TLP **5** (4 & 5): 595–600, 2005.
Printed in the United Kingdom

Book review

doi:10.1017/S1471068405002450

Concepts, Techniques, and Models of Computer Programming by Peter Van Roy and Seif Haridi, MIT Press, 2004, hard cover: ISBN 0-262-22069-5, xxvii + 900 pages, 55$.

When Krzysztof Apt asked me to review this book, I (Yves Deville) first refused for two reasons: (1) I am a colleague of the first author, and (2) the book is 900 pages long. But I finally accepted for two other reasons: (1) Krzysztof insisted, and (2) I found several researchers in my lab who accepted to do the review with me. Hence this review is a collective one.

The Oz language is widely used in our lab. It is of course used in Peter Van Roy's group, but also in our constraint programming and bioinformatics group, in program analysis, and in software engineering. The language is also used in several courses.

The book is based on more than 10 years of research on programming languages at DFKI, Saarland University, SICS, and UCL. The Oz language was originally designed by Gert Smolka et al. and Oz 1 was released in 1995. Ten years later, the Oz language is a mature programming language, supported by the Mozart Programming System, and released as free software (under a very liberal X11 style license).

What is this book about? This is not just a book on the Oz language. Rather, the language is used as a support for describing programming concepts and techniques. The focus is on computation models, that is, programming paradigms.

In the following, we provide an overview and an analysis of each chapter of the book. A global analysis is given at the end of the review.

Chapter 1 is a guided tour of topics developed in the book. It is also a short introduction to the Oz language. This is also where we first discover higher-order programming, concurrency, lazy execution, and dataflow programming.

In this first chapter, the objective of the authors is clear: to explain in a simple way various non-trivial programming paradigms.

Chapter 2 goes through the steps of defining a computation model. First, the steps are generally described, and then a computational model, namely the declarative computational model, is built using them. One of the main features of the language is the use of declarative variables; once bound, a declarative variable stays bound throughout the computation and is indistinguishable from its value. The semantics is presented as an operational semantics, and a kernel language is introduced. This simple kernel language will be extended in later chapters, in order to introduce enriched computation models.

The subject is presented in a very methodic way; concepts are defined and then applied to the proposed computational model. Design choices are strongly motivated,

illustrated by numerous examples, and alternatives are enumerated, evaluated, and when they exist in other languages, referenced. This allows the reader to get a holistic view of computer programming, situating the proposed model among other existing ones. This chapter is nice because while defining their computational model, the authors do not force the reader to accept their ideas, but every choice they make is motivated and compared to possible alternatives. This allows the reader to compare his ideas with the enumerated alternatives, and makes him feel like participating in the discussion.

Chapter 3 describes declarative programming techniques. Iterative and recursive computation are explained. Classical computational efficiency principles are then exposed. Higher-order programming principles are introduced, ranging from procedural abstraction to currying. Explicit lazy evaluation is explained. Abstract types, which are types defined only by their operations, are described and implemented in a declarative way. A special concern for secure languages leads to secure data types, where internal structure is protected, and to specific rights granted to part of the program (capabilities). The rest of the chapter studies nondeclarative needs, like I/O and GUI, and the design of an application by introducing the concept of modules and functors (the Oz terminology for a first-class software component).

Declarative programming is not a new concept, but the paradigm is clearly stated and motivated. This long chapter proposes numerous examples; they are practical, accurate, advanced, and constitute a solid base to start writing programs. Although the individual concepts are always clearly explained and illustrated, this chapter is quite dense and the links between parts are sometimes difficult to grasp, namely the introduction of special data types, time/space efficiency, and considerations about secure languages. In short, this chapter is a good and practical introduction to declarative programming, useful for reviewing principles and practice.

Chapter 4 extends the declarative model with concurrency. Dataflow threads, which are preemptive and suspend until data is available, are introduced to define declarative concurrency. Basic concurrent computations are presented along with practical examples. Streams are one of the main building blocks for declarative concurrent programs; they are useful for building producer/consumer programs, bounded buffers, process pipelines, and graphs. The next part presents concurrent composition of threads without streams, by using directly the concurrent declarative model. At this point a special interest is given to lazy execution, showing its use, and the complementarity between laziness and dataflow concurrency. Example of soft-real time programming problems are solved and a presentation is given of Haskell, as a good example of a declarative language. This chapter concludes by discussing the necessity to extend the declarative model and to mix models.

A nice chapter! It describes declarative concurrency in an uniform framework showing its overall power. The kernel language for declarative programming is extended minimally to also cover declarative concurrency. It therefore points out the very basic concepts of concurrency. Examples are well-chosen and the text is comprehensive. The section on laziness is perhaps somewhat harder to follow. This chapter offers a fresh view over concurrency, too often reduced to stateful concurrency.

https://doi.org/10.1017/S1471068405002450 Published online by Cambridge University Press

In **Chapter 5**, the authors present message passing as a programming style that allows building highly reliable systems. This style of programming is based on the asynchronous communication of independent entities (agents). After extending the kernel language and giving the formal semantics of the new concepts introduced, the authors show how the behavior of each independent entity can be defined by using declarative functions.

The presented topics are relevant as client-server applications are non-deterministic in general. This makes Declarative Concurrency not suitable for this need. The authors present the definition of the behavior of the agents and the non-determinism in their communication as two orthogonal concerns. This separation allows the behavior of each agent to be defined in a declarative way, and induces a smooth and pedagogical transition from Declarative Concurrency to Message Passing. Nevertheless, the reading of the previous chapters is mandatory and it is also recommended to be a bit familiar with higher-order programming in order to fully understand the abstractions presented in the chapter.

In **Chapter 6**, the concept of explicit state is introduced and defined, together with a sequential computation model. Data abstraction is introduced, and a comparison of both the stateless and stateful models (ADT vs object style) is given. Standard techniques are also described to reason about stateful programs, followed by case studies of programs that use state.

This chapter really focuses on the principle of abstraction, which is essential for building large systems. The comparison of the stateless and stateful models for building data abstractions is presented objectively. Both have advantages and drawbacks, and neither is preferred to the other. The text emphasizes the importance of state in large programs. Sometimes adding a bit of state in a component can improve encapsulation. The authors also demonstrate how to get the best of both worlds, i.e., how to combine successfully declarative and stateful components.

Chapter 7 defines an object-oriented computation model that supports multiple inheritance with static and dynamic binding. Inheritance is used to build data abstractions in an incremental way. The semantics of objects and classes is given in terms of the stateful computation model. The model is compared to other models and systems.

The computation model is defined in a clean way and is given a precise semantics. The authors also give a simple guideline for the correct use of inheritance. The examples are simple and well chosen. This presentation of object-oriented programming is nice, especially for teaching. It gives a clear intuition of what a class and an object should be. Moreover the model enforces good programming practice: object attributes are private, and classes are stateless. Finally, the model is as flexible as mainstream systems.

Chapter 8 proposes a computation model bringing together explicit state and concurrency. After showing the inherent difficulty of programming in this model, techniques are provided to deal with it. The concept of a lock is introduced and then extended to monitors and transactions.

Thanks to dataflow variables and atomic exchange on explicit state, no extra concept is needed to "control" concurrency. All the usual tools (locks, semaphores,

https://doi.org/10.1017/S1471068405002450 Published online by Cambridge University Press

monitors, etc.) can be programmed in this model. This model is a foundation for multi paradigm programming. It allows the programmer to use the appropriate style for each part of his program. The model also gives students a progressive approach to concurrency, first without state, then with message passing, and finally with full shared state.

In **Chapter 9**, it is shown how the kernel language can be extended to a relational kernel language, leading to a fully relational programming language. Such an extension is obtained by adding choice and failure operators, leading to search trees. The search is encapsulated in various solve functions. The relation to logic programming is discussed. Various examples in different fields are provided (puzzles, natural language, interpreters, databases). Finally, an introduction to the Prolog language is presented. It is namely shown how a Prolog program can be turned into an Oz relational program.

It is nice to see how the kernel language can be turned into a relational language by adding two constructs. If desired, the relational programs can be used with different directionalities (i.e. no pre-specified input or output arguments). However, for the classical Prolog programmer, switching from Prolog to relational Oz programs requires some practice as choice operations are only needed in non-deterministic predicates. The relational Oz has the advantages of the Oz language (higher-order programming, concurrency, etc.), which can make a difference for large applications.

Is Oz a logic programming language? It is a relational language, with (logical) variables and a declarative semantics. But this semantics is not based on classical first-order logic, and its computation model is also not based on resolution. Does a logic programming language need to be based on resolution? This is a debatable question. A significant subset of the Oz language has a logical semantics; in that sense, Oz can be seen as a logic language, like Prolog. For both Prolog and Oz, the whole language does not have a logical semantics. In this chapter, the authors clearly show that a logical semantics can be given to a relational program, and that Prolog programs can easily be turned into Oz programs. Oz is thus not a classical logic programming language, but it offers all the features of logic programming through its relational programming extension.

Chapter 10 describes simple and powerful way to do graphical user interface (GUI) programming by combining the declarative model together with the shared-state concurrent model. It is shown how using declarative and stateful models together helps simplify the programming of graphical user interfaces. The ideas have been worked out in a practical GUI tool, QTk, which is part of the Mozart system.

This chapter begins the second part of the book on specialized computation models.

In **Chapter 11** on distributed programming, the authors describe the various problems that arise when building distributed systems and how the Mozart system can handle some of these issues automatically for the programmer. The main topics covered are network transparency, handling of partial failures, and security. Oz provides network transparency by implementing a set of well-known distributed algorithms that are integrated deep within the language internals. Building a

https://doi.org/10.1017/S1471068405002450 Published online by Cambridge University Press

distributed application then becomes almost transparent to the programmer. Oz also detects and resolves some cases of partial failure of the distributed system, such as process failure or network inactivity. Another important aspect is security, especially when applications are deployed on public networks such as the Internet. The authors make a categorization of security issues that need to be tackled and explain that it is still the subject of ongoing research.

Distributed programming is one of the strong points of the Mozart platform. In particular, network transparency is much more developed than in other popular languages such as Java. However, we wish the section on security were more developed, since security requirements such as integrity, confidentiality, and privacy are a key aspect of today's commercial distributed systems.

Chapter 12 provides an introduction to the constraint part of the Oz language. It introduces the propagate-and-search approach to constraints, and the constraint-based computation domain. Computation spaces are central in the Oz model for constraints; it is used for the search engine; allowing a concurrent propagation of the constraints. Finally, the authors explain how this underlying search engine can be used for implementing the relational computational model of Oz.

Constraint programming is yet another extension of the Oz language. Constraints were known to be already present in early versions of Oz. Constraints are a very powerful scheme to solve CSPs. It is difficult however to fully describe the Oz constraint framework in a single 25-page chapter. This chapter is thus an introduction to the subject. Only a few examples are given. The interested reader would need more practical examples and complementary reading in order to use the full power of the Oz constraint system. This chapter is not as well-written as Chapter 9 on relational programming. We would have liked to see how the kernel relational language should be extended to cover constraint programming, and how the relational computational execution model can include propagators.

Chapter 13 on language semantics is the sole chapter in the third part of the book. A major achievement of Van Roy and Haridi's book is that they cover many programming concepts and many language features with a single elegant and extendable kernel language. The language is introduced throughout the book by example and given a semantics by reference to a simple abstract machine. It is extended, as required, to support new concepts as they are discussed. In this final chapter, the authors revisit their kernel language and give it a rigorous structural operational semantics.

Saraswat and Rinard's concurrent constraint language forms the basis of the kernel language. Its operational semantics is given as reduction rules operating on a constraint store. Required extensions for procedures, mutable state, demand-driven computations, exceptions, and so on, are pleasingly orthogonal. The specialized computation models (GUI programming, distributed programming, and constraint programming) are not covered here. Chaper 13 is formal and rigorous, it is succinct yet still very readable. It is unusual to include such a detailed semantics in a book aimed at a wide audience. We think readers will find the semantics useful and interesting, and we hope it encourages them to take a deeper look at the semantics of the languages they use.

https://doi.org/10.1017/S1471068405002450 Published online by Cambridge University Press

We mention that various exercises are proposed at the end of each chapter, and that the book includes more than 200 bibliographic references. The pedagogical qualities of this book make it especially suitable for teaching. It has been used as a textbook for a number of courses ranging from second-year undergraduate to graduate courses. The book's website gives a list of some of the universities that have used it. At UCL the book is used in a second-year course for engineering students (an introduction to programming concepts) and a third-year course for engineering and computing science students (mainly on concurrent programming). The second-year course is interesting because of the wide variety of concepts it introduces and because it gives a rigorous semantics for programs. The prerequisites for the second-year course are that the students have already had a previous introduction to programming (in any language) and an understanding of simple mathematical concepts (sets, trees, and graphs).

This book is a deep and profound book on computer programming. Within a single simple language, various programming paradigms are handled in a smooth and integrated way. The models in Chapters 1 through 9 are organized in a globally coherent way, using the creative extension principle (explained in Appendix D) which gives a way to decide when to add a new concept to the kernel language and which concept to add.

The three chapters of the second part of the book (Chapters 10 to 12) are not organized in the same holistic way as the rest of the book, but rather present three specialized computation models chosen because the authors consider them important.

In summary, a great book that deserves to be on the shelf of every computer scientist.

Yves Deville, with contributions from Raphal Collet, Jonathan Fallon, Kevin Glynn, David Janssens, Luis Quesada, and Stéphane Zampelli.
Université catholique de Louvain, Belgium

https://doi.org/10.1017/S1471068405002450 Published online by Cambridge University Press