



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

June 10, 2002

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	xv
List of Tables	xxi
Abstract	xxiii
Preface	xxv
I Introduction	1
1 Introduction to Programming Concepts	3
1.1 A calculator	3
1.2 Important information about Mozart	4
1.3 Variables	4
1.4 Functions	5
1.5 Lists	7
1.6 Functions over lists	9
1.7 Correctness	11
1.8 Complexity	12
1.9 Lazy evaluation	14
1.10 Higher-order programming	15
1.11 Concurrency	17
1.12 Dataflow	17
1.13 State	18
1.14 Objects	19
1.15 Classes	20
1.16 Nondeterminism and time	21
1.17 Atomicity	23
1.18 Where do we go from here	24
1.19 Exercises	24



II	General Computation Models	29
2	Declarative Computation Model	31
2.1	Defining practical programming languages	32
2.1.1	Language syntax	33
2.1.2	Language semantics	38
2.2	The single-assignment store	44
2.2.1	Declarative variables	45
2.2.2	Value store	45
2.2.3	Variable-value binding	46
2.2.4	Variable identifiers	46
2.2.5	Variable-value binding revisited	47
2.2.6	Partial values	49
2.2.7	Variable-variable binding	49
2.2.8	Dataflow variables	50
2.3	Kernel language syntax	51
2.3.1	Statements	51
2.3.2	Variable identifiers	51
2.3.3	Values and types	52
2.3.4	Basic types	54
2.3.5	Basic operations	55
2.4	Kernel language semantics	57
2.4.1	Basic concepts	57
2.4.2	The abstract machine	61
2.4.3	Non-suspending statements	63
2.4.4	Suspending statements	66
2.4.5	Basic concepts revisited	68
2.4.6	Last call optimization	73
2.4.7	Active memory and memory management	74
2.5	Practical matters	76
2.5.1	From kernel syntax to full syntax	77
2.5.2	Interactive interface (the declare statement)	81
2.5.3	Functions (the fun statement)	84
2.6	Exceptions	86
2.6.1	Motivation and basic concepts	87
2.6.2	The declarative model with exceptions	89
2.6.3	Full syntax	90
2.6.4	System exceptions	92
2.7	Advanced topics	93
2.7.1	Functional programming languages	93
2.7.2	Unification and entailment	95
2.7.3	Dynamic and static typing	101
2.7.4	Exceptions and declarativeness	103
2.8	Exercises	103



Declarative Programming Techniques	109
3.1 What is declarativeness?	112
3.1.1 A classification of declarative programming	113
3.1.2 Specification languages	114
3.1.3 Implementing components in the declarative model	115
3.2 Iterative computation	116
3.2.1 General schema	116
3.2.2 Iteration with numbers	117
3.2.3 Using local procedures	118
3.2.4 From general schema to control abstraction	120
3.3 Recursive computation	122
3.3.1 Growing stack size	123
3.3.2 Substitution-based abstract machine	124
3.3.3 Converting a recursive to an iterative computation	125
3.4 Programming with recursion	125
3.4.1 Type notation	126
3.4.2 Basic list techniques	127
3.4.3 Accumulators	136
3.4.4 Difference lists	139
3.4.5 Trees	145
3.4.6 Drawing trees	152
3.4.7 Parsing	155
3.5 Time and space efficiency	160
3.5.1 Execution time	160
3.5.2 Memory usage	166
3.6 Higher-order programming	168
3.6.1 Basic operations	168
3.6.2 Loop abstractions	174
3.6.3 Linguistic support for loops	179
3.6.4 Data-driven techniques	181
3.6.5 Explicit lazy evaluation	184
3.6.6 Currying	185
3.7 Abstract data types	186
3.7.1 A declarative dictionary	187
3.7.2 Using the declarative dictionary	191
3.7.3 Secure abstract data types	192
3.7.4 The declarative model with secure types	194
3.7.5 A secure declarative dictionary	198
3.7.6 The principle of independence	199
3.7.7 Capabilities and security	200
3.8 Nondeclarative needs	201
3.8.1 Programming with exceptions	202
3.8.2 Text input/output with a file	204
3.8.3 Text input/output with a graphic user interface	205

3.8.4	Stateless data input/output with files	209
3.9	Large-scale program structure	211
3.9.1	Modules and functors	211
3.9.2	Constructing functors and modules	213
3.9.3	Library modules	215
3.9.4	Standalone compilation	216
3.9.5	Example of a standalone application	217
3.10	More on efficiency	220
3.10.1	Reflections on optimization	220
3.10.2	Memory management	221
3.10.3	Garbage collection is not magic	224
3.11	Exercises	225
4	Declarative Concurrency	229
4.1	The data-driven concurrent model	232
4.1.1	Basic concepts	232
4.1.2	Semantics of threads	235
4.1.3	Example execution	238
4.2	Basic thread programming techniques	239
4.2.1	Creating threads	239
4.2.2	Threads and the browser	239
4.2.3	Dataflow computation with threads	240
4.2.4	Cooperative and competitive concurrency	244
4.2.5	Thread scheduling	245
4.2.6	Operations on threads	247
4.3	Streams	248
4.3.1	Basic producer/consumer	249
4.3.2	Transducers and pipelines	251
4.3.3	Managing resources and improving throughput	253
4.3.4	Stream objects	257
4.3.5	Digital logic simulation	258
4.4	Other programming techniques	264
4.4.1	Order-determining concurrency	264
4.4.2	Coroutines	266
4.4.3	Concurrent composition	267
4.5	Lazy execution	269
4.5.1	The demand-driven concurrent model	272
4.5.2	Reduction order	277
4.5.3	Lazy streams	279
4.5.4	Bounded buffer	281
4.5.5	Lazy reading of a file	283
4.5.6	The Hamming problem	284
4.5.7	List operations	285
4.5.8	List comprehensions	288





4.6	Soft real-time programming	290
4.6.1	Basic operations	290
4.6.2	Ticking	292
4.7	Limitations and extensions of declarative programming	295
4.7.1	Efficiency	295
4.7.2	Modularity	296
4.7.3	Nondeterminism	300
4.7.4	Interfacing with the real world	303
4.7.5	Picking the right model	303
4.7.6	Extended models	304
4.7.7	Using different models together	306
4.8	Advanced topics	307
4.8.1	Kinds of nondeterminism	307
4.8.2	The declarative concurrent model with exceptions	309
4.8.3	The model's "systolic" character	311
4.8.4	Synchronization	311
4.9	Exercises	316
5	Explicit State	323
5.1	What is state?	326
5.1.1	Implicit (declarative) state	326
5.1.2	Explicit state	327
5.2	State and system building	328
5.2.1	System properties	329
5.2.2	Component-based programming	330
5.2.3	Object-oriented programming	330
5.3	The declarative model with explicit state	331
5.3.1	Cells	331
5.3.2	Semantics of cells	333
5.3.3	Relation to declarative programming	334
5.4	Abstract data types	336
5.4.1	Eight ways to organize ADTs	337
5.4.2	Variations on a stack	338
5.4.3	Controlled security	343
5.4.4	Parameter passing	344
5.5	Some stateful data types	349
5.5.1	Iterators	349
5.5.2	Indexed collections	349
5.5.3	Choosing a collection	351
5.6	Reasoning with state	354
5.6.1	Invariant assertions	355
5.6.2	Example	356
5.6.3	Assertions	358
5.6.4	Proof rules	359



5.6.5	Normal termination	362
5.7	Component-based programming and system design	363
5.7.1	Components in general	365
5.7.2	System decomposition and computation models	365
5.7.3	What happens at the interface	366
5.7.4	System structure	371
5.7.5	Development methodology	373
5.7.6	Maintainable systems	375
5.8	Case studies	377
5.8.1	Transitive closure	377
5.8.2	Word frequencies (with stateful dictionary)	383
5.8.3	Generating random numbers	385
5.8.4	“Word of Mouth” simulation	389
5.9	Limitations of stateful programming	392
5.9.1	The real world is parallel	392
5.9.2	The real world is distributed	393
5.10	Exercises	393
6	Object-Oriented Programming	399
6.1	Motivations	400
6.1.1	Inheritance	400
6.1.2	Encapsulated state and inheritance	402
6.1.3	Objects and classes	403
6.2	Classes as complete ADTs	403
6.2.1	Example	404
6.2.2	Semantics of the example	405
6.2.3	Defining classes	406
6.2.4	Initializing attributes	408
6.2.5	First-class messages	410
6.2.6	First-class attributes	412
6.2.7	Programming techniques	413
6.3	Classes as incremental ADTs	414
6.3.1	Inheritance	414
6.3.2	Static and dynamic binding	416
6.3.3	Controlling encapsulation	418
6.3.4	Forwarding and delegation	423
6.3.5	Reflection	425
6.4	Programming with inheritance	428
6.4.1	The correct use of inheritance	428
6.4.2	Constructing a hierarchy by following the type	431
6.4.3	Generic classes	433
6.4.4	Multiple inheritance	436
6.4.5	Rules of thumb for multiple inheritance	442
6.4.6	The purpose of class diagrams	443



6.4.7	Design patterns	443
6.5	Relation to other computation models	447
6.5.1	Object-based and component-based programming	447
6.5.2	Higher-order programming	447
6.5.3	Functional decomposition versus type decomposition	450
6.5.4	Should everything be an object?	452
6.6	Implementing the object system	454
6.6.1	Abstraction diagram	454
6.6.2	Implementing classes	457
6.6.3	Implementing objects	458
6.6.4	Implementing inheritance	459
6.7	Exercises	459
7	Concurrency and State	461
7.1	The declarative model with concurrency and state	464
7.2	Programming with concurrency	465
7.2.1	Overview of the different approaches	466
7.2.2	Using the stateful concurrent model directly	469
7.3	Active objects	472
7.3.1	Example	474
7.3.2	Communication channels and ports	474
7.3.3	Defining an active object	475
7.3.4	Using a class to define behavior	476
7.3.5	The Flavius Josephus problem	476
7.3.6	Making active objects synchronous	480
7.3.7	Handling exceptions	480
7.3.8	A concurrent queue with ports	481
7.3.9	Event manager with active objects	483
7.3.10	Active objects sharing one thread	487
7.3.11	A thread abstraction with termination detection	490
7.3.12	Eliminating sequential dependencies	491
7.4	Atomic actions	493
7.4.1	Locks	495
7.4.2	Building stateful concurrent ADTs	496
7.4.3	Tuple spaces (“Linda”)	497
7.4.4	Implementing locks	503
7.4.5	Monitors	504
7.4.6	Bounded buffer	505
7.4.7	Programming with monitors	507
7.4.8	Implementing monitors	509
7.4.9	Another semantics for monitors	511
7.4.10	Transactions	511
7.4.11	Concurrency control	513
7.4.12	Transactions on cells	518



7.4.13	Implementing transactions on cells	521
7.4.14	More on transactions	525
7.5	Advanced topics	526
7.5.1	Memory management	526
7.5.2	The nondeterministic concurrent model	528
7.6	Case study: a bouncing ball application	533
7.6.1	The architecture	533
7.6.2	The program	534
7.6.3	Making a new bouncer	537
7.6.4	The ball manager	537
7.6.5	The ball active object	538
7.6.6	Animated active objects	539
7.6.7	Interface to the graphics subsystem	540
7.6.8	Making it standalone	541
7.7	Exercises	541
8	Relational Programming	549
8.1	The relational computation model	551
8.1.1	The choice and fail statements	551
8.1.2	Search tree	553
8.1.3	Encapsulated search	553
8.1.4	The <code>Search</code> module	553
8.2	Further examples	555
8.2.1	Numeric examples	555
8.2.2	Puzzles and the n -queens problem	557
8.2.3	Lazy execution	559
8.3	Relation to logic programming	560
8.3.1	Logic and logic programming	560
8.3.2	Operational and logical semantics	563
8.3.3	Nondeterministic logic programming	566
8.3.4	Relation to pure Prolog	567
8.3.5	Logic programming in other models	568
8.4	Natural language parsing	569
8.4.1	A simple grammar	570
8.4.2	Parsing with the grammar	571
8.4.3	Generating a parse tree	572
8.4.4	Generating quantifiers	572
8.4.5	Running the parser	575
8.4.6	Running the parser “backwards”	576
8.4.7	Unification grammars	576
8.5	A grammar interpreter	577
8.5.1	A simple grammar	578
8.5.2	Encoding the grammar	578
8.5.3	Running the grammar interpreter	579



8.5.4	Implementing the grammar interpreter	580
8.6	Databases	582
8.6.1	Defining a relation	583
8.6.2	Calculating with relations	584
8.6.3	Implementing relations	588
8.7	Exercises	589
9	Representative Languages (<i>incomplete</i>)	591
9.1	Erlang and concurrent programming	592
9.1.1	Computation model	592
9.1.2	The <code>receive</code> operation	593
9.2	Haskell and functional programming	597
9.2.1	Computation model	597
9.3	Java and object-oriented programming	598
9.3.1	Computation model	598
9.4	Prolog and logic programming	600
9.4.1	Context	601
9.4.2	Computation model	602
9.4.3	Translating Prolog into a relational program	603
9.5	Exercises	606
III	Specialized Computation Models	609
10	Graphic User Interface Programming (<i>incomplete</i>)	611
10.1	Declarative and procedural approaches	612
10.2	Basic concepts	614
10.2.1	Basic user interface elements	614
10.2.2	Building the user interface	615
10.2.3	Example	616
10.2.4	Declarative geometry	617
10.2.5	Declarative resize behavior	619
10.2.6	Dynamic behavior of widgets	620
10.3	Case studies	621
10.3.1	A simple calendar widget	621
10.3.2	Automatic generation of user interfaces	623
10.3.3	A context-sensitive clock	627
10.4	Implementing the GUI tool	632
10.5	Exercises	632
11	Distributed Programming (<i>incomplete</i>)	633
11.1	The distribution model (part I)	637
11.2	Hands-on introduction	640
11.2.1	Basic operations	640
11.2.2	Network awareness	647



11.2.3	Objects and servers	649
11.2.4	Practical matters	657
11.3	The distribution model (part II)	663
11.3.1	Global naming	666
11.3.2	Programmer support	667
11.3.3	Failure model	668
11.3.4	Implementation	669
11.4	Some related concepts	675
11.4.1	Parallelism	675
11.4.2	Mobility	676
11.5	Generic client/server	677
11.5.1	Application interface	678
11.5.2	A distributed expression evaluator	679
11.5.3	A chat room	681
11.6	Transactional object store	686
11.6.1	A scenario	687
11.6.2	Hands-on introduction	690
11.6.3	Application interface	694
11.6.4	Basic properties	696
11.7	Exercises	697
12	Constraint Programming (<i>incomplete</i>)	699
12.1	Propagate and search	700
12.1.1	Basic ideas	700
12.1.2	Example	702
12.1.3	Executing the example	704
12.1.4	Summary	704
12.2	Programming techniques	705
12.2.1	Example problem	705
12.2.2	Palindrome products revisited	707
12.2.3	Drawing trees revisited	708
12.3	Constraint-based computation model	713
12.3.1	Basic constraints	713
12.3.2	Computation spaces	713
12.3.3	Implementing the relational computation model	720
12.4	Case study: an intelligent minesweeper	720
12.4.1	Rules of the game	721
12.4.2	The architecture and implementation	722
12.4.3	The game	723
12.4.4	The user interface	727
12.4.5	The digital assistant	730
12.5	Exercises	735



IV	Semantics	737
13	Language Semantics	739
13.1	The stateful concurrent model	740
13.1.1	Store	740
13.1.2	Abstract syntax	741
13.1.3	Structural rules	742
13.1.4	Sequential and concurrent execution	743
13.1.5	Comparison with the abstract machine semantics	744
13.1.6	Variable introduction	744
13.1.7	Imposing equality (tell)	745
13.1.8	Conditional statements (ask)	748
13.1.9	Names	750
13.1.10	Procedural abstraction	750
13.1.11	Explicit state	752
13.1.12	By-need synchronization	753
13.1.13	Exception handling	758
13.1.14	Variable substitution	761
13.2	Eight computation models	762
13.3	Semantics of common abstractions	763
13.4	Historical background	764
13.5	Exercises	765
V	Appendices	769
A	Mozart System Development Environment	771
A.1	Interactive interface	771
A.1.1	Interface commands	771
A.1.2	Using functors interactively	772
A.2	Batch interface	773
B	Chapter Supplements	775
B.1	Chapter 1	775
B.1.1	Extensible memory store	775
B.2	Chapter 3	776
B.2.1	Simple file input/output	776
B.3	Chapter 6	777
B.3.1	Simple set operations	777
B.4	Chapter 7	778
B.4.1	Active object	778
B.4.2	Port with close operation	778
B.5	Chapter 11	779
B.5.1	Ticket distribution and distributed objects	779
B.5.2	Generic client/server	780



C Basic Data Types	785
C.1 Numbers (integers, floats, and characters)	785
C.1.1 Operations on numbers	787
C.1.2 Operations on characters	788
C.2 Literals (atoms and names)	789
C.2.1 Operations on atoms	790
C.3 Records and tuples	790
C.3.1 Tuples	791
C.3.2 Operations on records	791
C.3.3 Operations on tuples	793
C.4 Chunks (limited records)	793
C.5 Lists	793
C.5.1 Operations on lists	794
C.6 Strings	795
C.7 Virtual strings	796
D Language Syntax	799
D.1 Interactive statements	800
D.2 Statements and expressions	800
D.3 Nonterminals for statements and expressions	802
D.4 Operators	802
D.5 Keywords	803
D.6 Lexical syntax	805
D.6.1 Tokens	805
D.6.2 Blank space	806
E General Computation Model	807
E.1 Kernel language	808
E.2 Concepts	809
E.3 Layered language design	810
F Mozart System Particularities	813
F.1 Syntax differences	813
F.2 Memory properties and limitations	814
F.2.1 Memory use	814
F.2.2 Memory leaks	815
F.3 Numeric limitations	819
F.4 Distribution limitations and modifications	819
F.4.1 Performance limitations	820
F.4.2 Functionality limitations	820
F.4.3 Modification	821
Bibliography	822
Index	836



List of Figures

1.1	Taking apart the list [5 6 7 8]	8
1.2	Calculating the fifth row of Pascal's triangle	9
1.3	A simple example of dataflow execution	18
1.4	All possible executions of the first nondeterministic example	21
1.5	One possible execution of the second nondeterministic example	22
2.1	From characters to statements	33
2.2	The context-free approach to language syntax	35
2.3	Ambiguity in a context-free grammar	36
2.4	The kernel language approach to semantics	39
2.5	Translation approaches to language semantics	43
2.6	A single-assignment store with three unbound variables	44
2.7	Two of the variables are bound to values	44
2.8	A value store: all variables are bound to values	45
2.9	A variable identifier referring to an unbound variable	46
2.10	A variable identifier referring to a bound variable	47
2.11	A variable identifier referring to a value	47
2.12	A partial value	48
2.13	A partial value with no unbound variables, i.e., a complete value	48
2.14	Two variables bound together	49
2.15	The store after binding one of the variables	49
2.16	The type hierarchy of the declarative model	53
2.17	The declarative computation model	61
2.18	The memory use cycle	75
2.19	Declaring global variables	81
2.20	The Browser	83
2.21	Exception handling	87
2.22	Unification of cyclic structures	97
3.1	A declarative operation inside a general computation	110
3.2	Structure of the chapter	111
3.3	A classification of declarative programming	112
3.4	Finding roots using Newton's method (first version)	117
3.5	Finding roots using Newton's method (second version)	119
3.6	Finding roots using Newton's method (third version)	119



3.7	Finding roots using Newton's method (fourth version)	120
3.8	Finding roots using Newton's method (fifth version)	121
3.9	Sorting with mergesort	135
3.10	Control flow with threaded state	137
3.11	Deleting node Υ when one subtree is a leaf (easy case)	147
3.12	Deleting node Υ when neither subtree is a leaf (hard case)	148
3.13	Breadth-first traversal	150
3.14	Breadth-first traversal with accumulator	151
3.15	Depth-first traversal with explicit stack	152
3.16	The tree drawing constraints	153
3.17	An example tree	153
3.18	Tree drawing algorithm	155
3.19	The example tree displayed with the tree drawing algorithm	156
3.20	Delayed execution of a procedure value	169
3.21	Defining an integer loop	174
3.22	Defining a list loop	174
3.23	Simple loops over integers and lists	175
3.24	Defining accumulator loops	176
3.25	Accumulator loops over integers and lists	177
3.26	Folding a list	178
3.27	Declarative dictionary (with linear list)	187
3.28	Declarative dictionary (with ordered binary tree)	189
3.29	Word frequencies (with declarative dictionary)	190
3.30	Internal structure of binary tree dictionary in <code>WordFreq</code> (partial)	191
3.31	Doing <code>S1={Pop S X}</code> with a secure stack	196
3.32	A simple graphical I/O interface for text	207
3.33	Screen shot of the word frequency application	217
3.34	Standalone dictionary library (file <code>Dict.oz</code>)	218
3.35	Standalone word frequency application (file <code>WordApp.oz</code>)	219
4.1	The declarative concurrent model	232
4.2	Causal orders of sequential and concurrent executions	234
4.3	Relationship between causal order and interleaving executions	235
4.4	Execution of the <code>thread</code> statement	237
4.5	A concurrent map function	241
4.6	A concurrent Fibonacci function	242
4.7	Thread creations for the call <code>{Fib 6}</code>	242
4.8	The Oz Panel showing thread creation in <code>{Fib 26 X}</code>	243
4.9	Dataflow and rubber bands	244
4.10	Cooperative and competitive concurrency	244
4.11	Operations on threads	248
4.12	Producer-consumer stream communication	249
4.13	Filtering a stream	251
4.14	A prime-number sieve with streams	252



4.15	Pipeline of filters generated by {Sieve Xs 316}	253
4.16	Bounded buffer (illustration)	254
4.17	Bounded buffer (data-driven concurrent version)	255
4.18	Digital logic gates	259
4.19	A full adder	261
4.20	A latch	262
4.21	Tree drawing algorithm with order-determining concurrency	265
4.22	Procedures, coroutines, and threads	267
4.23	Concurrent composition	269
4.24	Different models for declarative programming	271
4.25	The by-need protocol	274
4.26	Bounded buffer (naive lazy version)	281
4.27	Bounded buffer (correct lazy version)	282
4.28	Lazy solution to the Hamming problem	284
4.29	A simple ‘Ping Pong’ program	291
4.30	A standalone ‘Ping Pong’ program	292
4.31	A standalone ‘Ping Pong’ program that exits cleanly	293
4.32	Changes needed for instrumenting procedure P1	298
4.33	How can two clients send to the same server? They cannot!	300
4.34	Impedance matching: example of a serializer	306
5.1	The declarative model with explicit state	331
5.2	Five ways to package a stack	339
5.3	Four versions of a secure stack	339
5.4	Different varieties of indexed collections	351
5.5	Extensible array implementation	353
5.6	An application structured as a hierarchical graph	364
5.7	Example of interfacing different execution models	367
5.8	Application structure – static and dynamic	371
5.9	A directed graph and its transitive closure	377
5.10	One step in the transitive closure algorithm	377
5.11	Transitive closure (first declarative version)	379
5.12	Transitive closure (stateful version)	381
5.13	Transitive closure (second declarative version)	381
5.14	Word frequencies (with stateful dictionary)	384
6.1	An example class Counter (with class syntax)	404
6.2	Defining the Counter class (without syntactic support)	405
6.3	Creating a Counter object	405
6.4	Illegal and legal class hierarchies	414
6.5	An example class Account	416
6.6	The meaning of “private”	419
6.7	Different ways to extend functionality	423
6.8	A simple hierarchy with three classes	429

6.9	Constructing a hierarchy by following the type	431
6.10	Lists in object-oriented style	432
6.11	A generic sorting class (with inheritance)	433
6.12	Making it concrete (with inheritance)	434
6.13	A class hierarchy for genericity	434
6.14	A generic sorting class (with higher-order programming)	435
6.15	Making it concrete (with higher-order programming)	436
6.16	Class diagram of the graphics package	438
6.17	Drawing in the graphics package	439
6.18	Class diagram with an association	441
6.19	The Composite pattern	444
6.20	Functional decomposition versus type decomposition	451
6.21	Abstractions in object-oriented programming	456
6.22	An example class <code>Counter</code> (again)	456
6.23	An example of class construction	457
6.24	An example of object construction	458
6.25	Implementing inheritance	459
7.1	The declarative model with concurrency and state	464
7.2	Different approaches to concurrent programming	466
7.3	Concurrent stack	470
7.4	Coroutines	470
7.5	Two active objects playing ball (definition)	473
7.6	Two active objects playing ball (illustration)	473
7.7	The Flavius Josephus problem	477
7.8	The Flavius Josephus problem (active object version)	478
7.9	The Flavius Josephus problem (data-driven concurrent version)	479
7.10	Queue (naive version with ports)	481
7.11	Queue (correct version with ports)	483
7.12	Event manager with active objects	484
7.13	Adding functionality with inheritance	485
7.14	Batching a list of messages and procedures	485
7.15	Active objects sharing one thread	488
7.16	Screenshot of the ‘Ping-Pong’ program	488
7.17	The ‘Ping-Pong’ program: active objects in one thread	489
7.18	A thread abstraction with termination detection	491
7.19	A concurrent filter without sequential dependencies	492
7.20	The hierarchy of atomic actions	493
7.21	Differences between atomic actions	494
7.22	Queue (declarative version)	496
7.23	Queue (sequential stateful version)	497
7.24	Queue (stateful concurrent version with lock)	498
7.25	Queue (concurrent object-oriented version with lock)	499
7.26	Queue (stateful concurrent version with exchange)	500





7.27	Tuple space (object-oriented version)	501
7.28	Lock (non-reentrant version without exception handling)	502
7.29	Lock (non-reentrant version with exception handling)	502
7.30	Lock (reentrant version with exception handling)	503
7.31	Bounded buffer (monitor version)	508
7.32	Monitor implementation	510
7.33	Architecture of the transaction implementation	521
7.34	Implementation of the transaction manager (part 1)	522
7.35	Implementation of the transaction manager (part 2)	523
7.36	Priority queue	525
7.37	Connecting two clients using a stream merger	530
7.38	Implementing nondeterministic choice (naive version)	532
7.39	Implementing nondeterministic choice (full version)	533
7.40	A bouncing ball application	534
7.41	Architecture of the bouncing ball application	535
7.42	Outline of the bouncer program	536
8.1	Search tree for the clothing design example	552
8.2	Two digit counting with depth-first search	555
8.3	The n -queens problem (when $n = 4$)	557
8.4	Solving the n -queens problem with relational programming	558
8.5	Natural language parsing (simple nonterminals)	573
8.6	Natural language parsing (compound nonterminals)	574
8.7	Encoding of a grammar	580
8.8	Implementing the grammar interpreter	581
8.9	A simple graph	584
8.10	Paths in a graph	586
8.11	Implementing relations (with first-argument indexing)	587
9.1	Translation of <code>receive</code> without timeout	595
9.2	Translation of <code>receive</code> with timeout	596
9.3	Translation of <code>receive</code> with zero time out	597
10.1	Building the user interface	615
10.2	Simple text entry window	616
10.3	Function for doing text entry	616
10.4	Windows generated with the <code>lr</code> and <code>td</code> widgets	617
10.5	Window generated with <code>newline</code> and <code>continue</code> codes	617
10.6	Declarative resize behavior	619
10.7	Window generated with the <code>glue</code> parameter	619
10.8	A simple calendar widget	622
10.9	Automatic generation of user interfaces	624
10.10	From the original data to the user interface	625
10.11	Architecture of the context-sensitive clock	628
10.12	Three views of FlexClock	629



10.13	View definitions for context-sensitive clock	630
10.14	The best view for any size clock window	631
11.1	The challenge: simplifying distributed programming	634
11.2	A simple taxonomy of distributed systems	636
11.3	The distributed computation model	638
11.4	Site-oriented view of the distribution model	639
11.5	Distributed locking	649
11.6	The advantages of asynchronous objects with dataflow	651
11.7	Graph notation for a distributed cell	672
11.8	Moving the state pointer	672
11.9	Graph notation for a distributed dataflow variable	673
11.10	Binding a distributed dataflow variable	673
11.11	Generic client/server	678
11.12	A distributed expression evaluator	680
11.13	Architecture of the chat room application	682
11.14	The original <code>ChatServer</code> class	683
11.15	The original <code>ChatClient</code> class	684
11.16	The client functor	685
11.17	The server functor	685
11.18	A collaborative graphic editor	687
12.1	Constraint definition of <i>Send-More-Money</i> puzzle	706
12.2	Tree drawing algorithm with constraint programming	709
12.3	Constraint-based computation model	713
12.4	Visibility of variables and bindings in nested spaces	715
12.5	Communication between a space and its distribution strategy	718
12.6	Depth-first single solution search	719
12.7	The minesweeper application	721
12.8	Architecture of the minesweeper application	722
12.9	An example for the value of <code>G.field</code>	723
12.10	The class <code>Game</code> which implements the game	724
12.11	The state diagram of a <code>GameStatus</code> object	725
12.12	The class <code>GameStatus</code> which implements a game monitor	726
12.13	Implementation of the graphical user interface	728
12.14	Creation of a new game	729
12.15	Making a single square	731
12.16	The application functor	731
12.17	The <code>Switchable</code> class	732
12.18	The implementation of an automatic playing agent	733
12.19	The implementation of an agent propagating knowledge	734
13.1	The kernel language of the stateful concurrent model	742
C.1	Graph representation of the infinite list <code>c1=a b c1</code>	796



List of Tables

2.1	The declarative kernel language	51
2.2	Value expressions in the declarative kernel language	52
2.3	Basic operations	55
2.4	Expressions for calculating with numbers	78
2.5	The if statement	79
2.6	The case statement	79
2.7	Interactive statements	81
2.8	Functions	84
2.9	The declarative kernel language with exceptions	89
2.10	Exceptions	90
2.11	Equality (unification) and equality test (entailment check)	95
3.1	A descriptive declarative kernel language	113
3.2	The parser's input language (which is a token sequence)	157
3.3	The parser's output language (which is a tree)	158
3.4	Execution times of kernel instructions	162
3.5	Memory consumption of kernel instructions	167
3.6	The declarative kernel language with secure types	194
3.7	Functors	211
4.1	The data-driven concurrent kernel language	233
4.2	The demand-driven concurrent kernel language	273
4.3	The declarative concurrent kernel language with exceptions	310
4.4	Correspondence between variable and communication channel	311
4.5	Classifying synchronization	312
5.1	The kernel language with explicit state	332
5.2	Cell operations	332
6.1	Classes	407
7.1	The kernel language with concurrency and state	465
7.2	The concurrent kernel language with nondeterministic choice	529
8.1	The relational kernel language	551
8.2	Translating a relational program to logic	564



8.3	The extended relational kernel language	588
11.1	Distributed algorithms	671
12.1	Primitive operations for computation spaces	717
12.2	The choice statement	718
12.3	The dis statement	719
13.1	Eight computation models	762
C.1	Characters (<i>lexical syntax</i>)	786
C.2	Operations on numbers	787
C.3	Some character operations	788
C.4	Literals	788
C.5	Atoms (<i>lexical syntax</i>)	789
C.6	Some atom operations	790
C.7	Records and tuples	790
C.8	Some record operations	792
C.9	Some tuple operations	793
C.10	Lists	793
C.11	Some list operations	795
C.12	Strings (<i>lexical syntax</i>)	796
C.13	Some virtual string operations	797
D.1	Interactive statements	800
D.2	Statements and expressions	800
D.3	Nestable constructs (no declarations)	801
D.4	Nestable declarations	801
D.5	Terms and patterns	802
D.6	Other nonterminals needed for statements and expressions	803
D.7	Operators with precedence and associativity	804
D.8	Keywords	804
D.9	Lexical syntax of variables, atoms, strings, and characters	805
D.10	Nonterminals needed for lexical syntax	805
D.11	Lexical syntax of integers and floating point numbers	806
E.1	The general kernel language	808



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*, a precise concept that captures the intuition of *programming paradigms*. The models are situated in a uniform framework with a complete and simple formal semantics that allows programmers to reason about correctness and efficiency. We examine the relationships between the models and show how and why to use different models together in the same program.

The simplest computation model covers the domain of declarative programming, which includes deterministic logic programming and strict functional programming. We add concurrency, leading to dataflow, streams, declarative concurrency, lazy execution, and coroutining. We add a nondeterministic choice operator, leading to search and nondeterministic logic programming. We add explicit state, leading to component-based programming, and inheritance, leading to object-oriented programming. Together with concurrency, this leads to active objects 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 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. Peter Van Roy is professor in the Department of Computing Science and Engineering (INGI) at UCL and part-time SICS member. Seif Haridi is professor in the Department of Microelectronics and Information Technology (IMIT) at the Royal Institute of Technology (KTH), Stockholm, and SICS chief scientific advisor.





Preface

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.

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 could pick a small number of languages that are representative of different programming paradigms. 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. The term computation model makes precise the imprecise notion of “programming paradigm”. The rest of the book talks about computation models and not programming paradigms.

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 many widely-used models as well as some less-used models. The main criterium for presenting a model is whether it is useful in practice.

Each computation model is based on a simple core language called its *kernel language*. The kernel languages are introduced in a progressive way, by adding concepts one by one. This lets us show the deep relationships between the different models. It also lets us use different models together in the same program. This is usually called *multiparadigm programming*. It is quite natural, since it means simply to use the right concepts for the problem, independent of what



computation model they originate from. Multiparadigm programming is an old idea. For example, the designers of Lisp and Scheme have long advocated a similar view. However, this book applies it in a much broader and deeper way than was previously done.

When stepping from one model to the next, how do we decide on what concepts to add to its kernel language? We will touch on this question many times in the book. One criterium is that adding a concept lets us write programs that are simpler and have a better structure. Another criterium is expressiveness. Often, just adding one new concept makes a world of difference in programming. For example, adding mutable variables (explicit state) to functional programming allows to do object-oriented programming.

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 many programming languages in the book and relate them to particular computation models. For example, Java and Smalltalk are based on an object-oriented model. Haskell and Standard 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 supports fault tolerant distributed programming.

We single out four languages as representatives of important computation models: Erlang, Haskell, Java, and Prolog. We identify the computation model of each language in terms of the book's uniform framework. 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 [2, 3] is the closest in spirit to ours. It covers functional programming, imperative (i.e., stateful) programming, and introduces objects, concurrency, and logic programming. The present book 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 uniform formal semantics for most of these models, thus putting them on a solid foundation. The semantics is carefully designed to allow reasoning about both correctness and complexity. 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.

Goals of the book

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 two parts: a technology and a science. The technology consists of tools, practical techniques, and standards, allowing to *do* programming. The science consists of a broad and deep theory with predictive power, allowing to *understand* programming. Teaching an engineering discipline means to teach both the current tools (the technology) and the fundamental concepts (the science). Knowing the tools prepares the student for the present. Knowing the science prepares the student for future developments.

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

We define *programming*, as a general human activity, to mean extending or changing a system’s functionality. Programming is a widespread activity that is done both by nonspecialists (e.g., consumers who change the settings of their alarm clock or cellular phone) and specialists (computer programmers, the audience of this book).

This book looks only at software systems. For these systems, programming is the step between specification and running program. This step consists in designing the program’s architecture and abstractions and coding them into a programming language. This is a broad view, perhaps broader than the usual connotation attached to the word programming. It covers both programming “in the small” and “in the large”. It covers both (language-independent) architectural issues and (language-dependent) coding issues. It is based more on concepts and their use rather than on any one programming language. We find that this general view is natural for teaching programming. It allows to look at many issues in a way unbiased by limitations of any particular language or design methodology. When used in a specific situation, the general view is adapted to the tools used, taking account their abilities and limitations.

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. What's more, different schools of thought have developed, based on different ways of viewing programming, called "paradigms": object-oriented, logic, functional, etc. The unity of programming as a single discipline has been lost.

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 suffer from poor design. We give an example based on Java, but the problem exists in all existing languages to some degree. Concurrency in Java is complex to use and expensive in computational resources. Because of these difficulties, Java-taught programmers conclude that concurrency is a fundamentally complex and expensive concept. Program specifications are designed around the difficulties, often in a contorted way. But these difficulties 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, the basic unit of concurrency, 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 language approach

Practical programming languages scale up to programs of millions of lines of code. They provide a rich set of abstractions and syntax. How can we separate the languages' fundamental concepts, which underlie their scalability, from their historical accidents? The kernel language approach shows one way. A practical language is translated into a *kernel language* that consists of a small number of *programmer-significant* elements. The rich set of abstractions and syntax is encoded into the small kernel language. This gives both programmer and student a clear insight into what the language does. The kernel language has a simple formal semantics that allows reasoning about program correctness and complexity. This gives a solid foundation to the programmer's intuition and the programming techniques built on top of it.

A wide variety of languages and programming paradigms can be modeled by a small set of closely-related kernel languages. It follows that the kernel language approach is a truly language-independent way to study programming. Because any given language translates into a kernel language that is a subset of the full 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, structural science lets one design *all* bridges (whether made of wood, iron, both,



or anything else) and predict their behavior in terms of simple concepts such as force, energy, stress, and strain, and the laws they obey [57].

Comparison with other approaches

Let us compare the kernel language approach with three other ways to give programming a broad scientific basis:

- A *foundational calculus*, like the λ -calculus or π -calculus, reduces programming to a minimal number of elements. The elements are chosen for ease in mathematical analysis, not for their programmer intuition. This is intended for mathematicians, not practicing programmers. Foundational calculi are useful for studying the fundamental properties and limits of programming a computer, not for writing or reasoning about general applications.
- A *virtual machine* defines a language in terms of an implementation on an idealized machine. It has concepts that are close to hardware, which makes it hard to reason about abstractions. Virtual machines are useful for designing computers, implementing languages, or for doing simulations, but not for writing or reasoning about general applications.
- A *multiparadigm language* is a language that encompasses several programming paradigms. For example, Scheme is both functional and imperative ([1]) and Leda has elements that are functional, object-oriented, and logical ([24]). The usefulness of a multiparadigm language depends on how well the different paradigms are integrated.

The kernel language approach combines features of all these approaches. Practical languages of different paradigms are defined with kernel languages, whose formal semantics are defined with virtual machines at a high level of abstraction.

Designing abstractions

The most difficult work of programmers, but also the most rewarding, is not writing programs but rather *designing abstractions*. Programming a computer is primarily designing and using abstractions to achieve new goals. We define an *abstraction* loosely as a tool or device that solves a particular problem. Usually the same abstraction can be used to solve many different problems. This versatility is one of the key properties of abstractions.

Abstractions are so deeply part of our daily life that we often forget about them. Some typical abstractions are books, chairs, screwdrivers, and automobiles.¹ Abstractions can be classified into a hierarchy depending on how specialized they are (e.g., “pencil” is more specialized than “writing instrument”, but both are abstractions).

¹Also, pencils, nuts and bolts, wires, transistors, corporations, songs, and differential equations. They do not have to be material entities!



Abstractions are particularly numerous inside computer systems. Modern computers are highly complex systems consisting of hardware, operating system, middleware, and application layers, each of which is based on the work of thousands of people over several decades. They contain an enormous number of abstractions, working together in a highly organized manner.

Designing abstractions is not always easy. It can be a long and painful process, as different approaches are tried, discarded, and improved. But the rewards are very great. It is not a great exaggeration to say that civilization is based on successful abstractions. New ones are being designed every day. Some ancient ones, like the wheel and the arch, are still with us today. Some modern ones, like the cellular phone, quickly become part of our daily life.

The second goal of the book is to teach how to design programming abstractions. We introduce most of the relevant concepts known today, in particular higher-order programming, compositionality, concurrency, encapsulation, explicit state, and inheritance. We show how to build programs as hierarchical graphs of interacting components, where each component is written using the computation model that is best for it. We give some general laws for building abstractions. Many of these general laws have counterparts in other engineering disciplines [50]. We build sequential, concurrent, and robust distributed abstractions. We build abstractions for constraint-based reasoning.

Distributed systems and intelligent agents

Two of the most challenging areas for programmers are distributed systems and intelligent agents. The third goal of the book is to show how programming in these areas can be made simpler, given the proper foundations. We start by making both distribution and inferencing integral parts of the computation model. On top of this foundation, we build powerful abstractions both for fault-tolerant distributed programming and constraint-based inferencing. We illustrate them with case studies of small, but complete and realistic applications. Since the late 1980's we have been doing research in constraint programming. Since 1995 we have been doing research in robust distributed programming. The third goal distills the essence of this research.

Main features

Pedagogical approach

There are two complementary approaches to teaching programming as a rigorous discipline:

- The *computation-based approach* presents programming as a way to define executions on machines. It grounds the student's intuition in the real world by means of actual executions on real systems. This is especially effective



with an interactive system: the student can create program fragments and immediately see what they do. Reducing the time between thinking “what if” and seeing the result is an enormous aid to understanding. Precision is not sacrificed, since the formal semantics of a program can be given in terms of an abstract machine.

- The *logic-based approach* presents programming as a branch of mathematical logic. Logic does not speak of execution but of program properties, which is a higher level of abstraction. Programs are mathematical constructions that obey logical laws. The formal semantics of a program is given in terms of a mathematical logic. Reasoning is done with logical assertions. The logic-based approach is harder for students to grasp yet it is essential for defining precise specifications of what programs do.

Like Abelson & Sussman [2, 3], this book mostly uses the computation-based approach. Concepts are illustrated with program fragments that can be run interactively on an accompanying software package, the Mozart Programming System [110]. Programs are constructed with a building-block approach, bringing together basic concepts to build more complex ones. A small amount of logical reasoning is introduced in later chapters, e.g., for defining specifications and for using invariants to reason about programs with state.

Formalism used

This book uses a single formalism for presenting all computation models and programs, namely the Oz language and its computation model. To be precise, the computation models of this book are all carefully-chosen subsets of Oz. Why did we choose Oz? The main reason is that it supports the kernel language approach well. Another reason is the existence of the Mozart Programming System.

Panorama of computation models

This book presents a broad overview of many of the most useful computation models. All 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.

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 can encode easily 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 concept 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 *explicit state* (mutable variables) 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. Explicit 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.

Importance of using models together

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 early point this answer may seem cryptic. It will become clear later in the book.

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 [1] and Standard ML [107]) and more recently with concurrency (e.g., Concurrent ML [127] and Concurrent Haskell [120, 119]).

Good programming style requires using models together

We find that a good programming style requires using programming concepts that are usually associated with different computation models. Languages that implement just one computation model make this difficult:

- Object-oriented languages encourage the overuse of state and inheritance. Objects are stateful by default. While this seems simple and intuitive, it actually complicates programming, e.g., it makes concurrency difficult (see Section 7.2). Design patterns, which define a common terminology for describing good programming techniques, are usually explained in terms of inheritance [53]. In many cases, simpler higher-order programming techniques would suffice (see Section 6.4.7). In addition, inheritance is often misused. For example, object-oriented graphic user interfaces often recommend using inheritance to extend generic widget classes with application-specific functionality (e.g., in the Swing components for Java). This is counter to separation of concerns.
- Functional languages encourage the overuse of higher-order programming. Typical examples are monads and currying. Monads are used to encode state by threading it throughout the program. This makes programs more intricate but does not achieve the modularity properties of true explicit



state (see Section 4.7). Currying lets you apply a function “partially” by giving only some of its arguments. This returns a new function that expects the remaining arguments. The function body will not execute until all arguments are there. The flipside is that it is not clear by inspection whether the function has all its arguments or is still curried (“waiting” for the rest).

- Logic languages in the Prolog tradition encourage the overuse of Horn clause syntax and search. These languages define all programs as collections of Horn clauses, which resemble simple logical axioms in an “if-then” style. Many algorithms are obfuscated when written in this style. Backtracking-based search must always be used even though it is almost never needed (see [163]).

These examples are to some extent subjective; it is difficult to be completely objective regarding goodness of programming style and language expressiveness. Therefore they should not be read as passing any judgement on these models. Rather, they are hints that none of these models is a panacea when used alone. This book tries to present a more balanced approach, using each concept when it is appropriate and not stretching it beyond its abilities.

Teaching from the book

We explain how the book fits in a computer science curriculum and what courses can be taught with it. We present the Mozart System, a software package that can be used to support these courses.

Role in CS curriculum

Let us consider the discipline of programming independent of any other domain in computer science. In our experience, it divides naturally into three core topics:

1. Concepts and techniques.
2. Algorithms and data structures.
3. Program design and software engineering.

The book gives a thorough treatment of topic (1) and an introduction to (2) and (3). In which order should the topics be given? There is a strong interdependency between (1) and (3). Experience shows that program design should be taught early on, so that students avoid bad habits. However, this is only part of the story since students need to know about concepts to express their designs. Parnas has used an approach that starts with topic (3) and uses an imperative computation model [116]. Because this book uses many computation models, we recommend using it to teach (1) and (3) concurrently, introducing new concepts and design principles gradually. In the engineering program at UCL, we attribute eight



Draft
June 10, 2003

semester-hours to each topic. This includes lectures and lab sessions. Together the three topics comprise one sixth of the full computer science curriculum.

Courses

We have used the book as a textbook for several courses ranging from second-year undergraduate to graduate courses [166, 126]. In its present form, this book is *not* intended as a first programming course, but the approach could be adapted for such a course.² Students should have a small amount of previous programming experience (e.g., a practical introduction to programming and knowledge of simple data structures such as sequences, sets, stacks, trees, and graphs) and a minimum level of mathematical maturity (e.g., a first course on analysis, discrete mathematics, or algebra). The book has enough material for four semester-hours worth of lectures and as many lab sessions. There are many ways to subset the material:

- An undergraduate course on programming concepts and techniques. Chapter 1 gives a light introduction. The course continues with Chapters 2–7. Depending on the desired depth of coverage, more or less emphasis can be put on algorithms (to teach algorithms along with programming), concurrency (which can be left out completely, if so desired), formal semantics (to make intuitions precise), or both.
- An undergraduate course on applied programming models. This includes relational programming (Chapter 8), specific programming languages (Chapter 9), graphic user interface programming (Chapter 10), distributed programming (Chapter 11), and constraint programming (Chapter 12). This course is a natural sequel to the previous one.
- An undergraduate course on concurrent and distributed programming (Chapters 4, 7 and 11). Students should have some programming experience with declarative and object-oriented systems. If desired, the book can be complemented with other texts on concurrent and distributed algorithms (e.g., [95], [31], [98], or [157]).
- A graduate course on computation models (the whole book, including the semantics in Chapter 13). The course can concentrate on the relationships between the models and on their semantics.

In addition, the book can be used as a complement to other courses.

- Part of an undergraduate course on constraint programming (Chapters 2–8, Chapter 12).

²We will gladly help anyone willing to tackle this adaptation.



- Part of a graduate course on intelligent collaborative applications (parts of the whole book, with emphasis on Part III). If desired, the book can be complemented by texts on artificial intelligence (e.g., [129]) or multi-agent systems (e.g., [172]).
- Part of an undergraduate course on semantics. All the models are formally defined in the chapters that introduce them, and this semantics is sharpened in Chapter 13. This gives a real-sized case study of how to define the semantics of a complete modern programming language.

The book, while it has a solid theoretical underpinning, 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 (see below). With these fragments, course lectures can have live interactive demonstrations of the concepts. We find that students very much appreciate this style of lecture.

Each chapter ends with a set of exercises that usually involve some programming. They can be solved on the Mozart system. To best learn the material in the chapter, we encourage students to do as many exercises as possible. Exercises marked (*advanced exercise*) can take from several days up to several weeks. Exercises marked (*research project*) are open ended and can result in significant research contributions.

Software

A useful feature of the book is that all program fragments 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 varieties of Unix and Windows. Distributed programs can be spread out over both 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 makes it ideal for comparing models, by writing programs in different models that solve the same problem, and for using 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. The book uses version 1.2.3, released in December 2001. The book should be compatible with all subsequent versions.

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 our colleagues 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 implements 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 Klinskog, Leif Kornstaedt, Michael 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, Frej Drejhammer, Sameh El-Ansary, Nils Franzén, Simon Lindblom, Benjamin Lorenz, Valentin Mesaros, Al-Metwally Mostafa, and Andreas Simon.

We would also like to thank the following researchers and indirect contributors. Their legacy is not just software quality but also scientific results, tools, course notes, applications, and encouragement. They include: Hassan Aït-Kaci, Andreas Franke, Claire Gardent, Fredrik Holmgren, Sverker Janson, Johan Montelius, Joachim Niehren, Luc Onana, Mathias Picker, Andreas Podelski, Christophe Ponsard, Mahmoud Rafea, Juris Reinfelds, Thomas Sjöland, and Jean Vanderdonckt. Finally, we 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 give a special thanks to Donatien Grolaux for the GUI case studies (Section 10.3) and to Raphaël Collet for the Minesweeper game (Section 12.4) and for his help on Chapter 13 (Language Semantics). We also give a special thanks to Frej Drejhammer, Sameh El-Ansary, and Dragan Havelka for their work on the lab sessions and exercises of DatalogiII, a course taught at KTH that is based on this book.

How did we manage to keep the result so simple with such a large crowd of developers 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 AC-



CLAIM 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 Sverker Janson & Seif Haridi in 1991 [83] (multiple paradigms in AKL), by Gert Smolka in 1995 [148] (building abstractions in Oz), and by Seif Haridi *et al* in 1998 [66] (dependable open distribution in Oz). The first paper on Oz was published in 1993 and already had many important ideas [72]. 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. The high quality of the implementation is due in large part to the systems building expertise of this lab. Among the developers, we mention Christian Schulte for his role in coordinating general development, Denys Duchier for his active support of users, and Per Brand for his role in coordinating development of the distributed implementation. In 1996, the German and Swedish labs 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 2_ε, flex, xfig, xv, vim, emacs, and Mozart, all running on Red Hat Linux with KDE. The first author thanks the Walloon Region of Belgium for their generous support of the Oz/Mozart work at UCL in the PIRATES project.

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, security, and implementation efficiency, we barely mention it. There is a good reason for this, which is explained in Section 2.7.3.
- *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 [129, 112], algorithms [38], object-oriented design patterns [53], multi-agent programming [172], databases [39], and numerical techniques [124].