CONCURRENT PROGRAMMING IN ML

Concurrent Programming in ML presents the language Concurrent ML (CML), which supports the union of two important programming models: concurrent programming and functional programming. CML is an extension of the functional language Standard ML (SML) and is included as part of the Standard ML of New Jersey (SML/NJ) distribution. CML supports the programming of process communication and synchronization using a unique higher-order concurrent programming mechanism which allows programmers to define their own communication and synchronization abstractions.

The main focus of the book is on the practical use of concurrency to implement naturally concurrent applications. In addition to a tutorial introduction to programming in CML, this book presents three extended examples of using CML for systems programming: a parallel software build system, a simple concurrent window manager, and an implementation of distributed tuple spaces.

This book includes a chapter on the implementation of concurrency using features provided by the SML/NJ system and provides many examples of advanced SML programming techniques. The appendices include the CML reference manual and a formal semantics of CML.

This book is aimed at programmers and professional developers who want to use CML, as well as students, faculty, and other researchers.

CONCURRENT PROGRAMMING IN ML

JOHN H. REPPY

Bell Labs, Lucent Technologies, Murray Hill, New Jersey

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521480895

First published 1999
This digitally printed version (with corrections) 2007

A catalogue record for this publication is available from the British Library

Library of Congress Cataloguing in Publication data
Reppy, John H.
Concurrent programming in ML John H. Reppy.
p. cm.
Includes bibliographical references.
ISBN 0 521 48089 2 hardback
1. ML (Computer program language) 2. Parallel processing
(Electronic computers) I. Title.
QA76.73.M6R47 1999
005.2´752 – dc21 99-20465
CIP

ISBN 978-0-521-48089-5 hardback
ISBN 978-0-521-71472-3 paperback

Contents

Preface

This book is about the union of two important paradigms in programming languages, namely, *higher-order* languages and *concurrent* languages. Higher-order programming languages, often referred to as "*functional programming*" languages,[1] are languages that support functions as first-class values. The language used here is the popular higher-order language *Standard ML* (**SML**) [MTH90, MTHM97], which is the most prominent member of the **ML** family of languages. In particular, the bulk of this book focuses on concurrent programming using the language *Concurrent ML* (**CML**), which extends **SML** with independent processes and higher-order communication and synchronization primitives. The power of **CML** is that a wide range of communication and synchronization abstractions can be programmed using a small collection of primitives.

A concurrent program is composed from two or more sequential programs, called *processes*, that execute (at least conceptually) in parallel. The sequential part of the execution of these processes is independent, but they also must interact via shared resources in order to collaborate on achieving their common purpose. In this book, we are concerned with the situation in which the concurrency and process interaction are explicit. This is in contrast with implicitly parallel languages, such as parallel functional languages [Hud89, Nik91, PvE93] and concurrent logic programming languages [Sha89]. The choice of language mechanisms used for process interaction is the key issue in concurrent programming language design. In this aspect, **CML** takes the unique approach of supporting *higher-order concurrent programming*, in which the communication and synchronization operations are first-class values, in much the same way that functions are first-class values in higher-order languages.

Concurrent programming is an especially important technique in the construction of systems software. Such software must deal with the unpredictable sequencing of external events, often from multiple sources, which is difficult to manage in sequential languages.

[1] I choose the term "higher-order" to avoid confusion with "pure" (*i.e.*, referentially transparent) functional languages.

Structuring a program as multiple threads of control, one for each external agent or event, greatly improves the modularity and manageability of the program. Concurrent programming replaces the artificial total ordering of execution imposed by sequential languages by a more natural partial ordering. The resulting program is nondeterministic, but this is necessary to deal with a nondeterministic external world efficiently.

This book differs from most books on concurrent programming in that the underlying sequential language, **SML**, is a higher-order language. The use of **SML** as the sequential sub-language has a number of advantages. **SML** programs tend to be "mostly-functional" and typically do not rely on heavy use of global state; this reduces the effort needed to migrate from a sequential to a concurrent programming style. The high-level features of **SML**, such as datatypes, pattern matching, the module system, and garbage collection, provide a more concise programming notation. Recent advances in implementation technology allow us to take advantage of the benefits of **SML**, without sacrificing good performance [SA94, Sha94, TMC+96]. One of the theses of this book is that efficient system software can be written in a language such as **SML**.

History

The language design ideas presented in this book date back to the language **PML** [Rep88], an **ML** dialect developed at AT&T Bell Laboratories as part of the **Pegasus** system [RG86, GR92]. The purpose of the **Pegasus** project was to provide a better foundation for building interactive systems than that provided by the **C**/UNIX world circa 1985. We believed then, and still do, that interactive applications are inherently concurrent, and that they should be programmed in a concurrent language. This was the motivation for designing a concurrent programming language. We finished an implementation of the **Pegasus** run-time system before the design of **PML** was complete. We tested our ideas on this run-time system by writing prototype applications in **C** with calls to our concurrency library. Our experience with these applications convinced us that the concurrency features of **PML** should be designed to support abstraction. It was this design goal that led me to develop "first-class synchronous operations" [Rep88].

Shortly thereafter, I began a graduate program at Cornell University, and started working with early versions of Standard ML of New Jersey (**SML/NJ**) [AM87, AM91]. In the spring of 1989, Appel and Jim developed a new back-end for **SML/NJ**, based on a *continuation-passing style* representation [AJ89, App92]. A key feature of this back-end is that the program stack was replaced by heap-allocated return closures. In the fall of 1989, this led to the addition of first-class continuations as a language extension in **SML/NJ** [DHM91], which made it possible to implement concurrency primitives directly in **SML**. Exploiting this feature, I implemented a coroutine version of the **PML** primitives on top of **SML/NJ** [Rep89]. Others also exploited the first-class con-

tinuations provided by **SML/NJ**: Ramsey, at Princeton, implemented **PML**-like primitives [Ram90], and Cooper and Morrisett, at Carnegie-Mellon, implemented **Modula 2+** style shared-memory primitives [CM90]. Morrisett and Tolmach later implemented a multiprocessor version of low-level shared-memory primitives [MT93].

While first-class continuations provided an important mechanism for implementing concurrency primitives, they did not provide a mechanism for preemptive scheduling, which is key to supporting modular concurrent programming. To address this problem, I added support for UNIX style signal handling to the **SML/NJ** run-time system [Rep90]. With this support, I modified my coroutine version of the **PML** primitives to include preemptive scheduling, and the first version of **CML** was born. It was released in November of 1990. This implementation evolved into the version of **CML** that was described in the first published paper about **CML** [Rep91a], and was the subject of my doctoral dissertation [Rep92]. In February of 1993, version 0.9.8 of **CML** was released as part of the **SML/NJ** distribution. After that release, a major effort was undertaken to redesign the Basis Library provided by **SML** implementations [GR04]. This effort grew into what is now known as *Standard ML 1997* (**SML**'97), which includes the new basis library, as well as a number of language improvements and simplifications. From the programmer's perspective, the most notable of these changes is the elimination of imperative type variables and the introduction of new primitive types for characters and machine words [MTHM97]. **CML** has also been overhauled to be compatible with **SML**'97 and the new Basis Library, and to use more uniform naming conventions. Although some of the names have changed since version 0.9.8, the core features and concepts are the same. Most recently, Riccardo Pucella has ported **CML** to run on Microsoft's Windows NT operating system.

Since its introduction, **CML** has been used by many people around the world. Uses include experimental telephony software [FO93], as a target language for a concurrent constraint programming language [Pel92], as a basis for distributed programming [Kru93], and for programming dataflow networks [Čub94b, Čub94a]. My own use of **CML** has focused on the original motivation of the **Pegasus** work: providing a foundation for user interface construction. Emden Gansner and I have constructed a multithreaded **X Window System** toolkit, called **eXene**, which is implemented entirely in **CML** [GR91, GR93].

CML has also been the focus of a fair bit of theoretical work. The semantics of the **PML** subset of the language has been formalized in several different ways [BMT92, MM94, FHJ96]. My dissertation also presents a full semantics of the **CML** concurrency mechanisms [Rep91b, Rep92] (Appendix B presents this semantics, but without the proofs). Nielson and Nielson have worked on analyzing the communication patterns (or *topology*) of **CML** programs [NN93, NN94] as well as on control-flow analysis for **CML** [GNN97]. Such analysis can be used to specialize communication operations to provide better performance.

While **CML** is not likely to change, there will continue to be improvements and enhancements to its implementation. The most important improvement is to provide the benefits of kernel-level threads to **CML** applications (*e.g.*, to mask the latency of system calls). To provide these benefits requires a new run-time system, which is under construction as of this writing. This new run-time system should also make a multiprocessor implementation of **CML** possible. The other major effort is to build useful libraries, particularly in the area of distributed programming and network applications. The **CML** home page (see below) will provide information about these, and other, improvements as they become available.

Getting the software

The **SML/NJ** system, **CML**, **eXene**, and other related software are all available, free of charge, on the internet.

Information about the latest and greatest version of **CML**, as well as user documentation, technical papers, and the sample code from this book can be found at the *Concurrent ML* home page:

```
http://cml.cs.uchicago.edu/index.html
```

CML is also available as part of the **SML/NJ** distribution, which can be found at the **SML/NJ** home page:

```
http://smlnj.org/index.html
```

This page also provides links to the **SML/NJ** Library documentation and to the online version of the *Standard ML Basis Library* manual.

Overview of the book

This book was written with several purposes in mind. The primary purpose of this book is to promote the use of **CML** as a concurrent language; it provides not only a tutorial introduction to the language, but also examples of more advanced uses. Although it is not designed as a teaching text, this book does provide an introduction to Concurrent Programming, and drafts of it have been used in courses at various universities. Because of the strong typing of **SML** and the choice of concurrency primitives, **CML** provides a friendlier introduction to concurrent programming than in many other languages. **CML** also provides a good example of systems programming using **SML/NJ**, and I hope that this book will inspire other non-traditional uses of the language. This book does not make an attempt to introduce or describe **SML**, as there are a number of books and technical

reports that already fill that purpose; a list of these can be found in the Chapter 1 notes (and on the **SML/NJ** home page).

The book is loosely organized into three parts: an introduction to concurrent programming, an expository description of **CML** (essentially a **CML** tutorial), and finally, a practicum consisting of example applications.

The first chapter motivates the rest of the book by arguing the merits of concurrent programming. Chapter 2 introduces various concepts and issues in concurrent programming and concurrent programming languages.

The next four chapters focus on the design and use of **CML**. First, Chapters 3 and 4 give a tutorial introduction to the basic **CML** features and programming techniques. Chapter 5 expands on this discussion by exploring various synchronization and communication abstractions. Finally, Chapter 6 describes the rationale for the design of **CML**; this chapter is mainly intended for those interested in language design issues, and may be skipped by the casual reader.

The subsequent three chapters present extended examples of **CML** programs. While space restrictions constrain the scope of these examples, each is a representative of a natural application area for concurrent programming. Furthermore, they provide examples of complete **CML** programs, rather than just program fragments. Chapter 7 describes a controller for a simple parallel software-build system. This illustrates the use of concurrency to manage parallel system-level processes. The next example, in Chapter 8, is a toy concurrent window manager, which illustrates the use of concurrency in user interface software. Chapter 9 describes an implementation of distributed tuple spaces. This provides both an illustration of how a distributed systems interface might fit into the **CML** framework, and how systems programming can be done in **SML** and **CML**.

The book concludes with a chapter on the implementation of concurrency in **SML/NJ** using its *first-class continuations*. **SML/NJ** provides a fairly unique test-bed for experimenting in concurrent language design, and this chapter provides a "how-to" guide for such experimentation.

There are two appendices, which provide a more concise description of **CML**. Appendix A is an abridged version of the *CML Reference Manual*; the complete manual is available from the **CML** home page. Appendix B gives an operational semantics for the concurrency features of **CML**, along with statements of some of its properties (proofs can be found in my dissertation [Rep92]).

Citations and a discussion of related work are collected in "**Notes**" sections at the end of each chapter. These notes also provide some historical context. The text is illustrated with numerous examples; the source code for most of these is available from the **CML** home page.

Acknowledgements

This book has taken a long time for me to write, and many people have helped along the way. Foremost, I would like to thank my wife, book coach, and primary proofreader, Anne Rogers. Without her encouragement, I doubt this book would have been finished. My graduate advisor at Cornell University, Tim Teitelbaum, provided support for the research that produced **CML**.[2] Andrew Appel and Dave MacQueen encouraged me to start this project, and have helped with implementation issues.

In addition to Anne, a number of other people provided feedback on portions of the text. Emden Gansner and Lorenz Huelsbergen performed the second pass of proofreading on many of the chapters. Greg Morrisett provided detailed feedback about Chapter 10, and Jon Riecke helped with the semantics in Appendix B. Nick Afshartous, Lal George, Prakash Panangaden, and Chris Stone also provided feedback on various parts of the book. Riccardo Pucella has helped with recent versions of the **CML** implementation, including the Windows NT port. I would also like to thank my editor at Cambridge University Press, Lauren Cowles, for her patience with this book — I hope the final result was worth the wait. I would also like to thank the many users of **CML**, who found those pesky bugs and used the language in ways that I never envisioned.

John H. Reppy
Murray Hill, NJ

[2] While a graduate student at Cornell, I was supported, in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under NSF grant CCR-89-18233.

Legend