Developing Applications With

Objective Caml

Emmanuel Chailloux Pascal Manoury Bruno Pagano

Developing Applications With **Objective Caml**

Translated by

Francisco Albacete • Mark Andrew • Martin Anlauf • Christopher Browne • David Casperson • Gang Chen • Harry Chomsky • Ruchira Datta • Seth Delackner • Patrick Doane • Andreas Eder • Manuel Fahndrich • Joshua Guttman • Theo Honohan • Xavier Leroy • Markus Mottl • Alan Schmitt • Paul Steckler • Perdita Stevens • François Thomasset

Éditions O'REILLY
18 rue Séguier
75006 Paris
FRANCE
france@oreilly.com
<url:http://www.editions-oreilly.fr/>

O'REILLY®

Cambridge • Cologne • Farnham • Paris • Pékin • Sebastopol • Taipei • Tokyo

The original edition of this book (ISBN 2-84177-121-0) was published in France by O'REILLY & Associates under the title *Dveloppement d'applications avec Objective Caml.*

Historique :

- Version 19990324???????????

Cover concept by Ellie Volckhausen.

Édition : Xavier CAZIN.

Preface

The desire to write a book on Objective Caml sprang from the authors' pedagogical experience in teaching programming concepts through the Objective Caml language. The students in various majors and the engineers in continuing education at Pierre and Marie Curie University have, through their dynamism and their critiques, caused our presentation of the Objective Caml language to evolve greatly. Several examples in this book are directly inspired by their projects.

The implementation of the Caml language has been ongoing for fifteen years. Its development comes from the Formel and then Cristal projects at INRIA, in collaboration with Denis Diderot University and the École Normale Supérieure. The continuous efforts of the researchers on these teams, as much to develop the theoretical underpinnings as the implementation itself, have produced over the span of years a language of very high quality. They have been able to keep pace with the constant evolution of the field while integrating new programming paradigms into a formal framework. We hope through this exposition to contribute to the widespread diffusion which this work deserves.

The form and the foundation of this book wouldn't be what they are without the help of numerous colleagues. They were not put off by rereading our first manuscripts. Their remarks and their comments have allowed this exposition to improve throughout the course of its development. We wish particularly to thank María-Virginia Aponte, Sylvain Baro, Christian Codognet, Hélène Cottier, Guy Cousineau, Pierre Crégut, Titou Durand, Christophe Gonzales, Michelle Morcrette, Christian Queinnec, Attila Raksany and Didier Rémy.

The HTML version of this book would not have seen the light of day without the tools `hevea` and `VideoC`. A big thank you to their respective authors, Luc Maranget and Christian Queinnec, who have always responded in the briefest intervals to our questions and our demands for changes.

Contents

Introduction

Objective Caml is a programming language. One might ask why yet another language is needed. Indeed there are already numerous existing languages with new ones constantly appearing. Beyond their differences, the conception and genesis of each one of them proceeds from a shared motivation: the desire to abstract.

To abstract from the machine In the first place, a programming language permits one to neglect the "mechanical" aspect of the computer; it even lets one forget the microprocessor model or the operating system on which the program will be executed.

To abstract from the operational model The notion of function which most languages possess in one form or another is borrowed from mathematics and not from electronics. In a general way, languages substitute formal models for purely computational viewpoints. Thus they gain *expressivity*.

To abstract errors This has to do with the attempt to guarantee execution safety; a program shouldn't terminate abruptly or become inconsistent in case of an error. One of the means of attaining this is strong static typing of programs and having an exception mechanism in place.

To abstract components (I) Programming languages make it possible to subdivide an application into different software components which are more or less independent and autonomous. Modularity permits higher-level structuring of the whole of a complex application.

To abstract components (II) The existence of programming units has opened up the possibility of their reuse in contexts other than the ones for which they were developed. *Object-oriented* languages constitute another approach to reusability permitting rapid prototyping.

Objective Caml is a recent language which takes its place in the history of programming languages as a distant descendant of Lisp, having been able to draw on the lessons of its cousins while incorporating the principal characteristics of other languages. It is developed at INRIA[1] and is supported by long experience with the conception of the languages in the ML family. Objective Caml is a general-purpose language for the expression of symbolic and numeric algorithms. It is object-oriented and has a parameterized module system. It supports the development of concurrent and distributed applications. It has excellent execution safety thanks to its static typing, its exception mechanism and its garbage collector. It is high-performance while still being portable. Finally, a rich development environment is available.

Objective Caml has never been the subject of a presentation to the "general public". This is the task to which the authors have set themselves, giving this exposition three objectives:

1. To describe in depth the Objective Caml language, its libraries and its development environment.
2. To show and explain what are the concepts hidden behind the programming styles which can be used with Objective Caml.
3. To illustrate through numerous examples how Objective Caml can serve as the development language for various classes of applications.

The authors' goal is to provide insight into how to choose a programming style and structure a program, consistent with a given problem, so that it is maintainable and its components are reusable.

Description of the language

Objective Caml is a functional language: it manipulates functions as values in the language. These can in turn be passed as arguments to other functions or returned as the result of a function call.

Objective Caml is statically typed: verification of compatibility between the types of formal and actual parameters is carried out at program compilation time. From then on it is not necessary to perform such verification during the execution of the program, which increases its efficiency. Moreover, verification of typing permits the elimination of most errors introduced by typos or thoughtlessness and contributes to execution safety.

Objective Caml has parametric polymorphism: a function which does not traverse the totality of the structure of one of its arguments accepts that the type of this argument is not fully determined. In this case this parameter is said to be *polymorphic*. This feature permits development of generic code usable for different data structures,

1. Institut National de Recherche en Informatique et Automatique (National Institute for Research in Automation and Information Technology).

such that the exact representation of this structure need not be known by the code in question. The typing algorithm is in a position to make this distinction.

Objective Caml has type inference: the programmer need not give any type information within the program. The language alone is in charge of deducing from the code the most general type of the expressions and declarations therein. This *inference* is carried out jointly with verification, during program compilation.

Objective Caml is equipped with an exception mechanism: it is possible to interrupt the normal execution of a program in one place and resume at another place thanks to this facility. This mechanism allows control of exceptional situations, but it can also be adopted as a programming style.

Objective Caml has imperative features: I/O, physical modification of values and iterative control structures are possible without having recourse to functional programming features. Mixture of the two styles is acceptable, and offers great development flexibility as well as the possibility of defining new data structures.

Objective Caml executes (threads): the principal tools for creation, synchronization, management of shared memory, and interthread communication are predefined.

Objective Caml communicates on the Internet: the support functions needed to open communication channels between different machines are predefined and permit the development of client-server applications.

Numerous libraries are available for Objective Caml: classic data structures, I/O, interfacing with system resources, lexical and syntactic analysis, computation with large numbers, persistent values, etc.

A programming environment is available for Objective Caml: including interactive toplevel, execution trace, dependency calculation and profiling.

Objective Caml interfaces with the C language: by calling C functions from an Objective Caml program and vice versa, thus permitting access to numerous C libraries.

Three execution modes are available for Objective Caml: interactive by means of an interactive toplevel, compilation to bytecodes interpreted by a virtual machine, compilation to native machine code. The programmer can thus choose between flexibility of development, portability of object code between different architectures, or performance on a given architecture.

Structure of a program

Development of important applications requires the programmer or the development team to consider questions of organization and structure. In Objective Caml, two models are available with distinct advantages and features.

The parameterized module model: data and procedures are gathered within a single entity with two facets: the code proper, and its interface. Communication between modules takes place via their interface. The description of a type may be hidden, not appearing in the module interface. These abstract data types facilitate modifications of the internal implementation of a module without affecting other modules which use it. Moreover, modules can be parameterized by other modules, thus increasing their reusability.

The object model: descriptions of procedures and data are gathered into entities called *classes*; an object is an instance (value) of a class. Interobject communication is implemented through "message passing", the receiving object determines upon execution (late binding) the procedure corresponding to the message. In this way, object-oriented programming is "data-driven". The program structure comes from the relationships between classes; in particular inheritance lets one class be defined by extending another. This model allows concrete, abstract and parameterized classes. Furthermore, it introduces polymorphism of inclusion by defining the subtyping relationship between classes.

The choice between these two models allows great flexibility in the logical organization of an application and facilitates its maintenance and evolution. There is a duality between these two models. One cannot add data fields to a module type (no extensibility of data), but one can add new procedures (extensibility of procedures) acting on data. In the object model, one can add subclasses of a class (extensibility of data) for dealing with new cases, but one cannot add new procedures visible from the ancestor class (no extensibility of procedures). Nevertheless the combination of the two offers new possibilities for extending data and procedures.

Safety and efficiency of execution

Objective Caml bestows excellent execution safety on its programs without sacrificing their efficiency. Fundamentally, static typing is a guarantee of the absence of runtime type errors and makes useful static information available to the compiler without burdening performance with dynamic type tests. These benefits also extend to the object-oriented language features. Moreover, the built-in garbage collector adds to the safety of the language system. Objective Caml's is particularly efficient. The exception mechanism guarantees that the program will not find itself in an inconsistent state after a division by zero or an access outside the bounds of an array.

Outline of the book

The present work consists of four main parts, bracketed by two chapters and enhanced by two appendices, a bibliography, an index of language elements and an index of programming concepts.

Chapter 1 : This chapter describes how to install version `2.04` of the Objective Caml language on the most current systems (Windows, Unix and MacOS).

Part I: Core of the language The first part is a complete presentation of the basic elements of the Objective Caml language. Chapter 2 is a dive into the functional core of the language. Chapter 3 is a continuation of the previous one and describes the imperative part of the language. Chapter 4 compares the "pure" functional and imperative styles, then presents their joint use. Chapter 5 presents the graphics library. Chapter 6 exhibits three applications: management of a simple database, a mini-Basic interpreter and a well-known single-player game, minesweeper.

Part II: Development tools The second part of the book describes the various tools for application development. Chapter 7 compares the various compilation modes, which are the interactive toplevel and command-line bytecode and native code compilers. Chapter 8 presents the principal libraries provided with the language distribution. Chapter 9 explains garbage collection mechanisms and details the one used by Objective Caml. Chapter 10 explains the use of tools for debugging and profiling programs. Chapter 11 addresses lexical and syntactic tools. Chapter 12 shows how to interface Objective Caml programs with C. Chapter 13 constructs a library and an application. This library offers tools for the construction of GUIs. The application is a search for least-cost paths within a graph, whose GUI uses the preceding library.

Part III: Organization of applications The third part describes the two ways of organizing a program: with modules, and with objects. Chapter 14 is a presentation of simple and parameterized language modules. Chapter 15 introduces Objective Caml object-oriented extension. Chapter 16 compares these two types of organization and indicates the usefulness of mixing them to increase the extensibility of programs. Chapter 17 describes two substantial applications: two-player games which put to work several parameterized modules used for two different games, and a simulation of a robot world demonstrating interobject communication.

Part IV: Concurrence and distribution The fourth part introduces concurrent and distributed programs while detailing communication between processes, lightweight or not, and on the Internet. Chapter 18 demonstrates the direct link between the language and the system libraries, in particular the notions of process and communication. Chapter 19 leads to the lack of determinism of concurrent programming while presenting Objective Caml's threads. Chapter 20 discusses interprocess communication via sockets in the distributed memory model. Chapter 21 presents first of all a toolbox for client-server applications. It is subsequently used to extend the robots of the previous part to the client-server model. Finally, we adapt some of the programs already encountered in the form of an HTTP server.

Chapter 22 This last chapter takes stock of application development in Objective Caml and presents the best-known applications of the ML language family.

Appendices The first appendix explains the notion of cyclic types used in the typing of objects. The second appendix describes the language changes present in the new version `3.00`. These have been integrated in all following versions of Objective Caml (`3.xx`).

Each chapter consists of a general presentation of the subject being introduced, a chapter outline, the various sections thereof, statements of exercises to carry out, a summary, and a final section entitled "To learn more" which indicates bibliographic references for the subject which has been introduced.