Concepts, Techniques, and Models of Computer Programming

—

with Practical Applications in Distributed Computing and Intelligent Agents

PETER VAN ROY[1]
Université catholique de Louvain, Belgium
Swedish Institute of Computer Science, Sweden

SEIF HARIDI[2]
Royal Institute of Technology (KTH), Sweden
Swedish Institute of Computer Science, Sweden

June 10, 2002

[1]Email: pvr@info.ucl.ac.be, Web: http://www.info.ucl.ac.be/~pvr
[2]Email: 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

List of Tables

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).

[1] Also, 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

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).

[2] We will gladly help anyone willing to tackly 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 Klintskog, 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].