in Chapter 6: the program should have no observable nondeterminism. All changes
in the store, i.e., for both dataflow variables and cells, are observable unless hidden by lexical scoping.

**Declarative programming in a stateful model (II)**

There is yet another way to do declarative programming in a stateful model. The idea is to have a declarative program that is *parameterized* by state. For example, a function can have an external reference to a cell. As long as the cell’s content is not changed, the function is declarative. Here is a simple example:

```plaintext
local
    C={NewCell 1}
in
    fun {Inc I}
        I+{Access C}
    end
end
```

The function `Inc` increments its input by 1, which is purely declarative as long as `C`’s content does not change. If `C`’s content is changed, then `Inc`’s behavior changes. If this happens during the execution of a program, then `Inc` is not declarative. But if it does not happen for some part of the execution, then `Inc` is declarative *in that part*. All of the reasoning techniques for declarative programs can be used in that part.

**Declarative programming in a stateful model (III)**

A third way is to use *memoization*, as explained in Section 8.9.2.

**8.4 Abstract data types**

As we saw in Chapter 5, an abstract data type is a set of values together with a set of operations on those values. In that chapter, we showed how to encapsulate data and write declarative ADTs. Now that we have introduced state, we can give a more general view of different kinds of ADTs.

**8.4.1 Different ADT structures**

While all ADTs have values and operations, there are different ways to *structure* an ADT. Each of these structures has different properties. This section explains four possible structures, which are a consequence of two independent choices to make when designing ADTs. The choices are whether it uses state and whether it bundles together or keeps separate the data and their operations. We explain these choices and give advice on the advantages and disadvantages of each structure.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
8.4 Abstract data types

Stateless and stateful ADTs

The first choice to make is whether the ADT is stateful:

- An ADT instance can be stateless. Chapter 5 gives examples: a declarative queue ADT and dictionary ADT. With this approach, ADT instances cannot be modified, but new ones must be created. When passing an ADT instance to a procedure, you can be sure about exactly what value is being passed. Once created, the instance never changes. On the other hand, this leads to a proliferation of instances that can be difficult to manage. The program is also less modular, since instances must be explicitly passed around, even through parts that may not need the instance themselves.

- An ADT instance can be stateful. Examples of stateful ADTs are components and objects, which are usually stateful. With this approach, ADT instances can change as a function of time. One cannot be sure about what value is encapsulated inside the instance without knowing what happened in all procedures that had access to the instance since its creation. On the other hand, there is only one instance. Furthermore, this one instance often does not have to be passed as a parameter; it can be accessed inside procedures by lexical scoping. This makes the program more concise. The program is also potentially more modular, since parts that do not need the instance do not need to mention it.

Bundled and unbundled ADTs

The second choice to make is whether the data is kept separate from the operations or whether they are bundled together:

- An ADT instance is unbundled when it separates the data from the operations. The instance is created together with a "key". The key is an authorization to access the internal data of an ADT (and update it, if the instance is stateful). All operations of the ADT know the key. It is possible to pass the key to other operations at any time during execution. This allows the program that uses the ADT to decide to extend its repertoire of operations at run time. To be precise, the program is divided into three parts: (1) the "inside" of the ADT, i.e., the implementation of the operations, (2) the part of the program that knows the key, and (3) the rest of the program. Usually the key is a name, which is an unforgeable constant (see Section B.2).

An advantage of encapsulating just the data is that it can be more efficient in time and space. For example, a file can contain just the data, without the operations. If the set of operations is very large, then this can take much less space than storing both the data and the operations. When the data is reloaded, then it can be used as before as long as the key is available.
An ADT instance can bundle together the data and the operations. As we will see in Chapter 9, this is what object-oriented programming does. Each object instance is bundled together with its operations, which are called “methods”. This approach is useful when we want to guarantee that the control of the repertoire of operations is inside the ADT and not in the program that uses the ADT (which is what the previous approach allows). If the ADT decides to have a fixed set of operations, then the program cannot change that. If the ADT decides to have an extensible set of operations, then it can do so. In object-oriented programming, there are two ways to extend the set of operations:

- First-class messages (Section 9.2.4). Objects can accept messages that are not part of their original definition and decide at run time what to do with them. Not all object-oriented languages support first-class messages.
- Inheritance (Section 9.3.1). With inheritance, the extensions are only available to new objects; existing objects have a fixed set of operations.

### 8.4.2 Encapsulation with chunks

The basic technique for building ADTs is to use secure encapsulation, i.e., to wrap the ADTs data inside a structure that can only be accessed and changed by authorized code. The authorization is done using lexical scoping together with name values. Lexical scoping guarantees that a reference will not leak outside a given area of program text. Using names guarantees that a reference cannot be forged. If forging were possible, it would compromise the encapsulation even if the reference does not leak outside.

The “wrapper” for ADT data is called a chunk. A chunk is simply a record whose label is a name and that does not have an \texttt{Arity} or \texttt{Label} operation. The label identifies the ADT. The fields of the chunk hold the encapsulated data of the ADT. If the features of the chunk are names, then they are completely hidden except to computations that know the name.

### 8.4.3 Security revisited

How the addition of state affects secure programs.

**Variations on a name**

Names are very useful, but they cannot be used outside of a computing system. Since they do not have an ASCII representation, they cannot be exchanged between programs except through purely electronic media. This is a strong limitation. Sometimes it can be useful to have name-like values that do have an ASCII representation. This can be done by using names, chunks, and lexical
We define a new kind of ‘name’ that can be transformed to an integer and vice versa. To keep the security properties, the integers (and their ASCII representations) should be well-guarded, e.g., only printed on documents handled by trusted middlemen. The transformation functions should be treated like capabilities: they should only be given out to trusted parties. For others, the new ‘names’ are just as impenetrable as built-in name values. Here is the definition as a stateful unbundled ADT:

```plaintext
local
    Key={NewName}
    Ctr={NewCell 0}

in
    fun (MyNewName) N1 N2 in
        {Exchange Ctr N1 N2} N2=N1+1
        {NewChunk foo(Key:N1)}
    end

    fun (ToInteger NN) NN.Key end
    fun (ToName N) {NewChunk foo(Key:N)} end
    fun (Equal N1 N2) N1.Key==N2.Key end
end
```

The function MyNewName works just like NewName, except that there also exist functions ToInteger and ToName to convert the ‘names’ to and from integers. Comparing ‘names’ can be done with the function Equal.

### 8.5 Indexed collections

An important ADT is the **indexed collection**, which groups a set of partial values into one compound entity. The individual values are accessible through an index. In the context of declarative programming, we have already seen two kinds of indexed collection, namely tuples and records. We can add state to these two data types. This allows them to be updated in certain ways. The stateful versions of tuples and records are called **arrays** and **dictionaries**.

In all, this gives four different kinds of indexed collection, each with its particular trade-offs between expressiveness and efficiency. With such a proliferation, how does one choose which to use? Section 8.5.3 compares the four and gives advice on how to choose among them.

#### 8.5.1 Arrays

An **array** is a mapping from integers to partial values. The domain is a set of consecutive integers from a lower bound to an upper bound. The domain is given
when the array is declared and cannot be changed afterwards. The range of the mapping can be changed. Both accessing and changing an array element are done in constant time.

If you need to change the domain or if the domain is not known when you declare the array, then you should use a dictionary instead of an array.

There is a close relationship between arrays and tuples. Each of them maps one of a set of consecutive integers to partial values. The essential difference is that tuples are stateless and arrays are stateful. A tuple has fixed contents for its fields, whereas in an array the contents can be changed. New tuples can be created with the `Adjoin` and `AdjoinAt` operations, but these copy the whole tuple, so they are not incremental. The `put` operation of an array is a constant time operation, and therefore much more efficient.

### 8.5.2 Dictionaries

A dictionary is a mapping from simple constants (atoms, names, or integers) to partial values. Both the domain and the range of the mapping can be changed. An item is a pair of one simple constant and a partial value. Items can be accessed, changed, added, or removed during execution. All operations are efficient: accessing and changing are done in constant time and adding/removal are done in amortized constant time. By amortized constant time we mean that a sequence of $n$ add or removal operations is done in time proportional to $n$, when $n$ becomes large. The memory needed by a dictionary is always proportional to the number of items in the mapping, with a small constant of proportionality. Other than system memory, there are no limits to the number of entries in the mapping. Section 5.7.1 gives some ballpark measurements comparing stateful dictionaries to declarative dictionaries. Operations on dictionaries are provided by the `Dictionary` module. Here are some of the most common operations:

- $D = \text{Dictionary.new}$ returns a new empty dictionary.
- $\{\text{Dictionary.put } D \ LI \ X\}$ puts in $D$ the mapping of $LI$ to $X$. This can also be written $D.LI := X$.
- $X = \text{Dictionary.get } D \ LI$ returns from $D$ the mapping of $LI$. This can also be written $X = D.LI$, i.e., with the same notation as for records.
- $X = \text{Dictionary.condGet } D \ LI \ Y$ returns from $D$ the mapping of $LI$, if it exists. Otherwise, it returns $Y$. This is minor variation of get, but it turns out to be extremely useful in practice.
- $\{\text{Dictionary.remove } D \ LI\}$ removes from $D$ the mapping of $LI$.
- $\{\text{Dictionary.member } D \ LI \ B\}$ tests in $D$ whether $LI$ exists, and binds $B$ to the boolean result.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Choose an indexed collection

- \( R = \{ \text{Dictionary.toRecord } L \ D \} \) returns a record with label \( L \) and the same items as the dictionary \( D \). The record is a “snapshot” of the dictionary’s state at a given moment in time.

- \( D = \{ \text{Record.toDictionary } R \} \) returns a dictionary with the same items as the record \( R \). This operation and the previous one are useful for saving and restoring dictionary state in pickles.

There is a close relationship between dictionaries and records. Each of them maps simple constants to partial values. The essential difference is that records are stateless and dictionaries are stateful. A record has a fixed set of fields and their contents, whereas in a dictionary the set of fields and their contents can be changed. New records can be created with the \texttt{Adjoin} and \texttt{AdjoinAt} operations, but these copy the whole record, so they are not incremental. The \texttt{put} operation of a dictionary is a constant time operation, and therefore much more efficient.

8.5.3 Choosing a collection

We have now seen many kinds of collections with different trade-offs in possible operations, memory use, and execution time. It is not always easy to decide which collection type is the best one in any given situation. This section examines the differences between these collections to make this decision easier. We first discuss indexed collections and then unindexed collections.

Indexed collections

We have seen four data types for indexed collections: tuples, records, arrays, and dictionaries. All provide constant-time access to their elements by means of indexes, which can be calculated at run time. But apart from this commonality
they are quite different. Figure 8.2 shows the principal differences. We compare them as follows:

- Tuples are the most restrictive. Their indexes are consecutive integers from 1 to a maximum, \( n \), which is specified when the tuple is created. They are useful for arrays where the contents do not have to be changed. Accessing a tuple entry is extremely efficient because the entries are stored consecutively.

- Records are more flexible than tuples because the indexes can be any literals and integers. The integers do not have to be consecutive. The record type, i.e., the label and set of indexes, is specified when the record is created. Accessing record entries is nearly as efficient as accessing tuple entries. To guarantee this, records entries are stored consecutively, like for tuples. This implies that creating a new record type (i.e., one for which no record exists yet) is much more expensive than creating a new tuple type. A hash table is created when the record type is created. The hash table maps each index to its offset in the record. To avoid having to use the hash table on each access, the offset is cached in the access instruction. Creating new records of an already-existing type is as inexpensive as creating a tuple.

- Arrays are more flexible than tuples because the contents of each entry can be changed. Accessing an array entry is extremely efficient because the entries are stored consecutively. The indexes are consecutive integers from any lower bound to any upper bound. The bounds are specified when the array is created.

- Dictionaries are the most general. They combine the flexibility of arrays and records. The indexes can be any literals and integers and the contents of each entry can be changed. Dictionaries are created empty. No indexes need to be specified. Indexes can be added and removed efficiently, in amortized constant time. On the other hand, dictionaries take more memory than the other data types (by a constant factor) and have slower access time (also by a constant factor).

As we will see, each of the four possibilities has its uses.

**Unindexed collections**

Indexed collections are not always the best choice. Sometimes it is better to use an unindexed collection. Two basic ones that we have seen are the list and the stream, which are both declarative data types that collect elements in a linear sequence. The sequence can be traversed from front to back. Any number of traversals can be done simultaneously on the same list or stream. Lists are of finite, fixed length. Streams are incomplete lists; their tails are unbound variables. This means they can always be extended, i.e., they are potentially unbounded. The stream is one of the most efficient extensible collections, in both memory

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
use and execution time. Extending a stream is more efficient than adding a new index to a dictionary and much more efficient than creating a new record type.

Streams are useful for representing ordered sequences of messages. This is an especially appropriate representation since the message receiver will automatically synchronize on the arrival of new messages. This is the basis of a powerful declarative programming style called stream programming (see Chapter 6) and its generalization to message passing (see Chapter 10).

8.6 Iterative computation

Declarative programming has to be done with care to guarantee that loops are iterative, i.e., that their “bookkeeping”, the management of the nested environments, takes up constant memory space. With state, on the other hand, loops can easily be done within a single environment, thus taking constant space. With state the distinction between iterative computation and recursive computation is no longer important.

8.6.1 Iterators

An iterator is an ADT for iterating over collections. This can be very powerful if higher-order programming is used.

8.7 Thinking about state

Programs that use state in a haphazard way are very difficult to understand. For example, if the state is visible throughout the whole program then it can be assigned anywhere. The only way to reason is to consider the whole program at once. Practically speaking, this is impossible for big programs.

The method of invariant assertions is a way of taming state. It allows to reason independently about parts of programs. This gets back one of the strongest properties of declarative programming. However, this property is achieved at the price of a rigorous organization of the program. The basic idea is to organize the program as a hierarchy of communicating ADTs. Each ADT has an invariant, which is a logical formula describing a relation among its arguments and its internal state. The ADT has an inside and an outside. The inside is responsible for making sure that the invariant is always true, when viewed from the outside. The outside uses the ADT and assumes that the invariant is always true. In this way, using the invariant decouples the inside from the outside.

The hierarchical organization of programs is good for more than just reasoning about programs. We will see it many times in the book. We find it again in the component-based programming of Section 8.9 and the object-oriented programming of Chapter 9.
To verify the truth of an invariant, we need to reason inside the ADT. We need to show that, if the invariant is true when entering the ADT, that the invariant is still true when exiting the ADT. This reasoning can be done in step-by-step fashion by using assertions. An assertion is a logical formula that is attached to a given point in the program, between two instructions. Assertions can contain variable and cell identifiers from the program as well as variables and quantifiers that do not occur in the program, but are used just for expressing a particular relation. For each instruction \( I \), we need to find an assertion \( A \) just before \( I \) and another assertion \( B \) just after \( I \) such that, if \( A \) is true and \( I \) is executed, then if this execution completes, \( B \) is true. We denote this by:

\[
\{ A \} I \{ B \}
\]

We call this a partial correctness assertion. \( A \) is called the precondition and \( B \) is called the postcondition. It is called partial because it does not say whether or not \( I \) terminates.

- Give a simple example.
- Analogy: this reasoning is like a 'symbolic execution' of a program, where the execution is done with algebraic formulas instead of actual values. One symbolic execution can summarize many 'real' executions and give useful information about them. Symbolic execution is more flexible than real execution. E.g., it does not have to be left-to-right. Give the example of an assignment statement, for which the right-to-left symbolic reasoning is easiest. In general, the symbolic execution can be done in any direction (left-to-right, right-to-left, top-down, bottom-up, or any mixture) as long as we end up with a contiguous chain of partial correctness assertions that are all correct according to their 'rules'.
- For each instruction \( I \) in the kernel language, we have a 'rule' that shows all possible correct forms of \( \{ A \} I \{ B \} \). We can prove that these rules are correct by using the semantics of Chapter 14.
- Explain how dataflow variables fit in this scheme.
- This scheme can be used for declarative as well as stateful programs.
- Practical tips: This technique depends on "guessing" the invariants. Usually, exploring the execution with small inputs is enough to find out what they must be. Loops use inductive reasoning similar to that of Chapter 4.
- Explain the difference between proving correctness and proving termination. These are two different properties that need different techniques. The above technique proves just correctness, not termination. To show termination, we need to show that execution always gets 'closer' to the result, i.e., there is some kind of 'distance' to the solution that always decreases. Often this is very simple, although in rare cases it can be tricky.
- Explain how to handle exceptions.
- For how to handle threads, see Chapter 9.
- Historical background: this technique is usually called 'axiomatic semantics'.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
It was invented by Floyd and Hoare in the 1960’s. Dijkstra also contributed. It is one of the most practical techniques for reasoning about stateful programs.

- The 'symbolic execution' analogy, when followed through, results in a powerful technique for program analysis called 'abstract interpretation'.

### 8.8 Programming with state

This section gives the basic techniques that complement the declarative techniques of Chapter 5 when state is added. We find that many algorithms can be efficiently written in a declarative way. This section concentrates on the differences between the declarative and stateful approaches.

#### 8.8.1 Stateful versus declarative programming

We compare the abilities of declarative and stateful programming, continuing the discussion of Section 5.14. For many problems, it is possible to write a declarative algorithm that has the same time and space complexity as a stateful algorithm. For example, Section 8.8.4 gives both a declarative and a stateful algorithm for the transitive closure problem. We find that the time complexity of both is the same, although the declarative version is harder to understand. This is because there are strong constraints on how it can be written in order to be both declarative and efficient: it has a “pipelined” structure, in which data passes through a series of stages. The stateful algorithm does not have this structure; it consists of three nested loops doing a straightforward calculation.

#### 8.8.2 Array algorithms

**In-place sorting**

An *in-place* algorithm is one that does all its calculations in the memory space allocated to store its result.

#### 8.8.3 Hash tables

#### 8.8.4 Transitive closure

Calculating the transitive closure is an example of a graph problem that can be solved reasonably well both with and without state. The problem is easy to state:

Consider any directed graph. Calculate a new graph, called the *transitive closure*, that has an edge between two nodes whenever the original graph has a path (a sequence of edges) between those same two nodes.
For example, Figure 8.6 shows a graph and its transitive closure. To solve this problem, we first choose a representation for the directed graph. Let us consider two possible representations:

- The **adjacency list** representation. The graph is a list with elements of the form $I#Ns$ where $I$ identifies a node and $Ns$ is an ordered list of its immediate neighbors. As we will see below, ordered lists of neighbors are more efficient to calculate with than unordered lists.

- The **matrix** representation. The graph is a two-dimensional array. The element with coordinates $(I,J)$ is true if there is an edge from node $I$ to node $J$. Otherwise, the element is false.

In what follows, we assume that all graphs have at least one node and that the nodes are consecutive integers. We give a first declarative algorithm that uses the adjacency list representation [70]. We then give an in-place stateful algorithm that uses the matrix representation [23]. We then give a second declarative algorithm that also uses the matrix representation. Finally, we compare the three algorithms.

**Converting between representations**

To make comparing the two algorithms easier, we first define routines to convert from the adjacency list representation to the matrix representation and vice versa. Here is the conversion from adjacency list to matrix:

```ocaml
fun {L2M GL}
  M={Map GL fun {§ I#_} I end}
  L={FoldL M Min M.1}
  H={FoldL M Max M.1}
in
  GM={Array.new L H unit}
  for X in GL do I#Ns=X in
    GM.I:={Array.new L H false}
    for J in Ns do
      GM.I.J:=true
    end
  end
end
```

In this routine, as in all following routines, we use $GL$ for the adjacency list representation and $GM$ for the matrix representation. Here is the conversion from matrix to adjacency list:

```ocaml
fun {M2L GM}
  L={Array.low GM}
  H={Array.high GM}
in
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
fun {DeclTrans InG}
  {FoldL InG fun {($ X V#_}
    {IncPath V {Find V X} X}
  end InG}

fun {IncPath V W InG}
  for X#Ns in InG collect:C do
    {C X# if {Member V Ns} then {Union Ns W} else Ns end}
  end
end

Figure 8.3: Transitive closure (first declarative version)

for I in L..H collect:C do
  {C I# for J in L..H collect:D do
    if GM.I.J then {D J} end
  end}
end

This uses the loop syntax including the accumulation procedure collect:C to good advantage.

Declarative algorithm

We first give a declarative algorithm for transitive closure. The algorithm needs two utility routines, for looking up neighbor lists in the adjacency list and for calculating the union of two ordered lists:

fun {Find V L}
  case L of X#W|L2 then
    if V==X then W else {Find V L2} end
  end
end

fun {Union A B}
  case A#B
    of nil#B then B
    [ ] A#nil then A
    [ ] (X|A2)#(Y|B2) then
      if X==Y then X|{Union A2 B2}
      elseif X<Y then X|{Union A2 B}
      elseif X>Y then Y|{Union A B2}
    end
  end
end

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
The **Find** routine assumes that \( V \) is always in the adjacency list, which is true in the algorithm given below. The **Union** routine’s execution time is proportional to the length of the smallest input list because its input lists are ordered. If the lists were not ordered, its execution time would be proportional to the product of their lengths (why?), which is usually much larger. Figure 8.3 gives the main algorithm. Again, this uses the loop accumulation procedure `collect:C` to good advantage. The function **IncPath** takes a partial transitive closure and adds those paths whose next-to-last edge is \( V \). The function **DeclTrans** just calls **IncPath** once for each edge. Note that the algorithm is written in such a way that only local list operations are needed at the top level, namely traversing a list and building a new list.

**Stateful algorithm**

We give a stateful algorithm for transitive closure. This algorithm assumes that the matrix contains the initial graph. It then calculates the transitive closure in-place, i.e., by updating the input matrix itself. Figure 8.4 gives the algorithm. For each node \( K \), this looks at each potential edge \((I, J)\) and adds it if there is both an edge from \( I \) to \( K \) and from \( K \) to \( J \).

**Second declarative algorithm**

Inspired by the stateful algorithm, we develop a second declarative algorithm. The second algorithm uses a sequence of tuples instead of an array to store the successive approximations of the transitive closure. A tuple is a record with fields numbered consecutively from 1 to a maximum value. Figure 8.5 gives the algorithm. This is somewhat more complicated than the stateful version. Each iteration of the outer loop uses the result of the previous iteration \((G_1)\) as input to calculate the next iteration \((G)\). The recursive function **Loop** passes the result from one iteration to the next.
8.8 Programming with state

fun {DeclTrans2 GM}
  H={Width GM}
  fun {Loop K G1}
    if K=<H then
      G={MakeTuple g H}
      in
      for I in 1..H do
        G.I={MakeTuple g H}
        for J in 1..H do
        end
      end
      {Loop K+1 G}
    else G1 end
  end
in
{Loop 1 GM}
end

Figure 8.5: Transitive closure (second declarative version)

Figure 8.6: A directed graph and its transitive closure

Example executions

Let us calculate the transitive closure of the graph \([1\#[2 3] 2\#[1] 3\#\text{nil}]\). Figure 8.6 shows the graph and its transitive closure. Here is how to use the declarative algorithms:

\[
\text{Browse \{DeclTrans [1\#[2 3] 2\#[1] 3\#\text{nil}]\}}
\]

Here is how to use the stateful algorithm:

\[
\text{declare GM in}
\{StateTrans GM={L2M [1\#[2 3] 2\#[1] 3\#\text{nil}]}}
\{Browse \{M2L GM\}\}
\]

This is slightly more complicated because of the calls to L2M and M2L, which we use to give both the input and output as an adjacency list. All three algorithms give the result \([1\#[1 2 3] 2\#[1 2 3] 3\#\text{nil}]\).
Discussion

Both the declarative and stateful algorithms are actually variations of the same conceptual algorithm, which is called the Floyd-Warshall algorithm. All three algorithms have an asymptotic running time of $O(n^3)$ for a graph of $n$ nodes. So which algorithm is better? Let us explore different aspects of this question:

- A first aspect is ease of understanding and reasoning. Perhaps surprisingly, the stateful algorithm is the easiest to understand: it consists of three simple nested loops that update a matrix in a straightforward way. Both declarative algorithms are harder to understand:
  - The first one takes an adjacency list and passes it through a sequence of stages (i.e., the calls to IncPath) in pipelined fashion. Each stage takes an input list and incrementally creates an output list.
  - The second one has a similar structure to the stateful algorithm, but creates a sequence of tuples.

Both declarative algorithms have a pipelined structure, passing values through a series of stages. This structure is typical of declarative algorithms.

- A second aspect is performance: running time and memory use. Both algorithms have the same asymptotic running time, but they may have different constant factors. We have measured the running times of both algorithms on several large random graphs. Given a random graph of 200 nodes in which there is an edge with probability $p$ between any node pair. For $p$ greater than about 0.05, the first declarative algorithm takes about 10 seconds, the second about 12 seconds, and the stateful algorithm about 15 seconds. For $p$ tending to 0, the first declarative algorithm tends towards 0 seconds and the other algorithms increase slightly, to 16 and 20 seconds, respectively.\(^3\) We conclude that the first declarative algorithm always has better performance than the two others. The adjacency list representation is better than the matrix representation when the graph is sparse.

Of course, the conclusions of this particular comparison are by no means definitive. We have chosen simple and clean versions of each style, but many variations are possible. For example, the first declarative algorithm can be modified to use a stateful \texttt{Union} operation. The stateful algorithm can be modified to stop looping when no more new edges are found. What then can we conclude from this comparison? First, that both the declarative and stateful models are reasonable. Second, that the choice of representation (adjacency list or matrix) can be more important than the choice of computation model.

\(^3\)All measurements using Mozart 1.1.0 under Red Hat Linux release 6.1 on a Dell Latitude CPx notebook computer with Pentium III processor at 500 MHz.
8.9 Component-based programming and system design

“Good software is good in the large and in the small, in its high-level architecture and in its low-level details.”
- Object-oriented software construction, 2nd ed., Bertrand Meyer (??)

“An efficient and a successful administration manifests itself equally in small as in great matters.”
- Memorandum, August 8, 1943, Winston Churchill (1874–1965)

What is the best way to build big applications? This is a big question. We will not do it justice here; that would take a whole book. If you are interested, we recommend [9] and [74] for their depth and insight. For component-based programming in particular, a good reference is Clemens Szyperski [91]. In this section we briefly introduce the main issues from the viewpoint of multiple computation models.

A large application is almost always built by a team. How should the team members partition the work amongst themselves? How should the team members communicate amongst themselves? The answers to these questions depend on the application’s structure. One way that works in practice is to structure the application as a hierarchical graph with well-defined interfaces at each level (see Figure 8.7). That is, the application consists of a set of nodes where each node interacts with some other nodes. Each node is an instance of a component. Each

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
node is itself a small application, and can itself be decomposed into a graph. The decomposition bottoms out when we reach primitive components that are provided by the underlying system.

When developing this application, each team member is responsible for a component. Each team member communicates with a well-defined set of other team members, depending on whether his component interacts with their components. Each component has an interface, which is visible from other components, and an implementation, which is invisible. Team members communicate at the component interface.

8.9.1 Components in general

It is easy to define components and component instances as a new abstraction. A component is a template from which one can create instances by specifying which other instances they need. For example, a component can be implemented as a record that contains the component name, a list naming the component instances it needs, and a higher-order procedure that returns a component instance when called with the component instances that it needs. An instance of a component is the set of language entities that are created when the component is instantiated. An interface is the set of all externally-visible entities of the component.

Components can reference each other in cyclic fashion, as Figure 8.7 illustrates. This is not a problem; it simply means that each component needs the other to be implemented. As long as the recursion bottoms out at run time, all is well.

8.9.2 System decomposition and computation models

At each level in the hierarchy, the system consists of a set of interacting components. Each component is written in a particular computation model. Section 5.2 introduces the major computation models used to program components. During development, a component’s internal structure may change drastically. It is not uncommon for its computation model to change. A stateless component can become stateful (or concurrent, or distributed, etc.), or vice versa. If such a change happens inside a component, then it is not necessary to change its interface. The interface needs to change only if the externally-visible functionality of the component changes. This is an important modularity property of the computation models. As long as the interface is the same, this property guarantees that it is not necessary to change anything else in the rest of the system. We consider this property as a basic design principle for the computation models:
Model independence principle

The interface of a component should be independent of the computation model used to implement the component. The interface should depend only on the externally-visible functionality of the component.

A good example of this principle is memoization. Assume the component is a function that calculates its result based on one argument. If the calculation is time consuming, then keeping a cache of argument-result pairs can greatly reduce the execution time. When the function is called, check whether the argument is in the cache. If so, return the result directly without doing the calculation. If not, do the calculation and add the new argument-result pair to the cache. Section ?? has an example of memoization. Since the memoization cache is stateful, changing a component to do memoization means that the component changes from using the declarative model to using the stateful model. The model independence principle implies that this can be done without changing anything else in the program.

All the computation models of this book are designed to satisfy this principle, if the models are not used to encode the abilities of more expressive models. With this principle, the system can be decomposed no matter which computation models it uses. It is therefore possible to develop each component and reason about it independently of its siblings. The reasoning techniques used depend on which computation model the component is implemented in. A component written in a declarative model is similar to a functional program. The reasoning techniques of Section 5.5 can be used. A component written in a stateful model is different. The techniques of Section 8.7 can be used.

8.9.3 What happens at the interface

The power of a component-based infrastructure depends in large part on the expressiveness of the interface between components. How does one component communicate with another? How do components “converse”, i.e., exchange a sequence of messages with a well-defined purpose? What kinds of information do they exchange?

- If the components are written in the same language using the same computation model then the interface is very simple. For example, an Oz component instance, i.e., a module, is a record whose fields contain the different parts of the module. To use the module, it suffices to have access to the record.

- If the components are written in different languages, but with similar computation models, e.g., both are object-oriented languages, then the interface is also simple. Its main work is to translate between the two language syntaxes. A popular example is the CORBA IDL (Interface Definition Language), which defines a correspondance from one language to another. The languages communicate as if they had a common object-based execution
model. CORBA also adds a distribution aspect, i.e., the two components can be running on different machines.

- The hard case is when the two components are written in very different computation models. In this case, the interface has to do real work. It not only translates data representations and syntax, but if well-designed it can allow each language to access the other while staying within its own computation model.

In what follows, we focus on the hard case since it subsumes the others. To understand the issues, let us look at a simple example. Assume that we have two components, C and S, where C is written in a concurrent model and S in a sequential model. They communicate through an interface. How can this work? Assume S wants to run a query on the C side. Should each of S’s queries create a new thread on the C side? Or should all of S’s queries run in the same thread on the C side? A choice has to be made. Sometimes no simple choice is possible. For example, assume an existing program written in C that S must interact with. In that case, the solution seems to be that S’s model must encode C’s model. That is, S’s program must do some explicit scheduling to handle the concurrency coming in from the C side. For example, if C wants to call S concurrently from several threads, then on the S side, these calls are put in a table, which the S side is free to handle sequentially. From the C point of view, it is as if S is also concurrent, to some degree. S’s apparent concurrency might be different from C’s, e.g., scheduling might be nonpreemptive. But the differences are manageable.

We discuss separately the problem of control structures and data structures at the interface.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Control structures at the interface

If one model is a strict subset of the other, it is not a problem. The problem is if each model has features that do not exist in the other. A good example is interfacing Prolog with Java (see Figure 8.8). Prolog is a sequential language with global backtracking. Java is a concurrent language.\(^4\) There are two basic approaches to interface them:

1. The interface handles only the common subset. For example, a Prolog program calls the sequential subset of Java (or vice versa).

2. The interface provides an encoding of the extra expressiveness. For example, a Java program can query Prolog with the added ability to ask for further solutions to the query. Or, inversely, a Prolog program could explicitly start Java threads and do thread operations.

Both approaches can be expressed in Oz, e.g., using the relational model of Chapter 7 for the Prolog side and the concurrent stateful model of Chapter 10 for the Java side.

There are many other possibilities. Here are six of the most popular organizations, in increasing order of component independence:

1. **Sequential.** The first organization is where the application is sequential and where one component calls the other like a procedure. The caller is not necessarily the only initiator; there can be nested calls where the locus of control passes back and forth between components. But there is only one global locus of control, which strongly ties together the two components.

2. **Coroutine.** A second organization is where two components each evolve independently, but still in a sequential setting. This introduces the concept of a coroutine (see Section 6.7.1). A component calls another, which continues where it left off. There are several loci of control, one for each coroutine. This organization is looser than the previous one, but the components are still dependent because they execute in alternation.

3. **Concurrent and synchronous.** A third organization is one in which each component evolves independently of the others and can initiate and terminate communications with another component, according to some protocol that both components agree on. The components execute concurrently. There are several loci of control, called threads, which evolve independently (see Chapters 6 and 10). Each component still calls the others synchronously, i.e., each call waits for a response.

4. **Concurrent and asynchronous.** A fourth organization is a set of concurrent components that communicate through asynchronous channels. Each

\(^4\)We simplify slightly, e.g., since Prolog has coroutining and allows programming with state, but the comparison is essentially correct.
component sends messages to others, but does not have to wait for a response. The channels can have FIFO order (messages received in order of sending) or be unordered. The channels are called streams in Chapter 6 and ports in Chapter 10. In this organization each component still knows the identity of the component with which it communicates.

5. **Concurrent mailbox.** A fifth organization is a variation of the fourth. The asynchronous channels behave like mailboxes. That is, it is possible to do pattern matching to extract messages from the channels, without disturbing the messages that remain. This is very useful for many concurrent programs. It is a basic operation in the Erlang language, which has FIFO mailboxes (see Chapter 11). Unordered mailboxes are also possible.

6. **Coordination model.** A sixth organization is where the components can communicate without the senders and receivers knowing each other’s identities. An abstraction called tuple space lives at the interface. The components are concurrent and interact solely through the common tuple space. One component can insert a message asynchronously and another can retrieve the message. Section 10.4.7 defines one form of tuple space abstraction and shows how to implement it.

**Data structures at the interface**

There are many levels of expressiveness for data structures that cross an interface:

- The first, simplest case is where only non-compound non-procedural values are sent across. For example, early Sun RPC (Remote Procedure Call) allowed integers, floating point numbers, and character strings. This is very limiting and cumbersome; it requires the application to do most of the encoding and decoding.

- A second case allows compound structures (pointer-based structures containing integers, floats, and strings). If the components are in different processes, this requires translating the pointers between the two components.

- A third case allows procedures and functions to cross the interface. A simple case of this is the “call-back”. A more advanced case is where the program fragment keeps a reference to its origin (back across the interface!) through lexical scoping (see next case).

- A fourth case allows compound structures that are spread over several components. The interface can translate pointers to “network pointers” that cross the interface. The distribution model of Chapter 13 uses this technique.
8.9 Component-based programming and system design

Embedding control in data

In addition to these levels of expressiveness, data structures can also specify control flow. Two important techniques for embedding control in any data are dataflow execution and lazy execution (see Chapter 6). Dataflow execution implicitly suspends a calculation until its operands are available. Lazy execution implicitly initiates a calculation when it is needed. Both can be used across an interface. Both allow to decouple components in an even stronger way than the previous techniques: they allow control to be dynamic (i.e., determined at run time by the data structures themselves) and more fine-grained (i.e., the control flow appears inside data structures).

Exceptions

If a component cannot satisfy its specification, then one way it can react is by raising an exception across the interface. This approach works well for synchronous interfaces. For asynchronous interfaces it is usually unworkable since the calling component will continue executing between the asynchronous invocation and the exception reply. Because the exception can happen at any time, this puts the component in an unpredictable situation. A workable alternative is to have a return argument in the invocation. The argument is bound to indicate whether the invocation has finished in a normal or an exceptional situation. The caller can use this argument any way it likes; it can even do a synchronous invocation by waiting for the argument to be bound.

Memory and computational resources

Components often share memory and computational resources. These form a kind of covert interface between the components. This interface can be tight or loose:

- If the components are in different processes or on different machines, possibly written in different languages, then the interface is loose. Memory management must be done across the interface, otherwise dangling pointers or memory leaks result. Memory management across the interface can be done by using finalization, which is explained in Section 10.6.1.

- If the components are in the same process, then the interface is tight. Memory management is done globally for all components, so it is usually simpler. Thread priorities in the components must be compatible.

How components find each other

Components are not always permanently connected to each other. For some applications, they have to find each other first. There are many ways for components to find each other.
8.9.4 System structure

How do we structure an application as shown in Figure 8.7, in practical terms? There are two aspects, a static one and a dynamic one:

- **Static structure.** This consists of the component graph that is known when the application is designed. These components can be linked as soon as the application starts up. Each component instance corresponds roughly to a bundle of functionality that is sometimes known as a *library* or a *package*. For efficiency, we would like each component instance to exist at most once in the system; if a library is needed by different parts of the application, then we want them to share the same library. For example, a component may implement a graphics package; the whole application can get by with just one instance.

- **Dynamic structure.** Often, an application will do calculations with components at run time. It may want to link new components, which are known only at run time. Or it may want to calculate a new component and store it. Component instances need not be shared; perhaps several instances of a component are needed. For example, a component may implement a database interface. Depending on whether there are one or more external databases, one or more instances of the component should be loaded. This is determined at run time, whenever a database is added.

Figure 8.9 shows the resulting structure of the application, with some components linked statically and others linked dynamically.

**Static structure**

To support the static structure, it is useful to make components into compilation units stored in files. We call these components *functors* and their instances *modules*. A functor is a component that can be stored in a file. It is a *compilation unit* because the file can be compiled independently of other functors. The functor’s dependencies are given as filenames. In order to be accessible by other functors, the functor *must* be stored in a file. This allows other functors to specify that they need it by giving the filename.

A functor has two representations: a source form, which is just a text, and a compiled form, which is a value in the language. If the source form is in a file `foo.oz`, whose entire content is of the form `functor ... end`, then it can be compiled to give another file, `foo.ozf`, that contains the compiled form. The content of file `foo.oz` looks like this:

```oz
functor
  import OtherComp1 at File1
  OtherComp2 at File2
  ...
  OtherCompN at FileN
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Component instance

Dependency

Sharing boundary

Figure 8.9: Application structure – static and dynamic

```
export op1:X1 op2:X2 ... opK:Xk
define
  % Define X1, ..., Xk
  ...
end
```

This component depends on the other components OtherComp1, ..., OtherCompN, stored respectively in files File1, ..., FileN. It defines a module with fields op1, ..., opK, referenced by X1, ..., Xk and defined in the functor body.

An application is just one compiled functor. To run the application, all the other compiled functors that it needs, directly or indirectly, must be brought together and instantiated. There are many ways to organize the components. One possibility is to bring them all together in one file. This is a convenience when sending the application to others; its disadvantage is that it keeps compiled functors from being shared between applications. A second possibility is to put them in a standard repository (e.g., directory) on the local machine. A third possibility is to put them in a standard repository on the Internet, e.g., in locations accessible by URL. The latter two possibilities keep local code size small; their disadvantage is that they make the application dependent on the repository that stores the functors.

When the application is run, it loads its components and instantiates them. The components can all be linked (i.e., loaded and instantiated) when the application starts up (static linking), or by need, i.e., one by one as they are needed (dynamic linking). We have found that dynamic linking is usually preferable, as long as all the compiled functors are quickly accessible (e.g., exist on the local
system). It has two advantages: the application starts up quickly and takes up only as much memory space as it needs.

**Dynamic structure**

In terms of the language, a functor is just another language entity. If a program contains a statement of the form \( x = \text{functor} \ldots \text{end} \) then \( x \) will be bound to the functor value. Like a procedure, the functor may have external references. Instantiating a functor gives a module, which is the set of language entities created by initializing the functor. An interface is a record giving the externally-visible language entities of the module. In a convenient abuse of notation, this record is sometimes called the module.

A functor can be instantiated, which returns a module. Instantiating a functor is done in the context of a set of existing modules with their module names:

- An environment is created that has variables for all the modules to be imported.
- These variables are bound to their respective modules.
- Then a record is created that will hold the new module.
- Then, the functor body is executed in the context of the new environment. This creates and initializes all the entities in the new module.

The functor body has two roles. Its first role is always to initialize the data structures of the module. Once initialized, the first role is done. Its second role is to implement an application. This role continues during the lifetime of the module. Often, an application will have only one functor that plays the second role, namely the application functor itself.

There exists a single operation to instantiate functors:

- \( \text{Ms} = \{ \text{NewModules Fs} \} \). Given a list \( \text{Fs} \) of functors (as language entities, filenames, or URLs), this instantiates all the functors and creates a list \( \text{Ms} \) of modules. Within the scope of a \( \text{NewModules} \) call, functors are shared. That is, if the same functor is mentioned more than once, then it is only linked once.

Each call to \( \text{NewModules} \) results in a new, fresh set of modules.

### 8.9.5 Development methodology

A big application takes a long time to develop. How is the structure of Figure 8.9 developed over time? There are many ways to organize the development. The techniques of top-down, bottom-up, and even “middle-out” development have been much discussed. None of these techniques is really satisfactory, and combining them does not help much. The main problem is that they all rely on the application’s requirements and specification to be fairly complete to start with, i.e.,
that the designers anticipated most of the functional and non-functional requirements. Unless the application is very well understood, this is almost impossible to achieve.

In our experience, the only approach that works well in practice is the thin-to-thick approach:

- Start with a small set of requirements, which is a subset of the complete set, and build a complete application that satisfies them. This application has a specification and an architecture that are “empty shells” with just enough functionality to do what they have to do.

- Then continuously extend the requirements, extending the other layers as needed to satisfy them. Using an organic metaphor, we say the application is “grown” or “evolved”. At all times, there is a running application that satisfies its specification and that can be evaluated by its potential users.

This method has many advantages:

- Bugs of all sizes are detected early and can be fixed quickly.

- Deadline pressure is much relieved, since there is always a working application.

- Developers are more motivated, because they have quick feedback on their efforts.

- Users are more likely to get what they really need, because they have a chance to use the application during the development process.

- The architecture is more likely to be good, since it can be corrected early on.

- The user interface is more likely to be good, since it is continuously improved during the development process.

For most applications, even very small ones, we find that it is almost impossible to determine in advance what the real requirements are, what a good architecture is to implement them, and what the user interface should be. The thin-to-thick approach works well partly because it makes very few assumptions up front.

Also discuss:
- Advantages of "many eyeballs" approach, also called "halo" approach, or "peer review" approach (as in the research community). In the area of software development this approach was discovered by Open Source developers, as documented by Eric Raymond~\cite{cathedral}.
- Techniques for designing abstractions: the dual criterium of ‘efficient implementation’ and ‘simple formal definition’, the life cycle of abstractions.
8.9.6 Maintainable systems

What is the best way to build systems that are maintainable, i.e., it is easy to correct and update them during their lifetimes to satisfy their users? From our experience, here are some of the most important principles.

**Fewest possible external dependencies**

A component that depends on another, i.e., it requires the other for its operation, is a source of maintenance problems. If the other is changed, then the first must be changed as well. This is a major source of “software rot”, i.e., once-working software that stops working. For example, it is very bad for \LaTeX\ documents to have links to style files in other, perhaps global, directories. If these are updated, then the documents can no longer be pageset. For maintenance, it is much preferable to have copies of the style files in each document directory. This satisfies a simple invariant: each document is guaranteed to be pagesettable at all times (“working software keeps working”). This invariant is an immense advantage that far outweighs the two disadvantages: (1) the extra memory needed for the copies and (2) the possibility that a document may use an older style file. If a style file is updated, the programmer is free to use the new version in the document, but only if necessary. Meanwhile, the document stays consistent. A second advantage is that it is easy to send the document from one person to another, since it is self-contained.

**Fewest possible levels of indirection**

This is related to the previous rule. When A points to B, then updating B requires updating A. Any indirection is a kind of “pointer”. The idea is to avoid the pointer becoming dangling, i.e., its destination no longer makes sense to the source. An action at B may cause A’s pointer to become dangling. B doesn’t know about A’s pointer and so cannot prevent such a thing. A stop-gap is never to change B, but only to make modified copies. This can work well if the system does automatic global memory management.

Two typical examples of problematic pointers are Unix symbolic links and URLs. Symbolic links are pernicious for system maintenance. They seem comfortable, in that they can refer to other mounted directory trees, but in fact they are a big cause of system problems. URLs are known to be extremely flaky. They are often referenced in printed documents, but their lifetime is usually much less than that of the document. They are an unreliable source of information.

**Dependencies should be predictable**

For example, consider a ‘localize’ command that is guaranteed to retrieve a file over a network and make a local copy. It has simple and predictable behavior, unlike the “page caching” done by Web browsers. Page caching is a misnomer,
because a true cache maintains coherence between the original and the copy. For any such "cache", the replacement policy should be clearly indicated.

**Fewest possible autonomous irrevocable decisions**

For example, time outs are bad. A time out is an irrevocable decision made at a low level in the system (a deeply-nested component) that propagates all the way to the top level without any way for the intermediate components to intervene. This behavior short-circuits any efforts the application designer may do to mask the problem or solve it. Section ?? says more about time outs.

**Documented violations**

Whenever one of the previous principles is violated, perhaps for a good reason (e.g., physical constraints such as memory limitations or geographic separation force a pointer to exist), then this should be documented! That is, all external dependencies, all levels of indirection, all non-predictable dependencies, and all irrevocable decisions, should be documented.

**Simple bundling hierarchy**

We define a simple hierarchy of how to bundle system components. We have found this hierarchy to be useful for documents as well as applications. The easiest-to-maintain design is first:

1. If possible, put the whole application in one file. The file may be structured in sections.

2. If the above is not possible (for example, there are files of different types or different people are to work on different parts simultaneously), then put the whole application in one directory.

3. If the above is not possible (for example, the application is to be compiled for multiple platforms), put the application in a directory hierarchy with one root.

**8.10 Case studies**

Here come some interesting case studies.

**8.10.1 Word frequencies (with stateful dictionary)**

In Section 5.7.1 we showed how to use dictionaries to count the number of different words in a text file. We compared the execution times of three versions of the word frequency counter, each one with a different implementation of dictionaries. The first two versions use declarative dictionaries (implemented lists and
fun {WordChar C} ... end

fun {WordToAtom PW} ... end

fun {CharsToWords PW Cs} ... end

Put=Dictionary.put
CondGet=Dictionary.condGet

proc {IncWord D W}
    {Put D W (CondGet D W 0)+1}
end

proc {CountWords D Ws}
    case Ws
    of W|Wr
    then
    {IncWord D W}
    {CountWords D Wr}
    [] nil then skip
    end
end

fun {WordFreq Cs}
    D={NewDictionary}
    in
    {CountWords D {CharsToWords nil Cs}}
    D
end

Figure 8.10: Word frequencies (with stateful dictionary)

binary trees, respectively) and the third uses the built-in definition of dictionaries (implemented with state). The version using stateful dictionaries, shown in Figure 8.10, is slightly different from the one using declarative dictionaries, shown in Figure 5.11:

- The stateful version needs to pass just one argument as input to each procedure that uses a dictionary.

- The declarative version has to use two arguments to these procedures: one for the input dictionary and one for the output dictionary. In Figure 5.11, the second output is realized by using functional notation.

The difference shows up in the operations Put, IncWords, CountWords, and WordFreq. For example, Figure 8.10 uses the stateful {Put D LI X}, which updates D. Figure 5.11 uses the declarative {Put D1 LI X D2}, which reads D1 and returns a new dictionary D2.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
8.10.2 Generating random numbers

A very useful primitive operation is a random number generator. It lets the computer “throw dice”, so to speak. How do we generate random numbers in a computer? Here we give the main insights; see Knuth [55] for a deep discussion of the theory underlying random number generators and of the concept of randomness itself.

Different approaches

One could imagine the following ways to generate random numbers:

- A first technique would be to use unpredictable events in the computer itself, e.g., related to concurrency, as explained in the previous chapter. Alas, their unpredictability does not follow simple laws. For example, using the thread scheduler as a source of randomness will give some fluctuations, but they do not have a useful probability distribution. Furthermore, they are intimately linked with the computation in nonobvious ways, so even if their distribution was known, it would be dependent on the computation. So this is not a good source of random numbers.

- A second technique would be to rely on a source of true randomness. For example, electronic circuits generate noise, which is a completely unpredictable signal with a known probability distribution. The noise comes from the depths of the quantum world, so for all practical purposes it is truly random. But there are two problems. First, the probability distribution is not exactly known: it might vary slightly from one circuit to the next or with the ambient temperature. The first problem is not serious; there are ways to “normalize” the random numbers so that their distribution is a constant, known one. The second problem is more serious: the randomness cannot be reproduced except by storing the random numbers and replaying them. It might seem strange to ask for reproducibility from a source of randomness, but it is perfectly reasonable. For example, the randomness might be input to a simulator. We would like to vary some parameter in the simulator such that any variation in the simulator depends only on the parameter, and not on any variation in the random numbers. For this reason, computers are not usually connected to truly random sources.

- It might seem that we have carefully worked ourselves into a tight corner. We would like true randomness and we would like it to be reproducible. How can we resolve this dilemma? The solution is simple: we calculate the random numbers. How can this generate truly random numbers? The simple answer is, it cannot. But the numbers can appear random, for all practical purposes. They are called pseudorandom numbers. What does this mean? It is not simple to define. Roughly, the generated numbers should
give the same behavior as truly random numbers, for the use we make of them.

The third solution, calculating random numbers, is the one that is almost always implemented. The question is, what algorithm do we use? Certainly not an algorithm chosen at random! Knuth [55] shows the pitfalls of this approach. It almost always gives bad results. We need an algorithm that has known good properties. We cannot guarantee that the random numbers will be good enough, but we can try to get what we can. For example, the generated random numbers should satisfy strong statistical properties, have the right distribution, and their period should be sufficiently long. The last point is worth expanding on: since a random number generator does a calculation with finite information, eventually it will repeat itself. Clearly, the period of repetition should be very long.

Uniformly distributed random numbers

A random number generator stores an internal state, with which it calculates the next random number and the next internal state. The state should be large enough to allow a long period. The random number is initialized with a number called its seed. Initializing it again with the same seed should give the same sequence of random numbers. If we do not want the same sequence, we can initialize it with information that will never be the same, such as the current date and time. Modern computers almost always have an operation to get this time information. Now we can define the abstract data type of a random number generator:

- \{NewRand Rand Init Max\} creates a new random number generator \(\text{Rand}\) with initialization procedure \(\text{Init}\) and maximum value given by \(\text{Max}\). Each generator has its own internal state.

- \{Init Seed\} initializes the generator with integer seed \(\text{Seed}\), that should be in the range \(0, 1, ..., \text{Max}\). To give many possible sequences, \(\text{Max}\) should be large. Initialization can be done at any time.

- \(\text{X} = \{\text{Rand}\}\) generates a new random number \(\text{X}\) and update the internal state. \(\text{X}\) is an integer in the range \(0, 1, ..., \text{Max}\) and has a uniform distribution. For best results, \(\text{Max}\) should be large. This allows the program to reduce the random numbers to the smaller domains it needs for its own purposes.

How do we calculate a new random number? It turns out that a good simple method is the linear congruential generator. If \(x\) is the internal state and \(s\) is the seed, then the internal state is updated as follows:

\[
x_0 = s \\
x_n = (ax_{n-1} + b) \mod m
\]
The constants $a$, $b$, and $m$ have to be carefully chosen so that the sequence $x_0$, $x_1$, $x_2$, ..., has good properties. The internal state $x_i$ is a uniformly distributed integer from 0 to $m – 1$. It is easy to implement this generator:

```plaintext
local
A = 333667
B = 213453322
M = 1000000000
in
  proc {NewRand Rand Init Max}
    X={NewCell 0}
  in
    proc {Init Seed}
      {Assign X Seed}
    end
  fun {Rand}
    S=(A*{Access X}+B) mod M
  in
    {Assign X S}
    S
  end
  Max=M-1
end

This is one of the simplest methods that has reasonably good behavior. More sophisticated methods are possible that are even better. Operating systems provide a random number generator. If the operating system is recent, this generator is usually quite good. The System module OS gives access to this generator:

- `{OS.srand S}` initializes the generator with integer seed $S$. If $S=0$ then the current time is used to initialize the generator.

- `{OS.rand}` generates a new random number $X$ and update the internal state. $X$ is an integer uniformly distributed in the range $Min$, ..., $Max$ (see `{OS.randLimits Min Max}` below).

- `{OS.randLimits Min Max}` binds $Min$ and $Max$ to the smallest and largest possible output of `{OS.rand}`.

Nonuniform distributions

A good technique to generate random numbers of any distribution is to start with a uniformly distributed random number. From this, we calculate a number with another distribution. Using this technique we explain how to generate Gaussian and exponential distributions. To make the rest easier, we first define some utilities:

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
declare
ITF = Int.toFloat
TwoPi = 4.0*{Float.acos 0.0}
Max = {ITF {OS.randLimits _}}
Rand = OS.rand

Now we can define random numbers of various distributions. First, a uniform distribution from 0 to 1 and an integer distribution from A to B:

fun {Uniform}
  {ITF {Rand}}/Max
end

fun {UniformI A B}
  ({Rand} mod (B-A+1)) + A
end

We will use Uniform to generate random variables with other distributions. First, let us generate random variables with an exponential distribution. For this distribution, the probability that $X \leq x$ is $D(x) = 1 - e^{-\lambda x}$, where $\lambda$ is a parameter called the intensity. It follows that the probability that $D(X) \leq D(x)$ is $D(x)$. Writing $y = D(x)$, it follows that the probability that $D(X) \leq y$ is $y$. Therefore $D(X)$ is uniformly distributed. Say $D(X) = U$ where $U$ is a uniformly distributed random variable. Then we have $X = -\ln(1-U)/\lambda$. This gives the following function:

fun {Exponential Lambda}
  ~{Log 1.0-{Uniform}}/Lambda
end

Now let us generate a Normal distribution with mean 0 and variance 1. This is also called a Gaussian distribution. We use the following technique. Given two variables $U_1$ and $U_2$, uniformly distributed from 0 to 1. Let $R = \sqrt{-2 \ln U_1}$ and $\phi = 2\pi U_2$. Then $X_1 = R \cos \phi$ and $X_2 = R \sin \phi$ are independent variables with a Gaussian distribution. The proof of this fact is beyond the scope of the book; it can be found in [55]. This gives the following function:

fun {Gauss}
  {Sqrt ~2.0*{Log {Uniform}}} * {Cos TwoPi*{Uniform}}
end

Since each call can give us two Gaussian variables, we can use a cell to remember one result for the next call:

local GaussCell={NewCell nil} in
  fun {Gauss}
    Prev={Exchange GaussCell $ nil} in
    if Prev\=nil then Prev
    else R Phi in
      R={Sqrt ~2.0*{Log {Uniform}}}
    end
  end

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
\[ \Phi = \text{TwoPi}\{\text{Uniform}\} \]
\[ \{\text{Assign GaussCell} \ R\{\cos \Phi}\} \]
\[ R\{\sin \Phi}\] end end end

Each call of Gauss calculates two independent Gaussian variables; we return one and store the other in a cell. The next call returns it without doing any calculation.

8.10.3 “Word of Mouth” simulation

Let us simulate how Web users “surf” on the Internet. To “surf” between Web sites means to successively load different Web sites. To keep our simulator simple, we will only look at one aspect of a Web site, namely its performance. This can be reasonable when surfing between Web portals, which each provide a large and similar set of services. Assume there are \( n \) Web sites with equal content and a total of \( m \) users. Each Web site has constant performance. Each user would like to get to the Web site with highest performance. But there is no global measure of performance; the only way a user can find out about performance is by asking other users. We say that information passes by “word of mouth”. This gives us the following simulation rules:

- Each site has a constant performance. Assume the constants are uniformly distributed.
- Each user knows which site it is on.
- Each site knows how many users are on it.
- Each user tries to step to a site with higher performance. The user asks a few randomly-picked users about the performance at their site. The user then goes to the site with highest performance. However, the performance information is not exact: it is perturbed by Gaussian noise.
- One round of the simulation consists of all users doing a single step.

With these rules, we might expect users eventually to swarm among the sites with highest performance. But is it really so? A simulation can give us answer.

Let us write a small simulation program. First, let us set up the global constants. We will use the functions \( \text{ITF, Uniform, UniformI, and Gauss} \) defined in the previous section. There are \( n \) sites, \( m \) users, and we will do \( t \) simulation rounds. We will initialize the random number generator and write information to the file `wordofmouth.txt` during the simulation. With 10000 sites, 500000 users, and 200 rounds, this gives the following:

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Next, we decide how to store the simulation information. We would like to store it in records or tuples, because they are easy to manipulate. But they cannot be modified. Therefore, we will store the simulation information in dictionaries. Dictionaries are very similar to records except that they can be changed dynamically (see Section 8.5.2). Each site picks its performance randomly. It has a dictionary giving its performance and the number of users on it. The following code creates the initial site information:

```plaintext
declare
Sites={Tuple.make sites N}
for I in 1..N do
  S.I={Record.toDictionary
    o(hits:0 performance:{ITF {UniformI 1 80000}})})
end
```

Each user picks its site randomly. It has a dictionary giving its current site. It updates the Sites information. The following code creates the initial user information:

```plaintext
declare
Users={Tuple.make users M}
for I in 1..M do
  S={UniformI 1 N}
in
  U.I={Record.toDictionary o(currentSite:S)}
  Sites.S.hits := Sites.S.hits + 1
end
```

Now that we have all the data structures, let us do one user step in the simulation. The function {UserStep I} does one step for user I, i.e., the user asks three other users for the performance of their sites, it calculates its new site, and then it updates all the site and user information.

```plaintext
proc {UserStep I}
  U = Users.I
  % Ask three users for their performance information
  L = (List.map [{UniformI 1 M} {UniformI 1 M} {UniformI 1 M}]
    fun($ X)
      (Users.X.currentSite) #
      Sites.(Users.X.currentSite).performance
      + (Gauss)*(ITF N)
    end)
```
end
% Calculate the best site
MS#MP = {List.foldL L
  fun ($ X1 X2) if X2.2>X1.2 then X2 else X1 end end
U.currentSite #
Sites.(U.currentSite).performance
  + {Abs {Gauss}*{ITF N}}
}

in
  if MS\=U.currentSite then
    Sites.(U.currentSite).hits :=
    Sites.(U.currentSite).hits - 1
    U.currentSite := MS
    Sites.MS.hits := Sites.MS.hits + 1
  end
end

Now we can do the whole simulation:

for J in 1..N do
  {Out {Record.adjoinAt {Dictionary.toRecord site Sites.J}
    name J}}
end

{Out endOfRound(time:0 nonZeroSites:N)}
for I in 1..T do
  X = {NewCell 0}
in
  for U in 1..M do {UserStep U} end
  for J in 1..N do
    H=Sites.J.hits in
    if H\=0 then
      {Out {Record.adjoinAt
        {Dictionary.toRecord site Sites.J} name J}}
      {Assign X 1+{Access X}}
    end
  end
  {Out endOfRound(time:I nonZeroSites:{Access X})}
end

{F close}

To make the simulator self-contained, we put all the above code in one procedure
with parameters N, M, T, and the output filename.

What is the result of the simulation? Will users cluster around the sites with
highest performance, even though they have only a very narrow and inaccurate
view of what is going on? Running the above simulation shows that the number of
non-zero sites (with at least one user) decreases smoothly in inverse exponential
fashion from 10000 initially to less than 100 after 83 rounds. Average perfor-
mance of user sites increases from about 40000 (half of the maximum) to more
than 75000 (within 6% of maximum) after just 10 rounds. So we can make a pre-
liminary conclusion that the best sites will quickly be found and the worst sites
will quickly be abandoned, even by word-of-mouth propagation of very approxi-
mate information. Of course, our simulation has some simplifying assumptions.
Feel free to change the assumptions and explore. For example, the assumption
that a user can pick any three other users is unrealistic—it assumes that each user
knows all the others. This makes convergence too fast. See the Exercises of this
chapter for a more realistic assumption on user knowledge.

8.10.4 Swiftfile

Digital assistant to aid an email user. A learning algorithm to guess in which
folder the user wants to put a received message.

8.10.5 Forward chaining rule-based system

Each variable and each rule is an object. Do forward chaining by following refer-
ences, no search needed.

8.11 Limitations of stateful programming

Stateful programming has some strong limitations due to its use of encapsulated
state. Object-oriented programming is a special case of stateful programming, so
it suffers from the same limitations.

8.11.1 The real world is parallel

The main limitation of the stateful model is that programs are sequential. In the
real world, entities are both stateful and execute in parallel. Stateful program-
ming does not model the parallel execution.

Sometimes this limitation is appropriate, e.g., when writing simulators where
all events must be coordinated (stepping from one global state to another in a
controlled way). In other cases, e.g., when interacting with the real world, the
limitation is an obstacle. To remove the limitation, the model needs to have both
state and concurrency. This gives the concurrent stateful model of Chapter 10.
As Section 5.2 explains, concurrency in the model can model parallelism in the
real world.

8.11.2 The real world is distributed

Encapsulated state is hard to use well in a distributed system. Chapter 13 ex-
plains this limitation in depth. Here we give just the main points. In a distributed
system, the store is partitioned into separate parts. Within one part, the store
behaves efficiently as we have seen. Between parts, communication is many orders of magnitude more expensive. The parts coordinate with one another to maintain the desired level of global consistency. For cells this can be expensive because cell contents can change at any time in any part. The programmer has to decide on both the level of consistency and the coordination algorithm used. This makes it tricky to do distributed programming with state.

The declarative model and its concurrent extension are much easier to use. As Section 5.2 explains, declarative programs can be decomposed into independent components with no internal history. This fits very well with the partitioned store of a distributed system. When programming a distributed system, we recommend to use the declarative model whenever possible for coordinating the parts. Chapter 13 explains how to program a distributed system and when to use the declarative and stateful models.

8.12 Exercises

1. **What is state.** Section 8.1 defines the function `SumList` that has two states, encoded in the successive values of arguments at recursive calls. For this exercise, rewrite the function `SumList` so that the two states are no longer encoded in arguments, but by cells.

2. **Emulating state with concurrency.** This exercise explores whether concurrency can be used to obtain encapsulated state.

   - First use concurrency to create an updatable container. We create a thread that uses a recursive procedure to read a stream. The stream has two possible commands: `access(X)`, which binds X to the container’s current content, and `assign(X)`, which assigns X as the new content. Here is how it is done:

     ```plaintext
     fun {MakeState Init}
       proc {Loop S V}
         case S of access(X) | S2 then
           X=V {Loop S2 V}
         [ ] assign(X) | S2 then
           {Loop S2 X}
         else skip end
       end
     end
     in
       thread {Loop S Init} end
     S
     end

     S={MakeState 0}
     ```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
The call \{\texttt{MakeState Init}\} creates a new container with initial content \texttt{Init}. We use the container by putting commands on the stream. For example, here is a sequence of three commands for the container \texttt{S}:

\begin{verbatim}
\texttt{declare S1 X Y in}
\texttt{S=access(X) | assign(3) | access(Y) | S1}
\end{verbatim}

This binds \texttt{x} to 0 (the initial content), puts 3 in the container, and then binds \texttt{y} to 3.

- Now rewrite \texttt{SumList} to use this container to count the number of calls. Can this container be encapsulated, i.e., can it be added without changing the arguments of \texttt{SumList}? Why or why not?

3. \textbf{Transitive closure}. Explain the similarities and differences between the declarative and stateful versions of the transitive closure algorithm. Is it reasonable to call them variations of the same algorithm?

4. \textbf{“Small World” simulation}. The “Word of Mouth” simulation of Section 8.10.3 makes some strong simplifying assumptions. For example, the simulation assumes that each user can choose any three users at random to ask them about their performance. This is much too strong an assumption. The problem is that the choice ranges over \textit{all} users. This gives each user a potentially unbounded amount of knowledge. In actuality, each user has bounded knowledge: a small network of acquaintances that changes but slowly. Each user asks only members of his network of acquaintances. Rewrite the simulation program to take this assumption into account. This can make convergence much slower. With this assumption, the simulation is called a “Small World” simulation [104].

5. \textbf{Performance effects in “Word of Mouth” simulation}. The “Word of Mouth” simulation of Section 8.10.3 assumes that site performance is constant. A better way to take performance into account is to assume that it is constant up to a given threshold number of users, which is fixed for each site. Beyond this threshold, performance goes down in inverse proportion to the number of users. This is based on the premise that for small numbers of users, Internet performance is the bottleneck, and for large numbers of users, site performance is the bottleneck.
Chapter 9

Object-Oriented Programming

“The fruit is too well known to need any description of its external characteristics.”
– From entry “Apple”, Encyclopedia Britannica (11th edition)

This chapter gives a case study of a particularly rich and useful style for structuring programs, namely object-oriented programming. It is primarily used to program with explicit state, but it is much more widely applicable. It is one of the most successful and pervasive abstractions in computing. From its timid beginnings in the 1960’s [7, 25], it has invaded every area of computing, both in scientific research and technology development. The currently most popular programming languages, Java and C++, are object-oriented [90, 89]. The most popular “language-independent” design aids, the Unified Modeling Language (UML) and Design Patterns, both implicitly assume that the underlying language is object-oriented [34, 76]. With all this exposure, one might feel that object-oriented programming is well understood (see chapter quote above). Yet, this is far from being the case.

The purpose of this chapter is not to cover all of object-oriented programming in 100 pages or less. This is impossible. Instead, we give an introduction that emphasizes areas where other programming books are weak: the relationship with other computation models, the precise semantics, and the possibilities of dynamic typing. The chapter is structured as follows:

- **Motivations** (Section 9.1). We give the principal motivation for object-oriented programming, namely to support inheritance, and how its features relate to this.

- **An object-oriented computation model** (Sections 9.2 and 9.3). We define an object system that takes advantage of dynamic typing to combine simplicity and flexibility. This allows us to explore better the limits of the object-oriented abstraction and situate existing languages within them. We give the object system syntactic and implementation support to make it easier to use and more efficient.
- **Principal programming techniques** (Section 9.4). We explain the basic principles and techniques for using inheritance to construct object-oriented programs. We illustrate them with realistic example programs. We also give pointers into the literature on object-oriented design.

- **Relation to other computation models** (Section 9.5). From the viewpoint of multiple computation models, we show how and when to use and not use object-oriented programming. We relate it to component-based programming, object-based programming, and higher-order programming. We give additional design techniques that become possible when it is used together with other models. We explain the pros and cons of the oft-repeated principle stating that every language entity should be an object. This principle has guided the design of several major object-oriented languages, but is often misunderstood.

- **Constructing the object system** (Section 9.6). We give a simple and precise semantics of our object system, by defining it in terms of the stateful computation model. This definition is ultimately based on the semantics of Chapter 15.

After reading this chapter, you will have a better view of what object-oriented programming is about and how to situate it among other computation models.

### 9.1 Motivations

#### 9.1.1 Inheritance

As we saw in the previous chapter, stateful abstract data types are a very useful concept for organizing a program. In fact, a program can be built in a hierarchical structure as ADTs that depend on other ADTs. This is the idea of component-based programming.

Object-oriented programming takes this idea one step further. It is based on the observation that components frequently have much in common. Take the example of sequences. There are many different ADTs that are “sequence-like”. Sometimes we want them to behave like stacks (adding and deleting at the same end). Sometimes we want them to behave like queues (adding and deleting at opposite ends). And so forth, with dozens of possibilities. All of these sequences share the basic, linear-order property of the concept of sequence. How can we implement them without duplicating the common parts?

Object-oriented programming answers this question by introducing the additional concept of *inheritance*. An ADT can be defined to “inherit” from other ADTs, that is, to have substantially the same functionality as the others, with possibly some modifications and extensions. Only the *differences* between the ADT and its ancestors have to be specified. Such an incremental definition of an ADT is called a *class*. 
Inheritance is the essential difference between object-oriented programming and most other kinds of stateful programming. It turns out that inheritance is a very rich concept that can be rather tricky. There are many ways that an ADT can be built by modifying other ADTs. The primary approach used in object-oriented programming is \textit{syntactic}: a new ADT is defined by doing simple syntactic manipulations of an existing ADT. Because the semantics does not always follow the syntax, these manipulations must be done with great care.

The component approach to building systems is much simpler. A component groups together any set of entities and treats them as a unit from the viewpoint of use dependency. A component is built from subcomponents, respecting their specifications.

The potential of inheritance is very great: it increases the possibilities of factoring an application, i.e., to make sure that each abstraction is implemented just once. Having more than one implementation of an abstraction does not just make the program longer. It is an invitation to disaster: if one implementation is changed, then the others must also be changed. What’s more, the different implementations are usually slightly different, which makes nonobvious the relationships among all the changes. This “code duplication” of an abstraction is one of the biggest sources of errors. Inheritance has the potential to remove this duplication.

The potential to factor an application is a two-edged sword. It comes at the price of “spreading out” an ADT’s implementation over large parts of the program. The implementation of an ADT does not exist in one place; all the ADTs that are part of it have to be considered together. Even stronger, part of the implementation may exist only as compiled code, with no access to the source code.

Early on, it was believed that inheritance would solve the problem of software reuse. That is, it would make it easier to build libraries that can be distributed to third parties, for use in other applications. This has not worked out in practice. The failure of inheritance as a reuse technique is clear from the success of other techniques such as components, frameworks, and design patterns. Inheritance remains most useful within a single application or closely-related family of applications.

Inheritance is not an unmixed blessing, but it takes its place next to higher-order programming as one of the most important techniques for structuring a program.

\section{Encapsulated state and inheritance}

The combination of encapsulated state and inheritance has led to the field of \textit{object-oriented programming}, which is presented in this chapter. This field has developed a rich theory and practice on how to write stateful programs with inheritance. Unfortunately, this theory tends to consider everything as being an object and to mix the notions of state and encapsulation. The advantages to
be gained by considering other entities than objects and by using encapsulation without state are often ignored. Chapters 5 and 6 explain well how to use these two ideas. This chapter follows the object-oriented philosophy and emphasizes how to build ADTs with both encapsulated state and inheritance.

Most object-oriented programming languages consider that ADTs should have encapsulated state by default. For example, Smalltalk, C++, and Java all consider variables to be stateful, i.e., mutable, by default. In Java it is possible to make variables immutable by declaring them as `final`, but it is not the default. This goes against the rule of thumb given in Section 5.2, and in our view it is a mistake. Encapsulated state is a complex notion which should not be the first one to program with. There are simpler ways to program, e.g., using variable identifiers to refer to values or dataflow variables. These simpler ways should be considered first before moving to encapsulated state.

### 9.1.3 Objects and classes

An *object* is an entity that encapsulates an explicit state so that it can only be accessed in a controlled way from outside the object. The access is provided by means of *methods*, which are procedures that are accessible from the outside and that can directly access the internal state. The only way to modify the state is by calling the methods. This means that the object can guarantee that the state always satisfies some invariant property.

A *class* is an entity that specifies an object in an incremental way, by defining the classes that the object inherits from and how the object is different from the inherited classes.

This chapter only talks about objects that are used sequentially, i.e., that are used in a single thread. Chapter 10 explains how to use objects in a concurrent setting, when multiple threads use the objects. In particular, object locking is explained there.
9.2 Classes as complete ADTs

The heart of the object concept is controlled access to encapsulated data. The behavior of an object is specified by a class. In the most general case, a class is an incremental definition of an ADT, that defines the ADT as a modification of other ADTs. There is a rich set of concepts for defining classes. We classify these concepts into two sets, according as they permit the class to define an ADT completely or incrementally:

- **Complete ADT definition.** These are all the concepts that permit a class, taken by itself, to define an ADT. There are two sets of concepts:
  - Defining the various elements that make up a class (Section 9.2.2), namely methods, attributes, features, and properties. Attributes and features can be initialized in several ways, per object or per class (Section 9.2.3).
  - Taking advantage of dynamic typing. This gives first-class messages (Section 9.2.4) and first-class attributes and features (Section 9.2.5). This allows powerful forms of polymorphism that are harder or impossible to do in statically-typed languages. This increased freedom comes with an increased responsibility of the programmer to use it correctly.

- **Incremental ADT definition.** These are all the concepts related to inheritance, that is, they define how a class is related to existing classes. They are given in Section 9.3.

9.2.1 Examples

To see how classes and objects work in the object system, we first define an example class. Figure 9.1 defines a class referred to by the variable `Counter`. This class has one attribute, `val`, that holds a counter’s current value, and three methods, `init`, `browse`, and `inc`, for initializing, displaying, and incrementing the counter. This seems quite similar to how other languages would do it, modulo a different syntax. But appearances can be deceiving!

The declaration of Figure 9.1 is actually executed at run time, i.e., it is a statement. Putting this declaration at the head of a program will declare the class before executing the rest, which is familiar behavior. But this is not the only possibility. The declaration can be put anywhere that a statement can be. For example, putting the declaration inside a procedure will create a new and distinct class each time the procedure is called. Later on we will use this possibility to make parameterized classes.

Now we can create an object of class `Counter` and do some operations with it:

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
\begin{verbatim}
declare
C={New Counter init(0)}
{C inc(6)} {C inc(6)}
{C browse}
\end{verbatim}

This creates the counter object \( C \) with initial value 0, increments it twice by 6, and then displays the counter’s value. The statement \( \{ C \, \text{inc}(6) \} \) is called an object application. The message \text{inc}(6)\) is sent to the object, which invokes the corresponding method. Now try the following:

\begin{verbatim}
local X in \{ C \, \text{inc}(X) \} X=5 end
{C browse}
\end{verbatim}

This displays nothing at all! The reason is that the object application

\( \{ C \, \text{inc}(X) \} \)

suspends inside the method \text{inc}. Can you see exactly where? Now try the following variation:

\begin{verbatim}
local X in thread \{ C \, \text{inc}(X) \} end X=5 end
{C browse}
\end{verbatim}

Things now work as expected. This is another example of dataflow execution.

\subsection*{9.2.2 Defining classes}

A \textit{class} is a data structure that defines an object’s internal state (attributes), its behavior (methods), and several other properties and operations that we will see later on. More generally, a \textit{class} is a data structure that describes an ADT and gives its partial or total implementation. Table 9.1 gives the syntax of classes. There can be any number of objects of a given class. They are called \textit{instances} of the class. These objects have different identities and can have different values for their internal state. Otherwise, all objects of a given class behave according to the class definition. An object \textit{Obj} is called with the syntax \( \{ \textit{Obj} \, \text{M} \} \), where \text{M} is a record that defines the message. Calling an object is also called \textit{passing a message} to the object.

A class defines the set of members that each instance will have. There are four kinds of members:

- **Attributes** (declared with the keyword “\texttt{attr}”). An attribute is an instance variable that is stateful and private. The attribute can reference any language entity. The attribute is visible only in the class definition and all classes that inherit from it. Every instance has a separate set of attributes. The instance can update the attribute \text{val} with the following operations:

  - An assignment statement: \texttt{val<- (expr)}. This evaluates the expression \texttt{(expr)} and assigns the result to \texttt{val}. 

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
(statement) ::= class (variable) { (classDescriptor) }
    { meth (methHead) [ `=´ (variable) ]
      ( (inExpression) | (inStatement) ) end }
  end

  lock [ (expression) then | (inStatement) end
  (expression) `<=´ (expression)
  (expression) `´, `´ (expression)
  ]

(expression) ::= class `$´ { (classDescriptor) }
    { meth (methHead) [ `=´ (variable) ]
      ( (inExpression) | (inStatement) ) end }
  end

  lock [ (expression) then | (inExpression) end
  (expression) `<=´ (expression)
  (expression) `´, `´ (expression)
  `@´ (expression)
  self
  ]

(classDescriptor) ::= from { (expression) }+ | prop { (expression) }+ 
  | attr { (attrFeat) }+ | feat { (attrFeat) }+ 

(attrFeat) ::= [ `!´ ] (variable) | (atom) | (integer) | unit | true | false 

(methHead) ::= ( [ `!´ ] (variable) | (atom) | unit | true | false )
  [ `´ (´ { (methArg) } [ `...´ ] `´ ]
  [ `=´ (variable) ]

(methArg) ::= [ (feature) `:` ] ( (variable) | `´ | `$´ ) [ `<=´ (expression) ]

Table 9.1: Classes
An access operation: \texttt{@val}. This can be used in any expression that is lexically inside the class definition. In particular, it can be used inside of procedures that are defined inside the class.

An atomic exchange operation: \texttt{x=val<- (expr)}. This first evaluates the expression \texttt{(expr)}. Then it atomically does two operations: it unifies the content of \texttt{val} with \texttt{x} and assigns the expression’s result as \texttt{val}’s new content.

- **Features** (declared with the keyword \texttt{"feat"}). A feature is an instance variable that is stateless and publicly accessible by default. The feature can reference any language entity. It is intended for information that needs to be inspectable outside of the object. Features are stateless by default to give them some protection from public access. This is not a limitation, since a feature can always be bound to a stateful entity such as a cell. Features are accessed by using the dot \texttt{."} notation on objects, as if the object was a record. This emphasizes their stateless property. Section 9.2.3 gives examples of features.

- **Methods** (declared with the keyword \texttt{"meth"}). A method is a kind of procedure that is called in the context of a particular object and that can access the object’s instance variables. By default, methods are visible in the whole program. Any thread that references the object can call its methods. All argument variables in the method head must be distinct, otherwise there is a syntax error.

- **Properties** (declared with the keyword \texttt{"prop"}). A property modifies how an object behaves. For example:

  - The property \texttt{locking} creates a new lock with each object instance. The lock can be accessed inside the class with the \texttt{lock} ... \texttt{end} construct. Locking is explained in Chapter 10.

  - The property \texttt{final} makes the class be a final class, i.e., it cannot be extended with inheritance. Inheritance is explained in Section 9.3.

Section 9.2.3 shows how to initialize attributes and features. In addition to having these kinds of members, Section 9.3 shows how a class can \texttt{inherit} members from other classes. An instance of a class is created with the operation \texttt{New}:

\begin{verbatim}
declar
MyObj={New MyClass init}
\end{verbatim}

This creates a new object \texttt{MyObj} of class \texttt{MyClass} and passes \texttt{init} as the first message to the object. This message is used to initialize the object.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
9.2 Classes as complete ADTs

9.2.3 Initializing attributes and features

Attributes and features can be initialized in two ways. We give examples using features, but they work equally well with attributes.

- **Per class.** A feature or attribute can be given a value that is the same for all instances of a class. This is done by initializing it with `:` in the class definition. For example:

```plaintext
class Apartment
meth init skip end
end
class YorkApt from Apartment
feat
streetName:york
streetNumber:100
wallColor:_
floorSurface:wood
end
Apt1={New YorkApt init}
Apt2={New YorkApt init}
```

All instances, including `Apt1` and `Apt2`, have the same values for all four features. This includes the feature `wallColor`, even though it is still an unbound variable. All instances refer to the *same* unbound variable. It can be bound by binding it in one of the instances, e.g., `Apt1.wallColor=white`. Then all instances will see this value.

- **Per instance.** A feature or attribute can be given a different value per instance. This is done by not initializing it in the class definition. For example:

```plaintext
class OneApt from Apartment
feat streetName
end
Apt3={New OneApt init}
Apt4={New OneApt init}
Apt3.streetName=drottninggatan
Apt4.streetName=rueNeuve
```

Each instance, including `Apt3` and `Apt4`, will initially reference a different unbound variable. These can be bound to *different* values for the features.

- **Per brand.** A *brand* is a set of classes that are related according to some family resemblance, but not by inheritance. Sometimes, but not always, these classes can inherit from a parent class. A feature or attribute can be
given a value that is the same for all members of a brand by initializing with the same variable for all members. For example:\footnote{With apologies to all omitted Linux distributions.} 

```plaintext
declare L=linux
class RedHat
  feat ostype:L
end
class SuSE
  feat ostype:L
end
class Debian
  feat ostype:L
end
```

Each instance of each class will be initialized to the same value.

Initializing an attribute with “;” will make it reference the same entity for all instances, but since an attribute is stateful, this reference can be changed.

### 9.2.4 First-class messages

The principle is simple: messages are records and method headers are patterns that match a record. As a consequence, the following possibilities exist for object calls and method definitions:

- In the object call `{Obj M}`, the following is possible:
  
  1. **Static record as message.** In the simplest case, \( M \) is a record that is known at compile time, e.g., like in the object call `{Counter inc(X)}`.

  2. **Dynamic record as message.** It is possible to call `{Obj M}` where \( M \) is a variable that references a record that is calculated at run time. Because of dynamic typing, it is possible to add new record types at run time (e.g., with Adjoin or List.toRecord).

- In the method definition, the following is possible:

  1. **Fixed argument list.** The method head has record syntax. For example:

     ```plaintext
     meth foo(a:A b:B c:C)
     $ Method body
     end
     ```

     the method header `foo(a:A b:B c:C)` is a static record that must match the message exactly (label `foo` and arity \([a,b,c]\) must match).
Because the header is a record, the features (a, b, and c) can be given in any order (see Section B.3). A class can only have one method definition with a given label, otherwise there is a syntax error.

2. **Flexible argument list.** The method head is a pattern that ends in “...”. For example:

   ```
   meth foo(a:A b:B c:C ...)  
   $ Method body
   end
   ```

   The “...” in the method header means that any message is accepted if it has at least the listed arguments. This means the same as the “...” in patterns, e.g., in a `case` statement.

3. **Variable reference to method head.** The whole method head is referenced by a variable. This is particularly useful with flexible argument lists, but it can also be used with a fixed argument list. For example:

   ```
   meth foo(a:A b:B c:C ...)=M  
   $ Method body
   end
   ```

   The new variable M references the full message as a record. M is declared with a scope equal to the method body.

4. **Optional argument.** A default is given for an argument. The default is used if the argument is not in the message. For example:

   ```
   meth foo(a:A b:B<=V)  
   $ Method body
   end
   ```

   The “<=V” in the method header means that the field b is optional in the call. The method can be called in two ways. The first way is including b, as in `foo(a:1 b:2)`, in which case the optional expression V is disregarded. The second way is without b, as in `foo(a:1)`, in which case the message `foo(a:1 b:V)` is used.

5. **Dynamic method name.** It is possible to calculate the method name of a class at run time, by using an escaped variable identifier for the method header. This is possible because class definitions are executed at run time. The method name has to be known when the class definition is executed. For example:

   ```
   meth !A(bar:X)  
   $ Method body
   end
   ```

   causes the method name to be whatever the variable A was bound to. The variable should be bound to an atom or a name. By using names, this technique can make methods secure (see Section 9.3.3).

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
6. **The otherwise method.** The method header with label otherwise is a catchall that accepts any message for which no other method exists. For example:

```plaintext
meth otherwise(M)  
  % Method body
end
```

A class can only have one method with header otherwise, otherwise there is a syntax error. This method must have just one argument, otherwise a run-time “arity mismatch” error is given. If this method exists, then the object accepts any message. If no method is defined for the message, then the otherwise(M) method is called with the full message in M as a record. This mechanism allows to implement delegation, an alternative to inheritance explained in Section 9.3.4.

All these possibilities are covered by the syntax of Table 9.1. In general, for the call {Obj M}, the compiler tries to statically determine what the object Obj and the method M are. If it can, then it compiles a very fast specialized call instruction. If it cannot, then it compiles a general object call instruction. The general instruction uses caching. The first call is slower, because it looks up the method and caches the result. Subsequent calls find the method in the cache and are almost as fast as the specialized call.

### 9.2.5 First-class attributes and features

Attribute and feature names can be calculated at run time. For example, it is possible to write methods to access and assign any attributes:

```plaintext
class Inspector
  meth get(A X)  
    X=@A
  end
  meth set(A X)  
    A<-X
  end
end
```

The get method can access any attribute and the set method can assign any attribute. Any class can inherit from Inspector to “open up” all its attributes to public use:

```plaintext
class C from AnyClass Inspector end
```

Class C is an opened-up version of AnyClass. This is dangerous for programming but can be very useful for debugging.
Using name values to avoid conflicts

The above technique will not work if \texttt{AnyClass} already has methods called \texttt{get} or \texttt{set}, because they will be overridden by \texttt{Inspector}. To solve this problem, we need to be sure that the methods of \texttt{Inspector} do not exist anywhere else. We can ensure this by using name values for the method names:

\begin{verbatim}
local
  Get=\{NewName\}
  Set=\{NewName\}
in
  class Inspector2
    meth !Get (A X)
      X=@A
    end
    meth !Set (A X)
      A<-X
    end
  end
end
\end{verbatim}

When we add \texttt{Inspector2} to any class, there will never be any conflicts with existing method names. This is because name values are guaranteed to be globally unique and unforgeable. The use of name values to avoid naming conflicts is a general technique that can be used in any context where some entity has to be extended. For example, it can be used to add fields to a record without conflicting with existing fields.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
9.3 Classes as incremental ADTs

As explained before, the main addition that object-oriented programming adds to component-based programming is inheritance. Object-oriented programming allows to define a class incrementally, by extending existing classes. It is not enough to say which classes are extended; to properly define a new ADT more concepts are needed. Our model includes three sets of concepts:

- The first is inheritance itself (Section 9.3.1), which defines which preexisting classes are extended.
- The second is method access control (Section 9.3.2), which defines how to access particular methods both in the new class and in the preexisting classes. It is done with static and dynamic binding and the concept of self.
- The third is visibility control (Section 9.3.3), which defines what part of a program can see a classes’ attributes and methods.

In addition, the model can use first-class messages to implement delegation, a completely different way to define ADTs incrementally (see Section 9.3.4).

9.3.1 Inheritance

Inheritance is a way to construct new classes from existing classes. It defines what attributes, features, and methods are available in the new class. We will restrict our discussion of inheritance to methods. Nonetheless, the same rules apply to attributes and features. The methods available in a class C are defined through a precedence relation on the methods that appear in the class hierarchy. We call this relation the overriding relation:

- A method in class C overrides any method with the same label in all of C’s superclasses.

Classes may inherit from one or more classes, which appear after the keyword from in the class declaration. A class that inherits from exactly one class is said to use single inheritance (sometimes called simple inheritance). Inheriting from more than one class is called multiple inheritance. A class B is a superclass of a class A if:

- B appears in the from declaration of A, or
- B is a superclass of a class appearing in the from declaration of A.

A class hierarchy with the superclass relation can be seen as a directed graph with the current class being the root. The edges are directed towards the subclasses. There are two requirements for the inheritance to be legal. First, the inheritance relation is directed and acyclic. So the following is not allowed:
Second, after striking out all overridden methods, each remaining method should have a unique label and is defined in only one class in the hierarchy. Hence, class C in the following example is illegal because the two methods labeled \texttt{m} remain:

\begin{verbatim}
class A1 meth m(...) ... end end
class B1 meth m(...) ... end end
class A from A1 end
class B from B1 end
class C from A B end
\end{verbatim}

Figure 9.2 shows this hierarchy and a slightly different one that is legal. The class C below is also illegal, since two methods \texttt{m} are available in C:

\begin{verbatim}
class A meth m(...) ... end end
class B meth m(...) ... end end
class C from A B end
\end{verbatim}

Late error detection and language flexibility

If a program containing the above class declaration is compiled in Mozart then the system will not complain. It is only when the program \textit{executes} the declaration that the system will raise an exception. If the program does not execute the declaration then no exception is raised. For example, a program that contains the following source code:

\begin{verbatim}
fun \{StrangeClass\}
  class A meth foo(X) X=a end end
  class B meth foo(X) X=b end end
  class C from A B end
in
  C
end
\end{verbatim}

can be successfully compiled and executed. It is only if \texttt{StrangeClass} is executed that an exception will be raised. This “late error detection” is not just a property of class declarations. It is a general property of the Mozart system that is a consequence of the dynamic nature of the language. Most “interesting events” happen at run time, not compile time. For example, it is possible to define classes whose method names are calculated at run time (see Section 9.2.4).

Because of this dynamic nature, the role of the compiler in Mozart is very small. Very little knowledge of the language semantics is built into the compiler. It does not “execute” any declarations. Some such knowledge could be added, for example the compiler could be extended to detect class hierarchy errors when it can deduce what the method names are. This kind of extension is not essential to the system’s operation. It is an optimization that allows some of the error detection to be done earlier.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
9.3.2 Static and dynamic binding

When executing inside an object, we often want to pass a message \( M \) to the object itself. There are two ways to do this: static and dynamic binding. We explain dynamic binding first, since it is the more usual case:

- **Dynamic binding.** This is written \{self \( M \}\}. This chooses the method matching \( M \) that is visible in the current object. This takes into account the overriding that has been done.

- **Static binding.** This is written \( C, M \) (with a comma), where \( C \) is a class that defines a method matching \( M \). This chooses the method matching \( M \) that is visible in the class \( C \). This takes overriding into account from the root class up to class \( C \), but no further. If the object is of a subclass of \( C \) that has overridden \( M \) again, then this is not taken into account.

Dynamic binding is the only possible behavior for attributes and features. Static binding is not possible for them since the overridden attributes and features simply do not exist, neither in a logical sense (the only object that exists is the instance of the final class) nor in a practical sense (the implementation allocates no memory for them).

When calling an object, how does one choose between static and dynamic binding? There is no general rule, but we find the following guidelines helpful:

- When the object is considered as being used “from the outside”, i.e., we are not implementing its basic functionality but using it, then dynamic binding should be used.

- When we are implementing the object, “from the inside”, then it is sometimes necessary to access overridden methods. For example, an overriding method might need to run the method it overrides. In this case, static binding should be used.

9.3.3 Controlling visibility

The principle of controlling visibility is that it should be possible to limit access to class members, i.e., attributes, methods, and features, according to the requirements of the application architecture. Each member is defined with a scope. The scope is that part of the program text in which the member is visible, i.e., can be accessed by mentioning its name. Usually, the scope is statically defined. It can also be dynamically defined if names are used (see below).

Programming languages usually give a default scope to each member when it is declared. This default can be altered with special keywords. Typical keywords used are public, private, and protected. Unfortunately, different languages use these keywords to define different scopes. Visibility in programming languages is a tricky concept. In the spirit of [32], we will try to bring order to this chaos.
To simplify the discussion, we first introduce some terminology. The attributes and features of an object, taken together, we will call instance variables. The attributes, features, and methods of an object, taken together, we will call instance members or members. This terminology is widely used in the object community.

**Private and public scopes (in the ADT sense)**

The two most basic scopes are private and public, in the following sense:

- A private member is one which is only visible in the object instance. The object instance can see all members defined in its class and its superclasses. Thus private defines a kind of vertical visibility.

- A public member is one which is visible anywhere in the program.

In both Smalltalk and Oz, attributes are private and methods are public according to this definition. In addition, features are public because they are intended to store immutable class properties that can be inspected from the outside.

These definitions of private and public are natural if classes are used to construct ADTs. Let us see why:

- First of all, a class is not the same thing as the ADT it defines! The class is an increment; it defines an ADT as an incremental modification of its superclasses. The class is only needed during the ADT’s construction. The ADT is not an increment; it stands on its own, with all its own attributes and methods. Many of these may come from the superclasses and not from the class.
Second, attributes are internal to the ADT and should be invisible from the outside. This is exactly the definition of private.

Finally, methods and features make up the external interface of the ADT, so they should be visible to all entities that reference the ADT. This is exactly the definition of public.

Constructing other scopes

Techniques for writing programs to control visibility are based essentially on two concepts: lexical scoping and name values. The private and public scopes defined above can be constructed with these two concepts. However, many other scopes can also be defined. For example, we can define the private and protected scopes of C++ and Java, as well as write programs to define more elaborate security policies. The basic technique is to let method headers be name values instead of atoms. A name is an unforgeable constant; the only way to know a name is if someone gives you a reference to it (see Sections B.2 and 5.10.3). In this way, a program can pass the reference in a controlled way, to exactly those areas of the program in which it should be visible.

In the examples of the previous sections, we have used atoms as method names. But atoms are not secure: if a third party finds out the atom’s print representation (either by guessing or by some other way) then he can call the method too. Names are a simple way to plug this kind of security leak. This is important for a software development project with well-defined interfaces between different components. It is even more important for open distributed programs, where code written at different times by different groups can coexist (see Chapter 13).

Private methods (in the C++ sense)

When a method header is a name value, then its scope is limited to all instances of the class, but not to subclasses or their instances. This is exactly private in the sense of C++ and Java. Because of its usefulness, the object system of this chapter gives syntactic support for this technique. There are two ways to write it, depending on whether the name is defined implicitly inside the class or comes from the outside:

- By using a variable identifier as the method header. This implicitly creates a name when the class is defined and binds it to the variable. For example:

```
class C
  meth A(X)
    % Method body
  end
end
```

Method header $A$ is bound to a name. The variable $A$ is only visible inside the class definition. An instance of $C$ can call method $A$ in any other instance.
of C. Method \( A \) is invisible to subclass definitions. This is a kind of horizontal visibility. It corresponds to the concept of \textit{private method} as it exists in C++ and Java (but not in Smalltalk). As Figure 9.3 shows, private in C++ and Java is very different from private in Smalltalk and Oz. In Smalltalk and Oz, private is relative to an \textit{object} and its classes, e.g., \( I3 \) in the figure. In C++ and Java, private is relative to a \textit{class} and its instances, e.g., \( \text{SubSubC} \) in the figure.

- By using an \textit{escaped} variable identifier as the method header. The variable identifier is declared and bound outside of the class. When the class is defined then the method header is bound to whatever the variable is bound to. This is a very general mechanism that can be used to protect methods in many ways. It can also be used for other purposes than security (see Section 9.2.4). Here is an example that does exactly the same as the previous case:

```plaintext
local
    A={NewName}
in
    class C
        meth !A(X)
            % Method body
    end
end
end
```

This creates a name at class definition time, just like in the previous case, and binds the method header to it. In fact, the previous definition is just a short-hand for this example.

Letting the programmer determine the method name allows to define a security policy at a very fine grain. The program can pass the method name to exactly those entities who need to know it.

\textbf{Protected methods (in the C++ sense)}

By default, methods in the object system of this chapter are public. Using names, we can construct the concept of a \textit{protected} method, as it exists in C++ and Java. A method is protected if it is accessible only in the class it is defined or in descendant classes (and all instance objects of these classes). The protected concept is a combination of the Smalltalk notion of private with the C++/Java notion of private: it has both a horizontal and vertical component. In the following class, method \( A \) is protected:

```plaintext
class C
    attr pa:A
    meth A(X) skip end
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
meth foo(...) {self A(5)} end
end

It is protected because the attribute `pa` stores a reference to `A`. Now create a subclass `C1` of `C`. We can access method `A` as follows in the subclass:

class C1 from C
meth b(...) {self @pa(5)} end
end

Method `b` accesses method `A` through the attribute `pa`, which exists in the subclass. The method name can be stored in the attribute because a name is just another kind of value.

Attribute scopes

Attributes are always private. The only way to make them public is by means of methods. Because of dynamic typing, it is possible to define two generic methods that together make all attributes public. Consider the class:

class MakeAttrPublic
meth getattr(A X) @A=X end
meth setattr(A X) A<-X end
end

Any class that inherits from `MakeAttrPublic` will have all its attributes be public. At this point, their scope can be reduced again by using name values for attributes, just like for methods.

Atoms or names as method headers?

When should one use an atom or a name as a method header? By default, atoms are visible everywhere and names are visible nowhere. Most popular object-oriented programming languages (e.g., Smalltalk, C++, and Java) only support atoms as method headers, not names. These languages make atoms usable by adding special operations to restrict their visibility (e.g., `private` and `protected` declarations). On the other hand, names are practical too. Their visibility can be extended by passing around references. But the capability-based approach exemplified by names has not yet become popular. Even in Oz, programmers have mainly kept to atoms. Let us investigate why this is so and see whether it is justified.

Atoms are uniquely identified by their print representations. This means they can be stored in emails, on Web pages, etc. In particular, they can be stored in the programmer’s head! When writing a large program, a method can be called from anywhere by just giving its print representation. On the other hand, with
names this is more awkward: the program itself has somehow to pass the name to the caller. This adds some complexity to the program as well as being a burden for the programmer. So atoms win out both for program simplicity and for the psychological comfort factor during development. Apparently these advantages have hampered the spread of the capability-based approach.

But let us not dismiss names too quickly. A case can be made for the importance of using names for method headers. There are two major advantages. First, it is impossible to have conflicts with inheritance (either single or multiple). Second, encapsulation can be better managed, since an object reference does not necessarily have the right to call all the object’s methods. Therefore, the program as a whole can be made less error-prone and better structured. In an object-oriented program written by more than one person, this may well compensate for the minor advantages of atoms. A final point is that there is no notational burden to using a name in Oz: just capitalize the method header.

### 9.3.4 Delegation

The standard way to achieve inheritance is to use classes to define ADTs incrementally, as explained above. An alternative approach is delegation (sometimes called inheritance by delegation). With this approach, if object \texttt{Obj1} does not understand message \texttt{M}, then \texttt{M} is passed transparently to object \texttt{Obj2}. Usually, an object is free to delegate to any other. No hierarchical structure or other constraint is imposed. In the object system of this chapter, this can be implemented with the \texttt{otherwise(M)} method (see Section 9.2.4). The argument \texttt{M} is a first-class message that can be passed to another object. Here is an example class each of whose instances delegates to a different object:

```oz
class Obj1Class
  attr del
  meth init(del:D ...) 
    del<-D
    ...
  end
  ...
  meth otherwise(M)
    {@del M}
  end
end
```

\texttt{Obj2}={\texttt{New Obj2Class init}}

\texttt{Obj1}={\texttt{New Obj1Class init(del:Obj2)}}

In this example, \texttt{Obj2} is installed as \texttt{Obj1}’s delegate when \texttt{Obj1} is initialized. Here is a second example that defines a function that takes any class and object, and returns a new class whose instances delegate to that object:

```oz
fun {MakeDelegate C Obj} 
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
We assume that class C does not do any delegation. This definition is simple but powerful. It takes advantage of higher-order programming, run-time class declarations, and first-class messages.

### 9.3.5 Checking class membership

Is there a way to check if an object is a member of a particular class? We would like a function `{InstanceOf Obj C}` that returns `true` if Obj is an instance of C and `false` otherwise. This function does not exist in the Mozart libraries, but can be programmed in Oz. The solution takes advantage of higher-order programming, run-time class declarations, and name values.

The solution is to add a feature to each class that identifies the class. All instances of the class have the feature. To check whether an object is an instance of a class, just check whether the feature exists. We define an abstraction that completely hides the feature:

```oz
local
  TypeFeat={NewName}
in
  fun {IsInstanceOf Obj C}
    {HasFeature Obj C.TypeFeat}
  end

  fun {NewClass C}
    class $ from C feat !TypeFeat:AName AName:unit end
  end
end
```

This abstraction defines two functions, `IsInstanceOf` and `NewClass`. The latter is used instead of `class ... end` to define new classes. The abstraction works as follows. Each class is associated with a new name `AName`, which is stored in the feature `TypeFeat`. Each object has one feature `AName` for each class that it inherits from. Using name values for `TypeFeat` and `AName` guarantees that there are no conflicts with other features. Here is an example of how the abstraction is used:

```oz
C1={NewClass class $
  meth init skip end
  meth foo(X) X=23 end
end}
```

```oz
C2={NewClass class $
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
This displays false. The definition of NewClass shows how easy it is to use classes as building blocks. It takes any class C and uses inheritance to create a new class identical to C except that it has an extra feature. It is interesting to unroll the body of NewClass to see where the new names are created:

```plaintext
local
    AName={NewName}

in
    class $ from C
        feat !TypeFeat:AName !AName:unit
    end
end
```

The value of TypeFeat is the name AName, which is created afresh for each new class.

### 9.4 Principal programming techniques

All the programming techniques of stateful programming and declarative programming are still possible in object-oriented programming. Particularly useful are techniques that are based on encapsulation to make programs modular. See the previous chapter, and especially the discussion of component-based programming, which relies on this technique.

This section focuses on the techniques that are specific to object-oriented programming. All these techniques center around the use of inheritance: first, using it correctly, and then, taking advantage of its power. There are two ways to view inheritance:

- **The type view.** In this view, classes are types and subclasses are subtypes. For example, take a LabeledWindow class that inherits from a Window class. All labeled windows are also windows. The type view is consistent with the principle that classes should model real-world entities or some abstract versions of them. In the type view, classes satisfy the substitution property: every operation that accepts an object of class C also accepts objects of a subclass of C. Most object-oriented languages, such as Java and Smalltalk, are designed for the type view [36, 35].

- **The structure view.** In this view, inheritance is just another programming tool that is used to structure programs. This view is more complex to
program with than the type view. Some proponents of object-oriented pro-
gramming reject this view because classes no longer satisfy the substitution
property. A few object-oriented languages, notably Eiffel, are designed to
allow both the type and structure views [65].

In the type view, each class stands on its own two feet, so to speak, as a bona
fide ADT. This is even true for classes that have subclasses; from the viewpoint
of the subclass, the class is an ADT, with sole access through the methods and
its attributes hidden.

This section considers only the type view. However, we consider the structure
view important as well. The structure view is used in the previous section to add
basic functionality to the class abstraction (see Sections 9.3.4 and 9.3.5).

9.4.1 Parameterized classes

The usual way to make classes more generic in object-oriented programming is to
define first an abstract class, i.e., a class in which some methods are left undefined.
Later these methods are defined in the subclasses. For example, suppose you have defined a generic class `GenericSort` for sorting in which the operator `less` is needed. The operator depends on the type of data that is sorted. The generic class leaves out the definition of `less`, which is a method. Other classes can inherit from `GenericSort` and add definitions of `less`, for example, for integers, rationals, or strings. In this case, we specialize the abstract class to form a concrete class, i.e., a class in which all methods are defined.

There is a second natural way to create generic classes, namely by using higher-order programming. Since classes are first-class values, we can define a function that takes some arguments and returns a class that is specialized with these arguments. Figure 9.4 defines the function `SortClass` that takes a class as its argument and returns a sorting class specialized for the argument. We now define two classes, for integers and rationals:

```plaintext
class IntCompare
  meth less(X Y $)
    X<Y
  end
end
class RatCompare
  meth less(X Y $)
    '//'(P Q) = X
    '//'(R S) = Y
    in
    P*S < Q*R
  end
end
```

Now we can execute the following statements:

```plaintext
declare
  ISort={New {SortClass IntCompare} init}
  RSort={New {SortClass RatCompare} init}

  {Browse {ISort qsort([1 2 5 3 4] $)}}
  {Browse {RSort qsort([ '//'(23 3) '//'(34 11) '//'(47 17)] $)}}
```

### 9.4.2 Multiple inheritance

Multiple inheritance is useful when an object has to be two different things in the same program. For example, consider a graphical display that can show a variety of geometrical figures, including circles, lines, and more complex figures. We would like to define a “grouping” operation that can combine any number of figures into a single, composite figure. How can we model this with object-oriented programming? We will design a simple, fully working program. We will use multiple inheritance to add the grouping ability to figures. The idea for this

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
design comes from Bertrand Meyer [65]. This program can easily be extended to a full-fledged graphics package.

Geometric figures

We first define the class Figure to model geometric figures, with methods init (initialize the figure), move(X Y) (move the figure), and display (display the figure):

```{}
class Figure
  meth otherwise(M)
    raise nonexistingMethodError end
end
end
```

This is an abstract class; any attempt to invoke its methods will raise an exception. Actual figures are instances of subclasses of Figure. For example, here is a Line class:

```{}
class Line from Figure
  attr canvas x1 y1 x2 y2
  meth init(Can X1 Y1 X2 Y2)
    canvas<-Can
    x1<-X1 y1<-Y1
    x2<-X2 y2<-Y2
  end
  meth move(X Y)
    x1<-@x1+X y1<-@y1+Y
    x2<-@x2+X y2<-@y2+Y
  end
  meth display
    @{canvas create(line @x1 @y1 @x2 @y2)}
end
end
```

and here is a Circle class:

```{}
class Circle from Figure
  attr canvas x y r
  meth init(Can X Y R)
    canvas<-Can
    x<-X y<-Y r<-R
  end
  meth move(X Y)
    x<-@x+X y<-@y+Y
  end
  meth display
    @{canvas create(oval @x-@r @y-@r @x+@r @y+@r)}
end
end
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Figure 9.5: Class diagram of the graphics package

Figure 9.5 shows how Line and Circle inherit from Figure. This kind of diagram is called a class diagram. It is a part of UML, the Uniform Modeling Language, a widely-used set of techniques for modeling object-oriented programs [32]. Class diagrams are a useful way to visualize the class structure of an object-oriented program. Each class is represented by a rectangle with three parts, containing the class name, the attributes it defines, and the methods it defines. Classes can be connected with inheritance links.

Linked lists

We define the class LinkedList to group figures together, with methods init (initialize the linked list), add(F) (add a figure), and forall(P) (execute {P F} for all figures):

```plaintext
class LinkedList
  attr elem next
  meth init(elem:E<=null next:N<=null)
    elem<-E next<-N
  end
  meth add(E)
    next<-{New LinkedList init(elem:E next:@next)}
  end
  meth forall(P)
    if @elem\=null then {P @elem} end
    if @next\=null then {@next forall(P)} end
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
A linked list is represented as a sequence of instances of this class. The next field of each instance refers to the next one in the list. The last element has the next field equal to null. There is always at least one element in the list, called the header. The header is not an element that it seen by users of the linked list; it is just needed for the implementation. The header always has the elem field equal to null. Therefore an empty linked list corresponds to a header node with both elem and next fields equal to null.

The forall(\(P\)) method is especially interesting. The procedure \(P\) is a “packet of work” to be done on all elements of the linked list. This is a good example of using the declarative model together with the object-oriented model. The simplest way to encapsulate any packet of work is by using procedural abstraction.

Composite figures

What is a composite figure? It is both a figure and a linked list of figures. Therefore we define a class CompositeFigure that inherits from both Figure and LinkedList:

```plaintext
class CompositeFigure from Figure LinkedList
  meth init
    LinkedList,init
  end
  meth move(X Y)
    {self forall(proc {$ F} {F move(X Y)} end)}
  end
  meth display
    {self forall(proc {$ F} {F display} end)}
  end
end
```

Figure 9.5 shows the multiple inheritance. The multiple inheritance is correct because the two functionalities are completely different and have no undesirable interaction. The init method is careful to initialize the linked list. It does not need to initialize the figure. As in all figures, there is a move and a display method. The move(X Y) method moves all figures in the linked list. The display method displays all figures in the linked list.

Do you see the beauty of this design? With it, a figure can consist of other figures, some of which consist of other figures, and so forth, to any number of levels. The inheritance structure guarantees that moving and displaying will always work correctly.

Example execution

Let us run this example. First, we set up a window with a graphics display field:
9.4 Principal programming techniques

\[\text{Figure 9.6: Drawing in the graphics package}\]

\begin{verbatim}
declare
W=250 H=150 Can
Window={Build
td(canvas(width:W height:H bg:white handle:Can)))}
{Window show}
This uses the QTk graphics tool, which is explained in Chapter 12. For now just assume that this sets up a canvas, which is the drawing field for our geometric figures. Next, we define a composite figure \( F_1 \) containing a triangle and a circle:

\begin{verbatim}
declare
F1={New CompositeFigure init}
{F1 add({New Line init(Can 50 50 150 50) })}
{F1 add({New Line init(Can 150 50 100 125) })}
{F1 add({New Line init(Can 100 125 50 50) })}
{F1 add({New Circle init(Can 100 75 20) })}
\end{verbatim}

We can display this figure as follows:

\begin{verbatim}
{F1 display}
\end{verbatim}

This displays the figure once. Let us move the figure around and display it each time:

\begin{verbatim}
for I in 1..10 do {F1 display} {F1 move(3^2)} end
\end{verbatim}

Figure 9.6 shows the result.

\textbf{Composite figures with single inheritance}

Instead of defining \texttt{CompositeFigure} with multiple inheritance, we can define it using single inheritance by putting the list of figures in an attribute. This gives:

\begin{verbatim}
class CompositeFigure from Figure
\end{verbatim}

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
attr figlist
meth init
  figlist<-{New LinkedList init}
end
meth add(F)
  {@figlist add(F)}
end
meth move(X Y)
  {@figlist forall(proc ${ F} {F move(X Y)} end)}
end
meth display
  {@figlist forall(proc ${ F} {F display} end)}
end
end

Figure 9.7 shows the class diagram for this case. The link between CompositeFigure and LinkedList is called an association. It represents a relationship between the two classes. The numbers attached to the two ends are cardinalities; each number says how many elements there are for a particular instance. The number 1 on the linked list side means that there is exactly one linked list per composite figure, and similarly for the other side. The association link is a specification; it does not say how it is implemented. In our case, each composite figure has a figlist attribute that references a linked list.

The example execution we gave before will also work in the single inheritance case. What are the trade-offs in using single or multiple inheritance in this example? In both cases, the figures that make up the composite figure are encapsulated. The main difference is that multiple inheritance brings the operations of linked lists up to the same level as figures:

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
With multiple inheritance, a composite figure is also a linked list. All the operations of the LinkedList class can be used directly on composite figures. This is important if we want to do linked list calculations with composite figures.

With single inheritance, a composite figure completely hides its structure. This is important if we want to protect the composite figure from any calculations other than those defined in its class.

Scaling it up

It is straightforward to extend this example to be a full-fledged graphics package. Here are some of the changes that should be made:

- Many more figures can be defined to inherit from Figure.

- In the current implementation, figures are tied to their canvas. This has the advantage that it allows figures to be spread over multiple canvases. But usually we will not want this ability. Rather, we would like to be able to draw the same figure on different canvases. This means that the canvas should not be an attribute of figure objects but be passed as argument to the display method.

- A journaling facility can be added. That is, it should be possible to record sequences of drawing commands, i.e., sequences of calls to figures. These recordings represent drawings at a high level of abstraction. They can then be manipulated by the application, stored on files, passed to other applications, etc.

- The display method should be able to pass arbitrary parameters from the application program, through the graphics package, to the underlying graphics subsystem. In the Line and Circle classes, we change it as follows:

```plaintext
meth display(...) = meth
  (@canvas {Adjoin M create(line @x1 @y1 @x2 @y2)})
end
```

The Adjoin operation combines two record arguments, where the second argument overrides the first in the case of conflicts. This allows arbitrary parameters to be passed through display to the canvas drawing command. For example, the call `{F display(fill:red width:3)}` draws a red figure of width 3.
9.4.3 Rules of thumb for multiple inheritance

- Give examples of 'mixin' classes (a good example is the animation class, but this requires concurrency: maybe put in Concurrent State chapter?)

Multiple inheritance is a powerful technique that has to be used with care. We recommend that you use multiple inheritance as follows:

- Multiple inheritance works well when combining two completely independent abstractions. For example, figures and linked lists have nothing in common, so they can be combined fruitfully.

- Multiple inheritance is much harder to use correctly when the abstractions have something in common. For example, creating a WorkStudy class from Student and Employee is dubious, because students and employees are both human beings. They may in fact both inherit from a common Person class. Even if they do not have a shared ancestor, there can be problems if they have some concepts in common.

- What happens when sibling superclasses share (directly or indirectly) a common ancestor that specifies a stateful object (i.e., it has attributes)? This is known as the implementation-sharing problem. This can lead to duplicated operations on the common ancestor. This typically happens when initializing an object. The initialization method usually has to initialize its superclasses, so the common ancestor is initialized twice. The only remedy is to understand carefully the inheritance hierarchy to avoid such duplication. Alternatively, you should only inherit from multiple classes that do not share a stateful common ancestor.

- When name clashes occur, i.e., the same method name is used between adjacent superclasses, then the program must define a local method that overrides the conflict-causing methods. Otherwise the object system gives an error message. A simple way to avoid name clashes is to use name values as method headers. This is a useful technique for classes, such as mixin classes, that are often inherited from by multiple inheritance.

9.4.4 Limitations of class diagrams

Class diagrams are excellent tools for visualizing the class structure of an application. They are at the heart of the UML approach to modeling object-oriented applications, and as such they enjoy widespread use. Their popularity has often masked their limitations. They have three clear limitations:

- They do not specify what classes do. For example, if the methods of a class enforce an invariant, then this invariant does not show up in the class diagram.
They do not model the dynamic behavior of the application, i.e., its evolution over time. Dynamic behavior is both large-scale and small-scale. Applications often go through different phases, for which different class diagrams are valid. Application are often concurrent, with independent parts that interact in coordinated ways.

- They only model one level in the application’s component hierarchy. As Section 8.9 explains, well-structured applications have a hierarchical decomposition. Classes and objects are near the base of this hierarchy. A class diagram explains the decomposition at this level.

The UML approach recognizes these limitations and provides tools that partially alleviate them, e.g., the interaction diagram and the package diagram. Interaction diagrams model part of the dynamic behavior. Package diagrams explain the decomposition at a higher level than class diagrams.

9.5 Relation to other computation models

Object-oriented programming is one way to structure stateful programs. Its primary characterizing feature is inheritance. There are other ways to organize state without using inheritance, for example as explained in Chapter 8. This section examines how these other ways relate to object-oriented programming.

9.5.1 Object-based and component-based programming

Object-based programming is object-oriented programming without inheritance. This is very close to component-based programming, but using the class syntax. This gives a convenient notation for encapsulating state and defining multiple operations on it. Without inheritance, the object abstraction becomes much simpler. There are no problems of overriding and conflict in multiple inheritance. Static and dynamic binding are identical.

9.5.2 Higher-order programming

Object-oriented programming and higher-order programming are closely related. For example, let us examine the case of a sorting routine that is parameterized by an order function. A new sorting routine can be created by giving a particular order function. In higher-order programming, this can be written as follows:

```plaintext
proc {NewSortRoutine Order SortRoutine}
  proc {SortRoutine InL OutL}
    $ ... {Order X Y} calculates order
  end
end
```

In object-oriented programming, this can be written as follows:

```plaintext
Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
```
class SortRoutineClass
  attr ord
  meth init(Order)
    ord<-Order
  end
  meth sort(InL OutL)
    % ... {&ord order(X Y &)} calculates order
  end
end

The order relation itself is written as follows:

proc {Order X Y B}
  B=(X<Y)
end

or as follows:

class OrderClass
  meth init skip end
  meth order(X Y B)
    B=(X<Y)
  end
end

Instantiating the sorting routine is then written as follows:

declare
  SortRoutine={NewSortRoutine Order}
end

or as follows:

declare
  SortRoutine={New SortRoutineClass init({New OrderClass init})}
end

It is clear that procedures and objects are closely related. Let us now compare higher-order and object-oriented programming more carefully. The main difference is that object-oriented programming “embellishes” higher-order programming. It is a richer abstraction that provides a collection of additional idioms beyond procedural abstraction:

- **Encapsulated state** can be defined and used easily.

- **Multiple methods** that share the encapsulated state can be defined easily. Invoking an object picks one of them.

- **Classes** are provided, which define a set of methods and can be instantiated. Each instance has a fresh encapsulated state. If objects are like procedures, then classes are like procedures that return procedures.

- **Inheritance** is provided, to define new sets of methods from existing sets, by extending, modifying, and combining existing ones. Static and dynamic binding make this ability particularly rich.
9.5 Relation to other computation models

- Different kinds of visibility can be defined between classes and objects. Attributes and methods can be private, public, protected or have some other, programmer-defined visibility.

It is important to note that these mechanisms do not provide any fundamentally new ability. They can be completely defined with higher-order programming and explicit state. On the other hand, the mechanisms are useful idioms and lead to a programming style that is often convenient.

Object-oriented programming is an abstraction that provides a rich notation to use any or all of these mechanisms together, whenever they are needed. This richness is a double-edged sword. On the one hand, it makes the abstraction particularly useful for programming. On the other hand, the abstraction has a complex semantics and is hard to reason about. For this reason, we recommend to use object-orientation only in those cases when it significantly simplifies program structure, e.g., when there is a clear need for inheritance: the program contains a set of closely-related abstract data types. In other cases, we recommend to use other, simpler computation models.

The object system defined in this chapter is particularly close to higher-order programming. Not all object systems are so close. In particular, the following characteristics are not always present:

- Classes as values in the language.

- Full lexical scoping. For example, allowing a class to be defined inside the scope of another class, or allowing an object to have external references. Full lexical scoping means that the language supports closures.

- First-class messages. Usually, all message headers and method headers both have to be known at compile time.

Many object-oriented languages, e.g., Java and C++, do not support full higher-order programming because they cannot define closures with lexical scoping at run time. In these languages, some of the abilities of higher-order programming can be obtained through encapsulation and inheritance:

- A closure can be encoded as an object. The object’s attributes represent the closure’s external references and the method arguments are the closure’s arguments. When the object is created, its attributes are initialized with the external references. The object can be passed around and called just like a closure. With a little bit of discipline from the programmer, this allows for programming with closures, just like in true higher-order programming.

- A generic procedure can be encoded as an abstract class. A generic procedure is one that takes procedure arguments and returns a specific procedure. For example a generic sorting routine can take a comparison operation for a given type and return a sorting routine that sorts arrays of that type. An abstract class is a class with undefined methods. The methods are defined in subclasses.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
9.5.3 Should everything be an object?

In the area of object-oriented programming, one sometimes hears claims that “everything should be an object”. This section takes a closer look at this intuitively appealing claim to see what it is really trying to say. Usually it is used in a fairly imprecise way. As far as we can tell, the two most common interpretations of it are as follows. Sometimes a narrow view is taken, that everything should be stateful, defined by a class, and accessed with a uniform object syntax. Sometimes a broad view is taken, that everything should be an ADT.

The importance of abstractions

It is clear that the narrow view is too naive. Let us examine its premises. First, stateless entities can play an important role. With them, the powerful reasoning techniques of declarative programming become possible. Modern object-oriented languages such as Java support stateless objects. In addition, they are essential for making transparent distributed programming practical (see Chapter 13).

Second, the formal simplicity of a uniform syntax is not relevant for readability. For example, Scheme has a uniform syntax which does not necessarily make it more readable. An important principle (among others) in syntax design is \textit{form mirrors content}: differences in semantics should be visible as differences in syntax. Having a uniform syntax does not aid this principle. On the contrary, the uniform syntax just moves the richness of the language away from the syntax and into the names of objects and classes. This adds a second layer of syntax, making the language more verbose.

Smalltalk-80 is a good example of a language which is sometimes used to justify the narrow view [35]. All data types in Smalltalk, including simple ones like integers, are objects. However, the designers of Smalltalk themselves have a more nuanced view that is closer to the broad view [47]. Not everything in Smalltalk is an object; there is a concept called “block” that is in fact a lexically-scoped closure used for higher-order programming. The syntax of Smalltalk is less uniform than Scheme’s and more readable.

The broad view is completely consistent with the use of multiple computation models advocated in this book. In this view, building a system consists primarily in designing and implementing abstractions. In this context, the main contribution of object-oriented programming is the concept of \textit{inheritance} as a new technique for structuring a system. The new concept is added to the programmer’s toolbox, where it sits aside other tools such as higher-orderness and concurrency.

The narrow view considers the object-oriented abstraction as a kind of all-purpose programming tool, a “Swiss Army knife”, to be used as often as possible. As this section explains, this view is too naive. In our experience the broad view is the correct one.

\footnote{In Java, they are objects in which all attributes are annotated \texttt{final}.}


```plaintext
class Queue
  attr front back
  meth init
    Q in
    front<-Q
    back<-Q
  end

  meth put(X)
    Q in
    @back=X|Q
    back<-Q
  end

  meth get(X)
    Q in
    {Wait @front}
    @front=X|Q
    front<-Q
  end
end
```

Figure 9.8: Queue (object-oriented version with stateless list)

Problems with a uniform object syntax

If a language uses a uniform object syntax, then programs tend to be more verbose, and arguably harder to read. This is because not all data types are easily represented as objects. For example, stateless values can be given a very natural, compact syntax. A list value can be created just by mentioning it, e.g.:

```
LV=[1 2 3]
```

This is approximately the syntax followed by languages that support symbolic programming, such as Prolog, Haskell, Erlang, and their relatives. This contrasts with the use of a uniform object syntax:

```
ListValue *lv=
    new ListValue(1, new ListValue(2, new ListValue(3, nil)));
```

This is the syntax of C++, which is similar to Java. For decomposing a list value, there is another natural notation using pattern matching:

```
case LV of X|LV2 then ... end
```

This also is cumbersome to do in a uniform object syntax. There is a further increase in verbosity when doing concurrent programming in the object syntax. This is because the uniform syntax requires explicit synchronization. This is not
class Pair
  attr elem next
  meth put(E N)
    elem<-E
    next<-N
  end
  meth get(E N)
    @elem=E
    @next=N
  end
end

class Queue
  attr front back
  meth init
    Q={New Pair put(dummy null)}
    in
      front<-Q
      back<-Q
  end

  meth put(X)
    Q={New Pair put(dummy null)}
    in
      {@back put(X Q)}
      back<-Q
  end

  meth get(X)
    if @front==@back then
      raise error end
    else Q in
      {@front get(X Q)}
      front<-Q
    end
  end
end

Figure 9.9: Queue (object-oriented version with object list)
true for the case syntax above, which is sufficient for concurrent programming if the model does implicit dataflow synchronization.

Another, more realistic example is the graphic user interface tool of Chapter 12. Inspired by this tool, Christophe Ponsard built a Java prototype of a similar tool. The Java version is more cumbersome to use than the Oz version, primarily because Java has no syntactic support for record values. Unfortunately, this verbosity is an inherent property of the Java language. There is no simple way around it.

We give a third example completely within the stateful model, using its syntax for both stateless values and objects. We show two implementations of a queue class, one using stateless lists and dataflow variables and the other using only objects. Figure 9.8 shows the queue class implemented with a stateless list. Figure 9.9 shows the queue class implemented with objects only. It is instructive to carefully compare these two definitions:

- The queue with stateless list is shorter and simpler. If \texttt{get} is called when the queue is empty, it blocks. This is because an empty queue is just an unbound dataflow variable.

- The queue with object list is longer and more complex. The queue requires a second class, \texttt{Pair}, to define the object list. In addition, the \texttt{put} and \texttt{get} methods are harder to understand. If \texttt{get} is called when the queue is empty, it raises an exception. To make it block in this case, a monitor would have to be added. Monitors are more complex than dataflow variables.

The use of objects together with stateless data has a long history, e.g., it is done in CLOS (the Common Lisp Object System) [71] and also in Objective Caml [15].

9.6 Constructing the object system

The complete object system can be constructed in a straightforward way from the declarative stateful computation model. In particular, the main characteristics come from the combination of higher-order programming with explicit state. With this construction, you will understand objects and classes completely. In addition, we show how to construct metaclasses, which are a way to parameterize the basic class operations. Metaclasses are a standard way to make object-oriented programming more flexible.

While the construction of this section works well and is reasonably efficient, a real implementation can add optimizations to do even better. For example, a real implementation can make an object invocation be as fast as a procedure call. This section does not give these optimizations.

9.6.1 Abstraction diagram

- Need to coordinate with the 'Relation to higher-order programming' section.
The first step in understanding how to build an object system is to understand how the different parts are related. Object-oriented programming defines a hierarchy of abstractions that are related to each other by a kind of “specification-implementation” relationship. There are many variations on this hierarchy. We give a simple one that has most of the main ideas. Here are the abstractions, in order from most concrete to most abstract:

- **Running object.** A running object is an active invocation of an object. It associates a thread to an object. It contains a set of environment frames (the part of the thread’s stack that is created while executing the object) as well as an object.

- **Object.** An object is a chunk that encapsulates a state (a cell) and a set of procedures that reference the state.

- **Class.** A class is a chunk that encapsulates a set of procedures named by literals and a set of attributes, which are just literals. The procedures are called *methods*. Methods take a state as argument for each attribute and modify that state. Methods can only call each other indirectly, through the literals that name them. Often the following distinction is useful:
  
  - **Abstract class.** An abstract class is a class in which some methods are called that have no definition in the class.

  - **Concrete class.** A concrete class is a class in which all methods that are called are also defined.

If first-class messages are supported by the language, then invocations of the form `{Obj M}` are possible where M is calculated at run time. If such an invocation exists in the program, then the conceptual distinction between abstract and concrete class disappears. Executing the invocation `{Obj M}` raises an exception if M does not exist in Obj.

- **Metaclass.** A metaclass is a class with a particular set of methods that correspond to the basic operations of a class, for example: object creation, inheritance policy (which methods to inherit), method call, method return, choice of method to call, attribute assignment, attribute access, self call. Writing these methods allows to customize the semantics of objects.

Figure 9.10 shows how these concepts are related. There are three relationships, “invocation of”, “instance of”, and “inherits from”. These relationships have the following intuitive meanings:

- A running object is created when a thread invokes an object. The running object exists until the thread’s execution leaves it. Multiple invocations of the same object can exist simultaneously.
9.6 Constructing the object system

- An object can be created as an instance of a class. If the object system distinguishes between abstract and concrete classes, then it is usually only possible to create instances of concrete classes. The object exists forever.\(^3\) The object encapsulates a cell that was created especially for it. Multiple instances of the same class can exist simultaneously.

- A class can be created that inherits from a list of other classes. The new class exists forever. Inheritance takes a set of methods and a list of classes and returns a new class with a new set of methods. Multiple classes that inherit from the same class can exist simultaneously. If one class can inherit from several classes, then we have multiple inheritance. Otherwise, if one class can inherit only from one class, we have single inheritance.

- A class can be created as an instance of a metaclass. The new class exists forever. The basic operations of the class are defined by particular methods of the metaclass. Multiple instances of the same metaclass can exist simultaneously.

9.6.2 Constructing classes

We first explain what a class is. A class is a chunk, i.e., a record that is protected from snooping because it has no arity operation (see Section B.4). The chunk contains:

- A set of methods in a method table.

\(^3\)In practice, until there are no more references to it from the actively running program. See the discussion on garbage collection in Section 5.13.2.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
**Figure 9.11: An example of class construction**

- A set of attribute names, giving the attributes that each class instance (object) will possess. Each attribute is a stateful cell that is accessed by the attribute name, which is either an atom or an Oz name.

- A set of feature names, giving the features that each instance of the class will possess. A feature is a reference that is accessed by the feature name, which is either an atom or an Oz name. Feature values are accessible from outside the objects defined by the class. Usually, features are used to reference important constants that belong to the class.

Figure 9.11 shows how a class is constructed according to this definition. In this basic form, classes are values, i.e., they are stateless. They are just specifications of how the objects should behave. The figure shows a `Counter` class that has exactly the same behavior as the `Counter` class defined in Section 9.2.1. This class has a single attribute accessed by the atom `val`. It has a method table, which has three methods accessed through the features `browse`, `init`, and `inc`. Each method is a three-argument procedure that takes a message, which is always a record, an extra parameter representing the state of the current object, and the object itself. As we can see, the method `init` assigns the value `Value` to the attribute `val`, the method `inc` increments the attribute `val`, and the method `browse` browses the current value of `val`. 

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
9.6 Constructing the object system

9.6.3 Constructing objects

Now we show how to create objects of class `Counter`. Figure 9.12 shows a generic procedure `NewObject` that creates an object from any class. It first creates an object state, a record, from the attributes of the class. Then it initializes each field of this record to a cell (with an unbound initial value). We use the iterator `Record.forAll` to iterate over all fields of a record. Calling `NewObject` returns a procedure `Obj` that identifies the object. The state of the object is visible only within `Obj`. One may say that `Obj` is a procedure that *encapsulates* the state.\(^4\)

We can try out `NewObject` as follows:

```oz
C={NewObject Counter init(0)}
{C inc(6)} {C inc(6)}
{C browse}
```

This behaves in exactly the same way as the example of Section 9.2.1.

9.6.4 Constructing and instantiating metaclasses

- Explain what part of object execution can be parameterized.
  In particular, the following: object creation, method call, method return, attribute assign, attribute access, inheritance policy. There are many variations on the theme. Show a simple one that gives the main idea and that can be useful.
- Define a version of `NewObject` that takes these parameters from methods in a `Class` argument. This class is the `MetaClass`.
- Define a version of `NewClass` that takes a `Class` argument, with method names corresponding to the parameters, creates the parameters, and returns a class definition function `NC` that puts the extra parameters in any class it creates.

\(^4\)This is a slight simplification; an object in Oz is actually a chunk that has the above procedure in one of its fields; other fields contain the object features.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
- Show how to create a class with NC.
- Give an example to show why this technique is useful.

9.6.5 Stateful classes

Sometimes it is useful to have stateful classes. This is usually done to share updatable data between all instances of the class. In Java this is called \textit{static variables}. There are two ways to make classes stateful in Oz:

- Features can reference stateful entities such as cells or objects. These references are accessible from outside the classes’ objects.

- Classes can have external references, in the same way that procedures have external references. Such classes are known as \textit{parameterized} classes. These references are inaccessible from outside the classes’ objects. The references can be to stateful entities. Parameterized classes are explained further in Section 9.4.1.

9.7 Case studies

9.8 Exercises

1. \textbf{Uninitialized objects}. The function \texttt{New} creates a new object when given a class and an initial message. Write another function \texttt{New2} that does not require an initial message. That is, the call \texttt{Obj={New2 Class}} creates a new object without initializing it. Hint: write \texttt{New2} in terms of \texttt{New}.
Chapter 10

Concurrency and State

The concurrent stateful model is a simple extension to the declarative model that adds both explicit threads and explicit state. This is the most expressive model of the book, but also the hardest to program in.

The inherent difficulty of the model

Let us first see exactly why the model is so difficult. Execution consists of multiple threads, all executing independently and all accessing shared cells. At some level, a thread’s execution can be seen as a sequence of atomic instructions. For a cell, these are Access, Assign, and Exchange. Because of the interleaving semantics, all execution happens as if there was one global order of operations. All operations of all threads are therefore “interleaved” to make this order. There are many possible interleavings; their number is limited only by data dependencies (calculations needing results of others). Any particular execution realizes an interleaving. Because thread scheduling is nondeterministic, there is no way to know which interleaving will be chosen.

But just how many interleavings are possible? Let us consider a simple case: two threads, each doing $k$ cell operations. Thread $T_1$ does the operations $a_1, a_2, ..., a_k$ and thread $T_2$ does $b_1, b_2, ..., b_k$. How many possible executions are there, interleaving all these operations? It is easy to see that the number is $\binom{2k}{k}$.

Any interleaved execution consists of $2k$ operations, of which each thread takes $k$. Consider these operations as integers from 1 to $2k$, put in a set. Then $T_1$ takes $k$ integers from this set and $T_2$ gets the others. This number is exponential in $k$.\footnote{Using Stirling’s formula we approximate it as $2^{2k}/\sqrt{\pi k}$.} For three or more threads, the number of interleavings is even bigger (see Exercises).

It is possible to write algorithms in this model and prove their correctness by reasoning on all possible interleavings. For example, given that the only atomic operations on cells are Access and Assign, then Baker’s algorithm implements mutual exclusion. Even though Baker’s algorithm is very short (about 20 lines of...}
code), the reasoning is already quite difficult. For bigger programs, this technique rapidly becomes impractical. It is unwieldy and interleavings are easy to overlook.

**Why not use a simpler model?**

Given the inherent difficulty of programming in the concurrent stateful model, an obvious question is why not stick with the concurrent declarative model of Chapter 6? It is enormously simpler to program in than the concurrent stateful model. It is almost as easy to reason in as the declarative model, which is sequential.

Let us briefly examine why the concurrent declarative model is so easy. It is because dataflow variables are monotonic: they can be bound to just one value. Once bound, the value does not change. Threads that share a dataflow variable, e.g., a stream, can therefore calculate with the stream as if it were a simple value. This is in contrast to cells, which are nonmonotonic: they can be assigned arbitrary values. Threads that share a cell cannot make any assumptions about its content: at any time, it can be completely different from any previous value.

The problem with the concurrent declarative model is that threads must communicate in a kind of “lock-step” or “systolic” fashion. Two threads communicating with a third thread cannot execute independently; they must coordinate with each other. This is a consequence of the fact that the model is still declarative, and hence deterministic.

To allow two threads to be completely independent, for example, to make independent queries to a common server or to independently increment a shared state, we have to leave the realm of declarative models. This is because two independent entities communicating with a third introduce an observable nondeterminism. One approach is to add nondeterministic choice (a “guarded command” in the terminology of Dijkstra) which allows to choose any successful alternative from among several. Another, more practical approach is to add explicit state. This leads us directly to the concurrent stateful model of this chapter.

**Getting around the difficulty**

Programming in the concurrent stateful model is largely a matter of managing the interleavings. This chapter introduces two successful approaches:

- **Message passing.** In this approach, programs consist of active objects that send asynchronous messages to each other. Internally, an active object executes in a single thread.

- **Atomic actions.** In this approach, programs consist of passive objects that are invoked by threads. Abstractions are used to build large atomic actions (e.g., using locking, monitors, or transactions) so that the number of possible interleavings is small.
Each approach has its advantages and disadvantages. We will investigate both approaches in this chapter. The technique of invariants, as explained in Chapter 8, can be used in both approaches to reason about programs. The two approaches are equivalent in a theoretical sense, but not in a practical sense: a program using one approach can be rewritten to use the other approach, but it may not be as easy to understand [58].

Explicit state can be added in many ways. This chapter uses two: cells and ports. A cell is a kind of shared assignable variable. Cells were introduced in Chapter 8. They naturally support the atomic action approach. A port is a kind of communication channel. It naturally supports the message passing approach. Cells can easily implement ports and vice versa, so the two concepts have the same expressive power. This chapter contains four sections:

- Section 10.1 defines the concurrent stateful model.
- Section 10.2 brings together and compares the different concurrent models that we have introduced in the book.
- Section 10.3 gives the principal programming techniques of message passing, using ports as the basic primitive.
- Section 10.4 gives the principal programming techniques of atomic actions, using cells as the basic primitive.

**Concurrent Programming in Java**

A useful companion to this chapter is the book *Concurrent Programming in Java, Second Edition*, by Doug Lea [59]. This book gives a rich panoply of programming techniques that are particularly well-suited to Java, a popular concurrent object-oriented language (see Chapter 11). However, they can be used in many other languages including Oz. Most of the techniques use atomic actions; message passing is mentioned only in passing.

The major difference between the Java book and this chapter is that the book assumes threads are expensive, which is the case for all current Java implementations. Because of this, the book adds a conceptual level between threads and procedures, called *tasks*, and advises the programmer to schedule multiple tasks on one thread. Assuming threads are cheap broadens the range of practical programming techniques and often allows for simpler solutions. For example, using active objects can often give a simpler program structure. Active objects are only practical if threads are very lightweight.
Concurrency and State

Figure 10.1: The declarative model with concurrency and state

![Diagram showing multiple statements (threads) with an immutable store (single-assignment) and a mutable store.]

| (statement) ::= |
| skip | Empty statement |
| (statement) (statement) | Statement sequence |
| local X in (statement) end | Variable declaration |
| X=Y | Bind to variable |
| X=f(l1:X1 ... ln:Xn) | Bind to record |
| if X then (statement) else (statement) end | Test for boolean |
| proc {P X1 ... Xn} (statement) end | Procedure declaration |
| {P X1 ... Xn} | Procedure call |
| try (statement) catch X then (statement) end | Exception handler |
| raise X end | Raise exception |
| thread (statement) end | Thread creation |
| {ByNeed P X} | Trigger creation |
| {NewCell X C} | Cell declaration |
| {Exchange C X Y} | Cell exchange |

Table 10.1: The kernel language with concurrency and state

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
10.1 The declarative model with concurrency and state

Chapter 8 adds explicit state to the declarative model. This allows to do object-oriented programming. Chapter 6 adds concurrency to the declarative model. This allows to have multiple active entities that evolve independently. The next step is to add both explicit state and concurrency to the declarative model. The resulting model, called the *concurrent stateful* model, is shown in Figure 10.1. Its kernel language is defined in Table 10.1. The model has both multiple threads and a mutable store, in addition to dataflow variables. This model allows to do concurrent object-oriented programming, which is not possible in any earlier model without doing complicated encodings.

The model adds state in the form of cells, in the same way as Chapter 8 does. Another way to add state is by means of ports, which are asynchronous communication channels. Ports are explained in Section 10.3.1, see below. Cells and ports are theoretically equivalent, i.e., each can be simply defined in terms of the other. But they show two quite different points of view for concurrent programming. Cells are useful when threads share state, e.g., when there are passive objects shared between threads. Ports are useful when threads send messages, e.g., when there are active objects sending messages.

10.2 Approaches to programming with concurrency

By now, we have seen many different ways to write concurrent programs. Before diving into programming with the concurrent stateful model, let us make a slight detour and put all these ways into perspective.

10.2.1 Four practical approaches

For the programmer, there are four main practical approaches to writing concurrent programs:

- **Sequential programming.** This is the baseline approach that has no concurrency.

- **Declarative concurrency.** This is concurrency without any observable nondeterminism.

- **Message passing.** This is message passing between active objects, which are single-threaded to limit the number of interleavings.

- **Atomic actions.** This is threads updating shared passive objects, using coarse-grained atomic actions to limit the number of interleavings.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Let us explore the relationships between these four approaches.

**Sequential programming**

In a sequential model, there is a total order among all operations. This is the strongest order invariant a program can have. We have seen two ways that this order can be relaxed a little, while still keeping a sequential model:

- **“Order-determining” concurrency** (Section 6.3). In this model, all operations execute in a total order, like with sequential execution, but the order is unknown to the programmer. Concurrent execution with dataflow finds the order dynamically. Preemption between threads is data-driven.

- **Coroutining** (Section 6.7.1). In this model, the program decides when to pass control to another thread. Preemption between threads is done explicitly. Section 10.2.2 shows how to implement coroutines.

Both of these variant models are still deterministic.

**Declarative concurrency**

The concurrent declarative model of Chapter 6 adds threads to the declarative model. There is nondeterminism but it is not observable. This allows to build a dynamic network of active objects communicating by means of streams. Each active object must at all times know from which stream its next input will come. Programs are deterministic, and therefore declarative, as long as they do not raise exceptions (since exceptions introduce an observable nondeterminism).

Lazy execution (Section 6.6) can be seen as part of this model. Execution order is determined by need, i.e., calculations are initiated when their results are needed. Lazy execution is implemented with threads and by-need triggers.

**Message passing**

Message passing is a basic programming style of the concurrent stateful model. It consists of a network of active objects communicating with each other through asynchronous message passing. Each active object has a stream or a mailbox of incoming messages. The object decides when to handle each messages. The object processes the messages sequentially. This limits the possible interleavings and allows us to reason using invariants.

**Atomic actions**

Atomic actions is another basic programming style in the concurrent stateful model. It consists of a set of threads accessing a set of shared passive objects. The threads coordinate among each other when accessing the shared objects. They do this by means of coarse-grained atomic actions, e.g., locks, monitors,
10.2 Approaches to programming with concurrency

and transactions. Again, this limits the possible interleavings and allows us to reason using invariants.

**Which concurrent model to use?**

How do we decide which approach to use when writing a concurrent program? Here are a few rules of thumb:

- Stick with the least concurrent model that suffices for your program. For example, if using concurrency does not simplify the architecture of the program, then stick with a sequential model. If your program does not have any observable nondeterminism, such as independent clients interacting with a server, then stick with the concurrent declarative model.

- If you absolutely need the concurrent stateful model, then try first to use either the message passing or atomic action approach. The message passing approach is often the best for multi-agent programs, i.e., programs that consist of autonomous entities (“agents”) that communicate with each other. The atomic action approach is often the best for data-centered programs, i.e., programs that consist of a large repository of data (“database”) that is accessed and updated concurrently. Both approaches can be used together for different parts of the same application.

- Modularize your program and concentrate the concurrency aspects in as few places as possible. Most of the time, large parts of the program can be sequential or use declarative concurrency. For example, active objects can be used as front ends to passive objects. If the passive objects are all called from the same active object then they can use a sequential model.

**Too much concurrency is bad**

There is a model that has even more concurrency than the concurrent stateful model. In this *maximally concurrent* model, each operation executes in its own thread. The execution order is constrained only by data dependencies. This has the greatest possible concurrency. But it is both hard to program in and hard to implement efficiently (see Exercise). It is rarely used except in special cases. We do not present this model in more detail here. By the way, constraint programming is an application of this model (see Chapter 14). All constraints execute together, each constraint concurrently adding information to the store, until no more information can be added. Because of the logical semantics of constraints, this is perfectly acceptable and easy to reason about.

**10.2.2 Implementing coroutines**

We saw how to use coroutines in Chapter 6, but not how to implement them. Now that we have both concurrency and explicit state, we can implement them.
Figure 10.2: Implementing coroutines

Figure 10.2 shows how to define \texttt{Spawn} and \texttt{Resume} for the simplest case, namely two coroutines that do not themselves call other coroutines. The first coroutine calls \{\texttt{Spawn P}\} to create the other. From then on, each calls \{\texttt{Resume}\} to transfer control to the other. See Exercises for the more general definition of \texttt{Spawn} and \texttt{Resume} in the case of many coroutines.

10.2.3 The nondeterministic concurrent model

This section explains the nondeterministic concurrent model, which is a stepping stone from the concurrent declarative model to the message passing model. This section can be skipped on first reading. The nondeterministic concurrent model is the model used by concurrent logic programming \[82\]. It is interesting mainly for historic reasons and for the insight it gives into practical concurrent programming. It adds just one operation, nondeterministic choice, to the concurrent declarative model. This lets it solve the stream communication problem of the concurrent declarative model, as we explain below.

Just like message passing, the model allows to express a dynamic network of active objects. An active object does not have to know from which stream its next input will come. Unlike message passing, we will see that the model is cumbersome to program in and no efficient implementation exists. This is why we recommend to use the concurrent stateful model instead.

We first introduce the nondeterministic concurrent model and show how it solves the stream communication problem of Section 6.5.6. We then show how to implement nondeterministic choice in the concurrent declarative model with exceptions, showing that the latter is at least as expressive as the nondeterministic concurrent model.
10.2 Approaches to programming with concurrency

Limitation of the concurrent declarative model

In Section 6.5.6 we saw a fundamental limitation of the concurrent declarative model: active objects must access input streams in a fixed pattern. Two streams cannot independently feed the same active object. How can we solve this problem? Consider the case of two client objects and a server object. We can try to solve it by putting a new active object, a stream merger, in between the two clients and the server. The stream merger has two input streams and one output stream. All the messages appearing on each of the input streams will be put on the output stream. Figure 10.3 illustrates the solution. This seems to solve our problem: each client sends messages to the stream merger, and the stream merger forwards them to the server. The stream merger is defined as follows:

```plaintext
proc {StreamMerger OutS1 OutS2 InS}
    case OutS1#OutS2
        of (M|NewS1)#OutS2 then NewS in
            InS=M|NewS
            {StreamMerger NewS1 OutS2 NewS}
        [] OutS1#(M|NewS2) then NewS in
            InS=M|NewS
            {StreamMerger OutS1 NewS2 NewS}
        [] nil#OutS2 then
            InS=OutS2
        [] OutS1#nil then
            InS=OutS1
    end
end
```

The stream merger is executed in its own thread. This is a complete definition: it handles the cases when either or both clients terminate. Yet, this solution has a basic difficulty: it does not work! Why not? Think carefully before reading the answer in the footnote.²

Adding nondeterministic choice

But this abortive solution has the germs of a working solution. The problem is that the case statement only waits on one condition at a time. A possible solution is therefore to extend the concurrent declarative model with an operation that allows to wait concurrently on more than one condition. We call this operation nondeterministic choice. One of the simplest ways is to add an operation that waits concurrently on two dataflow variables being bound. We call this operation WaitOr because it generalizes Wait (see Table 10.2). The function call {WaitOr A B} returns when either A or B is bound. It returns either 1 or 2. It can return

²It is because the case statement tests only one pattern at a time, and only goes to the next when the previous ones fail. While it is waiting on stream OutS1, it cannot accept an input from stream OutS2, and vice versa.
Table 10.2: The \texttt{WaitOr} operation: nondeterministic choice

1 when \( A \) is bound and 2 when \( B \) is bound. The concurrent declarative model extended with \texttt{WaitOr} is called the \textit{nondeterministic concurrent} model.

**Concurrent logic programming**

The nondeterministic concurrent model is the basic model of concurrent logic programming, as pioneered by IC-Prolog, Parlog, Concurrent Prolog, and GHC (Guarded Horn Clauses) [18, 19, 17, 80, 81, 96]. It is the principal computation model that was used by the Japanese Fifth Generation Project and many other substantial projects in the 1980’s [82, 33]. In the nondeterministic concurrent model, it is possible to write a stream merger. Its definition looks as follows:

```prolog
proc (StreamMerger OutS1 OutS2 InS)
F={WaitOr OutS1 OutS2}
in
  case F#OutS1#OutS2
  of 1#(M|NewS1)#OutS2 then NewS in
      InS=M|NewS
      {StreamMerger OutS2 NewS1 NewS}
  [] 2#OutS1#(M|NewS2) then NewS in
      InS=M|NewS
      {StreamMerger NewS2 OutS1 NewS}
  [] 1#nil#OutS2 then
      InS=OutS2
  [] 2#OutS1#nil then
      InS=OutS1
  end
end
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
This first calls \texttt{Waitor} to decide \textit{which} stream to listen to. Only after \texttt{Waitor} returns does it enter the \texttt{case} statement. Because of the argument \texttt{F}, alternatives that do not apply are skipped. Note that the recursive calls \texttt{reverse} the two stream arguments. This guarantees fairness between both streams even if the \texttt{Waitor} statement favors one or the other (which is often the case in an implementation, to allow performance tuning). A message appearing on an input stream will eventually appear on the output stream, independent of what happens in the other input stream.

\textbf{Is it practical?}

What can we say about practical programming in this model? Assume that new clients arrive during execution. Each client wants to communicate with the server. This means that a new stream merger must be created for each client! The final result is a tree of stream mergers feeding the server. Is this a practical solution? It has two problems:

- It is inefficient. Each stream merger is an active object that executes in its own thread. The tree of stream mergers is extended at run time each time a new object references the server. Furthermore, the tree is not necessarily balanced. To make it balanced is extra work.

- It lacks expressiveness. It is not possible to reference the server directly. For example, it is not possible to put a server reference in a data structure. The only way we have to reference the server is by referencing one of its streams. We can put this in a data structure, but only one client can use this reference. (Remember that declarative data structures cannot be modified.)

How can we solve these two problems? The first problem could hypothetically be solved by a very smart compiler that recognizes the tree of stream mergers and replaces it by a direct many-to-one communication in the implementation. However, after two decades of research in this area, such a compiler does not exist [95]. The second problem can be partially solved (see Exercises), but the solution is still cumbersome.

We seem to have found an inherent limitation of the nondeterministic concurrent model. Upon closer examination, the problem seems to be that there is no notion of \textit{explicit state} in the model, where explicit state associates a name with a store reference. Both the name and the store reference are immutable; only their association can be changed. There are many equivalent ways to introduce explicit state. One way is by adding the concept of \texttt{cell}, like in Chapter 8. Another way, which is more appropriate for active objects, is by adding the concept of \texttt{port}. A port associates a name to a stream, which makes it a kind of asynchronous communications channel. The port has a send operation, which adds a message to the stream. Ports and cells are equivalent; there are simple implementations of either in terms of the other.
fun \texttt{WaitOr A B}
\[
\begin{align*}
\text{X} \\
\text{thread } \{ \text{Wait A} \} & \text{ try } X=1 \text{ catch } _\_ \text{ then skip } \text{ end } \text{ end} \\
\text{thread } \{ \text{Wait B} \} & \text{ try } X=2 \text{ catch } _\_ \text{ then skip } \text{ end } \text{ end}
\end{align*}
\]
X \end{align*}
end

Figure 10.4: Implementing nondeterministic choice (naive version)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{NewPort S}</td>
<td>Return a new port with stream S.</td>
</tr>
<tr>
<td>{Send P M}</td>
<td>Add M to P’s stream.</td>
</tr>
<tr>
<td>{IsPort P}</td>
<td>Return a boolean saying whether P is a port.</td>
</tr>
</tbody>
</table>

Table 10.3: Port operations

Implementing nondeterministic choice

The \texttt{WaitOr} operation cannot be defined in the concurrent declarative model unless exceptions are used. Figure 10.4 gives a simple definition. This returns 1 or 2, depending on whether A is bound or B is bound. But this definition has the problem that it can leave threads “hanging around” forever, if one variable is never bound. Figure 10.5 gives a better definition that “cleans up” after itself. This version makes sure that both threads terminate after it completes. The third and fourth \texttt{try} statements are needed because \texttt{Thread.terminate} raises an exception if the thread is already terminated.

10.3 Message passing

This section gives the programming techniques for the message passing approach.

10.3.1 Asynchronous communication with ports

Chapter 8 has defined \textit{cells}, which are a form of explicit state. Another form is the \textit{port}. Ports are equivalent to cells in that a port can easily be implemented with a cell and vice versa. Ports are more appropriate primitives for message passing. A port P is an asynchronous communication channel that can be shared among multiple senders. A port is associated with a stream. Sending a message to the port, done by \{\texttt{Send P M}\}, adds the message M to the end of the stream. Table 10.3 gives the three operations defined for ports. The port keeps an internal reference to the stream’s unbound tail. The tail is a read-only variable (see Section 5.9.2). This guarantees that the stream is bound only by the port and
by no-one else. The port’s communication channel has the FIFO property, i.e., repeated sends from the same thread cause the messages to appear in the order of the sends.

Ports are important because they give us the ability to have named active objects [52]. An active object, in its simplest form, pairs an object with a thread. The thread reads a stream of internal and external messages, and invokes the object for each message. The Erlang language is based on this idea (see Section 11.1). Here is a simple example of a port:

```plaintext
declare S P
P={NewPort S}
{Browse S}
```

This displays S, i.e., the stream is empty. Now do one send:

```plaintext
{Send P 1}
```

This causes the display to be updated to \(1|\_\), i.e., the stream has one element.

```plaintext
{Send P 2}
```

Now the display becomes \(1|2|\_\).

With ports it is possible to make networks of active objects that communicate without needing stream mergers. This means that ports and threads together are more expressive than the nondeterministic concurrent model of Section 10.2.3.
Ports and threads have three abilities that the concurrent nondeterministic model does not have:

- They are *sharable*: they can be shared among multiple threads. This is not possible for streams: each stream has exactly one sender. To have multiple senders, it is necessary to define stream mergers, which take two streams as input and send to one stream as output.

- They are *embeddable*: a port reference can be part of a data structure. A port reference can be stored in a file and be sent across another port’s channel. This is impossible for streams. Streams cannot be referenced directly without naming them in some way, which amounts to reinventing ports.

- They are *efficient*: sending a message to an active object implemented with a port takes constant time, independent of the number of senders. With streams, a tree of stream mergers is needed, so the time grows with the number of senders.

### Causal ports

The simplest definition of a port is the causal port, which has a single global FIFO channel. All threads feed into this channel.

### FIFO ports

In a distributed system, a causal port is inefficient to implement because of the single global channel. Much more efficient and natural is the FIFO port, which has a FIFO channel for each thread. Each thread feeds into its own channel. FIFO ports are non-causal.

#### 10.3.2 A simple active object with ports

Here is an example that uses a port to make an active object:

```plaintext
proc {DisplayStream Xs}
  case Xs of X|Xr then {Browse X} {DisplayStream Xr}
  else skip end
end

declare P in % P has global scope
local Xs in % Xs has local scope
  (NewPort Xs P)
  thread {DisplayStream Xs} end
end
```

---

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
10.3 Message passing

fun {NewQueue}
Given GivePort={NewPort Given}
Taken TakePort={NewPort Taken}
in
Given=Taken
queue{put:proc {$ X} {Send GivePort X} end
   get:proc {$ X} {Send TakePort X} end)
end

Figure 10.6: Queue (naive version with ports)

This is a very simple active object: all it does is read the stream \( X \)s continuously and display each element as it arrives. Later on we will see active objects using a class definition, where the arriving elements invoke methods. Sending to \( P \) sends to the active object. Any number of clients can send to the active object concurrently:

\[
\begin{align*}
\text{thread} & \quad \{\text{Send } P \ 1\} \ \{\text{Send } P \ 2\} \ \ldots \ \text{end} \quad \% \ \text{Client 1} \\
\text{thread} & \quad \{\text{Send } P \ a\} \ \{\text{Send } P \ b\} \ \ldots \ \text{end} \quad \% \ \text{Client 2}
\end{align*}
\]

The elements 1, 2, a, b, etc., will appear fairly on the stream \( X \)s. 1 will always appear before 2, but there is no fixed order between 1 and a because they are sent from different threads. The port is fair, i.e., all sent messages will eventually be received. Port fairness is a consequence of thread fairness.

Here is a more compact way to define the active object’s thread:

\[
\begin{align*}
\text{thread} & \quad \text{for } X \ \text{in } Xs \ \text{do} \ \{\text{Browse } X\} \ \text{end} \\
\end{align*}
\]

The \texttt{for} statement is a syntactic short-cut for a call to \texttt{ForAll}. It synchronizes on the stream just like \texttt{DisplayStream}.

10.3.3 A concurrent queue with ports

The program shown in Figure 10.6 defines a thread that acts as a FIFO queue. The function \texttt{NewQueue} returns a new queue \( Q \), which is a record \( \text{queue(put:PutProc get:GetProc)} \) that contains two procedures, one for inserting an element in the queue and one for fetching an element from the queue. The queue is implemented with two ports. The use of dataflow variables makes the queue insensitive to the relative arrival order of \( Q.\text{get} \) and \( Q.\text{put} \) requests. For example, the \( Q.\text{get} \) requests can arrive even when the queue is empty. To insert an element \( X \), call \( \{Q.\text{put } X\} \). To fetch an element in \( Y \), call \( \{Q.\text{get } Y\} \).

Try the following sequence of statements. The program does not work. What is the problem?

\[
\begin{align*}
declare & \quad \text{thread } Q={\text{NewQueue}} \ \text{end}
\end{align*}
\]
The problem is that \texttt{Given=Taken} tries to impose equality between two read-only variables, i.e., bind them. But a read-only variable can only be read and not bound. So the thread defining the queue will suspend in the statement \texttt{Given=Taken}. This problem is remedied by running the equality in its own thread, as shown in Figure 10.7.\footnote{This design of a FIFO queue was given by Denys Duchier.} Now try the above sequence of statements again.

The correct version of the queue works as follows. Doing a series of put operations:

\begin{align*}
\{Q\text{.put }I_0\} \; \{Q\text{.put }I_1\} \ldots \; \{Q\text{.put }I_n\}
\end{align*}

incrementally adds the elements $I_0$, $I_1$, ..., $I_n$, to the stream \texttt{Given}, resulting in:

\begin{align*}
I_0 | I_1 | \ldots | I_n | F_1
\end{align*}

where $F_1$ is a read-only variable. In the same way, doing a series of get operations:

\begin{align*}
\{Q\text{.get }X_0\} \; \{Q\text{.put }X_1\} \ldots \; \{Q\text{.put }X_n\}
\end{align*}

adds the elements $X_0$, $X_1$, ..., $X_n$ to the stream \texttt{Taken}, resulting in:

\begin{align*}
X_0 | X_1 | \ldots | X_n | F_2
\end{align*}

where $F_2$ is another read-only variable. The equality constraint \texttt{Given=Taken} binds the $X_i$'s to $I_i$'s and suspends again for $F_1=F_2$.

This concurrent queue is completely \textit{symmetric} with respect to inserting and retrieving elements. That is, \texttt{Q.put} and \texttt{Q.get} are defined in exactly the same way. Furthermore, because they use dataflow variables to reference queue elements, these operations never suspend. This gives the queue the remarkable property that it can be used to insert and retrieve elements \textit{before} the elements are known. For example, if you do a \texttt{(Q.get X)} when there are no elements in the queue, then an unbound variable is returned in $X$. The next element that is inserted will be bound to $X$. To do a blocking retrieval, i.e., one that waits when there are no elements in the queue, the call to \texttt{Q.get} should be followed by a \texttt{Wait}:

\begin{align*}
\{Q\text{.get }X\} \\
\text{Wait } X
\end{align*}

Similarly, if you do \texttt{(Q.put X)} when $X$ is unbound, i.e., when there is no element to insert, then the unbound variable $X$ will be put in the queue. Binding $X$ will make the element known. To do an insert only when the element is known, the call to \texttt{Q.put} should be preceded by a \texttt{Wait}:

\begin{align*}
\{Q\text{.put }X\} \\
\text{Wait } X
\end{align*}
fun {NewQueue}
  Given GivePort={NewPort Given}
  Taken TakePort={NewPort Taken}
in
  thread Given=Taken end
  queue{put:proc {$ X} {Send GivePort X} end
       get:proc {$ X} {Send TakePort X} end)
end

Figure 10.7: Queue (correct version with ports)

{Wait X}
{Q.put X}

We have captured the essential asymmetry between put and get: it is in the Wait operation. Another way to see this is that put and get reserve places in the queue. The reservation can be done independent of whether the values of the elements are known or not.

10.3.4 Active objects

An active object is a concurrent entity to which one can send asynchronous messages. The active object reads its messages in the order of arrival and performs an action for each message. An active object is defined by a class definition, just like a passive object. The methods of the class define the actions for each message. As part of an action, the active object can send messages to other active objects.

Active objects are created in a similar way to standard, passive objects. Instead of using the operation New, the operation NewActive is used. We pass it a class and an initialization message:

declare
  A={NewActive Class Init}

Implementing active objects

An active object is implemented with a thread, a port, and a passive object. From the outside, the active object is referenced by its port. Active objects send asynchronous messages to each other’s ports. Each active object’s thread reads messages from its port’s stream and invokes its object for each message. An active object executes one method at a time. Therefore locking is not needed. An active object can synchronize with another by putting a dataflow variable in the message and waiting until the variable is bound.

The NewActive function can be defined as follows:

fun {NewActive Class Init}
  S P={NewPort S}
in

thread

for M in S do {Obj M} end

end

proc {$ M} {Send P M} end

end

10.3.5 Networks of active objects

Large applications can be built as communicating active objects. This is a case of component-based programming, as introduced in Section 8.9. In a network of active objects, certain patterns recur frequently. We summarize some of the simplest and most common of these patterns. Many others exist; when they become complicated they are often called distributed algorithms [93, 62]. The systematic study of these patterns, namely designing them, programming with them, and reasoning about them, is called multi-agent systems [105].

Client/server organization

An active object that provides a service to others is called a server. The objects that use this service are called its clients. Client and server activities are roles, not objects in themselves. One active object can play the role of many clients, thus using many services, and many servers, thus providing many services.

Peer-to-peer organization

Active objects that each have their own independent purpose, but that occasionally communicate with each other.

- Example: implementation of e-mail, file sharing, gossip.

Group organization

A set of active objects together implement some part of the application.

- Example: collaborative work application.
- Example: fault tolerance abstraction.

10.3.6 Event manager with active objects

We can use active objects to implement a simple concurrent event manager. The event manager contains a set of event handlers. Each handler is a triple Id#F#S, where Id uniquely identifies the handler, F defines the state update function, and S is the handler’s state. When an event E occurs, each triple Id#F#S is replaced
class EventManager
attr
    handlers
meth init handlers<-nil end
meth event (E)
    handlers<-{
        Map @handlers fun {$ Id#F#{F E S} Id#F#{F E S} end}
    end
meth add (F S ?Id)
    Id={NewName}
    handlers<-Id#F#{F E S}@handlers
end
meth delete (DId ?DS)
    handlers<-{List.partition
        @handlers fun {$ Id#F#{F E S} DId==Id end [_,_,_,DS]} }
end
end

Figure 10.8: Event manager with active objects

by Id#F#{F E S}. That is, each event handler is a finite state machine, which
does a transition from state S to state {F E S} when the event E occurs.

The event manager was originally written in Erlang [5]. The Erlang computation model is based on communicating active objects (see Chapter 11). The translation of the original code to the concurrent stateful model was straightforward.

We define the event manager EM as an active object with four methods:

• (EM init) initializes the event manager.

• (EM event(E)) posts the event E at the event manager.

• (EM add(F S Id)) adds a new handler with update function F and initial
  state S. Returns a unique identifier Id.

• (EM delete(Id S)) removes the handler with identifier Id, if it exists.
  Returns the handler’s state in S.

Figure 10.8 shows how to define the event manager. We show how to use the
event manager to do error logging. First we define a new event manager:

EM={NewActive EventManager init}

We then install a memory-based handler. It logs every event in an internal list:

MemH=fun {$ E Buf} E|Buf end
Id={EM add (MemH nil $)}
We can replace the memory-based handler by a disk-based handler during execution, without losing any of the already-logged events. In the following code, we remove the memory-based handler, open a log file, write the already-logged events to the file, and then define and install the disk-based handler:

```
DiskH = fun {S E S} {S write(vs:E)} S end
File = {New Open.file init(name:`event.log` flags:[write create])}
Buf = {EM delete(Id $)}
for E in {Reverse Buf} do {File write(vs:E)} end
Id2 = {EM add(DiskH File $)}
```

Adding functionality with inheritance

The event manager of Figure 10.8 has the defect that if events occur during a replacement, i.e., between the delete and add operations, then they will not be logged. How can we remedy this defect? A simple solution is to add a new method, `replace` to `EventManager` that does both the delete and add. Because all methods are executed sequentially in the active object, this ensures no event will occur between the delete and add. We have the choice to add the new method directly, to `EventManager`, or indirectly, to a subclass by inheritance. Which possibility is the right solution depends on several factors. First, whether we have access to the source code of `EventManager`. If we do not, then inheritance is the only possibility. If we do have the source code, inheritance may still be the right answer. It depends on how often we need the replace functionality. If we almost always need it in event managers, then we should modify `EventManager` directly and not create a second class. If we rarely need it, then its definition should not encumber `EventManager`, and we can separate it by using inheritance.

Let us use inheritance for this example. Figure 10.9 defines a new class `ReplaceEventManager` that inherits from `EventManager` and adds a new method `replace`. Instances of `ReplaceEventManager` have all methods of `EventManager` as well as the method `replace`. The `insert` field is optional; it can be used to insert an operation to be done between the delete and add operations. We define a new event manager:

```plaintext
class ReplaceEventManager from EventManager
meth replace (NewF NewS OldId NewId
    insert:P<=
      proc {$ _} skip end)
  Buf=EventManager,delete(OldId $)
in
  {P Buf}
  NewId=EventManager,add(NewF NewS $)
end
end
```

Figure 10.9: Adding functionality with inheritance
Now we can do the replacement as follows:

```
DiskH=fun {E S} {S write(vs:E)} S end
File={New Open.file init(name:`event.log` flags:[]write create])}
Id2
{EM replace(DiskH File Id Id2
    insert:
        proc {$ S}
        for E in {Reverse S} do
            {File write(vs:E)} end
    end)
```

Because replace is executed inside the active object, it is serialized with all the other messages to the object. This ensures that no events can arrive between the delete and add methods.

### Batching operations using a mixin class

A second way to remedy the defect is to add a new method that does *batching*, i.e., it does a list of operations. Figure 10.10 defines a new class `Batcher` that has just one method, `batch(L)`. The list `L` can contain messages or zero-argument procedures. When `batch(L)` is called, the messages are passed to `self` and the procedures are executed, in the order they occur in `L`. This is an example of using first-class messages. Since messages are also language entities (they are records), they can be put in a list and passed to `Batcher`. We define a new class that inherits from `EventManager` and brings in the functionality of `Batcher`:

```
class BatchingEventManager from EventManager Batcher end
```

We use multiple inheritance because `Batcher` can be useful to *any* class that needs batching, not just to event managers. Now we can define a new event manager:

```
EM={NewActive BatchingEventManager init}
```

All instances of `BatchingEventManager` have all methods of `EventManager` as well as the method `batch`. The class `Batcher` is an example of a *mixin* class: it
adds functionality to an existing class without needing to know anything about the class. Now we can replace the memory-based handler by a disk-based handler:

```plaintext
DiskH=fun \{ E S \} \{ S write(vs:E) \} S end
File={New Open.file init(name:´event.log´ flags:[write create])}
Buf Id2
{EM batch([delete(Id Buf)
    proc {$_}
        for E in (Reverse Buf) do {File write(vs:E)} end
    end
    add(DiskH File Id2)])}
```

The `batch` method guarantees atomicity in the same way as the `replace` method, i.e., because it executes inside the active object.

What are the differences between the replacement solution and the batching solution? There are two:

- The replacement solution is more efficient because the `replace` method is hard-coded. The `batch` method, on the other hand, adds a layer of interpretation.

- The batching solution is more flexible. Batching can be added to any class using multiple inheritance. No new methods have to be defined. Furthermore, any list of messages and procedures can be batched, even a list that is calculated at run time. However, the batching solution requires that the language support first-class messages.

**Combining computation models**

The event manager is an interesting combination of the declarative, object-oriented, and concurrent stateful computation models:

- Each event handler is defined declaratively by its state update function. Even stronger, each method of the event manager can be seen as a declarative definition. Each method takes the event manager’s internal state from the attribute `handlers`, does a declarative operation on it, and stores the result in `handlers`.

- All methods are executed in sequential fashion, as if they were in a stateful model with no concurrency. All concurrency is managed by the active object abstraction, as implemented by `NewActive`. This abstraction guarantees that all object invocations are serialized. Especially, no locking or other concurrency control is needed.

- New functionality, for example replacement or batching, is added by using object-oriented inheritance. Because the new method executes inside the active object, it is guaranteed to be atomic.
fun {MakeScheduler}
S P={NewPort S}
in
thread
  for msg(Obj Msg) in S do
    try {Obj Msg} catch _ then skip end
  end
end
proc ${ Obj Msg}
  {Send P msg(Obj Msg)}
end
Call={MakeScheduler}

Figure 10.11: Active objects sharing one thread

The result is that event handlers are defined sequentially and declaratively, and yet can be used in a concurrent stateful environment. This is an interesting example of the general principle of separation of concerns. Here, the concerns of state and concurrency are separated from the definition of the event handlers. It is good programming practice to separate concerns as much as possible. Using different computation models together often helps to achieve separation of concerns.

10.3.7 Active objects sharing one thread

It is possible to run a whole set of active objects in just one thread, if the thread serializes all messages in the system. According to David Wood of Symbian Ltd., this solution was used in the operating system of the Psion Series 3 palmtop computers, where memory is at a premium [109]. This takes less memory than the solution of Section 10.3.4, which uses one thread per active object. Execution is efficient since no scheduling has to be done. Objects can access shared data without any particular precautions since they all run in the same thread. The main disadvantage is that synchronization is harder since execution cannot wait inside a method for a calculation done in another object. Otherwise there would be deadlock. This disadvantage means that programs must be written in a particular way, in which messages are like continuations.

Figure 10.11 sets up the mechanism to call all objects inside a single thread. The function MakeScheduler sets up the thread and returns a procedure Call. Doing {Call Obj Msg} sends message Msg to Obj. All invocations to Call execute in a single thread. In this version, exceptions raised during {Obj Msg} are simply ignored.

Figure 10.12 gives a screenshot of a small concurrent application, Ping-Pong, that uses Call. Figure 10.13 gives the full source code of the Ping-Pong appli-
The Window class defines the graphic user interface using the QTk tool of Chapter 12. The window displays a text label and a checkbutton. The text label is updated only when the checkbutton is enabled. The PingPong class defines two methods, ping and pong. Each displays a text and then sends a message to another object. Two objects are created, Ping and Pong, each of which sends a message to the other. Ping asynchronously sends a pong(N) message to Pong and vice versa. The integer argument N is incremented with each message. The ping-ponging of method executions is started with the call \{Call Ping ping(0)\}.

The user interface lets execution progress be monitored. The window displays a term of the form ping(123) or pong(123). When the checkbutton is enabled, then each term is displayed for 50 ms. When the checkbutton is disabled, then the messages are passed internally at a much faster rate, limited only by the speed of the Mozart run-time system. With Mozart 1.2.0 on a 500 MHz Pentium III processor, the rate is about 100000 asynchronous method calls per second.

10.3.8 A thread abstraction with termination detection

Thread creation with \texttt{thread (Stmt) end} can itself create new threads during the execution of (Stmt). We would like to detect when all these new threads have terminated. This does not seem easy: new threads may themselves create new threads, and so forth. A termination detection algorithm like the one of Section 6.7.2 is needed. That algorithm requires explicitly passing variables between threads. We require a solution that is encapsulated, i.e., it does not have this awkwardness. To be precise, we require a procedure \texttt{NewThread} with the following properties:

- The call \{NewThread P SubThread\} creates a new thread that executes the zero-argument procedure P. It also returns a one-argument procedure SubThread.

- During the execution of P, new threads can be created by calling \{SubThread P1\}, where P1 is the thread body. We call these \textit{subthreads}. SubThread can be called recursively.
class Window
    feat hand
    attr see
    meth init
        D H in
        see<-true
        D=td(label(text:nil handle:self.hand)
            checkbutton(  
                text:"See ping-pong" handle:H init:true  
                action:proc {$} see<-{(H get($)) end})  
            )
        )
    end
    meth setlbl(S)
        if @see then
            {Delay 50}
            {self.hand set(text:S)}
        end
    end
end

class PingPong
    feat wind other
    meth init(W P) self.wind=W self.other=P end
    meth ping(N)
        {self.wind setlbl("ping(#N#")})
        {Call self.other pong(N+1)}
    end
    meth pong(N)
        {self.wind setlbl("pong(#N#")})
        {Call self.other ping(N+1)}
    end
end

W={New Window init}
Ping={New PingPong init(W Pong)}
Pong={New PingPong init(W Ping)}
{Call Ping ping(0)}

Figure 10.13: The Ping-Pong application: active objects in one thread

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.


Figure 10.14: A thread abstraction with termination detection

- The NewThread call only returns when the new thread and all subthreads terminate.

That is, there are three ways to create a new thread:

```
thread (Stmt) end

{NewThread proc {} (Stmt) end SubThread}
{SubThread proc {} (Stmt) end}
```

They have identical behavior except for NewThread, which has a different termination behavior. One possible definition of NewThread using the concurrent stateful model is given in Figure 10.14. This definition uses a port. When a subthread is created, then 1 is sent to the port. When a subthread terminates, then −1 is sent. The procedure ZeroExit accumulates a running total of these numbers. If the total ever reaches zero, then all subthreads have terminated and ZeroExit returns.

This definition of NewThread has two restrictions. First, P1 should always call SubThread to create subthreads, never any other operation (such as thread ... end or a SubThread created elsewhere). Second, SubThread should not be called anywhere else in the program. The definition can be extended to relax these restrictions or to check them. We leave this as an exercise for the reader.
### 10.3 Message passing

#### 10.3.9 Eliminating sequential dependencies

Let us examine how to remove useless sequential dependencies between different parts of a program. We take as example the procedure \( \text{Filter} \ L \ F \ L2 \), which takes a list \( L \) and a one-argument boolean function \( F \). It outputs a list \( L2 \) that contains the elements \( X \) of \( L \) for which \( \{F \ X\} \) is true. This is a library function (it is part of the List module) that can be defined declaratively as follows:

```plaintext
fun \{Filter L F\}
  case L
  of nil then nil
  [] X|L2 then
    if \{F X\} then X|\{Filter L2 F\} else \{Filter L2 F\} end
  end
end
```

or equivalently, using the loop syntax:

```plaintext
fun \{Filter L F\}
  for X in L collect:C do
    if \{F X\} then \{C X\} end
  end
end
```

This definition is efficient, but it introduces dependencies: \( \{F X\} \) can be calculated only after it has been calculated for all elements of \( L \) before \( X \). These dependencies are introduced because all calculations are done sequentially in the same thread. But these dependencies are not really necessary. For example, in the call:

\( \text{Filter} \ [A \ 5 \ 1 \ B \ 4 \ 0 \ 6] \ \text{fun} \ \{X \} X>2 \ \text{end Out} \)

it is possible to deduce immediately that 5, 4, and 6 will be in the output, without waiting for A and B to be bound. Later on, if some other thread does \( A=10 \), then 10 could be added to the result immediately.

We can write a new version of Filter that avoids these dependencies. It constructs its output incrementally, as the input information arrives. We use two building blocks:

- **Concurrent composition** (see Section 6.7.3). The procedure \texttt{Barrier} implements concurrent composition: it creates a concurrent task for each list element and waits until all are finished.

- **Asynchronous channels** (ports, see Section 10.3.1). The module \texttt{Port} implements ports. They are used to build the output list incrementally, adding each element independently of the others.

Here is the definition:

```plaintext
proc \{ConcFilter L F L2\}
  C P=\{NewPortClose L2 C\}
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
This first creates a port whose stream is the output list. Then, \texttt{Barrier} is called with a list of procedures, each of which adds \(x\) to the output list if \(F X\) is true. Finally, when all list elements are taken care of, the output list is ended by closing the port.

Is \texttt{ConcFilter} declarative? As it is written, certainly not, since the output list can appear in any order (an observable nondeterminism). This is why we define \texttt{ConcFilter} with procedure syntax and not function syntax. It can be made declarative by hiding this nondeterminism, for example by sorting the output list. There is another way, using the properties of ADTs. If the rest of the program does not depend on the order (e.g., the list is a representation of a set data structure), then \texttt{ConcFilter} can be treated as if it were declarative. This is easy to see: if the list were in fact hidden \textit{inside} a set ADT, then \texttt{ConcFilter} would be deterministic and hence declarative.

### 10.4 Atomic actions

This section gives the programming techniques for the atomic action approach. The basic techniques use just dataflow variables, cells, and threads. These concepts are sufficient for all concurrent programs. We also introduce derived concepts such as reentrant locks or monitors. Using derived concepts can sometimes simplify concurrent stateful programs that use explicit synchronization.

#### 10.4.1 Simulating a slow network

The object invocation \{\texttt{Obj} \(M\)\} calls \texttt{Obj} immediately and returns when the call is finished. We would like to modify this to simulate a slow, asynchronous network, where the object is called asynchronously after a delay that represents the network delay. Here is a simple solution that works for any object:

\[
\text{fun } \{\texttt{SlowNet1} \texttt{Obj} D\} \\
\text{proc } \{\texttt{$ M$}\} \\
\text{thread} \\
\{\texttt{Delay} D\} \\
\{\texttt{Obj} M\} \\
\text{end}
\]
The call \( \text{SlowNet1 Obj D} \) returns a “slow” version of \( \text{Obj} \). When the slow object is invoked, it waits at least \( D \) milliseconds before calling the original object.

**Preserving message order with token passing**

The above solution does not preserve message order. That is, if the slow object is invoked several times from within the same thread, then there is no guarantee that the messages will arrive in the same order as they are sent. Here is a solution that does preserve message order:

\[
\text{fun } \{ \text{SlowNet2 Obj D} \} \\
\text{C=\{NewCell unit\}} \\
in \\
\text{proc } \{ M \} \\
\text{Old New in} \\
\text{\{Exchange C Old New\} } \\
\text{thread} \\
\text{\{Delay D\} } \\
\text{\{Wait Old\} } \\
\text{\{Obj M\} } \\
\text{New=unit} \\
\text{New=} \\
\text{end} \\
\text{end} \\
\text{end}
\]

This solution uses a general technique, called *token passing*, to extract an execution order from one part of a program and impose it on another part. The token passing is implemented by creating a sequence of dataflow variables \( X_0, X_1, X_2, \ldots \), and passing consecutive pairs \( (X_0, X_1), (X_1, X_2), \ldots \) to the operations that should be done in the same order. An operation that receives the pair \( (X_i, X_{i+1}) \) then brackets its computation as follows:

- First, wait until the token arrives, i.e., until \( X_i \) is bound \( \{\text{Wait } X_i\} \).
- Then do the computation.
- Finally, send the token to the next pair, i.e., bind \( X_{i+1} = \text{unit} \).

In the definition of \( \text{SlowNet2} \), each time the slow object is called, a pair of variables \( (\text{Old, New}) \) is created. This is inserted into the sequence by the call \( \text{\{Exchange C Old New\}} \). Because \text{Exchange} is atomic, this works also in a concurrent setting where many threads call the slow object. Each pair shares one variable with the previous pair \( (\text{Old}) \) and one with the next pair \( (\text{New}) \). This effectively puts the object call in an ordered queue. Each call is done in a new thread. It first waits until the previous call has terminated \( \{\text{Wait Old}\} \), then invokes the object \( \{\text{Obj M}\} \), and finally signals that the next call can continue...
The \{Delay D\} call must be done before \{Wait Old\}; otherwise each object call would take at least $D$ milliseconds, which is incorrect.

### 10.4.2 Locks

It often happens that threads wish to access a shared resource, but that the resource can only be used by one thread at a time. To help manage this situation, we introduce a language concept called \textit{lock}, to help control access to the resource. A lock dynamically controls access to part of the program, called a \textit{critical region}. The basic operation of the lock is to ensure exclusive access to the critical region, i.e., that only one thread at a time can be executing inside it. If the shared resource is only accessed from within the critical region, then the lock can be used to control access to the resource.

The shared resource can be either inside the program (e.g., an object) or outside it (e.g., an operating system resource). Locks can help in both cases. If the resource is inside the program, then the programmer can guarantee that it cannot be referenced outside the critical region, using lexical scoping. This kind of guarantee can in general not be given for resources outside of the program. For those resources, locks are an aid to the programmer, but he must follow the discipline of only referencing the resource inside the critical region.

There are many different kinds of locks that provide different kinds of access control. Most of them can be implemented in Oz using language entities we have already seen (i.e., cells, threads, and dataflow variables). However, a particularly useful kind of lock, the thread-reentrant lock, is directly supported by the language. The following operations are provided:

- \{NewLock L\} returns a new lock.
- \{IsLock X\} returns \texttt{true} if and only if \(X\) references a lock.
- \texttt{lock} \(X\) \texttt{then} \(S\) \texttt{end} guards \(S\) with lock \(X\). If no thread is currently executing any statement guarded by lock \(X\), then any thread can enter. If a thread is currently executing a guarded statement, then the same thread can enter again, if it encounters the same lock in a nested execution. A thread blocks if it attempts to enter a guarded statement while there is another thread in a statement guarded by the same lock.

Note that \texttt{lock} \(X\) \texttt{then} \ldots \texttt{end} can be called many times with the same lock \(X\). That is, the critical section does not have to be contiguous. The lock will ensure that at most one thread is inside any of the parts that it guards.

### 10.4.3 Implementing locks

Locks can be defined in the concurrent stateful model by using cells and dataflow variables. We first show the definition of a simple lock, then a simple lock that handles exceptions correctly, and finally a thread-reentrant lock. The built-in...
A simple lock is a procedure \( \{L \; P\} \) that takes a zero-argument procedure \( P \) as argument and executes \( P \) in a critical section. Any thread that attempts to enter the lock while there is still one thread inside will block. The lock is called simple because a thread that is inside a critical section cannot enter any other section protected by the same lock. It has to leave the critical section first. Simple locks can be created by the procedure SimpleLock defined in Figure 10.15. If multiple threads attempt to access the lock body, then only one is given access, and the others are queued. This works as follows. The queue is a sequence of dataflow variables. Each thread gets two of these dataflow variables, \( \text{Old} \) and \( \text{New} \), through the Exchange. The \( \text{Old} \) of one thread is the \( \text{New} \) of the preceding thread. Each thread blocks on its \( \text{Old} \) variable (until the preceding thread has bound it) and

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
proc \{NewLock \?Lock\}
    Token=(NewCell unit $)
    Current=(NewCell unit $)
in
    proc \{Lock P\}
        ThisThread=(Thread.this $)
        LockedThread
        in
        \{Access Current LockedThread\}
        if ThisThread==LockedThread then
            \{P\}
        else Old New in
            \{Exchange Token Old New\}
            \{Wait Old\}
            \{Exchange Current _ ThisThread\}
            try
                \{P\}
            finally
                \{Exchange Current _ unit\}
                New=unit
            end
        end
    end
end

Figure 10.17: Lock (reentrant version with exception handling)

binds its \texttt{New} variable after it has executed the lock body. Each thread desiring the lock therefore references two dataflow variables: \texttt{Old} to wait for the lock and \texttt{New} to pass the lock to the next thread.

But what happens if the lock body \{\texttt{P}\} raises an exception? The lock of Figure 10.15 does not work since \texttt{New} will never be bound. We can fix this problem with a \texttt{try} statement. Figure 10.16 gives a version of the simple lock that handles exceptions. The \texttt{try} (S1) \texttt{finally} (S2) \texttt{end} is syntactic sugar that ensures (S2) is executed in both the normal and exceptional cases, i.e., an exception will not prevent the lock from being released.

A \textit{thread-reentrant} lock extends the simple lock to allow the same thread to enter other critical sections protected by the same lock. It is even possible to nest critical sections protected by the same lock. Other threads trying to acquire the lock will queue until \texttt{P} is completed. When the lock is released, it is granted to the thread standing first in line. Figure 10.17 shows how to define thread-reentrant locks. This assumes that each thread has a unique identifier \texttt{T} that is different from the literal \texttt{unit} and that is obtained by calling the procedure \{Thread.this \texttt{T}\).

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
10.4 Atomic actions

fun \{NewQueue\}
X in
q(0 X X)
end

fun \{Insert q(N S E) X\}
E1 in
E=X|E1 q(N+1 S E1)
end

fun \{Delete q(N S E) X\}
S1 in
S=X|S1 q(N-1 S1 E)
end

Figure 10.18: Queue (declarative version)

10.4.4 Using locks to build concurrent stateful ADTs

Let us see how to extend declarative ADTs and stateful ADTs to be used in a concurrent setting. We will show the different techniques by means of an example, a queue. This is not a limitation since these techniques work for any ADT. We start from a declarative implementation and show how to convert this to a stateful implementation that can be used in a concurrent setting:

- Figure 10.18 is essentially the declarative queue of Section 5.7.3. (For brevity we leave out the function \texttt{EmptyQueue}.) Delete operations never block: if the queue is empty when an element is deleted, then a dataflow variable is returned which will be bound to the next inserted element. The size field \(N\) is positive if there are more inserts than deletes and negative otherwise. All the functions of this version have the form \(Q_{out}=\{QueueOp \ Q_{in} \ldots\}\), taking an input queue \(Q_{in}\) and returning an output queue \(Q_{out}\). Therefore this queue can be used in a concurrent setting without changing anything.

- Figure 10.19 shows the same queue, but in a stateful version that encapsulates the queue’s data. This version cannot be used in a concurrent setting without some changes. The problem is that encapsulating the state requires to read the state (\texttt{Access}), do the operation, and then to write the new state (\texttt{Assign}). If two threads each do an insert, then both reads may be done before both writes, which is incorrect. A correct concurrent version requires the read-operation-write sequence to be atomic.

- Figure 10.20 shows a concurrent version of the queue, using a lock to ensure atomicity of the read-operation-write sequence.

- Figure 10.21 shows the same version, written with object-oriented syntax.
The cell is replaced by the attribute queue and the lock is implicitly defined by the locking property.

- Figure 10.22 shows another concurrent version, using an exchange to ensure atomicity. This takes advantage of the single-assignment property of dataflow variables. An important detail: the arithmetic operations $N-1$ and $N+1$ must be done after the exchange (why?).

We discuss the advantages and disadvantages of these solutions:

- The declarative version of Figure 10.18 is the simplest, but it has the disadvantage that the queues must be explicitly threaded in the program. The stateful version is more flexible because the queue threading is implicit and dynamic.

- Both concurrent versions of Figure 10.20 and 10.22 are reasonable. Figure 10.20’s use of a lock is more general, since a lock can be used to make atomic any set of operations. Furthermore, the object-oriented syntax of Figure 10.21 makes this version more convenient to write. Figure 10.22’s version with exchange is compact but less general; it is only possible for operations that manipulate a single data sequence.

### 10.4.5 A concurrent stack and queue

This section defines two useful data types, a concurrent stack and queue, that we will need for implementing tuple spaces and monitors. Both the concurrent stack and queue use cells to store their state and rely on Exchange for concurrent invocations. For simplicity, both data types are concrete. The stack provides
fun {NewQueue ?Insert ?Delete}
X C={NewCell q(0 X X)}
L={NewLock}
in
proc {Insert X}
N S E1 in
  lock L then
    q(N S X|E1)={Access C}
    {Assign C q(N+1 S E1)}
  end
end
end

proc {Delete X}
N S1 E in
  lock L then
    q(N X|S1 E)={Access C}
    {Assign C q(N-1 S1 E)}
  end
end
end

Figure 10.20: Queue (concurrent stateful version with lock)

push and pop operations. (Both push and pop are nonblocking, although the
nonblocking pop is not needed by the monitor.) Using exchange, the stack is
very compact:

fun {NewStack} {NewCell nil} end
proc {Push X Stack} S in {Exchange Stack S X|S} end
fun {Pop Stack} X S in {Exchange Stack X|S S} X end

The queue is an extension of the declarative queue defined in Figure 10.18. It
adds three operations, DeleteNonBlock, GetAll, and Size. The first addition
is a nonblocking delete operation:

fun {DeleteNonBlock q(N S E) X}
  if N>0 then H S1 in
    X=[H] S=H|S1 q(N-1 S1 E)
  else
    X=nil q(N S E)
  end
end

This binds X to a one-element list if the delete was successful and to nil otherwise.
The second addition is an operation to return all elements in the queue, thus
emptying the queue:

fun {GetAll q(_ S E) L} X in
```pascal
class Queue
  attr queue
  prop locking

  meth init
    queue<-q(0 X X)
  end

  meth insert(X)
    lock N S E1 in
    q(N S X|E1)=@queue
    queue<-q(N+1 S E1)
  end

  meth delete(X)
    lock N S1 E in
    q(N X|S1 E)=@queue
    queue<-q(N-1 S1 E)
  end
end

Figure 10.21: Queue (concurrent object-oriented version with lock)

L=S E=nil
q(0 X X)
end
```

The third addition is a function that returns the size of the queue, i.e., how many elements it contains:

```pascal
fun {Size q(N _ _)} N end
```

### 10.4.6 Concurrent object-oriented programming

Locks are separate entities from objects. This lets one lock control access to many objects. It is often useful to have one lock per object. The object system has syntactic support for this, through the `locking` property.

### 10.4.7 Tuple spaces (“Linda”)

Tuple spaces are a popular abstraction for concurrent programming. The first tuple space abstraction, called *Linda*, was introduced by David Gelernter in the late 1980’s [13, 14]. This abstraction is both one of the first models of concurrent programming and a practical abstraction for concurrent programs. A tuple
fun {NewQueue ?Insert ?Delete}
    X C={NewCell q(0 X X)}
in
    proc {Insert X}
        N S E1 N1 in
            {Exchange C q(N S X|E1) q(N1 S E1)}
            N1=N+1
        end
    end

    proc {Delete X}
        N S1 E N1 in
            {Exchange C q(N X|S1 E) q(N1 S1 E)}
            N1=N-1
    end
end

Figure 10.22: Queue (concurrent stateful version with exchange)

space abstraction is sometimes called a coordination model and a programming language that contains a tuple space abstraction is sometimes called a coordination language. In its basic form, the abstraction is simple to define. It consists of a multiset $TS$ of tuples with three basic operations:

- $\{TS \ \text{write}(T)\}$ adds the tuple $T$ to the tuple space.

- $\{TS \ \text{read}(T)\}$ waits until the tuple space contains at least one tuple that has the same label as $T$. It then removes one such tuple and binds it to $T$.

- $\{TS \ \text{readnonblock}(T \ B)\}$ does not wait, but immediately returns. It returns with $B=\text{false}$ if the tuple space contains no tuple with the same label as $T$. Otherwise, it returns with $B=\text{true}$, removes one tuple with the right label and binds it to $T$.

This abstraction has two important properties. The first property is that it provides a content-addressable memory: tuples are identified only by their labels. The second property is that the readers are decoupled from the writers. The abstraction does no communication between readers and writers other than that defined above.

Tuple spaces can be implemented using dictionaries together with the concurrent queue of Section 10.4.5. Figure 10.23 shows a simple implementation in object-oriented style. This implementation is completely dynamic; at any moment it can read and write tuples with any labels. The implementation uses a dictionary to store the tuples. Each entry is indexed by the tuple’s label and stores a queue of tuples with that label. The capitalized methods EnsurePresent and Cleanup are private to the TupleSpace class and invisible to users of tuple space objects (see Section 9.3.3). The implementation does correct memory
```python
class TupleSpace
    prop locking
    attr tupledict

    meth init tupledict<-{NewDictionary} end

    meth EnsurePresent(L)
        if {Not {Dictionary.member @tupledict L}} then
            @tupledict.L:={NewQueue} end
    end

    meth Cleanup(Q L)
        @tupledict.L:=Q
        if {Size Q}==0 then
            {Dictionary.remove @tupledict L} end
    end

    meth write(Tuple)
        lock L={Label Tuple} in
        {self EnsurePresent(L)}
        @tupledict.L:={Insert @tupledict.L Tuple}
    end

    meth read(Tuple) X in
        lock Q L={Label Tuple} in
        {self EnsurePresent(L)}
        Q={Delete @tupledict.L X}
        {self Cleanup(Q L)}
        {Wait X} X=Tuple
    end

    meth readnonblock(Tuple ?B)
        lock U Q L={Label Tuple} in
        {self EnsurePresent(L)}
        Q={DeleteNonBlock @tupledict.L U}
        case U of [X] then
            {self Cleanup(Q L)} B=true X=Tuple
        else B=false end
    end
end
```

Figure 10.23: Tuple space (object-oriented version)
Example execution

We first create a new tuple space:

\[ \text{TS} = \{ \text{New TupleSpace init} \} \]

In \( \text{TS} \) we can read and write any tuples in any order. The final result is always the same: the reads see the writes in the order they are written. Doing \( \{ \text{TS write(foo(1 2 3))} \} \) adds a tuple with label \( \text{foo} \). The following code waits until a tuple with label \( \text{foo} \) exists, and when it does, it removes and displays it:

\[ \text{thread T in } \{ \text{TS read(T=foo(_ _ _))} \} \{ \text{Browse T} \} \text{ end} \]

The following code immediately checks if a tuple with label \( \text{foo} \) exists:

\[ \text{local T B in } \{ \text{TS readnonblock(T=foo(_ _ _) B)} \} \{ \text{Browse T#B} \} \text{ end} \]

This does not block, so it does not need to be put in its own thread.

10.4.8 Monitors

One of the traditional ways to do concurrent programming is by means of monitors. A monitor is a lock extended with a gating mechanism for threads to control more precisely how they exit and enter the critical section. The monitor adds a wait and a notify operation. Notify is sometimes called signal. Wait and notify are only possible from inside the lock. A thread can explicitly do a wait; this suspends the thread until another thread does a notify.

It is interesting to compare this approach with the dataflow approach of the concurrent stateful model. We will first implement monitors and then give some examples programmed both with the monitor approach and with the dataflow approach.

Definition

There are several slightly different varieties of monitors. We explain the Java variety. The following definition is taken from [60]. In Java, a monitor is always tied to an object—it is an object with an internal lock. The object methods can be invoked within the lock (i.e., inside a critical section) by annotating them as synchronized. There are three operations to manage the lock: wait, notify, and notifyAll. These operations can only be called by threads that hold the lock. They have the following meaning:

- The wait operation does the following:
  - The current thread is suspended.
  - The thread is placed in an internal wait queue associated with the object.
- The lock for the object is released.

- The **notify** operation does the following:
  - If one exists, an arbitrary thread \( T \) is removed from the object’s internal wait queue.
  - \( T \) proceeds to get the lock, just as any other thread. This means that \( T \) will always block for a short time, until the notifying thread releases the lock.
  - \( T \) resumes execution at the point it was suspended.

- The **notifyAll** operation is similar to **notify** except that it does the above steps for *all* threads in the internal wait queue. The wait queue is then emptied.

For the examples that follow, we suppose that a procedure `NewMonitor` exists that creates monitor locks in the following way:

- `{NewMonitor LockM WaitM NotifyM NotifyAllM}` returns a new lock procedure `LockM`, and its corresponding monitor operations, namely `WaitM`, `NotifyM`, and `NotifyAllM`, each of which is a zero-argument procedure.

In the same way as for locks, we assume that the monitor lock is thread-reentrant and handles exceptions correctly. Section 10.4.9 explains how the monitor lock is implemented.

**Comparison with implicit synchronization**

We now give some examples of concurrent programs implemented with the monitor approach and with the dataflow approach.

### 10.4.9 Implementing monitors

We give two implementations of monitors. Unlike the Java approach, we do not tie monitors to objects, but we consider them as independent entities. Our implementations build monitors with cells and dataflow variables. The first implementation is naive; it is not thread-reentrant and it does not take exceptions into account. The second implementation is complete; it works with exceptions and is thread-reentrant. Both implementations use the concurrent stack and queue defined in Section 10.4.5.

Figure 10.24 gives the first, naive monitor implementation. The procedure `NewMonitor` has four outputs, namely the `LockM`, `WaitM`, `NotifyM`, and `NotifyAllM` procedures. Internally, it uses the `GetLock` and `ReleaseLock` procedures to implement the others. For correct operation, each thread must call the `GetLock` and `ReleaseLock` operations in *pairs*. They can be called in succession:
10.4 Atomic actions

\begin{verbatim}
  Stack={NewStack}
  Q={NewCell {NewQueue}}
  Token={NewCell unit}

proc {GetLock}
  Old New in
    {Exchange Token Old New}
    {Push New Stack}
    {Wait Old}
  end
proc {ReleaseLock}
  unit={Pop Stack}
  end
in
proc {LockM P}
  {GetLock}
  {P}
  {ReleaseLock}
end

proc {WaitM}
  X in
    {Assign Q {Insert {Access Q} X}}
    {ReleaseLock}
    {Wait X}
    {GetLock}
end

proc {NotifyM}
  X in
    {Assign Q {DeleteNonBlock {Access Q} X}}
    case X of [U] then
      U=unit
    else skip end
end

proc {NotifyAllM}
  L in
    {Assign Q {GetAll {Access Q} L}}
    {ForAll L proc {$ X} X=unit end}
end
end
\end{verbatim}

Figure 10.24: Monitor (naive version)
GetLock - ReleaseLock - GetLock - ReleaseLock

or nested:

GetLock - GetLock - ReleaseLock - ReleaseLock

If there is some thread that is between a GetLock - ReleaseLock pair, then we say the lock is active. The complete monitor implementation of Figure 10.25 ensures these ordering constraints. It is thread-reentrant and handles exceptions. Finally, it verifies that WaitM, NotifyM, and NotifyAllM are executed inside an active lock and raises an exception if this is not true. The main difference between this version and the naive version is the implementation of LockM, which has two extensions:

- LockM checks whether the lock is already held by the current thread, before attempting to get the lock.
- LockM executes P in the context of a try - finally operation, to ensure the lock is released if an exception is raised.

This implementation will work correctly, and even fairly efficiently. It can of course be optimized in many ways, which is important if monitor operations are frequent (see Exercises).

When writing concurrent programs in the concurrent stateful model, it is usually simpler to use the dataflow approach rather than monitors. The implementation therefore does no particular optimizations to improve monitor performance.

10.4.10 Bounded buffers

In Chapter 6, we showed how to implement bounded buffers in a declarative way, in the context of stream communication. In this section we show how to implement them with a monitor and we compare this with the declarative implementations.

10.5 Threaded state and named state

Explain the difference between ‘‘threaded’’ state and ‘‘named’’ state.

Threaded state exists only in the mind of the programmer, i.e., it is just a way to view the program. That is, a particular sequence of variable locations in the program is considered as representing the different values of a stateful entity.

A typical example is a recursive procedure with an accumulating parameter.

The value of this parameter at successive calls
10.5 Threaded state and named state

```plaintext
  Stack={NewStack}
  Q={NewQ}
  Token={NewCell unit}
  Current={NewCell unit}

proc {GetLock} ... /* As before */ end
proc {ReleaseLock} ... /* As before */ end

in

% Extended to handle exceptions and be thread-reentrant
proc {LockM P}
  if {Thread.this}=={Access Current} then
    {P}
  else
    {GetLock}
    {Assign Current {Thread.this}}
    try
      {P}
    finally
      {Assign Current unit}
      {ReleaseLock}
    end
  end
end

proc {WaitM}
  if {Thread.this}!='{Access Current} then
    raise error end
  else ... /* As before */ end
end

proc {NotifyM}
  if {Thread.this}!='{Access Current} then
    raise error end
  else ... /* As before */ end
end

proc {NotifyAllM}
  if {Thread.this}!='{Access Current} then
    raise error end
  else ... /* As before */ end
end
end
```

Figure 10.25: Monitor (full version)

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
gives the content of the stateful entity.

Named state is supported by the computation model: it exists as an entity in the language. There exists a ‘‘name’’ attached to a mutable content. For example, a pointer in C++ is a constant, namely a memory address, with a mutable content. The pointer is the ‘‘name’’.

We explain why threaded state, which exists in the declarative model, is insufficient by itself for real-world concurrent programming. Named state is needed, resulting in the stateful model [52]. The canonical way to encode state in the declarative model is by using streams. An active object is a recursive predicate that reads an internal stream. The object’s current state is this stream’s most-recently-calculated element. A reference to an active object is a stream that is read by that object. This reference can only be used by one sender object, which sends messages by binding the stream’s tail. Two sender objects sending messages to a third object are coded as two streams feeding a stream merger, whose output stream then feeds the third object. Whenever a new reference is created, a new stream merger has to be created. The system as a whole is therefore more complex:

- The communications graph of the active objects is encoded as a network of streams and stream mergers. In this network, each object has a tree of stream mergers feeding into it. The trees are created incrementally during execution, as object references are passed around the system.

- To regain efficiency, the compiler and run-time system must be smart enough to discover that this network is equivalent to a much simpler structure in which senders send directly to receivers. This “decompilation” algorithm is so complex that to our knowledge it has never been implemented. Pure concurrent logic programming, which is a subset of the declarative model, has been the subject of much implementation effort, but this decompilation algorithm has never been implemented in it.

On the other hand, adding state directly to the computation model makes the system simpler and more uniform. In that case, programmer-visible state (e.g., active objects) is mapped directly to computation model state (e.g., threads and cells), which is compiled directly into machine state. Both the compiler and the run-time system are simple. The counterargument that the stateful model is no longer “pure” is true but of no practical use, since the stateful model allows simpler reasoning than the “pure” declarative model.
10.6 Memory management

As explained in Section 5.13.2, garbage collection is a technique for automatic memory management that recovers memory for all entities inside the computation model that no longer take part in the computation. This is not good enough for entities outside the computation model. Such entities exist because there is a world outside of the computation model, which interacts with it. How can we do automatic memory management for them? There are two cases:

- From inside the computation model, there is a reference to an entity outside it. We call such a reference a resource pointer. Here are some examples:
  - A file descriptor, which points to a data structure held by the operating system. When the file descriptor is no longer referenced, we would like to close the file.
  - A handle to access an external database. When the handle is no longer referenced, we would like to close the connection to the database.
  - A pointer to a block of memory allocated through the Mozart C++ interface. When the memory is no longer referenced, we would like it to be freed.

- From the external world, there is a reference to inside the computation model. We call such a reference a ticket. Tickets are used in distributed programming as a means to connect processes together (see Chapter 13).

In the second case, there is no way in general to recover memory. The external world is so big that the computation model cannot know whether the outside reference still exists or not. The solution we have adopted is to add the language entity to the root set, i.e., to make it permanently unrecoverable. This is acceptable because usually very few tickets are used: they are needed only as entry points for connecting to a running application. As such, only one ticket is needed per application, and the ticket remains valid as long as the application runs.

In the first case, there is a simple solution based on parameterizing the garbage collector. This solution, called finalization, gives the ability to perform a user-defined action when a language entity has become unreachable. This is implemented by the System module Finalize. We first explain how the module works. We then give some examples of how it is used.

10.6.1 Finalization

Finalization is supported by the Finalize module. The design of this module is inspired by the guardian concept of [29]. Finalize has the following two operations:

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
• {Finalize.register \(X\) \(P\)} registers a reference \(X\) and a procedure \(P\). When \(X\) becomes otherwise unreachable (otherwise than through finalization), \({P \ X}\) is eventually executed in its own thread. During this execution, \(X\) is reachable again until its reference is no longer accessible.

• {Finalize.everyGC \(P\)} registers a procedure \(P\) to be invoked eventually after every garbage collection.

In both of these operations, you cannot rely on how soon after the garbage collection the procedure \(P\) will be invoked. It is in principle possible that the call may only be scheduled several garbage collections late if the system has very many live threads and generates garbage at a high rate.

There is no limitation on what the procedure \(P\) is allowed to do. This is because \(P\) is not executed during garbage collection, when the system’s internal data structures can be temporarily inconsistent, but is scheduled for execution after garbage collection. \(P\) can even reference \(X\) and itself call Finalize.

An interesting example is the everyGC operation itself, which is defined in terms of register:

\[
\begin{align*}
\text{proc } \&\text{EveryGC } P \\
\text{proc } \&\text{DO } \_ \{P\} \{\text{Finalize.register DO DO}\} \end{align*}
\]

This creates a procedure \(DO\) and registers it using itself as its own handler. When EveryGC exits, the reference to \(DO\) is lost. This means that \(DO\) will be invoked after the next garbage collection. When invoked, it calls \(P\) and registers itself again.

### 10.6.2 Laziness and external resources

To make lazy evaluation practical for external resources, like files, we need to use finalization to release the external resources when they are no longer needed. For example, in Section 6.6.3 we defined a function LazyRead that reads a file lazily. This function closes the file after it is completely read. But this is not good enough: if only part of the file is needed, then the file should also be closed.

We can implement this with finalization. We extend the definition of LazyRead to close the file when it becomes inaccessible:

\[
\begin{align*}
\text{fun } \&\text{LazyRead } FN \\
\text{InFile}=&\{\text{New Open.file init(name:FN)}\} \\
\{\text{Finalize.register InFile proc }\&\text{F close}\} \end{align*}
\]

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
10.7 Case studies

Here come some interesting case studies.

10.7.1 A bouncing ball application

- Change this to use QTk.

Chapter 2 introduces a bouncing ball application. This application animates a set of balls bouncing in a window, as if they were subject to the law of gravity. Programming this application requires both active objects and a graphical user interface. This section gives the program and explains how it works. Figure 10.26 shows the architecture of the application. It gives all the active entities and the messages they send to each other. Here is the program that implements this architecture:

```plaintext
declare
CanvasWidth = 400.0
CanvasHeight = 400.0
BallDiameter = 50.0
XDelta = 2.0

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
```
This program starts by declaring important constants. Then it introduces Ball and Manager classes, which we will use to define the active objects. We will fill in the blanks one by one, explaining what is going on. This program is all declarations, except for the {NewBouncer} at the very end, which is a procedure call. Calling this opens an empty bouncer window. The procedure NewBouncer itself has zero arguments and is declared just before the call, by proc {NewBouncer} ... end. The list BallColors is an infinite list, used to cycle through the colors:

```
BallColors = red | blue | yellow | green |
            | plum  | cyan | tan | bisque | BallColors
```

The list could have been finite, for example:

```
EightColors = red | blue | yellow | green |
            | plum | cyan | tan | bisque | nil
```
This is a list of eight elements. But BallColors is not like this! It ends with a reference to itself. This means that it is a cyclic data structure. It represents an infinite list, which repeat the same eight elements over and over again. The manager uses this list to choose colors for new balls. This means that after creating eight balls, the colors will repeat.

The class Ball is local to the function NewBall. The function uses the Ball class to create a new active ball object. The NewBall function is visible to the whole program.

Making a new bouncer

The NewBouncer procedure makes a new bouncer, which is a running instance of the application. Here is the procedure's definition:

```haskell
proc {NewBouncer}
  Win = {New Window 
    init(title:`Oz Bouncer`
      delete:proc {$} {Man killAll} {Win close} end)}
  Can = {New Canvas init(parent:Win bd:3 relief:sunken
    width:CanvasWidth height:CanvasHeight)}
  Man = {NewActive Manager init(Can)}
  in
    {Can bind(action:Man#newBall
      event:`<1>`
      args:[float(x) float(y)])}
    {Can bind(action:Man#killBall
      event:`<3>`)}
    {Win pack(Can)}
  end
end
```

This creates a new window Win, a new canvas Can, which is the area on which the balls are displayed, and a new manager Man, which is an active object. With the event of pressing the left mouse button in the canvas, it associates the action of sending the message newBall to Man. Likewise, the message killBall is associated with the right mouse button. Each time one of these buttons is pressed, the associated message is sent.

Finally, the canvas is packed into the window, and the call to NewBouncer terminates. At this point, we have two active entities, the manager Man and the graphic subsystem, and no balls.

We will highlight some of the interesting details of the definition, and leave the rest for later. Consider the following fragment:

```haskell
Win = {New Window 
    init(title:`Oz Bouncer`
      delete:proc {$} {Man killAll} {Win close} end)}
```

From the Window class, which is part of the graphic library, this creates a new window that has a title and a delete procedure. The notation init(title:
... delete: ...) is a record that represents a method call, used to initialize the window. The argument to delete is an anonymous procedure `proc {$} ... end`, where the "$" notation replaces the procedure’s name. When the window is closed, this procedure is called. This sends the message `killAll` to `Man` and `close` to `Win`, thus cleanly terminating the application. (What happens is that `Man` will kill all the balls.)

### Passive and active objects

A passive object is an abstract data type: it is an encapsulated state together with a set of operations, which are called methods. When a passive object is called, it executes in the caller’s thread. We say that it executes synchronously, i.e., there is only one thread of control.

An active object is also an abstract data type, like a passive object. However, it also has its own internal thread. When an active object is called, then the caller continues immediately. The active object will eventually receive the message and execute it in its thread. This means that the active object and the caller will execute concurrently.

The `NewBouncer` procedure creates a new active object:

```plaintext
Man = {NewActive Manager init(Can)}
```

This makes a new manager for a given canvas `Can`. Here is the class `Manager`:

```plaintext
class Manager
  attr
    balls:nil colors:BallColors canvas
  
  meth init(Can)
    canvas <- Can
  end

  meth newBall(X Y)
    C|Cr = @colors
    BallClose={NewBall X Y @canvas C}
    in
      balls <- BallClose|@balls
      colors <- Cr
  end

  meth killBall
    case @balls of nil then skip
    [] BallClose|Br then {BallClose} balls<-Br
  end

  meth killAll
    {ForAll @balls proc {$ BC} {BC} end}
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
This class has three attributes, \texttt{balls}, \texttt{colors}, and \texttt{canvas}, and four methods, \texttt{init}, \texttt{newBall}, \texttt{killBall}, and \texttt{killAll}. The attributes \texttt{balls} and \texttt{colors} are initialized when the class is defined. On the other hand, \texttt{canvas} is initialized separately for each new object. The attribute \texttt{balls} maintains a list of all living balls, to be precise, of their close procedures. The attribute \texttt{colors} gives a list of colors to be used subsequently for new balls. Notice that it is initialized to be the cyclic list \texttt{BallColors}.

Attributes can be read and assigned multiple times, unlike single-assignment variables. We say they are \textit{stateful}. In fact, the \textit{only} way to make stateful variables is to organize a group of them inside a class. They are local to the class, i.e., they cannot read or assigned from outside.

The \texttt{newBall} method creates a new ball, the \texttt{killBall} method destroys the most recently created ball, and the \texttt{killAll} method destroys all balls. The \texttt{newBall} method introduces several new concepts:

\begin{verbatim}
meth newBall(X Y)
    C|Cr = @colors
    BallClose={NewBall X Y @canvas C}
    in
        balls <- BallClose|@balls
        colors <- Cr
end
\end{verbatim}

This definition declares three single-assignment variables \texttt{C}, \texttt{Cr}, and \texttt{BallClose}, and immediately assigns them. Notice how \texttt{C} and \texttt{Cr} get their values through \textit{pattern matching}. That is, since \texttt{@colors} is a list, \texttt{C} will be bound to its first element and \texttt{Cr} to the rest of the list.

The \texttt{NewBall} function creates a new ball as a special active object, one that is animated. The function's arguments give the ball's initial position, its canvas, and its color. The function returns a procedure with a very special role. Executing it will destroy the ball. This is an example of a \textit{capability}. A capability is an unforgeable reference that gives its owner the right to perform a particular operation.

\textbf{The ball active object}

The ball object has two parts: a class, \texttt{Ball}, that encapsulates the ball's state, and a function, \texttt{NewBall}, that defines a new ball together with its active behavior. We explain each of these in turn.

The \texttt{Ball} class encapsulates a ball's state, how to calculate the next state, and how to draw the ball's position:

\begin{verbatim}
class Ball
    from CanvasObject
\end{verbatim}
attr
  x y
  xd: XDelta yd: YDelta
  mouth: Mouth d: ~MouthDelta

meth init(X Y Can Color)
  CanvasObject, init (parent: Can)
  {Can create (arc X Y X+BallDiameter Y+BallDiameter
              fill: Color start: Mouth extent: 360-2*Mouth id: self)}
  x <- X y <- Y
end

meth step
  %%% Calculate new ball state
  % increment @x and @xd
  x <- @x + @xd
  if @x <= 0.0 then
    x <- 0.0
    xd <- ~@xd
  elseif @x >= EffWidth then
    x <- EffWidth
    xd <- ~@xd
  end
  % increment @y and @yd
  y <- @y - @yd
  yd <- @yd - Gravity
  if @y >= EffHeight then
    y <- EffHeight
    yd <- Fraction * ~@yd
  end
  % set the new mouth
  mouth <- @mouth + @d
  if @mouth >= Mouth then
    mouth <- Mouth d <- ~@d
  elseif @mouth <= MouthDelta then
    mouth <- MouthDelta d <- ~@d
  end
  %%% Draw new ball
  {self place (@x @y @x+BallDiameter @y+BallDiameter)}
  {self configure (start: @mouth extent: 360-2*@mouth)}
end

meth close
  CanvasObject, close
end

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
This class inherits from the CanvasObject class. It extends the behavior of a canvas object, i.e., a graphical object displayed in a canvas. Its new behavior is very simple. It keeps the state describing the position (attributes $x$ and $y$) and velocity ($x_d$ and $y_d$) of a PacMan figure, how far open its mouth is ($\text{mouth}$), and how fast the mouth opens or closes ($d$). It knows how to calculate the next state and how to draw the figure.

The NewBall function defines a new ball as an active entity. A ball is very like an active object, except that it does something even when no messages are sent to it.

```
fun {NewBall X Y Can Color}
  B={New Ball init(X Y Can Color)}
  S P={NewPort S}
  proc {Loop S}
    case S of
      step|S2 then
        {B step}
        {Delay RepeatTime}
        {Send P step}
        {Loop S2}
      [] close|_ then
        {B close}
    end
  end
  proc {BallClose} {Send P close} end
in
  thread {Loop S} end
  {Send P step}
  BallClose
end
```

This function first defines and binds four variables, $B$, which is a passive Ball object, $P$, which is a port, $S$, which is the port’s stream, and $\text{Loop}$, which is a recursive procedure used as a loop.

The procedure $\text{Loop}$ reads the stream $S$. Whenever a new element appears on the stream, then a case statement chooses what to do. If the message close appears, then close is sent to ball $B$. If the message step appears, then step is sent to $B$, the thread waits for a given time (“delay”), and a new step is sent to the port. Then, a recursive call to $\text{Loop}$ ensures that the next stream element will be read.

The body of $\text{NewBall}$ starts a thread that runs $\{\text{Loop S}\}$. To start the animation, the message step is sent to the port. This sets off an infinite sequence of steps: each step is read, the ball is stepped, and a new step is done. Finally, an anonymous procedure

```
proc {} {Send P close} end
```

is returned that sends a close message to stop the ball and close its object.
We have seen this style of concurrent programming in Chapter 10. It is interesting because it has no locking! There are threads that communicate through shared variables (like \( S \)) and that synchronize through dataflow. Threads are very inexpensive, so the programmer can create them whenever they are needed, like here to run \( \text{Loop} \).

The port \( P \) is hidden inside the function. There’s no way to get access to it. This makes it easy to be sure that the function works like it should. No interference is possible from outside.

## Making it standalone

We can make the bouncing ball into a standalone application. A standalone application can be run as is or as an applet. The interactive interface gives the programmer a set of modules already there. In the standalone interface, all modules (except for Base modules) have to be explicitly imported because they could come from many places. This is done using a functor. Here is how this is done for the bouncing ball:

```haskell
functor
  import Graphics(window:Window canvas:Canvas canvasObject:CanvasObject)
  define
      ...
      % the interactive program as given before
  end
```

Here’s how to make a standalone application.

Here’s how to make an applet.

### 10.7.2 Multi-agent systems

The area of multi-agent systems is vast. Loosely, any set of communicating concurrent entities is a multi-agent system. More specifically, any set of communicating active objects is a multi-agent system. An example is the bouncing ball program of Chapter 2. Any distributed algorithm over an asynchronous network is a multi-agent system.

The essential goal of multi-agent systems is to achieve some global goal through local cooperation of the agents. The agents have no global information; they just communicate with some other agents. Consider a set of communicating active objects. Each active object is a self-contained entity with its own state and its own “goals”. The crucial aspect is to make the step from local reasoning on each active object to global reasoning on the set of active objects. How can we take this step?

There are many different approaches to take this step. The key is to have a formalism that allows to reason about all the agents together. Then, from the known local behavior we can deduce what the global behavior will be. Here is a list of useful formalisms:
• Use the theory of algorithms. The agent system is considered to be executing a distributed algorithm. Standard techniques for reasoning about algorithms can be used, e.g., using axiomatic semantics or structured operational semantics.

• Use logic. A typical logic for intelligent agents is a combination of a modal logic (reasoning with the operators “believes” and “knows”) with a temporal logic (reasoning over a time sequence of events).

• Use thermodynamics and statistical mechanics. This is a useful approach when dealing with very large communities of agents.

• Use economic theory. The agent system is considered as a physical system that obeys a system of equations. These equations can be nonlinear differential equations.

We can also go in the other direction: from a desired global behavior, we can deduce what the local behavior must be. This is more difficult.

In this book, we will limit ourselves to the distributed algorithm approach and use standard techniques for reasoning about discrete algorithms.

what is an agent?
examples
simple transport scenario
agent styles
scheduling
question-answering agents (servers)
game-playing agents
path-finding agents
decision support example?
basic multi-agent idioms
cooperation, negociation, ...
agent communication
agent interaction protocols
goal: handle common contingencies in simple way
ambulance dispatching scenario
agent platforms and societies of agents
simple agent platform example
handling contingencies
representing agent knowledge

10.7.3 SugarScape

An agent-based simulation.
10.8 Exercises

1. **Number of interleavings.** Generalize the argument used in the chapter introduction to calculate the number of possible interleavings of \( n \) threads, each doing \( k \) operations. Using Stirling’s formula for the factorial function, 
   \[ n! \approx \sqrt{2\pi n} n^{n+1/2} e^{-n}, \]
   calculate a closed form approximation to this function.

2. **Limitations of stream communication.** In this exercise, we explore the limits of stream communication in the nondeterministic concurrent model. Section 10.2.3 claims that we can partially solve the problem of putting server references in a data structure. How far can we go? Consider the following active object:

   ```prolog
   declare NS
   thread {NameServer NS nil} end
   
   where NameServer is defined as follows:
   
   ```prolog
   proc {NameServer NS L}
   case NS
   of register(A S)|NS1 then
     {NameServer NS1 A#S|L}
   [] getstream(A S)|NS1 then L1 OldS NewS in
     L1={Replace L A OldS NewS}
     thread {StreamMerger S NewS OldS} end
     {NameServer NS1 L1}
   [] nil then
     skip
   end
   end
   
   fun {Replace InL A OldS NewS}
   case InL
   of !A#S|L1 then
     OldS=S
     A#NewS|L1
   [] E|L1 then
     E|{Replace L1 A OldS NewS}
   end
   end
   ```

   The `NameServer` object understands two commands. Assume that \( S \) is a server’s input stream and `foo` is the name we wish to give the server. Given a reference `NS` to the name server’s input stream, doing `NS=register(foo S)|NS1` will add the pair `foo#S` to its internal list \( L \). Doing `NS=getstream(foo S1)|NS1` will create a fresh input stream, \( S1 \), for the server whose name is `foo`, which the name server has stored on its internal list \( L \). Since `foo` is a

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
constant, we can put it in a data structure. Therefore, it seems that we can put server references in a data structure, by defining a name server. Is this a practical solution? Why or why not? Think before reading the answer in the footnote.\footnote{It’s not possible to name the name server! It has to be added as an extra argument to all procedures. Eliminating this argument needs explicit state.}

3. **Maximal concurrency and efficiency.** In between the concurrent stateful model and the maximally concurrent model, there is an interesting model called the *job-based* concurrent model. The job-based model is identical to the concurrent stateful model, except that whenever an operation would suspend, a new thread is created with only that operation (this is called a *job*) and the original thread continues execution.\footnote{The initial Oz language used the job-based model [84].} Practically speaking, the job-based model has all the concurrency of the maximally concurrent model, and in addition it can easily be implemented efficiently. For this exercise, investigate the job-based model. Is it a good choice for a concurrent programming language? Why or why not?

4. **Non-deterministic choice.** Section 48 shows how to implement `WaitOr` in the concurrent declarative model with exceptions. Exceptions do not play an essential role in this solution. Rather, what is essential is that they are associated with an observable non-determinism. To see this more clearly, implement `WaitOr` in the concurrent stateful model without using exceptions.

5. **Implementing coroutines.** The `Spawn` and `Resume` operations of Section 10.2.2 only work if there are exactly two coroutines. Write general versions of `Spawn` and `Resume` that work for any number of coroutines. That is, `Spawn` adds an entry for the coroutine in a global table and `Resume` chooses an entry from this table. The result of this exercise is a simple thread scheduler.

6. **Active objects sharing one thread.** Section 10.3.7 gives a small application, Ping-Pong, that has two active objects. Each object executes a method and then asynchronously calls the other. When one initial message is inserted into the system, this causes an infinite ping-pong of messages to bounce between the objects. What happens if two (or more) initial messages are inserted? For example, what happens if these two initial calls are done:

\begin{verbatim}
{Call Ping ping(0)}
{Call Pong pong(10000000)}
\end{verbatim}

Messages will still ping-pong indefinitely, but how? Which messages will be sent and how will the object executions be interleaved? Will the interleaving
be in lock-step (alternating between objects strictly), looser (subject to fluctuations due to thread scheduling), or something in between?

7. **Concurrent filter.** Section 10.3.9 defines a concurrent version of Filter, called ConcFilter, that calculates each output element independently, i.e., without waiting for the previous ones to be calculated.

- What happens when the following is executed:

```plaintext
declare Out
{ConcFilter [5 1 2 4 0] fun {$ X} X>2 end Out}
{Show Out}
```

How many elements are displayed by the Show What is the order of the displayed elements? If several displays are possible, give all of them. Is the execution of ConcFilter deterministic? Why or why not?

- What happens when the following is executed:

```plaintext
declare Out
{ConcFilter [5 1 2 4 0] fun {$ X} X>2 end Out}
{Delay 1000}
{Show Out}
```

What is displayed now by Show? If several displays are possible, give all of them.

- What happens when the following is executed:

```plaintext
declare Out A
{ConcFilter [5 1 A 4 0] fun {$ X} X>2 end Out}
{Delay 1000}
{Show Out}
```

What is displayed now? What is the order of the displayed elements? If, after the above, A is bound to 3, then what happens to the list Out?

- If the input list has \( n \) elements, what is the complexity (in “big-Oh” notation) of the number of operations of ConcFilter? Discuss the difference in execution time between Filter and ConcFilter.

8. **Simulating slow networks.** Section 10.4.1 defines a function SlowNet2 that creates a “slow” version of an object. But this definition imposes a strong order constraint. Each slow object defines a global order of its calls and guarantees that the original objects are called in this order. This constraint is often too strong. A more refined version would only impose order among object calls within the same thread. Between different threads, there is no reason to impose an order. Define a function SlowNet3 that creates slow objects with this property.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
9. **Monitors.** The monitor implementation of Figure 10.25 uses a concurrent stack to store nested lock tokens. Yet, because it is reentrant, only one lock token is needed at any moment in time. Simplify the monitor implementation to store the lock token with a cell instead of a concurrent stack.

10. **Fast and slow balls.** Modify the bouncing ball example to allow both fast and slow balls. Double-clicking the left mouse button will create a fast ball. The other actions are the same as before, e.g., clicking on the right button will remove the most-recent ball. The mouse event for double-clicking is called ???.

11. **Attracting balls.** Modify the bouncing ball example to give the balls a tendency to stay near, but not too near, the average position of all balls. Each ball updates its velocity \((x_d, y_d)\) by giving it a nudge in the direction of the average position, towards it if it is too far away and away from it if it is too close.

Implementing this example is done in two parts: first, decide on the extra messages needed between the manager and the balls. For example, the manager can ask each ball for its position, and when it has received all responses, calculate the average position. Each ball can ask the manager for the average position. Be careful about deadlock! The ball should send the manager its position before asking for the average position.
Chapter 11

Representative Languages

We have now introduced the full computation model and explained its useful subsets. At this point, it is interesting to see how other languages fit in this model. We have chosen four very different languages: Erlang, Haskell, Java, and Prolog. All four languages are strongly typed and have automatic memory management. We summarize their strong points:

- Erlang is an example of the active object paradigm. It is dynamically typed and designed for symbolic computation, fine-grained concurrency, and fault tolerance.

- Haskell is an example of the functional paradigm. It is statically typed, sequential, and designed for symbolic computation and problems that can be easily formulated as functions.

- Java is an example of the object-oriented paradigm. It is statically typed, concurrent, and designed for large applications that may be distributed.

- Prolog is an example of the logic paradigm. It is dynamically typed, sequential, and designed for symbolic computation and problems that have a simple logical formulation.

Each of these languages is a good representative example of its class. There are many other interesting languages. We apologize for leaving out most of them, but it could not be otherwise in the book: our purpose is not to explain all languages, an impossible task, but to explain computation models, a much more reasonable task since they are much fewer.

- For each language: summarize the language’s execution model, summarizes the particular highlights of the language, and give some small program examples to give a flavor of the language.
- Then show how to do these examples in the general computation model of the book (or in a particular computation model--explain which).
11.1 Erlang and concurrent programming

The Erlang language was developed by Ericsson for telecommunications applications, in particular, for telephony [4, 106]. Its implementation, the Ericsson OTP (Open Telecom Platform), features fine-grained concurrency (efficient threads), extreme reliability (high performance software fault tolerance), and hot code replacement ability (update software while the system is running). It is a high-level language that hides the internal representation of data and does automatic memory management. It has been used successfully in several Ericsson products.

11.1.1 Computation model

The Erlang computation model has an elegant layered structure. We describe first the centralized model and then show how it is extended for distribution and fault tolerance.

Centralized model

The Erlang computation model consists of active objects that communicate with each other through message passing. The language can be divided into two layers:

- **Functional core.** Active objects are programmed in a dynamically-typed strict functional language. Each active object contains one thread that runs a recursive function whose arguments are the thread’s state. Functions can be passed in messages.

- **Message passing extension.** Threads communicate by sending messages to other threads asynchronously in FIFO order. Each thread has a unique identifier, its PID, which is a constant that identifies the receiving thread, but can also be embedded in data structures and messages. Messages are values in the functional core. They are put in the receiving thread’s mailbox. Receiving can be synchronous or asynchronous; the receiving thread uses pattern matching to wait for and then remove messages that have a given form from its mailbox, without disturbing the other messages. This means that messages are not necessarily received in the order that they are sent.

An active object consists of a thread with its associated mailbox. An active object is called a process in Erlang terminology. The concept of class or inheritance does not exist. A process that spawns a new process specifies which function should be initially executed inside it.

Extensions for distribution and fault tolerance

The centralized model is extended for distribution and fault tolerance:
• **Transparent distribution.** Processes can be on the same machine or on different machines. A single machine environment is called a *node* in Erlang terminology. In a program, communication between local or remote processes is written in exactly the same way. The PID encapsulates the destination and allows the run-time system to decide whether to do a local or remote operation. Processes are stationary; sending a message to a remote process requires exactly one network operation. This means that Erlang is network transparent and network aware.

• **Fault detection.** A process can be set up to detect faults in another process. In Erlang terminology this is called *linking* the two processes. When the second process fails, a message is sent to the first, which can receive it. This fault detection ability means that fault tolerance can be programmed entirely in Erlang.

• **Persistence.** The Erlang run-time system comes with a database, called Mnesia, that helps to build highly available applications.

### 11.1.2 The receive operation

Much of the unique flavor and expressiveness of concurrent programming in Erlang is due to the mailboxes and how they are managed. Messages are taken out of a mailbox with the *receive* operation, which uses pattern matching to pick out the desired messages. Using *receive* gives particularly compact, readable, and efficient code. We show how to translate *receive* into the computation models of this book. There are two reasons for giving the translation. First, it gives a precise semantics for *receive*, which aids the understanding of Erlang. Second, it shows how to do Erlang-style programming in Oz.

Because Erlang is functional, *receive* is an expression that returns a value. The *receive* expression has the following general form [4]:

```erlang
receive
  Pattern1 [when Guard1] -> Body1;
  ...
  PatternN [when GuardN] -> BodyN;
  [ after Expr -> BodyT; ]
end
```

The guards and the *after* clause are optional. This suspends until a message matching one of the patterns arrives in the current thread’s mailbox. It then removes this message, binds the corresponding variables in the pattern, and executes the body. Patterns are very similar to patterns in the *case* statement of this book: they introduce new single-assignment variables whose scope ranges over the corresponding body. For example, the Erlang pattern `\{rectangle, [X,Y]\}` corresponds to the pattern `rectangle([X, Y])`. Identifiers starting with lowercase letters correspond to atoms and identifiers starting with capital letters...
correspond to variables, like the notation of this book. Compound terms are enclosed in braces \{ and \} and correspond to tuples.

The optional \texttt{after} clause defines a time out; if no matching message arrives after a number of milliseconds given by evaluating the expression \texttt{Expr}, then the time-out body is executed. If zero milliseconds are specified, then the \texttt{after} clause is executed immediately if there are no messages in the mailbox.

\section*{General remarks}

Each Erlang process is translated into one thread with one port. Sending to the process means sending to the port. This adds the message to the port’s stream, which represents the mailbox contents. All forms of \texttt{receive}, when they complete, either take exactly one message out of the mailbox or leave the mailbox unchanged. We model this by giving each translation an input stream and an output stream. All translations have two free variables, \texttt{Sin} and \texttt{Sout}, that reference the input stream and the output stream. These streams do not appear in the Erlang syntax, but they exist in the semantics. There are two possibilities for the value of the output stream. Either it is the same as the input stream or it has one less message than the input stream. The latter occurs if the message matches a pattern in the \texttt{receive} expression.

There are three fundamentally different forms of \texttt{receive} that result in different translations. In each form the translation can be directly inserted in a program and it will behave like the respective \texttt{receive}. The first two translations use the declarative model. The third form needs \texttt{WaitOr} and therefore uses the nondeterministic concurrent model (see Section 10.2).

\subsection*{First form}

The first form of the \texttt{receive} expression is as follows:

\begin{verbatim}
receive
  Pattern1 -> Body1;
  ...
  PatternN -> BodyN;
  Var -> BodyV;
end
\end{verbatim}

The last pattern is a variable identifier \texttt{Var}. This means that the expression never suspends. It will execute \texttt{BodyV} immediately if there is no message matching one of the other patterns. This is the simplest case, since there is no out-of-order reception. The translation is as follows:

\begin{verbatim}
case Sin of M|S then Sout=S
  case M
    of TPattern1 then TBody1
    ...
end
\end{verbatim}

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
The pattern \( T_{\text{Pattern1}} \) corresponds to \( \text{Pattern1} \) through a straightforward change of syntax. Similarly, the expression \( T_{\text{Body1}} \) corresponds to \( \text{Body1} \).

**Second form**

The second form has no variable identifier in the patterns. The \texttt{receive} expression suspends until a message arrives that matches one of the patterns. Out-of-order reception is possible: if messages of a wrong pattern arrive, then they are put in the output stream and do not cause the \texttt{receive} to complete. The expression has the following form:

\[
\text{receive}
\quad \text{Pattern1} \rightarrow \text{Body1};
\quad \ldots
\quad \text{PatternN} \rightarrow \text{BodyN};
\text{end}
\]

The translation is as follows:

\[
\text{local}
\quad \text{fun} \ (\text{Loop} \ \text{S} \ \text{T})
\quad \text{case} \ \text{S} \ \text{of} \ \text{M}\mid \text{S2} \ \text{then}
\quad \quad \text{case} \ \text{M}
\quad \quad \quad \text{of} \ \text{TPattern1} \ \text{then} \ \text{T=S2} \ \text{TB}ody1
\quad \quad \quad \ldots
\quad \quad \quad \quad \quad \text{[]} \ \text{TPatternN} \ \text{then} \ \text{T=S2} \ \text{TB}odyN
\quad \quad \quad \quad \quad \quad \text{[]} \ \text{M} \ \text{then} \ \text{T2} \ \text{in} \ \text{T=M}\mid \text{T2} \ \{\text{Loop} \ \text{S2} \ \text{T2}\}
\quad \quad \text{end}
\quad \text{end}
\quad \text{in}
\quad \{\text{Loop} \ \text{Sin} \ \text{Sout}\}
\text{end}
\]

This uses the local function \texttt{Loop} to manage the out-of-order reception: if a message \( \text{M} \) is received that does not match any pattern, then it is put in the output stream and \texttt{Loop} is called recursively.

**Third form**

The third form does a time out. The \texttt{receive} expression suspends until a message arrives that matches one of the patterns. If no match is done after \( N \) milliseconds, then the time-out action is executed. If a match is done before \( N \) milliseconds,
then it is handled as if there were no time out. The expression has the following form:

\[
\text{receive} \\
\text{Pattern}_1 \rightarrow \text{Body}_1; \\
\ldots \\
\text{Pattern}_N \rightarrow \text{Body}_N; \\
\text{after} \text{Expr} \rightarrow \text{Body}_T; \\
\text{end}
\]

The translation is as follows:

\[
\text{local} \\
\text{Cancel} = \text{Alarm TExpr} \\
\text{fun} \{\text{Loop} \ S \ T\} \\
\quad \text{if} \{\text{WaitOr} \ S \ \text{Cancel}\}==1 \ \text{then} \\
\quad \text{case} \ S \ \text{of} \ M | S2 \ \text{then} \\
\quad \quad \text{case} \ M \ \text{of} \ T\text{Pattern}_1 \ \text{then} \ T=S2 \ T\text{Body}_1 \\
\quad \quad \ldots \\
\quad \quad [ ] \ T\text{Pattern}_N \ \text{then} \ T=S2 \ T\text{Body}_N \\
\quad \quad [ ] \ M \ \text{then} \ T2 \ \text{in} \ T=M | T2 \ \{\text{Loop} \ S2 \ T2\} \\
\quad \text{end} \\
\quad \text{end} \\
\quad \text{else} \\
\quad \ T=S \ T\text{Body}_T \\
\quad \text{end} \\
\text{end} \\
\text{in} \\
\{\text{Loop} \ S2 \ S2\}
\]

The expression TExpr corresponds to Expr. We explain how the translation works. It uses a timer interrupt. The Alarm call waits and binds the unbound variable Cancel after a delay (see Section 6.8). The WaitOr call (explained in Section 10.2.3) simultaneously waits for two events: either a message arrives (S is bound) or time is up (Cancel is bound). It returns 1 or 2 depending on whether its first or second argument is determined. If a message arrives that does not match, then it is put in the output stream and the wait operation is repeated.

If N= 0 then a more efficient translation is possible. Remove the Alarm call and replace the WaitOr call by \{IsDet S\}. This immediately executes TBodyT if there are no messages in the mailbox.

### 11.2 Haskell and functional programming

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
11.2.1 Computation model

The Haskell computation model consists of functions that take complete values and return complete values. Haskell is a strongly-typed language, i.e., the types of all variables are enforced by the implementation. This model is closest to the concurrent declarative model of Chapter 6. Haskell differs from this model in the following ways:

- There is syntactic and implementation support for implicit state threading using monads (see Section 8.3.2). Monads allow to hide the explicit state threading that would be necessary in the declarative model.

- There is syntactic and implementation support for currying (see Section 5.8.6).

- All functions are lazy by default. There is compiler support, called strictness analysis, to determine when the laziness is not necessary for termination or resource control. Functions that do not need laziness are compiled as eager ("strict") functions, which is much more efficient.

- There is no concurrency or dataflow. There is an extension called Concurrent Haskell that supports both concurrency and dataflow.

- Type enforcement is done at compile time; we say that Haskell is statically typed. No type mismatches are possible at run time. Furthermore, Haskell does type inferencing. It is not necessary for the programmer to declare types; the compiler is powerful enough to infer all the types in the program, by inspecting the program. This eliminates the drudgery of having to declare all types.

11.3 Java and object-oriented programming

11.3.1 The Java language

Java is a concurrent object-oriented language with a syntax that resembles C++. Unlike C++, Java is a relatively clean language with a simple semantics. Its implementations have automatic memory management and support for distributed computing on multiple platforms.

11.3.2 Computation model

The Java computation model is close to the concurrent stateful model. It consists of statically-typed object-oriented programming with explicit threads. Java implementations are optimized for the atomic action approach; threads are too heavyweight to efficiently support the message passing approach. The basic concurrency primitive is the monitor. Monitors are lightweight constructs that can be associated to any object.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
11.4 Prolog and logic programming

Despite many extensions and new ideas, Prolog is still the most popular language for practical logic programming [87]. This is partly because Prolog has a quite simple operational model that easily accommodates many extensions and partly because no consensus has yet been reached on a successor. The computation model of Prolog is very close to the relational programming model of Chapter 7. Relational programming has a logical semantics, and is therefore a kind of purely declarative logic programming. Operationally, it subsumes “pure” Prolog programming, i.e., Prolog minus the nonlogical features.

Modern implementations of Prolog are efficient and provide rich functionality for application development (e.g., [12]). It is possible to compile Prolog with similar execution efficiency as C; the Aquarius and Parma systems are constructive proof of this [100, 92]. The successful series of conferences on Practical Applications of Prolog is witness to the usefulness of Prolog in industry.

Prolog is generally used in application areas in which complex symbolic manipulations are needed, such as expert systems, specialized language translators, program generation, data transformation, knowledge processing, deductive databases, and theorem proving. There are two application areas in which Prolog is still predominant over other languages: natural language processing and constraint programming. The latter in particular has matured from a subdomain of logic programming into a research domain in its own right.

Prolog has many advantages for such applications. The bulk of programming can be done cleanly in its pure declarative subset. Programs are concise due to the expressiveness of unification and the term notation. Memory management is dynamic and implicit. Powerful primitives exist for useful non-declarative operations. The call/1 provides a form of higher-orderness (first-class procedures, but without lexical scoping). The setof/3 provides a form of encapsulated search that can be used as a database query language.

In addition to Prolog’s computation model, there are three other popular computation models for logic programming: concurrent logic programming, constraint logic programming, and concurrent constraint programming. We treat them elsewhere in the book:

- The nondeterministic concurrent model of Section 10.2.3 subsumes concurrent logic programming and concurrent constraint programming. See Section 10.1 for more information.

- The constraint-based computation model of Chapter 14 subsumes constraint logic programming. See Chapter 14 for more information.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
11.4 Prolog and logic programming

11.4.1 Context

The two programming styles

Logic programming languages have traditionally been used in two very different ways:

- For *algorithmic problems*, i.e., for which efficient algorithms are known. Most applications written in Prolog, including expert systems, are of this kind.

- For *search problems*, i.e., for which efficient algorithms are not known, but that can be solved with search. For example, combinatorial optimization or theorem proving. Most applications in constraint programming are of this kind.

Prolog was originally designed as a compromise between these two styles. It provides backtracking execution, which is just built-in depth-first search. This compromise is not ideal. For algorithmic problems the search is not necessary. For search problems the search is not good enough. This problem has been recognized to some extent since the original conception of the language in 1972. The first satisfactory solution, encapsulating search with computation spaces, was given by the AKL language in 1990 [38, 50]. The unified model of this book simplifies and generalizes the AKL solution (see Chapter 14).

Future developments

Three major directions in the evolution of logic programming languages:

- **Mercury.** An evolution of Prolog that is completely declarative, statically typed and moded, and higher-order. It focuses on the algorithmic programming style. It keeps the Horn clause syntax and global backtracking.

- **Oz.** An evolution of Prolog and concurrent logic programming that cleanly separates the algorithmic and search programming styles. It also cleanly integrates logic programming with other computation models. It replaces the Horn clause syntax with a syntax that is closer to functional languages.

- **Constraint programming.** An evolution of Prolog that consists of a set of constraint algorithms and ways to combine them to solve complex optimization problems. This focuses on the search programming style. Constraint techniques can be presented as libraries (e.g., ILOG Solver is a C++ library) or language extensions (e.g., SICStus Prolog and Oz).

11.4.2 The Prolog language

Logical core

Explain SLDNF resolution.
Nonlogical features

Explain cut, var, negation as failure, assert/retract, input/output.

Critical assessment

The foundation of Prolog’s success is the high abstraction level of its declarative subset, namely first-order Horn clause logic with SLDNF resolution. What’s missing from Prolog is that little attempt is made to give the same foundation to anything outside the declarative subset. Two decades of research have resulted in a solid understanding of the declarative subset and only a partial understanding of the rest.\(^1\) This results in two main flaws of Prolog:

- The operational aspects are too deeply intertwined with the declarative. The control is naive (depth-first search) and eager. The interactive top level has a special status: it is lazy and is not accessible to programs. There is no efficient way to modify the search strategy.

- To express anything beyond the declarative subset requires ad hoc primitives that are limited and do not always do the right thing. The \texttt{freeze/2} operation provides coroutining as a limited form of concurrency. The \texttt{call/1} and \texttt{setof/3} operations provide a limited form of higher-orderness.\([?)\]. There are no complete forms of concurrent and higher-order programming.

11.4.3 Translating Prolog into a relational program

We show how to translate any pure Prolog program into a relational program with the same logical semantics and operational semantics. The translation is a purely syntactic one. The opposite is not true because the declarative model is higher-order and compositional. We also show how to translate properly-written Prolog programs with cut into relational programs.

Translating pure Prolog

To translate a pure Prolog program to a relational program in the declarative model, follow these three rules for each of the program’s predicates:

- Translate deterministic predicates, i.e., those that do not do backtracking, into relational programs using \texttt{if} and \texttt{case} but not \texttt{choice}.

- Translate nondeterministic predicates, i.e., those that are backtracked into, into procedures with \texttt{choice}. Each clause is one alternative of the \texttt{choice} statement.

\(^1\)The non-declarative aspect has received some attention, e.g., [68, ?, 72, 3].

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
If you have doubts whether the predicate is deterministic or nondeterministic, then your Prolog program may have a memory leak. This is because it may be accumulating choice points forever. We strongly suggest that you rewrite the predicate so that it is clearly deterministic or nondeterministic. If you do not or cannot do this, then translate it as if it were nondeterministic. The resulting relational program will have a memory leak if the Prolog program does.

Here are examples of a deterministic and a nondeterministic Prolog predicate, translated according to these rules. Consider the following deterministic Prolog predicate:

```prolog
place_queens(0, _, _, _) :- !.
place_queens(I, Cs, Us, [_;|Ds]) :-
    I>0, J is I-1,
    place_queens(J, Cs, [_;|Us], Ds),
    place_queen(I, Cs, Us, Ds).
```

This predicate has a blue cut according to O'Keefe [70], i.e., the cut is needed to inform naive implementations that the predicate is deterministic, so they can improve efficiency, but it does not change the program's results. The predicate is translated as:

```prolog
proc {PlaceQueens N Cs Us Ds}
    if N==0 then skip
    elseif N>0 then
        Ds2 Us2=_|Us
        Ds=_|Ds2
        {PlaceQueens N-1 Cs Us2 Ds2}
        {PlaceQueen N Cs Us Ds}
    else fail end
end
```

The following nondeterministic Prolog predicate:

```prolog
placequeen(N, [N|_], [N|_], [N|_]).
placequeen(N, [_|Cs2], [_|Us2], [_|Ds2]) :-
    placequeen(N, Cs2, Us2, Ds2).
```

is translated as:

```prolog
proc {PlaceQueen N Cs Us Ds}
    choice N|_ =Cs N|_ =Us N|_ =Ds
    [] _|Cs2=Cs _|Us2=Us _|Ds2=Ds in
    {PlaceQueen N Cs2 Us2 Ds2}
end
end
```

In both examples, the logical semantics and the operational semantics of the Prolog and relational versions are identical.
The cut operation ("!")

If your Prolog program uses cut "!", then the translation to a relational program is often simple if the cut is a grue cut, i.e., a blue or green cut, as defined by O’Keefe [70]. Grue cuts do not have any effect on logical semantics but they improve the program’s efficiency. Let us translate the following predicate:

\[
\text{foo}(X, Z) :- \text{guard1}(X, Y), !, \text{body1}(Y, Z).
\]
\[
\text{foo}(X, Z) :- \text{guard2}(X, Y), !, \text{body2}(Y, Z).
\]

The guards must not bind any head variables. We say that the guards are quiet. It is good Prolog style to postpone binding head variables until after the cut. The translation has two cases, depending on whether the guards are deterministic or not. If a guard is deterministic (it has no choice), then write it as a deterministic boolean function. This gives the following simple translation:

\[
\text{proc } \{\text{Foo } X \ Z\}
\]
\[
\text{if } \ y \ \text{in } \{\text{Guard1 } X \ Y\} \ \text{then } \{\text{Body1 } Y \ Z\}
\]
\[
\text{elseif } \ y \ \text{in } \{\text{Guard2 } X \ Y\} \ \text{then } \{\text{Body2 } Y \ Z\}
\]
\[
\text{else fail end}
\]

If a guard is nondeterministic (it uses choice), then write it so that it has one input and one output argument, like this: \{\text{Guard1 In Out}\}. It should not bind the input argument. This gives the following translation:

\[
\text{proc } \{\text{Foo } X \ Y\}
\]
\[
\text{case } \{\text{Search.base.one fun } \{\} \ {\text{Guard1 } X}\} \text{ end} \ \text{of } [Y] \ \text{then}
\]
\[
\{\text{Body1 } Y \ Z\}
\]
\[
\text{elseif } \{\text{Search.base.one fun } \{\} \ {\text{Guard2 } X}\} \text{ end} \ \text{of } [Y] \ \text{then}
\]
\[
\{\text{Body2 } Y \ Z\}
\]
\[
\text{else fail then end}
\]

If neither of these two cases apply to your Prolog program, e.g., either your guards bind head variables or you use cuts in other ways (i.e., as red cuts), then it likely does not have a logical semantics. Its behavior is defined only by its operational semantics. In that case, the translation is not automatic. You will have to understand the program’s operational semantics and use this knowledge to translate into an equivalent relational program.

The bagof predicate

Prolog’s bagof predicate corresponds to using Search.base.all inside a relational program. Consider the following small biblical database (taken from [87]):

\[
\text{father(terach, abraham).}
\]
\[
\text{father(terach, nachor).}
\]
\[
\text{father(terach, haran).}
\]
This can be written as follows as a relational program:

\[
\text{proc \{Father F C\}}
\]
\[
\text{choice F=terach C=abraham}
\]
\[
\text{[] F=terach C=nachor}
\]
\[
\text{[] F=terach C=haran}
\]
\[
\text{[] F=abraham C=isaac}
\]
\[
\text{[] F=haran C=lot}
\]
\[
\text{[] F=haran C=milcah}
\]
\[
\text{[] F=haran C=yiscah}
\end
\]
\end

Calling \text{bagof} without existential quantification, for example:

\[
\text{children1(X, Kids) :- bagof(K, father(X,K), Kids).}
\]
is defined as follows with \text{Search.base.all}:

\[
\text{proc \{Children1 X Kids\}}
\]
\[
\text{\{Search.base.all proc \{Father X K\} end Kids\}}
\end
\]

The \text{Children1} definition is deterministic; it assumes \text{X} is known and it returns \text{Kids}. To search over different values of \text{X} the following definition should be used instead:

\[
\text{proc \{Children1 X Kids\}}
\]
\[
\text{\{Father X _\}}
\]
\[
\text{\{Search.base.all proc \{Father X K\} end Kids\}}
\end
\]

The call \{Father X _\} creates a choice point on \text{X}. The "_" is syntactic sugar for \text{local X in X end}, which is just a new variable with a very small scope. Calling \text{bagof} with existential quantification, for example:

\[
\text{children2(Kids) :- bagof(K, X^father(X,K), Kids).}
\]
is defined as follows with \text{Search.base.all}:

\[
\text{proc \{Children2 Kids\}}
\]
\[
\text{\{Search.base.all proc \{Father _ K\} end Kids\}}
\end
\]

The relational solution uses _ to add a new existentially-scoped variable. The Prolog solution, on the other hand, introduces a new concept, namely the "existential quantifier" \text{X^}, which only has meaning in terms of \text{setof/3} and \text{bagof/3}. The fact that this notation denotes an existential quantifier is arbitrary. The
relational solution introduces no new concepts. It really does existential quantification inside the search query.

In addition to doing all-solutions bagof, relational programs can do a *lazy bagof*, i.e., where each new solution is calculated on demand. Lazy bagof can be done by using the Search.object to create a search engine and asking it for solutions.

### 11.5 Exercises

1. Section 11.1.2 shows how to translate Erlang’s receive expression. The third form of this expression is translated with WaitOr. Another way to translate the third form would be to insert a unique message (using a name) after N milliseconds. This requires some care to keep the unique message from appearing in the output stream. Write another translation of the third form that uses this technique. What are the advantages and disadvantages of this translation with respect to the one in the book?
Part III

Specialized Computation Models
Chapter 12

Graphic User Interface Programming
Chapter 13

Distributed Programming
Chapter 14

Constraint Programming
Part IV

Semantics
Chapter 15

Language Semantics

“Formal semantics is the ultimate authority: it brings order to chaos, it knows all and tells all, and the buck stops with it.”
– The Enlightened Programmer

“Formal semantics is the ultimate programming language.”
– The Enlightened Language Designer

“All good programmers are language designers and all good language designers are programmers.”
– Computer science folk proverb.

“Always make things as simple as possible, but no simpler.”
– Albert Einstein (1879–1955)

This chapter defines a formal semantics for all the computation models introduced in the previous chapters.¹ For most of the computation models, we give only a structural operational semantics. The exceptions are the constraint-based model and its simplification, the relational model. For them we give a logical semantics as well.

15.1 Historical background

In the late 1980’s, a new model of computation known as the concurrent constraint model arose from research in concurrent logic programming and constraint logic programming [79, 63, 48]. The computation models of the book are based on this model. The concurrent stateful model adds first-class procedures, stateful data structures, and by-need synchronization to the basic concurrent constraint model of Saraswat [84]. The unified model adds further concepts for constraints and distribution.

¹This chapter’s primary author is Raphaël Collet.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
15.2 The concurrent stateful model

This section gives a structural operational semantics for the concurrent stateful model. The semantics of each earlier model, namely declarative, concurrent declarative, and stateful, can be gotten by taking just the rules for the language constructs that exist in those models.

15.2.1 Store

A computation in the concurrent stateful consists of several tasks connected to a shared store. A task, also called thread, is the basic unit of sequential calculation. A computation is a sequence of task reductions. At each step, one task is chosen among all reducible tasks to do a reduction step. A computation is a sequence of reduction steps. The execution of different tasks is interleaved in this sequence. We say that the model has an interleaving semantics. The model is concurrent but makes no assumption about parallelism.

\[
\begin{array}{c}
task \cdots task \\
\downarrow \\
store
\end{array}
\]

The store is a repository of information about the program variables. For instance, the store can say: “\( V \) is bound to 3, and \( V \) and \( W \) are equal”\(, \) which is written \( V=3 \land V=W \). An essential feature of the store is that it entails information. For example, the store \( \sigma \equiv V=3 \land V=W \) entails “\( W \) equals 3”, even if it does not explicitly contain that information. This relation is written \( \sigma \models W=3 \). Entailment is the general relation we use to query the store.

The store is monotonic: information can be added but not changed. Consequently, entailment is monotonic too: when the store entails some information, it continues to entail this information forever.

The name “concurrent constraint model” comes from the fact that the tasks generally impose constraints on variables. The constraints are simply logic formulas specifying relations between the variables. Consequently, the store can be seen as a logic conjunction of constraints, and entailment is nothing more than logic entailment.

The store provides two primitive operations, called tell and ask:

- **Tell.** The tell operation is a mechanism to add information to the store. A task telling the information \( \beta \) to store \( \sigma \) updates the store to \( \sigma \land \beta \), provided that the new store is consistent. For instance, a task may not tell \( W=7 \) to the store \( V=3 \land V=W \). It may however tell \( W=3 \), which is consistent with the store. An inconsistent tell leaves the store unchanged. It is signaled with some mechanism, typically by raising an exception.

- **Ask.** The ask operation is a mechanism to query the store for the presence of some information. A task asking store \( \sigma \) for information \( \gamma \) becomes...
15.2 The concurrent stateful model

Figure 15.1: The kernel language of the concurrent stateful model

reducible when $\sigma$ entails either $\gamma$ or its negation $\neg\gamma$. For instance, with the store $V=3 \land V=W$, asking for $W=3$ will give an affirmative answer (the information is present). Asking for $W=4$ will give a negative answer (the information will never be present). The task will not reduce until either an affirmative or negative answer is possible. Therefore the ask operation is a synchronization mechanism. The task doing the ask is said to synchronize on $\gamma$, which is called its guard.

Monotonicity of the store implies a strong property of the concurrent constraint model: task synchronization is monotonic. It means that once a task is reducible, it stays reducible even if other tasks are reduced before it. This is an excellent basis for dataflow concurrency, where tasks synchronize on the availability of data.

15.2.2 Abstract syntax

Figure 15.1 defines the abstract syntax for the kernel language of the concurrent stateful model. The metavariables $S, S_i$ denote statements, $C, P, X, X_i, Y$ denote variable identifiers, $z$ denotes an integer constant, and the indices are integers $i, n \geq 0$. In the record $f(l_1:X_1 \ldots l_n:X_n)$, the label $f$ denotes an atom, and each one of the features $l_i$ denotes an atom or integer constant. We use $\equiv$ to denote equality between semantic objects, in order to avoid confusion with $=$ in
the equality statement.

We assume that in any statement defining a lexical scope for a list of variable identifiers, the identifiers in the list are pairwise distinct. To be precise, in the three statements:

\[
\begin{align*}
\text{local } X_1 \cdots X_n \text{ in } S \text{ end} \\
\text{case } X \text{ of } f(l_1:X_1 \ldots l_n:X_n) \text{ then } S_1 \text{ else } S_2 \text{ end} \\
\text{proc } \{ P X_1 \cdots X_n \} \text{ S end}
\end{align*}
\]

we must have \( X_i \neq X_j \) for \( i \neq j \).

### 15.2.3 Structural rules

The system advances by successive reduction steps. A possible reduction step is defined by a reduction rule of the form:

\[
\frac{T_{\sigma} \mid T'_{\sigma'}}{\text{if } C}
\]

stating that the computation makes a transition from a multiset of tasks \( T \) connected to a store \( \sigma \), to a multiset of tasks \( T' \) connected to a store \( \sigma' \). The rule can have an optional boolean condition \( C \), which has to be true for the rule to reduce.

We use a very light notation for multisets of tasks: the multiset is in calligraphic style, disjoint union is denoted by a comma, and singletons are written without curly braces. This allows to write “\( T_1, T_2 \)” for \( \{T_1\} \cup \{T_2\} \). We generally write “\( \sigma \)” to denote a store, leaving implicit the set of its variables, say \( V \). To make this set explicit, we write the store with \( V \) as a subscript: \( \sigma_V \).

The following two rules express model properties that are independent of the programming language. They can be considered as formalism conveniences.

\[
\frac{T, U \mid T', U}{\sigma \mid \sigma'} \quad \text{if } \frac{T \mid T'}{\sigma \mid \sigma'} \quad \text{if } \frac{T \mid T'}{\sigma \mid \sigma'} \text{ if } \sigma \text{ and } \sigma' \text{ are equivalent}
\]

The first rule expresses concurrency in our model: a subset of the threads can reduce without directly affecting or depending on the others. The second rule states that the store can be replaced by an equivalent one. Two stores \( \sigma \) and \( \sigma' \) are equivalent if they entail one another, that is \( \sigma \models \sigma' \) and \( \sigma' \models \sigma \). This is possible because entailment makes the store independent of its representation: if \( \sigma \) and \( \sigma' \) are equivalent, then \( \sigma \models \gamma \) if and only if \( \sigma' \models \gamma \).

There are two ways to express that a rule has the entailment condition \( \sigma \models \beta \). The condition can be written as part of the store or as an explicit condition to the right:

\[
\frac{T, \sigma \models \beta \mid T', \sigma \models \beta}{\sigma \models \beta} \quad \text{or} \quad \frac{T, \sigma \models \beta \mid T', \sigma \models \beta}{\sigma \models \beta}
\]

In the definitions that follow, we use whichever notation is the most convenient.
15.2.4 Sequential and concurrent composition

A thread is characterized by a list of statements $S_i$ with the termination symbol $\bullet$, that is, $S_1 (S_2 (S_3 \cdots (S_n \bullet)\cdots))$. The abstract syntax of threads is

$$T ::= \bullet | ST.$$ 

A terminated thread has the form $\bullet$. Its reduction simply leads to an empty set of threads. A non-terminated thread has the form $ST$. Its reduction replaces its topmost statement $S$ by its reduction $S'$:

$$\frac{\bullet }{\sigma} \mid \sigma \quad \frac{ST}{\sigma} \mid \sigma' \quad \frac{S T}{\sigma} \mid \sigma' \quad \frac{ST}{\sigma} \mid \sigma'$$

The empty, sequential, and concurrent compositions are intimately linked to the notion of thread. Their reduction needs a more specific definition than the one given above for $S$:

$$\frac{skip T}{\sigma} \mid \sigma \quad \frac{(S_1 S_2) T}{\sigma} \mid \sigma \quad \frac{(S_1 T)}{\sigma} \mid \sigma \quad \frac{(thread S end) T}{\sigma} \mid \sigma \quad \frac{(S \bullet)}{\sigma} \mid \sigma$$

The empty statement $\text{skip}$ is removed from the thread’s statement list. A sequence $S_1 S_2$ makes $S_1$ the thread’s topmost statement, while $\text{thread } S \text{ end}$ creates a new thread with statement $S$, that is $(S \bullet)$.

15.2.5 Variable introduction

As we saw before, the store contains information about the variables of the program. But a program text contains identifiers, not variables. A statement containing a free identifier, i.e., an identifier that is not declared, cannot be reduced.\(^2\) The local statement does variable introduction: it creates new variables in the store and maps its variable identifiers to these variables. An informal definition of this operation is given in Section 4.4.2.

To avoid confusing variables and identifiers in the semantics rules, we use “$V$” to denote a variable and “$X$” to denote an identifier. In the following statement, the identifier Foo in $S_2$ refers to a variable which is distinct from the one referred by Foo in $S_1$ and $S_3$:

$$\begin{align*}
\text{local Foo Bar in} \\
\text{local Foo in } S_2 \text{ end}
\end{align*}
\equiv Stmt$$

\(^2\)Feeding such a statement to Mozart gives a “variable not introduced” error.
The gap between variable identifiers and variables is filled by the way we reduce statements defining a lexical scope for identifiers. The idea is to explicitly substitute a new variable for every identifier:

\[
\text{local } X_1 \ldots X_n \text{ in } S \text{ end}\]

\[
\sigma_V \left[ S[V_1 \ldots V_n/X_1 \ldots X_n] \right] \quad V_1, \ldots, V_n \text{ fresh variables}
\]

A variable is fresh if it is different from all existing variables in the store. So the condition of the rule states that all the variables \( V_i \) are distinct and not in \( \mathcal{V} \).

The notation \( S[V_1 \ldots V_n/X_1 \ldots X_n] \) stands for the simultaneous substitution of the free occurrences of \( X_1 \) by \( V_1 \), \( X_2 \) by \( V_2 \), \ldots, \( X_n \) by \( V_n \). For instance, the substitution of \( \text{Foo} \) by \( V \), and \( \text{Bar} \) by \( W \) in the statement \( Stmt \) defined above gives:

\[
Stmt[V,W/\text{Foo},\text{Bar}] \equiv S_1[V,W/\text{Foo},\text{Bar}] \\
\text{local } \text{Foo} \text{ in } S_2[W/\text{Bar}] \text{ end} \\
S_3[V,W/\text{Foo},\text{Bar}]
\]

The definition of the semantics only requires substituting identifiers by variables. Since variables and identifiers are disjoint, the simultaneous substitution \( S[V_1 \ldots V_n/X_1 \ldots X_n] \) is equivalent to the composition of single substitutions \( S[V_1/X_1] \cdots [V_n/X_n] \). A complete definition of the substitution operation \( S[V/X] \) is given in Section 15.2.12.

### 15.2.6 Imposing equality (tell)

According to Section 15.2.5, a variable introduced by `local` has no initial value. The variable exists but the store simply has no information about it. Adding information about the variable is done by the `tell` operation. Let \( \beta \) denote a statement imposing equality. This statement has three possible forms:

\[
\beta ::= V = V' \mid V = z \mid V = f(l_1:V_1 \ldots l_n:V_n)
\]

This states that \( V \) is equal to either another variable \( V' \), an integer constant \( z \), or a record with label \( f \), features (i.e., field names) \( l_i \), and fields \( V_i \). Doing a tell operation adds the information in \( \beta \) to the store, provided that it does not lead to an inconsistent store. This is also called binding the variable \( V \).

It is possible that the new information in \( \beta \) conflicts with what the store already knows about \( V \). We say that \( \beta \) is inconsistent with \( \sigma \). This happens whenever \( \beta \wedge \sigma \leftrightarrow \text{false} \). For example, take \( \beta \equiv V=10 \) and \( \sigma \equiv V=20 \). Instead of adding \( \beta \) to the store, we signal this as an error, e.g., by raising an exception. Therefore the store is always consistent.

In practice, most tell operations are very simple: telling \( \beta \) just binds one variable, \( V \), without binding any others. For example, telling \( V=23 \) where \( \sigma \) has no binding for \( V \). But the tell operation is actually much more general. It can cause many bindings to be done. For example, take \( \sigma \equiv V=f(V_1 V_2) \land \ldots \land \).
15.2 The concurrent stateful model

\[ V' = f(V_1', V_2'). \] Then telling \( V = V' \) does three bindings: \( V = V', V_1 = V_2, \) and \( V_1' = V_2' \).

This generality has an important consequence for inconsistent tells. For example, take \( \beta \equiv V = V' \) and \( \sigma \equiv V = f(V_1 V_2) \wedge V' = f(V_1' V_2') \wedge V_2 = a \wedge V_2' = b. \) The tell is inconsistent. Does the tell add \( V_1 = V_1' \) to the store? It would be nice if the tell did nothing at all, i.e., \( \sigma \) is unchanged afterwards. But this is very expensive to implement: it means the tell operation would be a transaction, which is rolled back if an inconsistency is detected. The system would have to do a transaction for each variable binding. It turns out that implementing tell as a transaction is not necessary. If \( \beta \wedge \sigma \) is inconsistent, practical experience shows that it is perfectly reasonable that some bindings are already done when the inconsistency is detected.

For the semantics of a tell operation we therefore need to distinguish the binding of \( V \), which we call a basic binding, and the other bindings, which we call nonbasic bindings. We first define precisely what are the nonbasic bindings. Then we give the reduction rules for basic bindings and nonbasic bindings.

Nonbasic bindings

The nonbasic bindings are the extra bindings that happen as part of a tell operation, i.e., the bindings of other variables than \( V \). For a store \( \sigma \), we write \( \beta \rightarrow_{\sigma} \gamma \) to say that the binding \( \beta \) involves the extra binding \( \gamma \). The relation \( \rightarrow_{\sigma} \) is defined as the least reflexive transitive relation satisfying:

\[
\begin{align*}
V = f(l_1:V_1 \ldots l_n:V_n) & \rightarrow_{\sigma} V_i = W_i & \text{if } \sigma \models V = f(l_1:W_1 \ldots l_n:W_n) \\
V = W & \rightarrow_{\sigma} V_i = W_i & \text{if } \sigma \models V = f(l_1:V_1 \ldots l_n:V_n) \\
& & \wedge W = f(l_1:W_1 \ldots l_n:W_n)
\end{align*}
\]

We then define \( \text{sub}_{\sigma}(\beta) \), the set of subbindings strictly involved by \( \beta \), as

\[
\text{sub}_{\sigma}(\beta) = \left\{ \gamma \mid \beta \rightarrow_{\sigma} \gamma \text{ and } \gamma \not\rightarrow_{\sigma} \beta \text{ and } \sigma \not\models \gamma \right\}.
\]

Rules for basic bindings

The following rules decide whether to add \( \beta \) to the store, in the case where \( \beta \) does not involve other constraints.

\[
\begin{align*}
\text{skip} & \quad \frac{\beta \text{ \,\,\,\,} \text{skip}}{\sigma} \quad \text{if } \text{sub}_{\sigma}(\beta) = \emptyset \text{ and } \sigma \wedge \beta \text{ is consistent} \\
\text{fail} & \quad \frac{\beta \text{ \,\,\,\,} \text{fail}}{\sigma} \quad \text{if } \text{sub}_{\sigma}(\beta) = \emptyset \text{ and } \sigma \wedge \beta \text{ is inconsistent}
\end{align*}
\]

If only basic bindings are done, then these rules are sufficient. On the other hand, if there are nonbasic bindings, we need one more rule, which is explained next.
Rule for nonbasic bindings

The following rule applies when $\beta$ involves other bindings. It prescribes to decompose $\beta$ into simpler bindings, which can be told first.

\[
\frac{\beta}{\sigma} \xrightarrow{\gamma} \frac{\beta}{\sigma} \quad \text{if} \quad \gamma \in \text{sub}_\sigma(\beta)
\]

With the three binding rules, we can now completely explain how a tell operation works. Telling $\beta$ consists of two parts. If $\beta$ is basic, then the two basic binding rules explain everything. If $\beta$ is nonbasic, then the nonbasic binding rule is used to "peel off" basic bindings, until the tell is reduced to basic bindings only. The rule allows basic bindings to be peeled off in any order, so the implementation is free to choose an order that it can handle efficiently.

Determined variables

We say that a variable is determined if it is bound to an integer, a name, or a record. We say an equality determines a variable if it results in the variable becoming determined. We define the predicate $\text{det}(V)$ which is entailed by the store when the given variable $V$ is determined.

\[
\sigma \models \text{det}(V) \quad \text{iff} \quad \sigma \models V=z \quad \text{for some integer or name } z \\
\sigma \models V=f(l_1;V_1\ldots l_n;V_n) \quad \text{for some } f,l_i,V_i
\]

It is useful to introduce a statement that blocks until a variable is determined. We provide $\text{Wait}$ and $\text{WaitQuiet}$ for this. Their semantics is extremely simple: they can both reduce to $\text{skip}$ when their argument is determined.

\[
\frac{\{\text{Wait } V\}}{\sigma} \xrightarrow{\text{skip}} \sigma \models \text{det}(V)
\]

\[
\frac{\{\text{WaitQuiet } V\}}{\sigma} \xrightarrow{\text{skip}} \sigma \models \text{det}(V)
\]

The difference between the two appears when the variable’s value is computed by need, as explained in Section 15.2.10. $\text{Wait}$ requests the computation of the value, while $\text{WaitQuiet}$ does not.

Modification to support by-need synchronization

The first basic binding rule has to be changed slightly if the model supports by-need synchronization, as defined in Section 15.2.10. The tell operation should only be done after any by-need calculation on $V$. That is, a by-need calculation gets the first try at determining a variable. Section 15.2.10 defines the condition
blocking\textsubscript{e}(\beta), which is true if \(\beta\) would bind a variable that is blocked under a by-need synchronization. We can now modify the first binding rule:

\[
\begin{align*}
\beta & \not\models \text{skip} \\
\sigma & \models \sigma \land \beta \\
\text{if} \quad \text{sub}_\sigma(\beta) = \emptyset \quad \text{and} \quad \sigma \land \beta \quad \text{is consistent} \\
\quad \text{and} \quad \text{blocking}_\sigma(\beta) = \emptyset
\end{align*}
\]

### 15.2.7 Conditional statements (ask)

There is a single conditional statement that does an ask operation, namely the \textbf{if} statement. The reduction of an \textbf{if} statement depends on its condition variable:

\[
\begin{align*}
\text{if } V \text{ then } S_1 \text{ else } S_2 \text{ end} & \models S_1 \\
\sigma \land V = \text{true} & \models \sigma \land V = \text{true} \\
\text{if } V \text{ then } S_1 \text{ else } S_2 \text{ end} & \models S_2 \\
\sigma \land V = \text{false} & \models \sigma \land V = \text{false}
\end{align*}
\]

This statement synchronizes on the value of the variable \(V\). The first rule applies when the store entails \(V = \text{true}\) and the second rule applies when the store entails \(V = \text{false}\). The value of \(V\) can be determined by a boolean function, as in \(V = (V_1 < V_2)\) (Section 15.2.8). What happens if \(V\) is different from the atoms \textbf{true} and \textbf{false} is explained later.

The \textbf{if} statement only becomes reducible when the store entails sufficient information to decide whether \(V\) is \textbf{true} or \textbf{false}. If there is not enough information in the store, then neither rule can reduce. Such an execution is said to be \textit{dataflow}. Because variables are the basis for dataflow execution, they are called \textit{dataflow variables}.

### The \textbf{case} statement

The \textbf{case} statement is a linguistic abstraction for pattern matching that is built on top of \textbf{local} and \textbf{if}. Its semantics can be deduced from the semantics of \textbf{local} and \textbf{if}. Because pattern matching is such an interesting concept, though, we also give the semantics of \textbf{case} directly as reduction rules:

\[
\begin{align*}
\text{case } V \text{ of } f(l_1:X_1 \ldots l_n:X_n) & \text{ then } S_1 \text{ else } S_2 \text{ end} & \models S_1 \left[V_1 \ldots V_n / X_1 \ldots X_n \right]
\end{align*}
\]

\[
\sigma \land V = f(l_1:V_1 \ldots l_n:V_n) \quad \sigma \land V = f(l_1:V_1 \ldots l_n:V_n)
\]

\[
\text{case } V \text{ of } f(l_1:X_1 \ldots l_n:X_n) & \text{ then } S_1 \text{ else } S_2 \text{ end} & \models S_2
\]

\[
\sigma \quad \sigma \quad \text{if } \sigma \models \exists V_1 \ldots V_n : V = f(l_1:V_1 \ldots l_n:V_n)
\]

The semantics of pattern matching uses entailment. We say that \(V\) \textit{matches} the pattern \(f(l_1:X_1 \ldots l_n:X_n)\) if there exist \(V_1, \ldots, V_n\) such that the store entails
If the match is successful, then the case statement reduces to $S_1$ where the identifiers $X_i$ are replaced by the corresponding $V_i$. This implies that the lexical scope of the $X_i$ covers the whole statement $S_1$. Otherwise, if the match fails, the case reduces to $S_2$. If there is not enough information to decide one way or another, then neither rule can reduce. This is the dataflow behavior of case.

15.2.8 Procedural abstraction

A procedure is created by the execution of a proc statement. This puts a closure $\lambda X_1 \cdots X_n.S$ in a compartment of the store called the procedure store. When the procedure is invoked, its closure is copied from the procedure store and its argument identifiers $X_i$ are substituted by its effective arguments. The result must of course contain no free occurrence of any identifier. An informal definition of procedure creation and invocation is given in Section 4.4.5.

The procedure is associated to a variable. We do this by giving the procedure a name, i.e., a unique, unforgeable constant. Names are so useful that they are also provided directly to the programmer; see Section 15.4.1. We pair the name $\xi$ with the closure, giving $\xi: \lambda X_1 \cdots X_n.S$ which is added to the store. A variable that refers to the procedure is bound to $\xi$. The procedure store consists of pairs name:closure. They define a mapping from names to closures.

\[
\begin{align*}
\text{proc } \{V_p \ X_1 \cdots X_n\} \ S \ \text{end} & \quad \Rightarrow \quad V_p = \xi \\
\sigma & \quad \Rightarrow \quad \xi \ \text{fresh name}
\end{align*}
\]

\[
\begin{align*}
\{V_p \ V_1 \cdots V_n\} & \quad \Rightarrow \quad S[V_1 \cdots V_n/X_1 \cdots X_n] \\
\sigma \land V_p = \xi & \quad \Rightarrow \quad \sigma \land V_p = \xi \land \xi: \lambda X_1 \cdots X_n.S
\end{align*}
\]

It is interesting to see the dataflow behavior of the procedure call. The invocation statement $\{V_p \ V_1 \cdots V_n\}$ synchronizes on the value of $V_p$. So the procedure can be created in a concurrent thread, provided that no other thread binds $V_p$ to a value.

Built-in procedures

A practical implementation of the concurrent stateful model has to define built-in procedures, such as arithmetic operators, comparisons, etc. For instance, the sum operation can be written as $V = V_1 + V_2$, which is actually a shorthand for the procedure call $\{\text{Add} \ V_1 \ V_2 \ V\}$ that is defined by:

\[
\begin{align*}
\{\text{Add} \ V_1 \ V_2 \ V\} & \quad \Rightarrow \quad V = z \\
\sigma \land V_1 = z_1 & \quad \Rightarrow \quad \sigma \land V_1 = z_1 \land V_2 = z_2 \quad \text{if } z = z_1 + z_2
\end{align*}
\]

Another built-in procedure is the equality test, which is often used in conjunction with an if statement. Equality test is the general form of the ask operation.
defined in Section 15.2.1. It is usually written as a boolean function in infix notation, as in \( V = (V_1 = V_2) \) which is shorthand for \{Equal \( V_1 \) \( V_2 \) \( V \}\).

\[
\frac{\{\text{Equal } V_1 \ V_2 \ V\}}{\sigma} \ \overset{\downarrow}{\text{true}} \quad \text{if } \sigma \models V_1 = V_2 \\
\frac{\{\text{Equal } V_1 \ V_2 \ V\}}{\sigma} \ \overset{\downarrow}{\text{false}} \quad \text{if } \sigma \models V_1 \neq V_2
\]

An algorithm to implement the Equal operation is given in Section 15. Notice that both Add and Equal have dataflow behavior.

Examples of higher-orderliness

You can always pass a procedure as an argument to another procedure. You only have to pass a variable equal to the procedure’s name. Procedures are said to be first-class citizens.

As a first example, the procedure ForAll below is a way to implement loops in a pure declarative setting. It takes as arguments a list and a procedure, and applies the procedure to every element of the list. The empty list is the atom nil. The list with head \( x \) and tail \( x_r \) is the record \( x | x_r \), which is a shorthand for \( \{1: x, 2: x_r\} \).

\[
\text{proc } \{\text{ForAll } Xs \ P\} \\
\text{case } Xs \ \text{of nil then} \\
\quad \text{skip} \\
\text{[] } X | Xr \ \text{then} \\
\quad \{P \ X\} \\
\quad \{\text{ForAll } Xr \ P\} \\
\text{end} \\
\text{end}
\]

In the second example, the procedure MakeAdder uses its second argument \( P \) as an output for a procedure:

\[
\text{local } \text{MakeAdder } P3 \ X \ \text{in} \\
\text{proc } \{\text{MakeAdder } N \ P\} \\
\quad \text{proc } \{P \ A \ B\} \\
\quad \quad B = A + N \\
\quad \text{end} \\
\quad \text{end} \\
\quad \{\text{MakeAdder } 3 \ P3\} \\
\quad \{P3 \ 5 \ X\} \\
\text{end}
\]

15.2.9 Encapsulated state

All the statements introduced up to now define a pure declarative language that calculates with a monotonic store. Information can be added to this store but
not changed. It is possible to define another kind of store in which information can be changed. We call this a nonmonotonic store or mutable store. Adding this store to the model greatly increases its expressive power (allowing object-oriented programming, for instance). The nonmonotonic store contains stateful entities called cells. We call them stateful since they have contents that can be changed.

A cell is named in the same way as a procedure: when the cell is created, a fresh name $\xi$ is associated with it. A pair $\xi:V$ is put in the mutable store, where the variable $V$ defines the current value of the cell. One changes a cell’s value to $W$ by replacing the pair $\xi:V$ in the mutable store by $\xi:W$. Cells need two primitive operations only, namely cell creation and exchange:

$$
\frac{\{\text{NewCell} V V_c\}}{\sigma} \quad \frac{V_c=\xi}{\sigma \land \xi:V} \quad \xi \text{ fresh name}
$$

$$
\frac{\{\text{Exchange} V_c V_{old} V_{new}\}}{\sigma \land V_c=\xi \land \xi:V} \quad \frac{V_{old}=V}{\sigma \land V_c=\xi \land \xi:V_{new}}
$$

Having just one operation to use cells, Exchange, is rather minimal. It is often convenient to assume that two other operations exist, namely $\{\text{Assign} V_c V_{new}\}$ and $\{\text{Access} V_c V_{old}\}$.

It is interesting to see the dataflow behavior of Exchange. It blocks until its first argument references a cell. It never blocks on the second or third arguments. This allows it to manipulate the cell’s contents even before they are determined.

**Example of a stream**

Using cells and dataflow variables together permits some remarkable programming techniques. We give a small example that uses a stream. Assume that the cell $C$ contains the tail of a stream. Then the following statement adds the atom one to the stream:

```
local X Old New in
  {Exchange C Old New}
  X=one
  Old=X|New
end
```

The three instructions inside this `local` statement can be executed in any order and the final result is exactly the same. What’s more, several threads can independently add elements to the stream by each executing the same `local` statement. No matter how the executions are interleaved, the final result is exactly the same.

**15.2.10 By-need synchronization**

By-need synchronization is the basic operation used to define both lazy execution and read-only variables. We give the semantics of $\text{ByNeed}$, the basic primitive
that provides by-need synchronization. This semantics is defined with three reduction rules. In addition, the basic binding rule of Section 15.2.6 is modified, as shown there. After giving the semantics, we use \textit{ByNeed} to define lazy execution and read-only variables.

\section*{Basic concepts}

To model by-need synchronization in the store, we introduce the predicates \textit{sync} and \textit{need}:

\textit{sync}(X) states that \(X\) is read-only. Each thread that attempts to read or write \(X\) will suspend. The thread can continue as soon as \textit{sync}(X) has been removed from the store.

\textit{need}(X, P) states that the value of \(X\) is computed when it is needed by the one-argument procedure \(P\). We call this a by-need trigger.

When a thread needs \(X\), then the following three-step protocol is done:

- \(X\) is made read-only. The thread suspends, along with any other threads needing \(X\).
- \(P\) is evaluated and its result is bound to \(X\). During the evaluation of \(P\), \(X\) remains read-only.
- \(X\) is made no longer read-only. All threads needing \(X\) can continue.

The semantics are defined so that \(V\) cannot be read-only and determined at the same time. This gives the following theorem:

\textbf{Theorem 15.2.1 (Read-only invariant).} Let \(\sigma\) be a store obtained from \texttt{true} and the reduction of rules. Then, for any variable \(V\),

\[ \sigma \not\models \text{sync}(V) \land \text{det}(V). \]

\section*{Rules for by-need synchronization}

By-need synchronization is defined by three reduction rules. The first rule defines the operation \textit{ByNeed}, which creates a read-only variable together with a \textit{need} trigger.

\[ \frac{\text{\{ByNeed } V_p \text{ } V\}}{\sigma} \quad \frac{V=W}{\sigma \land \text{sync}(W) \land \text{need}(W, V_p)} \quad W \text{ fresh variable} \]

The second rule defines the two-argument procedure \textit{Synchronize}, which synchronizes its first (read-only) argument with its second argument. This primitive is not accessible to the programmer: it is intended for semantic use only. This rule atomically does two things: it binds the read-only variable \(W\) and it removes the
read-only predicate \( \text{sync}(W) \). This transfers the result of the by-need calculation to \( W \) in a way consistent with the read-only invariant.

\[
\begin{array}{c}
\{ \text{Synchronize } W \, V \} \\
\sigma \land \text{sync}(W) \\
\text{skip} \\
\sigma \land W=V
\end{array}
\]

The third rule activates the by-need trigger. Its effect is to create a thread that computes the procedure \( V_p \) and synchronizes the by-need variable with its result.

\[
S \\
\sigma \land \text{need}(W, V_p) \\
\text{Synchronize } WX \\
S' \in \text{requested}_\sigma(S)
\]

where \( S' \) is the statement:

\[
\text{thread} \\
\text{local } X \text{ in} \\
\{ V_p \, X \} \{ \text{Synchronize } W \, X \} \\
\text{end} \\
\text{end}
\]

The set \( \text{requested}_\sigma(S) \) is the set of requested variables of statement \( S \) in the context of store \( \sigma \). We give its precise definition below. We first define the concept of a “blocked variable”.

**Blocked variables**

A *blocked variable* is a variable that cannot be bound because doing so would violate the read-only invariant. The basic binding rule of Section 15.2.6 is modified so that it cannot bind a blocked variable. We now give a precise definition of the set of blocked variables. Given a store \( \sigma \), only the reduction of a basic binding \( \beta \) can violate the invariant. Therefore we must not reduce \( \beta \) if one of the following situations occurs with its variable \( V \):

- \( V \) is read-only and bound to a determined value, i.e.,
  \[
  \sigma \land \beta \models \text{sync}(V) \land \text{det}(V);
  \]
  \[
  \text{(15.1)}
  \]

- \( V \) is read-only and bound to another read-only variable, i.e.,
  \[
  \text{for some } W, \quad \sigma \land \beta \models V=W \land \text{sync}(V) \land \text{sync}(W) \text{ and } \sigma \not\models V=W.
  \]
  \[
  \text{(15.2)}
  \]

We define the set of variables blocking a binding \( \beta \) as:

\[
\text{blocking}_\sigma(\beta) = \left\{ V \mid \text{(15.1) or (15.2) holds} \right\}.
\]
Requested variables

We define $\text{requested}_\sigma(S)$ as follows. We first give a definition for $\text{WaitQuiet}$, $\text{Wait}$, and the tell operation. The statement $\text{WaitQuiet}$ waits for determinacy, but does not request the value of its argument. The statement $\text{Wait}$ also waits, but it does request the value of the variable. Note that the whole equivalence class of the variable is requested. Finally, a basic binding requests the variables on which it blocks.

\[
\text{requested}_\sigma(\{\text{WaitQuiet } X\}) = \emptyset
\]  (15.3)

\[
\text{requested}_\sigma(\{\text{Wait } X\}) = \{ V \mid \sigma \models V=X \}
\]  (15.4)

\[
\text{requested}_\sigma(\beta) = \begin{cases} 
\text{blocking}_\sigma(\beta) & \text{if } \text{sub}_\sigma(\beta) = \emptyset \\
\emptyset & \text{otherwise}
\end{cases}
\]  (15.5)

The following statements request variables. We define them by referring to a $\text{Wait}$ statement:

\[
\text{requested}_\sigma(\text{if } X \text{ then } \ldots) = \text{requested}_\sigma(\{\text{Wait } X\})
\]  (15.6)

\[
\text{requested}_\sigma(\text{case } X \text{ of } \ldots) = \text{requested}_\sigma(\{\text{Wait } X\})
\]  (15.7)

\[
\text{requested}_\sigma(\{P \ X \ldots X_n\}) = \text{requested}_\sigma(\{\text{Wait } P\})
\]  (15.8)

\[
\text{requested}_\sigma(\{\text{Exchange } C \ X \ Y\}) = \text{requested}_\sigma(\{\text{Wait } C\})
\]  (15.9)

\[
\text{requested}_\sigma(\{\text{ByNeed } P \ X\}) = \text{requested}_\sigma(\{\text{Wait } P\})
\]  (15.10)

Finally, for all remaining statements $S$ in the kernel language, we have

\[
\text{requested}_\sigma(S) = \emptyset.
\]

Example

Consider the following program. We show that our semantics gives the expected behavior. The store resulting from the execution of a statement is given on the right of that statement.

1. $\text{local } P \ U \ V \ \text{in}$
2. $\text{proc } \{P \ X\} \ X=U \ \text{end}$ $\sigma$
3. $\{\text{ByNeed } P \ V\}$ $\sigma \land \text{sync}(V) \land \text{need}(V, P)$
   statement on line 4 requests $V$,
   a new thread computes its value: $\sigma \land \text{sync}(V)$
   \begin{verbatim}
   local X in
   \{P \ X\} $\sigma \land \text{sync}(V) \land X=U$
   \{Synchronize V X\} $\sigma \land V=X \land X=U$
   end
   \end{verbatim}
4. $V=\text{unit}$ $\sigma \land V=X \land X=U \land V=\text{unit}$
5. $\text{end}$
Lazy functions

A lazy function is implemented by attaching a by-need trigger to the variable that will contain the function result. The “lazy” annotation is a syntactic short-cut for this technique. Any lazy function, e.g.:

```hl
fun lazy {F X1 ... Xn} ⟨Expr⟩ end
```

behaves as if it were defined by:

```hl
fun {F X1 ... Xn}
   {ByNeed fun {§} ⟨Expr⟩ end}
end
```

When written in full, this becomes:

```hl
proc {F X1 ... Xn X} local P in
   proc {P X} X=⟨Expr⟩ end
   {ByNeed P X}
end
end
```

Read-only variables

A read-only variable is always linked to another variable. The read-only variable blocks any attempts to bind it until its variable gets determined. The variable’s value then becomes the read-only variable’s value.

Read-only variables can be defined easily with a by-need computation. The procedure `ReadOnlyView` below is a synonym for the function `!!`. It creates a by-need variable `F`. When `F` is requested, the computation waits until the variable `X` is determined, and then binds `F` to `X`.

```hl
fun {ReadOnlyView X} {ByNeed fun {§} {Wait X} X end}
end
```

When written in full, this becomes:

```hl
proc {ReadOnlyView X F} local P in
   proc {P Z} {Wait X} Z=X end
   {ByNeed P F}
end
end
```

This definition has an interesting property: when the read-only variable is requested, its internal variable is requested too. This is due to the `Wait` statement, which itself triggers by-need calculations. We can define a variant that avoids this behavior. Its definition is similar to the first one, except that it uses `WaitQuiet` instead of `Wait`.}

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
fun {ReadOnlyViewQuiet X}
  {ByNeed fun {($) {WaitQuiet X} X end} end}

A property common to both definitions is that one knows nothing about a read-only variable until it gets determined. Indeed, each call to ReadOnlyView or ReadOnlyViewQuiet returns a new read-only view of the variable. Two read-only views of the same variable are not considered equal as long as the variable is not determined. The same happens with a read-only view of a read-only view.

local X F F1 F2 in
  {ReadOnlyView X F} % two views of the same variable
  F1=F F2
end

local X F FF F in
  {ReadOnlyView X F} % read-only view of a read-only view
  F=F F FF
end

An equivalent semantics for read-only variables

We can define a semantics for read-only variables that does not depend on by-need synchronization. We first introduce the predicate future:

\[ \text{future}(X, Y) \] states that \( Y \) is a read-only view of \( X \). Once \( X \) is determined, the predicate is removed from the store and replaced by \( X=Y \).

Two semantic rules are needed. A read-only view is created with the procedure Future, whose semantics is given by the first rule. This creates a new read-only variable for \( V_2 \) together with a future link.

\[
\frac{}{\{\text{Future} \ V_1 \ V_2\}} \quad \frac{}{\sigma \models \text{future}(V_1, W)} \quad W \text{ fresh variable}
\]

The second rule synchronizes the read-only variable on determinacy.

\[
\frac{}{\sigma \models \text{future}(V, W)} \quad \frac{}{\sigma \models \text{sync}(W)} \quad \frac{}{\sigma \models \text{det}(V)}
\]

These read-only variables behave like those created with ReadOnlyViewQuiet above: requesting the read-only view does not imply requesting its variable. In order to request the variable, the set \( \text{requested}_\sigma(S) \) must be augmented such that

\[ W \in \text{requested}_\sigma(S) \text{ and } \sigma \models \text{future}(V, W) \implies V \in \text{requested}_\sigma(S). \]
15.2.11 Exception handling

The exception mechanism is closely bound to sequential composition. Indeed, raising an exception modifies the sequence of statements in the thread where it has been thrown. It skips every statement inside the scope defined by the most enclosing \texttt{try/catch} block.

The following rule for the \texttt{try/catch} statement is a first attempt towards its semantics:

\[
\frac{\text{try } S_1 \text{ catch } X \text{ then } S_2 \text{ end \quad try } S'_1 \text{ catch } X \text{ then } S_2 \text{ end}}{\sigma \quad \sigma'} \quad \text{if } S \sigma \vdash S' \sigma' \quad \text{if } S \sigma \vdash S' \sigma'
\]

It defines the reduction of the nested statement. We then just need to add two rules for the cases where \( S_1 \) is \texttt{skip} and \texttt{raise V end}. But that definition is not complete: it does not handle the thread creation inside the \texttt{try/catch} block. And up to now, we found no elegant way to cope with it.

So we decide to “unfold” the \texttt{try/catch} statement, in order to match the reduction of the nested statement with the usual rules for sequence and thread creation. We say that the statement

\[
\text{try } S_1 \text{ catch } X \text{ then } S_2 \text{ end}
\]

unfolds to a sequence of two statements, the first one being \( S_1 \), and the second one a “\texttt{catch}” statement:

\[
S_1 \text{ (catch } X \text{ then } S_2 \text{ end)}
\]

The new \texttt{catch} statement is for semantic use only: it is a marker that stops a \texttt{raise} statement exactly at the place it must.

The unfolding technique works even when \texttt{try/catch} blocks are nested. For instance, the statement:

\[
\begin{align*}
\text{try} \\
\text{try } S_1 S_2 \\
\text{catch } X \text{ then } \ldots \text{ end} \\
S_3 S_4 \\
\text{catch } Y \text{ then } \ldots \text{ end} \\
S_5 S_6
\end{align*}
\]

when put in a thread unfolds to:

\[
\begin{align*}
\text{scope of outer} \\
\text{try/catch} \\
S_1 S_2 (\text{catch } X \text{ then } \ldots \text{ end}) S_3 S_4 (\text{catch } Y \text{ then } \ldots \text{ end}) S_5 S_6
\end{align*}
\]
The following two rules define the unfolding of a try/catch statement and the simplification of a catch statement when no exception is raised:

\[
\begin{align*}
\text{try } S_1 \text{ catch } X \text{ then } S_2 \text{ end} & \quad \Rightarrow \quad S_1 (\text{catch } X \text{ then } S_2 \text{ end}) \\
\text{catch } X \text{ then } S_2 \text{ end} & \quad \Rightarrow \quad \text{skip}
\end{align*}
\]

We now define the behavior of a raise statement. As we said earlier, it should skip every statement following it, except a catch statement. As the following statements reside in the current thread’s tail, we must use a “thread-level” reduction:

\[
\begin{align*}
(\text{raise } V \text{ end}) (S T) & \quad \Rightarrow \quad (\text{raise } V \text{ end}) T \quad S \neq \text{catch...end} \\
(\text{raise } V \text{ end}) (S T) & \quad \Rightarrow \quad S_2[V/X] T \quad S \equiv \text{catch } X \text{ then } S_2 \text{ end}
\end{align*}
\]

What happens if the thread’s statement sequence is done (i.e., there is only the termination symbol)? The behavior in this case is implementation-dependent. The implementation should have a rule like this one:

\[
\begin{align*}
(\text{raise } V \text{ end}) \cdot & \quad \Rightarrow \quad \ldots \\
& \quad \Rightarrow \quad \ldots
\end{align*}
\]

The Mozart system has a rule that halts the process with an error message (“un-caught exception”).

Examples

The following two examples show the expressiveness of the language:

```
try <S>
  catch X then
    case X of e1 then <S1>
    else case X of e2 then <S2>
    else case X of e3 then <S3>
    else raise X end
  end end end
end
```

```
try
  try
    raise e1 end
  catch X then
    catch Y then
      {Show Y}
    end
end
```

The example on the left shows exception catching like many modern languages (such as Java) do: the catch statement is combined with a case statement. But there are two different concepts involved: exceptions and pattern matching. It is interesting to consider them separately. The example on the right shows two nested try statements.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Sources of exceptions

Several statements can raise an exception when their reduction is definitively impossible. The first source of exceptions is imposing an equality that would lead to an inconsistent store:

\[
\frac{\beta \mid \text{raise failure(...) end}}{\sigma \mid \sigma} \quad \text{if } \sigma \land \beta \text{ is inconsistent}
\]

where the \( \ldots \) stands for some debugging information that is not specified here.\(^3\)

The second source of exceptions in the kernel language is type inconsistencies. They are defined with the following rules. An exception is raised when the condition variable of an \textbf{if} statement is not a boolean:

\[
\frac{\text{if } V \text{ then } S_1 \text{ else } S_2 \text{ end} \mid \text{raise error(...) end}}{\sigma \mid \sigma} \quad \text{if } \sigma \models V \notin \{\text{true, false}\}
\]

A procedure invocation fails if it is performed on something that is either not a procedure, or a procedure with a wrong number of arguments:

\[
\frac{\{V_p \, V_1 \cdots V_n\} \mid \text{raise error(...) end}}{\sigma \mid \sigma} \quad \text{if } \sigma \models (V_p \text{ is not a procedure})
\]

\[
\frac{\{V_p \, V_1 \cdots V_n\} \mid \text{raise error(...) end}}{\sigma \mid \sigma} \quad \text{if } \sigma \models V_p = \xi \land \xi : \lambda X_1 \cdots X_m.S \text{ and } m \neq n
\]

You cannot use \text{Exchange} on something that is not a cell:

\[
\frac{\{\text{Exchange } V_c \, V_{\text{old}} \, V_{\text{new}}\} \mid \text{raise error(...) end}}{\sigma \mid \sigma} \quad \text{if } \sigma \models (V_c \text{ is not a cell})
\]

With by-need synchronization, \text{BindUnblock} cannot add an inconsistent binding, but it will still remove the binding blocker:

\[
\frac{\{\text{BindUnblock } V \, V'\} \mid \text{raise error(...) end}}{\sigma \land \text{block}(V) \mid \sigma} \quad \text{if } \sigma \land V = V' \text{ is inconsistent}
\]

15.2.12 Variable substitution

This section defines the substitution of an identifier by a variable in a statement. The notation \( S[V/X] \) stands for the substitution of \( X \) by \( V \) in statement \( S \). For convenience, we first define substitution for variables and identifiers:

\[
X[V/X] \equiv V \\
Y[V/X] \equiv Y \\
V'[V/X] \equiv V'
\]

\(^3\) This \text{raise} statement is shorthand for \textbf{local} \( X \) \textbf{in} \( X = \text{failure(...) raise } X \) \textbf{end} \textbf{end}.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
The following substitutions do not involve lexical scoping, so their definition is easy. We assume that $E$ denotes either a variable or an identifier.

\[
\begin{align*}
\text{(skip)}[V/X] &\equiv \text{skip} \\
(S_1 \ S_2)[V/X] &\equiv S_1[V/X] \ S_2[V/X] \\
\text{(thread S end)}[V/X] &\equiv \text{thread S[V/X] end} \\
(E_1=E_2)[V/X] &\equiv E_1[V/X] = E_2[V/X] \\
\text{(if E then S_1 else S_2 end)}[V/X] &\equiv \text{if E[V/X] then S_1[V/X] else S_2[V/X] end} \\
\{E_1 \ldots E_n\}[V/X] &\equiv \{E[V/X] \ E_1[V/X] \ldots E_n[V/X]\} \\
\{\text{NewCell} \ E_1 \ E_2\}[V/X] &\equiv \{\text{NewCell} \ E_1[V/X] \ E_2[V/X]\} \\
\{\text{Exchange} \ E_1 \ E_2 \ E_3\}[V/X] &\equiv \{\text{Exchange} \ E_1[V/X] \ E_2[V/X] \ E_3[V/X]\} \\
\{\text{ByNeed} \ E_1 \ E_2\}[V/X] &\equiv \{\text{ByNeed} \ E_1[V/X] \ E_2[V/X]\} \\
\text{(raise E end)}[V/X] &\equiv \text{raise E[V/X] end}
\end{align*}
\]

We assume that the identifiers \text{NewCell} and \text{Exchange} never change their meaning. The remaining substitutions deal with lexical scoping. We first define the case where the identifier substituted falls into the lexical scope of a statement:

\[
\begin{align*}
\text{(local X_1 \ldots X_n in S end)}[V/X_i] &\equiv \text{local X_1 \ldots X_n in S end} \\
\text{(case E of f(l_1:X_1 \ldots l_n:X_n) then S_1 else S_2 end)}[V/X_i] &\equiv \text{case E[V/X_i] of f(l_1:X_1 \ldots l_n:X_n) then S_1 else S_2[V/X_i] end} \\
\text{(proc \{E X_1 \ldots X_n\} S end)}[V/X_i] &\equiv \text{proc \{E[V/X_i] X_1 \ldots X_n\} S end} \\
\text{(try S_1 catch X then S_2 end)}[V/X_i] &\equiv \text{try S_1[V/X_i] catch X then S_2 end}
\end{align*}
\]

Finally, we define the case when the identifier is not subject to the lexical scoping:

\[
\begin{align*}
\text{(local X_1 \ldots X_n in S end)}[V/X] &\equiv \text{local X_1 \ldots X_n in S[V/X] end} \\
\text{(case E of f(l_1:X_1 \ldots l_n:X_n) then S_1 else S_2 end)}[V/X] &\equiv \text{case E[V/X] of f(l_1:X_1 \ldots l_n:X_n) then S_1[V/X] else S_2[V/X] end} \\
\text{(proc \{E X_1 \ldots X_n\} S end)}[V/X] &\equiv \text{proc \{E[V/X] X_1 \ldots X_n\} S[V/X] end} \\
\text{(try S_1 catch Y then S_2 end)}[V/X] &\equiv \text{try S_1[V/X] catch Y then S_2[V/X] end}
\end{align*}
\]

### 15.3 Eight computation models

The previous section gives the semantics of the concurrent stateful model, which is the most expressive general-purpose model of the book. This semantics is factorized so that the semantics of most of the earlier models are subsets of it. To make these subsets easy to understand, we distinguish three properties: concurrency, state, and laziness. Each of these properties is defined by a part of the semantics:

- **Concurrency** is introduced by the \text{thread} statement. Having concurrency implies that there is a multiset of tasks.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
<table>
<thead>
<tr>
<th>C</th>
<th>L</th>
<th>S</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Declarative model (Chapters 4–5).</td>
</tr>
<tr>
<td>×</td>
<td></td>
<td></td>
<td>Stateful model (Chapters 8–9, Scheme, ML, Prolog, Mercury, Pascal).</td>
</tr>
<tr>
<td>×</td>
<td></td>
<td></td>
<td>Lazy declarative model (Haskell).</td>
</tr>
<tr>
<td>×</td>
<td>×</td>
<td></td>
<td>Lazy stateful model.</td>
</tr>
<tr>
<td>×</td>
<td>×</td>
<td></td>
<td>Eager concurrent declarative model.</td>
</tr>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>Eager concurrent stateful model (Erlang, Java, FCP).</td>
</tr>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>Concurrent declarative model (Chapter 6).</td>
</tr>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>Concurrent stateful model (Chapter 10, Oz).</td>
</tr>
</tbody>
</table>

Table 15.1: Eight computation models

- **State** is introduced by the `NewCell` operation and handled by the `Exchange` operation. Having state implies that there is a mutable store.

- **Laziness** is introduced by the `ByNeed` operation. Having laziness implies that there is a trigger store.

Each of the three properties can be left out of the model by removing its statements. This gives *eight* useful models of varying degrees of expressiveness (!). Table 15.1 lists these eight models. All of these models are practical and most have been used in real programming languages. In this table, C means concurrency, L means laziness, and S means state. An × means the property is in the model, a blank means it is not.

In the concurrent stateful model of this book, the three properties are all *explicit*. That is, the programmer controls whether or not they are used by means of explicit commands. This is not true of all the languages mentioned. For example, laziness is implicit in Haskell and concurrency is implicit in FCP (Flat Concurrent Prolog).

Languages can be based on the same computation model and yet “feel” very differently to the programmer:

- Scheme, ML, Prolog, Mercury, and Pascal are all based on the stateful model. Pascal is a simple imperative language. Scheme and ML are “mostly-functional” languages. Prolog and Mercury are “mostly-logical” languages with predicates written in Horn clause syntax. Both Prolog and Mercury have global backtracking, which is not mentioned in the table. By “mostly” we mean that state is intended to be used in a limited way. Scheme and Prolog are dynamically typed. ML, Mercury, and Pascal are statically typed.

- Erlang, Java, and FCP are all based on the eager concurrent stateful model. Erlang is based on active objects that are programmed in a functional model.
and communicate with asynchronous message passing. Java is based on passive objects referenced by threads. FCP is based on the process model of logic programming, with predicates in Horn clause syntax that communicate through shared streams.

This table does not give the semantics of the relational model of Chapter 7 (the declarative model with don’t know nondeterminism). We delay this one until we give the semantics of constraint programming, in Section 15.7. The logical semantics of Prolog and Mercury are closely related to the relational model.

15.4 Kernel extensions

This section defines several other important operations that can be added to the kernel language.

- **Names.** Names are used to identify procedures and cells. They are also made available to the programmer directly. An informal definition of names is given in Section B.2.

- **Chunks.** An informal definition of chunks is given in Section B.4.

- **Ports.** An informal definition of ports is given in Section 10.3.1.

Names and chunks are especially useful to define ADTs.

15.4.1 Names

Names are atomic values used in the semantics of procedures and cells (see Sections 15.2.8 and 15.2.9). They are used to represent procedures and cells in the store and to identify procedures and cells uniquely. But their usefulness goes much beyond this semantic role. Names can be used explicitly by the program. They behave as capabilities, because they do not have a concrete representation and cannot be forged. A thread cannot guess a name value: a thread can know a name only if it references it via one of its variables.

There are just two operations on a name: creation and equality test. A name is equal only to itself. New names can be created at will. We use the metavariable $\xi$ to denote a name, and we extend the equality statement for names:

$$\beta ::= \cdots \mid V=\xi.$$  

This statement cannot be typed explicitly by the programmer, but only created indirectly through the `NewName` operation, which creates a new name:

$$\frac{}{\sigma \llbracket \text{NewName } V \rrbracket \sigma} V=\xi \quad \xi \text{ fresh name}$$

The `NewName` operation is not needed for the semantics of procedures and cells. It is not part of the basic concurrent stateful model.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
15.4.2 Chunks

15.4.3 Ports

The port entity can either be defined as a new kernel operation or as a programming idiom. We define it as a kernel operation because this simplifies the definition of its distributed semantics.

15.5 Common programming idioms

This section defines the most common programming idioms of the concurrent stateful model.

15.5.1 Classes and objects

15.5.2 Reentrant locks

15.5.3 Loop constructs

15.6 The distributed computation model

15.7 The constraint-based computation model

15.7.1 The store as constraint store

15.7.2 Computation spaces

15.7.3 The relational programming model

15.8 Exercises

1. Let us investigate the case statement, whose semantics is defined in Section 15.2.7.

   - Show how the semantic rules of case can be derived from the rules for local and if.

   - In the first rule for the case, we could have explicitly introduced variables for the $X_i$ by:

     \[
     \text{case } V \text{ of } f(l_1:X_1 \ldots l_n:X_n) \quad \begin{array}{ll}
     \text{then } S_1 \text{ else } S_2 \text{ end} \\
     \sigma \land V = f(l_1:V_1 \ldots l_n:V_n)
     \end{array}
     \quad \begin{array}{ll}
     \text{local } X_1 \ldots X_n \text{ in} \\
     \quad \begin{array}{ll}
     X_1 = V_1 \ldots X_n = V_n \quad S_1 \text{ end} \\
     \sigma \land V = f(l_1:V_1 \ldots l_n:V_n)
     \end{array}
     \end{array}
     \]

   Do the rules lead to the same possible executions? What are the differences (if any)?
- It is possible to write an `if` statement in terms of a `case` statement. How? This implies that `case` could have been put in the kernel language instead of `if`, and `if` could have been defined as a linguistic abstraction.

2. Write the consecutive computation steps (rule reductions) for the execution of the `ForAll` and `MakeAdder` definitions in Section 15.2.8.

3. Section 15.2.9 mentions that two more cell operations can be added for programming convenience, namely `{Access Vc Vold}` to read the content and `{Assign Vc Vnew}` to update the content. Define the semantics of these two operations.

4. Section 15.2.9 gives an example of a `local` statement that adds an element to a stream. Prove that executing two of these statements in different threads always gives exactly the same final result as if they were executed sequentially in the same thread.

5. Define a stream datatype that does not use dataflow variables. That is, it is a list in which each tail is a cell whose content points to the rest of the list. The last cell contains a marker saying that the stream is not yet complete, e.g., the atom `incomplete`. (This is not the same as the atom `nil` which means that the stream is complete.) There is a global cell `c` whose contents is always the last cell in the stream. Write an operation that adds an element to the stream and that works in a concurrent setting. **Hint:** assume that there exists a statement `lock S end` such that only one thread at a time can be executing `S`; all others block if needed to make this true. Can you do it without using a `lock` statement? Compare your solution to that of the previous exercise. Which is simpler?

6. Section 15.2.10 defines read-only variables in terms of by-need synchronization:

```haskell
fun {ReadOnlyView X}
  {ByNeed fun {} {Wait X} X end}
end
F={ReadOnlyView X}
```

The read-only variable `F` is “lazy” in the sense that the by-need trigger does not disappear when `x` is bound, but also waits until `F` is needed. For this exercise, modify the definition to become “eager”, i.e., so that the by-need trigger disappears as soon as `x` is bound. With the solution to this exercise, we have four variants of the ‘read-only’ concept, along two orthogonal axes:

- Whether or not requesting `F` will also request `x`.
- Whether or not `F` is lazy or eager.
7. Section 15.2.11 defines the `try/catch` statement:

```
try S1 catch X then S2 end
```

which executes $S_2$ if an exception is raised in $S_1$. Another, very useful statement relating to exceptions is the `try/finally` statement:

```
try S1 finally S2 end
```

which _always_ executes $S_2$, whether or not $S_1$ raises an exception. If $S_1$ raises an exception, then this exception is raised again after executing $S_2$. Define the `try/finally` statement in terms of the `try/catch` statement.
Chapter 16

Virtual Machines

- Give a virtual machine for the concurrent stateful model
- Explain something about spaces implementation
- Explain something about distribution implementation
Part V

Appendices
Appendix A

Development Environment

Interactive use:
  OPI, Browser, Panel, Compiler Panel
Batch use:
  compiler, linker, functors
Functors in the OPI
Appendix B

Basic Data Types

This appendix explains the most common basic data types in Oz together with some common operations. The types explained are numbers (including integers and floating point numbers), characters (which are represented as small integers), literals (constants of two types, either atoms or names), records, tuples, chunks (records with a limited set of operations), lists, strings (which are represented as lists of characters), and virtual strings (strings represented as tuples).

For each data type discussed in this appendix, there is a corresponding Base module in the Mozart system that defines all operations on the data type. This appendix gives some but not all of these operations. See the Mozart system documentation for complete information [27].

B.1 Numbers (integers, floats, and characters)

The following code fragment introduces four variables I, H, F and C. It binds I to an integer, H to an integer in hexadecimal notation, F to a float, and C to the character t in this order. It then displays I, H, F, and C:

```
declare I H F C in
I = ~5
H = 0xDadBeddedABadBadBabe
F = 5.5
C = & t
{Browse I} {Browse H} {Browse F} {Browse C}
```

Note that ~ (tilde) is the unary minus symbol. This displays the following:

```
~5
1033532870595452951444158
5.5
116
```

Oz supports binary, octal, decimal, and hexadecimal notation for integers, which can have any number of digits. An octal integer starts with a leading 0 (zero), followed by any number of digits from 0 to 7. A binary integer starts with a
(character) ::= (any integer in the range 0 ... 255)  
    | `'&` (charChar)  
    | `'&` (pseudoChar)  

(charChar) ::= (any inline character except \ and NUL)  

(pseudoChar) ::= (‘\’ followed by three octal digits)  
    | (‘\x’ or ‘\X’ followed by two hexadecimal digits)  
    | ‘\a’ | ‘\b’ | ‘\f’ | ‘\n’ | ‘\r’ | ‘\t’  
    | ‘\v’ | ‘\ ’ | ‘\ ’ | ‘\ ’ | ‘\ ’ | ‘\ ’  

Table B.1: Characters (character syntax)

leading 0b or 0B (zero followed by the letter b or B) followed by any number of binary digits, i.e., 0 or 1. A hexadecimal integer starts with a leading 0x or 0X (zero followed by the letter x or X). The hexadecimal digits from 10 to 15 are denoted by the letters a through f and A through F.

Floats are different from integers in that they approximate real numbers. Here are some examples of floats:

-3.14159265359  3.5E3  -12.0e-2  163.

Note that ~ (tilde) is the unary minus symbol for floats as well as integers. Floats are internally represented in double precision (64 bits) using the IEEE floating point standard. A float must be written with a decimal point and at least one digit before the decimal point. There may be zero or more digits after the decimal point. Floats can be scaled by powers of ten by appending the letter e or E followed by a decimal integer (which can be negative with a ‘-’).

Characters are a subtype of integers that range from 0 to 255. The standard ISO 8859-1 coding is used. This code extends the ASCII code to include the letters and accented letters of most languages whose alphabets are based on the Roman alphabet. Unicode is not currently used, but may be in the future. Unicode extends the ASCII code to include the letters and writing directions of most of the alphabets used in the world. There are five ways to write characters:

- A character can be written as an integer in the range 0, 1, ..., 255, in accord with the integer syntax given before.
- A character can be written as an ampersand & followed by a specific character representation. There are four such representations:
  - Any inline character except for \ (backslash) and the NUL character. Some examples are &t, & (note the space), and &+. Inline control characters are acceptable.
  - A backslash \ followed by three octal digits, e.g., &\215 is a character. The first digit should not be greater than 3.
B.1 Numbers (integers, floats, and characters)

Table B.2: Operations on numbers

- A backslash \ followed by the letter x or X, followed by two hexadecimal digits, e.g., \x3f is a character.
- A backslash \ followed by one of the following characters: a (= \007, bell), b (= \010, backspace), f (= \014, formfeed), n (= \012, newline), r (= \015, carriage return), t (= \011, horizontal tab), v (= \013, vertical tab), \ (= \134, backslash), ' (= \047, single quote), " (= \042, double quote), ` (= \140, backquote), and & (= \046, ampersand). For example, \\ is the backslash character.

Table B.1 summarizes these possibilities.

There is no automatic type conversion in Oz, so 5.0 = 5 will raise an exception. The next section explains the basic operations on numbers, including the primitive procedures for explicit type conversion. The complete set of operations for characters, integers, and floats are given in the Base modules Char, Float, and Int. Additional generic operations on all numbers are given in the Base module Number. See the Base module documentation for more information.

B.1.1 Operations on numbers

To express a calculation with numbers, we use two kinds of operations: binary operations, such as addition and subtraction, and function applications, such as type conversions. Table B.2 gives the syntax of these expressions. All numbers, i.e., both integers and floats, support addition, subtraction, and multiplication:

\[
\text{declare } I \ Pi \ Radius \ Circumference \text{ in} \\
I = 7 \times 11 \times 13 + 27 \times 37 \\
\Pi = 3.1415926536 \\
Radius = 10. \\
Circumference = 2.0 \times \Pi \times Radius
\]

Integer arithmetic is to arbitrary precision. Float arithmetic is of a fixed precision given in Appendix F. Integers support integer division (\text{div} symbol) and modulo (\text{mod} symbol). Floats support floating division (/ symbol). Integer division truncates the fractional part. Integer division and modulo satisfy the following identity:

\[
A = B \times (A \text{ div } B) + (A \text{ mod } B)
\]

There are several operations to convert between floats and integers.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{IsChar C}</code></td>
<td>Return boolean saying whether C is a character</td>
</tr>
<tr>
<td><code>{Char.toAtom C}</code></td>
<td>Return atom corresponding to C</td>
</tr>
<tr>
<td><code>{Char.toLower C}</code></td>
<td>Return lowercase letter corresponding to C</td>
</tr>
<tr>
<td><code>{Char.toUpper C}</code></td>
<td>Return uppercase letter corresponding to C</td>
</tr>
</tbody>
</table>

Table B.3: Some character operations

\[
\text{<expression>} ::= \text{unit} \; | \; \text{true} \; | \; \text{false} \; | \; \text{<atom>} \; | \; ...
\]

Table B.4: Literals

- There is one operation to convert from an integer to a float, namely \texttt{IntToFloat}. This operation finds the best float approximation to a given integer. Because integers are calculated with arbitrary precision, it is possible for an integer to be larger than a representable float. In that case, the float \texttt{inf} (infinity) is returned.

- There is one operation to convert from a float to an integer, namely \texttt{FloatToInt}. This operation follows the default rounding mode of the IEEE floating point standard, i.e., if there are two possibilities, then it picks the even integer. For example, \{\texttt{FloatToInt 2.5}\} and \{\texttt{FloatToInt 1.5}\} both give the integer 2. This eliminates the bias that would result by always rounding half integers upwards.

- There are three operations to convert a float into a float that has zero fractional part: \texttt{Floor}, \texttt{Ceil} (ceiling), and \texttt{Round}.
  
  - \texttt{Floor} rounds towards negative infinity, e.g., \{\texttt{Floor \_\_3.5}\} gives \_\_4.0 and \{\texttt{Floor 4.6}\} gives 4.0.
  
  - \texttt{Ceil} rounds towards positive infinity, e.g., \{\texttt{Ceil \_\_3.5}\} gives \_\_3.0 and \{\texttt{Ceil 4.6}\} gives 5.0.
  
  - \texttt{Round} rounds towards nearest even, e.g., \{\texttt{Round \_\_4.6}\} gives \_\_5.0. \texttt{Round} is identical to \texttt{FloatToInt} except that it returns a float, i.e., \{\texttt{Round X}\} = \{\texttt{IntToFloat FloatToInt X}\}.

B.1.2 Operations on characters

All integer operations also work for characters. There are a few additional operations that work only on characters. Table B.3 lists some of them. The Base module \texttt{Char} gives them all.
B.2 Literals (atoms and names)

Atomic types are types whose members have no internal structure. The previous section has given one kind of atomic type, namely numbers. In addition to numbers, literals are a second kind of atomic type (see Table B.4 and Table B.5). Literals can be either atoms or names. An atom is a value whose identity is determined by a sequence of printable characters. An atom can be written in two ways. First, as a sequence of alphanumeric characters starting with a lower case letter. This sequence may not be a keyword of the language. Second, by arbitrary printable characters enclosed in single quotes. Here are some valid atoms:

```
a foo
'\Oz 3.0'
'Hello World'
'if'
'\n,\n'
```

There is no confusion between the keyword `if` and the atom `'if'` because of the quotes. The atom `'\n,\n'` consists of four characters. Atoms are ordered lexicographically, based on the underlying ISO 8859-1 encoding for single characters.

Names are a second kind of literal. A name is a unique atomic value that cannot be forged or printed. Unlike numbers or atoms, names are truly atomic, in the original sense of the word: they cannot be decomposed at all. Names have just two operations defined on them: creation and equality comparison. The only way to create a name is by calling the function `{NewName}`, which returns a new name that is guaranteed to be unique. Note that Table B.4 has no representation for names. The only way to reference a name is through a variable that is bound to the name. As Chapter 5 explains, names play an important role for secure encapsulation in ADTs.

There are three special names that have keywords reserved to them. The keywords are `unit`, `true`, and `false`. The names `true` and `false` are used to denote boolean true and false values. The name `unit` is often used as a synchronization token in concurrent programs. Here are some examples:

```
local X Y B in
  X = foo
```

1But like physical atoms, atomic values can sometimes be decomposed if the right tools are used, e.g., numbers have a binary representation as a sequence of zeroes and ones and atoms have a print representation as a sequence of characters.

Table B.5: Atoms (character syntax)
### Operation Description

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{IsAtom A}</td>
<td>Return boolean saying whether A is an atom</td>
</tr>
<tr>
<td>{AtomToString A}</td>
<td>Return string corresponding to atom A</td>
</tr>
<tr>
<td>{StringToAtom S}</td>
<td>Return atom corresponding to string S</td>
</tr>
</tbody>
</table>

Table B.6: Some atom operations

$$
\begin{array}{l}
\langle \text{expression} \rangle ::= \langle \text{label} \rangle \ (\ (\ \{ \ (\ \langle \text{feature} \rangle \ : \ : \ \} \ (\ \langle \text{expression} \rangle \ ) \ ) \ ) \ ) \ | \ ...
\\
\langle \text{label} \rangle ::= \text{unit} | \text{true} | \text{false} | \langle \text{variable} \rangle | \langle \text{atom} \rangle
\\
\langle \text{feature} \rangle ::= \text{unit} | \text{true} | \text{false} | \langle \text{variable} \rangle | \langle \text{atom} \rangle | \langle \text{integer} \rangle
\\
\langle \text{binaryOp} \rangle ::= \cdot | \cdot | \langle \text{consBinOp} \rangle | ...
\\
\langle \text{consBinOp} \rangle ::= \# | ...
\end{array}
$$

Table B.7: Records and tuples

```plaintext
\{\text{Name Y}\}
\ B = \text{true}\n\{\text{Browse [X Y B]}\}
end
```

#### B.2.1 Operations on atoms

Table B.6 gives the operations in the Base module `Atom` and some of the operations relating to atoms in the Base module `String`.

#### B.3 Records and tuples

Records are data structures that allow to group together language references. Here is a record that groups four variables:

```
tree(key:I value:Y left:LT right:RT)
```

It has four components and the label `tree`. To avoid ambiguity, there should be no space between the label and the left parenthesis. Each component consists of an identifier, called `feature`, and a language reference. A feature can be either a literal or an integer. Table B.7 gives the syntax of records and tuples. The above record has four features, `key`, `value`, `left`, and `right`, that identify four language references, `I`, `Y`, `LT`, and `RT`.

It is allowed to omit features in the record syntax. In that case, the feature will be an integer starting from 1 for the first such component and incrementing by 1 for each successive component that does not have a feature. For example, the record `tree(key:I value:Y LT RT)` is identical to `tree(key:I value:Y 1:LT 2:RT)`.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
The order of labeled components does not matter; it can be changed without changing the record. We say that these components are unordered. The order of unlabeled components does matter; it determines how the features are numbered. It is as if there were two “worlds”: the ordered world and the unordered world. They have no effect on each other and can be interleaved in any way. All the following notations denote the same record:

\[
\begin{align*}
& \text{tree(key:I value:Y LT RT)} & \text{tree(value:Y key:I LT RT)} \\
& \text{tree(key:I LT value:Y RT)} & \text{tree(value:Y LT key:I RT)} \\
& \text{tree(key:I LT RT value:Y)} & \text{tree(value:Y LT RT key:I)} \\
& \text{tree(LT key:I value:Y RT)} & \text{tree(LT value:Y key:I RT)} \\
& \text{tree(LT key:I RT value:Y)} & \text{tree(LT value:Y RT key:I)} \\
& \text{tree(LT RT key:I value:Y)} & \text{tree(LT RT value:Y key:I)}
\end{align*}
\]

Two records are the same if the same set of components is present and the ordered components are in the same order.

It is an error if a feature occurs more than once. For example, the notations \text{tree(key:I key:J)} and \text{tree(1:I value:Y LT RT)} are both in error. The error is discovered when the record is constructed. This can be either at compile time or at run time. However, both \text{tree(3:I value:Y LT RT)} and \text{tree(4:I value:Y LT RT)} are correct since no feature occurs more than once. Integer features do not have to be consecutive.

Tuples

If the record has only consecutive integer features starting from 1, then we call it a tuple. All these features can be omitted. Consider this tuple:

\[
\text{tree(I Y LT RT)}
\]

It is exactly the same as the following tuple:

\[
\text{tree(1:I 2:Y 3:LT 4:RT)}
\]

Tuples whose label is ‘#’ have another notation using the # symbol as an “mixfix” operator (see Section C.4). This means that \text{a#b#c} is a tuple with three arguments, namely ‘#’(a b c). Be careful not to confuse it with the pair \text{a#(b#c)}, whose second argument is itself the pair b#c. The mixfix notation can only be used for tuples with at least two arguments. It is used for virtual strings (see Section B.7).

B.3.1 Operations on records

Table B.8 gives a few basic record operations. Many more operations exist in the Base module Record. This appendix shows only a few, namely those concerning extracting information from records and building new records. To select a field of a record component, we use the infix dot operator, e.g., \text{tree(key:I value:Y}
### Table B.8: Some record operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R.F</td>
<td>Return field of feature F from R</td>
</tr>
<tr>
<td>{HasFeature R F}</td>
<td>Return boolean saying whether feature F is in R</td>
</tr>
<tr>
<td>{IsRecord R}</td>
<td>Return boolean saying whether R is of record type</td>
</tr>
<tr>
<td>{MakeRecord L Fs}</td>
<td>Return record with label L and features Fs</td>
</tr>
<tr>
<td>{Label R}</td>
<td>Return the label of R</td>
</tr>
<tr>
<td>{Arity R}</td>
<td>Return the list of features (arity) of R</td>
</tr>
<tr>
<td>{Width R}</td>
<td>Return the number of features (width) of R</td>
</tr>
<tr>
<td>{AdjoinAt R F X}</td>
<td>Return R augmented with feature F and value X</td>
</tr>
<tr>
<td>{Adjoin R1 R2}</td>
<td>Return R1 augmented with all fields of R2</td>
</tr>
</tbody>
</table>

LT RT).value returns Y. To compare two records, we use the equality test operation. Two records are the same if they have the same set of features and the language references for each feature are the same.

The *arity* of a record is a list of the features of the record sorted lexicographically. To display the arity of a record we use the function Arity. Calling (Arity R) will execute as soon as R is bound to a record, and will return the arity of the record. Feeding the statement:

```plaintext
declare T W L R in
T=tree(key:a left:L right:R value:1)
W=tree(a L R 1)
{Browse {Arity T}}
{Browse {Arity W}}
```

will display:

```
[key left right value]
[1 2 3 4]
```

The function (AdjoinAt R1 F X) returns the record resulting from adjoining (i.e., adding) the field X to R1 at feature F. The record R1 is unchanged. If R1 already has the feature F, then the result is identical to R1 except for the field R1.F, whose value becomes X. Otherwise the feature F is added to R1. For example:

```plaintext
declare T W in
T=tree(key:a left:L right:R value:1)
W=tree(a L R 1)
{Browse {AdjoinAt T 1 b}}
{Browse {AdjoinAt W key b}}
```

will display:

```
tree(b key:a left:L right:R value:1)
tree(a L R 1 key:b)
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
### B.4 Chunks (limited records)

A *chunk* is a record type with a limited set of operations. There are only two basic operations: create a chunk from any record and extract information with the field selection operator ".":

```latex
\text{declare}
\begin{align*}
C &= \text{Chunk.new anyrecord}(a \ b \ c) \quad \% \text{Chunk creation} \\
F &= C.2 \quad \% \text{Chunk field selection}
\end{align*}
```

The *Label* and *Arity* operations are not defined and unification is not possible. Chunks are a basic way of “wrapping” information so that access to the information is restricted, i.e., not all computations can access the information. When used together with names, chunks are important for abstract data types (see Sections 5.9 and 8.4) and security (see Section 5.10).

### B.5 Lists

A *list* is either the atom `nil` representing the empty list or a tuple with infix operator `|` and two arguments which are respectively the head and the tail of the list. The two arguments have field numbered 1 and 2. The head can be any data type and the tail is a list. We call the tuple a *list pair*. Often it is called a *cons cell* because creating one in Lisp is done with an operation called *cons*. Lisp is the oldest list-processing language and pioneered many list concepts and their terminology. When the second argument is not necessarily a list, then it is often called a *dotted pair*, because Lisp writes it in infix with a dot operator. A list of the first three letters is represented as:

```latex
\text{List: `abc'}
```
Table B.11: Some list operations

\[
\begin{array}{|c|c|}
\hline
\text{Operation} & \text{Description} \\
\hline
\{\text{Append } L_1 \ L_2\} & \text{Return the concatenation of } L_1 \text{ and } L_2 \\
\{\text{Member } X \ L\} & \text{Return boolean saying whether } X \text{ is in } L \\
\{\text{Length } L\} & \text{Return the length of } L \\
\{\text{List.drop } L \ N\} & \text{Return } L \text{ minus the first } N \text{ elements, or nil if it is shorter} \\
\{\text{Sort } L \ F\} & \text{Return } L \text{ sorted according to boolean comparison function } F \\
\{\text{Map } L \ F\} & \text{Return the list obtained by applying } F \text{ to each element of } L \\
\{\text{ForAll } L \ P\} & \text{Apply the unary procedure } P \text{ to each element of } L \\
\{\text{Filter } L \ F\} & \text{Return the list of elements of } L \text{ for which } F \text{ gives } \text{true} \\
\{\text{FoldL } L \ F \ N\} & \text{Return the value obtained by inserting } F \text{ between all elements of } L \\
\hline
\end{array}
\]

This list can be written concisely as:

\[
[a \ b \ c \ nil]
\]

Table B.10 shows the syntax of these two ways of writing a list. The partial list containing elements \(a\) and \(b\) and whose tail is the variable \(X\) looks like:

\[
a|b|X
\]

One can also use the standard record notation for lists:

\[
(a | (b \ X))
\]

Circular lists are allowed. For example, the following is legal:

```
define X = a|b|X
(browse X)
```

By default, the browser displays the list without taking into account that part of the list may refer to another part. We call this “sharing”. In the list \(X\), after the first two elements \(a\) and \(b\), we find \(X\) again. By default, the browser ignores all sharing. It displays \(X\) as:

\[
a|b|a|b|a|b|a|b|\ldots
\]

To avoid infinite loops, the browser has an adjustable depth limit. The three commas \(\ldots\) represent the part of the list that is not displayed. Select Graph in the Representation entry of the browser’s Options menu and feed the fragment again. This will display the list as a graph (see Figure B.1):

\[
C_1 = a|b|C_1
\]

The browser introduces the new variable \(C_1\) to refer to another part of the list. See the browser manual for more information on what the browser can display.
B.6 Strings

Lists whose elements are character codes are called strings. For example:

"Mozart 1.2.0"

is the list:

[77 111 122 97 114 116 32 49 46 50 46 48]

Figure B.1: Graph representation of the infinite list C1=a\|b\|C1

B.5.1 Operations on lists

Table B.11 gives a few basic list operations. Many more operations exist in the Base module List. Here is a simple symbolic calculation with lists:

\begin{verbatim}
declare A B in
A=[a b c]
B=[1 2 3 4]
{Browse {Append A B}}
\end{verbatim}

This displays the list [a b c 1 2 3 4]. Like all operations, these all have correct dataflow behavior. For example, (Length a\|b\|X) suspends until X is bound. The operations Sort, Map, ForAll, Filter, and FoldL are examples of higher-order operations, i.e., operations that take functions or procedures as arguments. We will talk about higher-order execution in Chapter 5. For now, here's an example to give a flavor of what is possible:

\begin{verbatim}
declare L in
L=[john paul george ringo]
{Browse {Sort L Value.´<´}}
\end{verbatim}

sorts L according to the comparison function `<` and displays the result:

[george john paul ringo]

As an infix operator, comparison is written as X<Y, but the comparison operation itself is in the Base module Value. Its full name is Value.´<´. Modules are explained in Section 5.12.2.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
\textbf{Table B.12: Strings \textit{(character syntax)}}

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{VirtualString}.toString\ VS)</td>
<td>Return a string with the same characters as VS</td>
</tr>
<tr>
<td>(\text{VirtualString}.toAtom\ VS)</td>
<td>Return an atom with the same characters as VS</td>
</tr>
<tr>
<td>(\text{VirtualString}.length\ VS)</td>
<td>Return the number of characters in VS</td>
</tr>
<tr>
<td>(\text{Value}.toVirtualString\ X D W)</td>
<td>Return a string representing the partial value (X), where records are limited in depth to (D) and in width to (W)</td>
</tr>
</tbody>
</table>

or equivalently:

\[
&\&M &\&o &\&z &\&a &\&t &\&1 &\&2 &\&0
\]

Using lists to represent strings is convenient because all list operations are available for doing symbolic calculations with strings. Character operations can be used together with list operations to calculate on the internals of strings. String syntax is shown in Table B.12. The \texttt{NUL} character mentioned in the table has character code 0 (zero). See Section B.1 for an explanation of the meaning of \texttt{\a}, \texttt{\b}, etc.

There exists another, more memory-efficient representation for character sequences called \textit{bytestring}. This representation should only be used if memory limitations make it necessary.

\section*{B.7 Virtual strings}

A \textit{virtual string} is a tuple with label \texttt{#} that represents a string. The virtual string brings together different substrings that are concatenated with virtual concatenation. That is, the concatenation is never actually performed, which saves time and memory. For example, the virtual string:

\(123\texttt{#}-\texttt{#23}\) is \texttt{#(123-23)}
represents the string:

"123-23 is 100"

Except in special cases, a library operation that expects a string can always be given a virtual string instead. For example, virtual strings can be used for all I/O operations. The components of a virtual string can be numbers, strings, virtual strings (i.e., '#'-labeled tuples), and all atoms except for nil and '#'. Table B.13 gives a few virtual string operations.
Appendix C

Language Syntax

“The devil is in the details.”
– Traditional proverb.

“God is in the details.”
– Traditional proverb.

“I don’t know what is in those details, but it must be something important!”
– Irreverent proverb.

This appendix defines the syntax of the complete language including all syntactic conveniences. This appendix is divided into seven sections:

- Section C.1 defines the syntax of interactive statements, i.e., statements that can be fed into the interactive interface.

- Section C.2 defines the syntax of statements and expressions.

- Section C.3 defines the syntax of the nonterminals needed to define statements and expressions.

- Section C.4 lists the operators of the language with their precedence and associativity.

- Section C.5 lists the keywords of the language.

- Section C.6 defines the character syntax of variable identifiers, atoms, strings, and characters in strings. Unlike the previous sections which define token sequences, this section defines character sequences.

- Section C.7 defines the character syntax of integers and floating point numbers.
To be precise, this appendix defines a context-free syntax for a superset of the language. This keeps the syntax simple and easy to read. The disadvantage of a context-free syntax is that it does not capture all syntactic conditions for legal programs. For example, take the statement \texttt{local X in S end}. The statement that contains this one must declare all the free variable identifiers of \texttt{S}, possibly minus \texttt{X}. This is not a context-free condition.

This appendix is strongly inspired by the syntactic definitions given in [42, 26]. This appendix differs from [42] in several ways: it introduces \textit{nestable constructs}, \textit{nestable declarations}, and \textit{terms} to factor the common parts of statement and expression syntax, it defines interactive statements and for loops, it leaves out the translation to the kernel language (which is given for each linguistic abstraction in the main text of the book), and it makes other small simplifications for clarity (but without sacrificing precision).

## C.1 Interactive statements

Table C.1 gives the syntax of interactive statements. An interactive statement is a superset of a statement; in addition to all regular statements, it can contain a \texttt{declare} statement. The interactive interface must always be fed interactive statements. All free variable identifiers in the interactive statement must exist in the global environment, otherwise the system gives a \texttt{variable not introduced} error.
\textbf{Table C.3: Nestable constructs (no declarations)}

\begin{enumerate}
\item \texttt{nestCon(\alpha)} ::= \langle \text{expression} \rangle \ (\ `=` | `\leq` | `\geq`) \langle \text{expression} \rangle \\
\item \quad \texttt{local} \{ \langle \text{declarationPart} \rangle \} + \texttt{in} \langle \text{statement} \rangle \langle \alpha \rangle \texttt{end} \\
\item \quad `\{` \langle \text{expression} \rangle \} \ `\} `\texttt{\end{expression}} \\
\item \texttt{if} \langle \text{expression} \rangle \texttt{then} \langle \text{in(\alpha)} \rangle \\
\quad \langle \text{elseif} \langle \text{expression} \rangle \texttt{then} \langle \text{in(\alpha)} \rangle \} \\
\quad \texttt{else} \langle \text{in(\alpha)} \rangle \texttt{end} \\
\item \texttt{case} \langle \text{expression} \rangle \texttt{of} \langle \text{pattern} \rangle \ [ \texttt{andthen} \langle \text{expression} \rangle \] \texttt{then} \langle \text{in(\alpha)} \rangle \\
\quad \langle \text{else} \langle \text{in(\alpha)} \rangle \} \texttt{end} \\
\item \texttt{for} \{ \langle \text{loopDec} \rangle \} + \texttt{do} \langle \text{in(\alpha)} \rangle \texttt{end} \\
\item \texttt{try} \langle \text{in(\alpha)} \rangle \\
\quad \langle \texttt{catch} \langle \text{pattern} \rangle \texttt{then} \langle \text{in(\alpha)} \rangle \} \\
\quad \langle \texttt{finally} \langle \text{in(\alpha)} \rangle \} \texttt{\end{expression}} \\
\item \texttt{raise} \langle \text{inExpression} \rangle \texttt{end} \\
\item \texttt{thread} \langle \text{in(\alpha)} \rangle \texttt{end} \\
\item \texttt{lock} \langle \text{expression} \rangle \texttt{then} \langle \text{in(\alpha)} \rangle \texttt{end}
\end{enumerate}

\textbf{Table C.4: Nestable declarations}

\begin{enumerate}
\item \texttt{nestDec(\alpha)} ::= \texttt{proc} `\{` \langle \text{pattern} \rangle `\} `\texttt{\end{inStatement}} \texttt{end} \\
\item \quad \texttt{fun} \ [ \texttt{lazy} `\{` \langle \text{pattern} \rangle `\} `\texttt{\end{inExpression}} \texttt{end} \\
\item \quad \texttt{functor} \ \langle \alpha \rangle \\
\quad \langle \texttt{import} \ {\langle \text{variable} \rangle \ [ \texttt{at} \langle \text{atom} \rangle \} \\
\quad \langle \text{variable} \rangle `\{` `\} + `\} `\texttt{\end{inExpression}} \texttt{end} \\
\quad \langle \texttt{export} \ {\langle \langle \text{atom} \rangle \ [ \texttt{\end{integer}} \} `:` `\texttt{\end{variable}} \} + `\} `\texttt{\end{expression}} \texttt{end} \\
\item \quad \texttt{define} \ {\langle \text{declarationPart} \rangle} + \texttt{in} \langle \text{statement} \rangle \texttt{end} \\
\item \quad \texttt{class} \ \langle \alpha \rangle \ {\langle \texttt{classDescriptor} \rangle} \\
\quad \langle \texttt{meth} \langle \texttt{methHead} \rangle \ `=` \langle \texttt{variable} \rangle \} \\
\quad \langle \texttt{\end{inExpression}} \texttt{\end{inStatement}} \texttt{end} \texttt{end}
\end{enumerate}

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
\[\langle \text{term} \rangle ::= [ \text{"!' '} ] \langle \text{variable} \rangle | \langle \text{integer} \rangle | \langle \text{float} \rangle | \langle \text{character} \rangle | \langle \text{atom} \rangle | \langle \text{string} \rangle | \langle \text{unit} \rangle | \langle \text{true} \rangle | \langle \text{false} \rangle | \langle \text{label} \rangle \cdot ( [ \langle \text{feature} \rangle \cdot : ] \langle \text{expression} \rangle ) \cdot \langle \text{expression} \rangle \cdot [ \langle \text{consBinOp} \rangle \langle \text{expression} \rangle ] \cdot \langle \text{label} \rangle \cdot ( [ \langle \text{feature} \rangle \cdot : ] \langle \text{pattern} \rangle ) [ \text{\ldots } ] \cdot \langle \text{pattern} \rangle \cdot [ \langle \text{consBinOp} \rangle \langle \text{pattern} \rangle ] \cdot \langle \text{label} \rangle \cdot ( [ \langle \text{feature} \rangle \cdot : ] \langle \text{pattern} \rangle ) [ \cdot + ] \cdot\]

Table C.5: Terms and patterns

C.2 Statements and expressions

Table C.2 gives the syntax of statements and expressions. Many language constructs be used in either a statement position or an expression position. We call such constructs \textit{nestable}. We write the grammar rules to give their syntax just once, in a way that works for both statement and expression positions. Table C.3 gives the syntax for nestable constructs, not including declarations. Table C.4 gives the syntax for nestable declarations. The grammar rules for nestable constructs and declarations are \textit{templates} with one argument. The template is instantiated each time it is used. For example, \(\langle \text{nestCon}()\rangle\) defines the template for nestable constructs without declarations. This template is used twice, as \(\langle \text{nestCon(statement)}\rangle\) and \(\langle \text{nestCon(expression)}\rangle\), and each corresponds to one grammar rule.

C.3 Nonterminals for statements and expressions

Tables C.5 and C.6 defines the nonterminal symbols needed for the statement and expression syntax of the preceding section. Table C.5 defines the syntax of terms and patterns. Note the close relationship between terms and patterns. Both are used to define partial values. There are just two differences: (1) patterns can contain only variable identifiers whereas terms can contain expressions, and (2) patterns can be partial (using \text{"\ldots\"}) whereas terms cannot.

Table C.6 defines nonterminals for the declaration parts of statements and loops, for binary operators ("constructing" operators \(\langle \text{consBinOp} \rangle\) and "evaluating" operators \(\langle \text{evalBinOp} \rangle\)), for records (labels and features), and for classes (descriptors, attributes, methods, etc.).
C.4 Operators

Table C.6 lists the operators in order of increasing tightness of binding. The operators lower in the table bind tighter. We define the associativities as follows:

- **Left.** For binary operators, this means that repeated operators group to the left. For example, 1+2+3 means the same as ((1+2)+3).

- **Right.** For binary operators, this means that repeated operators group to the right. For example, a|b|X means the same as (a|(b|X)).

- **Mixfix.** Repeated operators are actually just one operator, with all expressions being arguments of the operator. For example, a#b#c means the same as ´#´(a b c).

- **None.** For binary operators, this means that the operator cannot be repeated. For example, 1<2<3 is an error.

Parentheses can be used to override the default precedence.

C.5 Keywords

Table C.8 lists the keywords of the language in alphabetic order.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>=</code></td>
<td>right</td>
</tr>
<tr>
<td><code>&lt;-</code></td>
<td>right</td>
</tr>
<tr>
<td><code>orelse</code></td>
<td>right</td>
</tr>
<tr>
<td><code>andthen</code></td>
<td>right</td>
</tr>
<tr>
<td><code>/</code></td>
<td>right</td>
</tr>
<tr>
<td><code>#</code></td>
<td>mixfix</td>
</tr>
<tr>
<td><code>+</code></td>
<td>left</td>
</tr>
<tr>
<td><code>*</code></td>
<td>left</td>
</tr>
<tr>
<td><code>div</code></td>
<td>left</td>
</tr>
<tr>
<td><code>mod</code></td>
<td>left</td>
</tr>
<tr>
<td><code>@</code></td>
<td>left</td>
</tr>
</tbody>
</table>

Table C.7: Operators with precedence and associativity

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Keyword</th>
<th>Keyword</th>
<th>Keyword</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>andthen</code></td>
<td><code>attr</code></td>
<td><code>break</code></td>
<td><code>case</code></td>
<td><code>catch</code></td>
</tr>
<tr>
<td><code>class</code></td>
<td><code>collect</code></td>
<td><code>continue</code></td>
<td><code>declare</code></td>
<td><code>default</code></td>
</tr>
<tr>
<td><code>div</code></td>
<td><code>do</code></td>
<td><code>else</code></td>
<td><code>elsif</code></td>
<td><code>end</code></td>
</tr>
<tr>
<td><code>export</code></td>
<td><code>false</code></td>
<td><code>feat</code></td>
<td><code>finally</code></td>
<td><code>for</code></td>
</tr>
<tr>
<td><code>fun</code></td>
<td><code>functor</code></td>
<td><code>if</code></td>
<td><code>import</code></td>
<td><code>in</code></td>
</tr>
<tr>
<td><code>local</code></td>
<td><code>lock</code></td>
<td><code>meth</code></td>
<td><code>mod</code></td>
<td><code>of</code></td>
</tr>
<tr>
<td><code>onelse</code></td>
<td><code>proc</code></td>
<td><code>prop</code></td>
<td><code>raise</code></td>
<td><code>return</code></td>
</tr>
<tr>
<td><code>self</code></td>
<td><code>skip</code></td>
<td><code>then</code></td>
<td><code>true</code></td>
<td><code>try</code></td>
</tr>
<tr>
<td><code>unit</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.8: Keywords

Table C.9: Variables, atoms, strings, and characters (character syntax)
\[ \text{variableChar} ::= \text{(any inline character except ', \ and NUL)} \]
\[ \text{atomChar} ::= \text{(any inline character except ', \ and NUL)} \]
\[ \text{stringChar} ::= \text{(any inline character except ', \ and NUL)} \]
\[ \text{charChar} ::= \text{(any inline character except \ and NUL)} \]
\[ \text{pseudoChar} ::= ' \ \text{odigit} \ \text{odigit} \ \text{odigit} \ (\text\'x\text\' | \text\'X\text\') \ \text{hexdigit} \ \text{hexdigit} \]
\[ | \text\'a\text\' | \text\'b\text\' | \text\'f\text\' | \text\'n\text\' | \text\'r\text\' | \text\'t\text\' \]
\[ | \text\'v\text\' | \text\'\\\text\' | \text\'\text\'' | \text\'\text\'' | \text\'\text\'' | \text\'\&\text\' \]

Table C.10: Nonterminals needed for character syntax

\[ \text{integer} ::= [ \text\'~\text\' ] \text{nzdigit} \{ \{ \text{digit} \} \}
| [ \text\'~\text\' ] \text\'0\text\' \{ \{ \text{octdigit} \} \} +
| [ \text\'~\text\' ] \text\'0x\text\' | \text\'0X\text\' \{ \{ \text{hexdigit} \} \} +
| [ \text\'~\text\' ] \text\'0b\text\' | \text\'0B\text\' \{ \{ \text{bindigit} \} \} +
\]
\[ \text{float} ::= [ \text\'~\text\' ] \{ \{ \text{digit} \} \} + \text\'\text\'. \{ \{ \text{digit} \} \} [ ( \text\'e\text\' | \text\'E\text\' ) [ \text\'~\text\' ] \{ \{ \text{digit} \} \} + ] \]
\[ \text{digit} ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \]
\[ \text{nzdigit} ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 \]
\[ \text{octdigit} ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 \]
\[ \text{hexdigit} ::= \{ \{ \text{digit} \} \} | \text\'a\text\' | \text\'b\text\' | \text\'c\text\' | \text\'d\text\' | \text\'e\text\' | \text\'f\text\' \]
\[ | \text\'A\text\' | \text\'B\text\' | \text\'C\text\' | \text\'D\text\' | \text\'E\text\' | \text\'F\text\' \]
\[ \text{bindigit} ::= 0 | 1 \]

Table C.11: Integers and floating point numbers (character syntax)
C.6 Variables, atoms, strings, characters

Table C.9 defines the character syntax for variable identifiers, atoms, strings, and characters in strings. Unlike the previous sections which define token sequences, this section defines character sequences. It follows from this syntax that an atom cannot have the same character sequence as a keyword unless the atom is quoted. Table C.10 defines the nonterminals needed for Table C.9. “Any inline character” includes control characters and accented characters. The \texttt{NUL} character has character code 0 (zero).

C.7 Integers and floating point numbers

Table C.11 defines the character syntax of integers and floating point numbers. Note the use of the ‘\~’ (tilde) for the unary minus symbol.
Appendix D

Distribution Model

basic categories
stateful entities
  objects (mobile, stationary, ...)
  other stateful entities
  sited entities
single-assignment entities
stateless entities
global naming
Appendix E

Failure Model

fault states
basic model: exceptions
advanced model: watchers and handlers
language entities
basic abstractions: guards, stationary objects
Appendix F

Mozart System Properties and Limitations

“In der Beschränkung zeigt sich der Meister.”
“Any fool can do well with plenty; it takes a master to do well with limits.”
– German proverb.

For most practical purposes, the Mozart system implements all the computation models in the book exactly, respecting the semantics of Chapter 15. In particular, all code fragments and programs given in the book will run as expected. A few limitations remain; they are listed in this appendix. These limitations apply to Mozart Version 1.2.0 on both Unix and Windows platforms, unless indicated otherwise.

F.1 Memory properties and limitations

Memory allocation is automatic: it is allocated when needed for data and recovered when the data is no longer needed. Most variable sizes, such as the length of lists, the number of record fields, the size of arrays and dictionarys, the number of characters in an atom, the number of threads, and the precision of integers, have no upper limit except that imposed by available memory.

A Mozart process can grow and shrink. When the system is first started, it is several megabytes in size. During execution, it requests memory from the operating system according to need. If memory use decreases, then all unused memory is eventually returned to the operating system.

The following subsections list the few limitations that remain regarding memory usage.
F.1.1 Maximum memory size

Mozart has been ported to 32-bit and 64-bit architectures. On 32-bit architectures, a program can have a maximum of 2048 MB of active data each time a garbage collection starts. Active data is the total size of all data structures that are needed by the executing program. In practice, the actual limit is somewhat smaller than this; it depends on how much garbage the program generates. To be precise, if the amount of garbage is \( g \) MB when a garbage collection starts then the maximum active memory is \( 2048 - g/2 \) MB. This limit is due to the automatic memory management, which uses a copying dual-space garbage collection algorithm.

The distributed programming abilities give a crude but effective way to go beyond this limit. First use the \texttt{Remote} module to create new sites (i.e., OS processes), as explained in Chapter 13. Then partition the application over these sites. Parts of the application that communicate heavily with each other should be put on the same site. If the partitioning is done carefully, this will have almost no effect on performance. Furthermore, because of network transparency, it will require almost no change to the application program.

F.1.2 Garbage collection

Mozart implements a dual-space copying garbage collector. The execution time of a garbage collection is proportional to the size of the active data, not to the total memory size. During a garbage collection, no other execution can take place.

Guaranteeing quick reaction

If it is important to guarantee a quick reaction to external events, then an effective technique is to use the \texttt{Remote} module. First create a new site and then make sure that the new site only has a small amount of active data. This will guarantee that the new site can react quickly. The original site can be slow, as long as it delegates quick reactions to the new site.

Open distributed programming

The memory area used to store program instructions is garbage collected. This means that distributed programs can transfer closures, procedures, objects, classes, and functors between sites indefinitely without resulting in any memory leaks. This is particularly important for open programs, such as client/server applications in which new clients can come and go indefinitely.

Record arities and atoms

All memory areas of the virtual machine are garbage collected except for the atom table and the table of record arities. A record’s arity is the set of its features (see Section 8.5.3).
We recommend to avoid as much as possible dynamically creating atoms (with \texttt{String.toAtom} or \texttt{VirtualString.toAtom}) or records with new arities (for example, with \texttt{Record.filter}). Tuples do not have entries in the arity table, so they are not subject to this restriction.

This does not affect most programs, since records (instances of record arities) are garbage collected. But programs that create new record arities indefinitely, such that the number of no-longer-needed record arities increases indefinitely, will have a memory leak. Programs that interface with databases, such that new atoms are created indefinitely, will also have memory leaks. In the latter case, we recommend to use strings instead of atoms.

### F.1.3 Compiler memory usage

The compiler is part of the interactive run-time system. It may use large amounts of memory when compiling large programs or programs that access large data structures in the run-time environment. This memory use is temporary. After the compiler finishes, the memory is reclaimed.

### F.1.4 Loaded modules are permanent

Modules are loaded on demand. If they are not needed, then they are not loaded and they will not take up any execution time or memory space. But once loaded and installed, a module remains in the system permanently.

### F.1.5 Environment trimming

- Can also occur with thread creation or unification of streams. Explain.
- The explanation below is not quite right. Correct it.

There is a problem with environment trimming that can in some cases lead to a memory leak. This problem is due to incomplete handling of variable identifier lifetimes in environments: some identifiers that are not alive are still considered alive by the system. The problem will be fixed in a future release. Here we explain when the problem occurs and give a simple fix.

The problem is that environments are not correctly trimmed when doing a procedure call, except for the last call which does a correct last call optimization (see Section 5.3.4). This can cause a memory leak if before the call there is a reference to a continuously growing data structure, such as the head of a stream, and after the call the reference is absent. A simple fix is to make sure that all suspicious references are removed by last call optimization. This can be done by modifying the program text to insert a last call. For example, the following code may lead to a memory leak:

```prolog
local P S T in
   P={NewPort S}
```

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
If messages are sent to \( P \) indefinitely, then \( S \) is the head of a growing stream and will cause a leak. The fix is to ensure that \( S \) disappears by inserting a last call to an anonymous procedure:

\[
\text{local } P \ S \ T \ \text{in}
\]

\[
P=\{\text{NewPort } S\}
\]

\[
\{\text{SomeProc0 } S\}
\]

\[
\{\text{proc } \{ $ P \ T\} \}
\]

\[
\{\text{SomeProc1 } P \ T\}
\]

\[
\{\text{SomeProc2 } T\}
\]

\[
\text{end } P \ T)
\]

\[
\text{end}
\]

We indent this in a funny way to show how easy it is to modify the original code. Only two lines have to be added; nothing else has to be changed. The last call invokes an anonymous procedure that does not reference \( S \). It follows that the reference to \( S \) is removed by last call optimization. The memory leak has gone away.

### F.2 Numeric limitations

There are a few numeric limitations:

- Procedures can have no more than 4096 arguments.

- Integers of 28-bit or less precision (small integers) are stored in registers. Floating-point numbers and integers needing more than 28-bit precision (big integers) are stored on the heap. This means that calculations are significantly faster and memory-efficient when done with small integers rather than with floating-point numbers or big integers.

- Floating-point numbers are 64-bit precision, usually in the IEEE floating point standard.

- The range of a finite domain variable is from 0 to \( 2^{27} - 2 \), i.e., from 0 to 134217726.

- The range of a finite set domain variable is from \( \{\} \) (the empty set) to \( \{0, \ldots, 2^{27} - 2\} \).
F.3 Distribution limitations and modifications

A limitation is an operation that is specified but is not possible or has lower performance in the current release. A modification is an operation that is specified but behaves differently in the current release.

F.3.1 Performance limitations

This class of limitations affects only performance, not the semantics. They can safely be ignored if performance is not an issue.

- The code of functions, procedures, classes, and functors (but not objects) is always inserted in messages that are passed between sites, even if the code is already present at the destination. In future releases, the code will be copied across the network only if it is not present at its destination. In both current and future releases, this limitation does not cause a memory leak since at most a single copy of the code can exist per site.

- The distributed garbage collection algorithm reclaims all unused entities except for cycles of stateful entities on at least two different owner sites (a cross-site cycle). For example, if two sites each own an object that references the other, then they will never be reclaimed. It is up to the programmer to break the cycle by updating one of the objects to no longer reference the other.

- If a site crashes that has references to entities created on other sites, then these entities are not garbage-collected. Future releases will incorporate a lease-based or similar technique to recover such entities.

F.3.2 Functionality limitations

This class of limitations affects what operations are available to the programmer. They document where the full language specification is not implemented.

- Objects created with NewStat or NewAsync should not use self. The simple fix is to add an attribute selfattr and initialize it with the object reference. That is, replace the following initialization:

  ```
  meth init
  skip
  end
  ```

  by this one:

  ```
  meth init(S)
  selfattr<-S
  end
  ```
and create the object as follows (with \texttt{NewStat} or \texttt{NewAsync}, whichever is desired):

\[
S = \{\texttt{NewStat MyClass init(S)}\}
\]

In each method that needs \texttt{self}, use \texttt{@selfattr} instead.

- On Windows, the \texttt{Remote} module has limited functionality. Only a single option is possible for fork, namely \texttt{sh}. Future releases will add more options.

- The \texttt{Connection} module can establish a connection across a firewall if the site that initiates the connection is allowed to communicate with the other site. This partially solves the firewall problem of Release 1.1.0. In a future release, this problem will be completely solved by making the connection protocol be a user-defined Oz procedure.

- Threads, dictionaries, arrays, and spaces are sited, even though they are in Base modules. In future releases, it is likely that dictionaries and arrays will be made unsited. Threads and spaces will be made stationary entities that can be called remotely (like ports).

- When a reference to a constrained variable (finite domain, finite set, or free record) is passed to another site, then this reference is converted to a read-only variable on that site. The read-only variable will be bound when the constrained variable becomes determined.

- If an exception is raised or a handler or watcher is invoked for an \texttt{object}, then the \texttt{Entity} argument is undefined. For handlers and watchers, this limitation can be bypassed by giving the handler and watcher procedures an external reference to the object.

- If an exception is raised or a handler is invoked for an \texttt{object}, then the attempted object operation cannot be retried. This limitation can be bypassed by programming the object so that it is known where and in which method the error was detected.

### F.3.3 Modification

There is currently only one modification:

- A handler installed on a variable will \texttt{retry} the operation (i.e., bind or wait) after it returns. That is, the handler is inserted before the operation instead of replacing the operation.

Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.
Bibliography


Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.


Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.


[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.


Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.


Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.


Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.


Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.


Copyright © 2001 by P. Van Roy and S. Haridi. All rights reserved.


