*Interpreter with Lazy Evaluation*, dovetailed nicely with this books's Chapter 14 *A Normal-Order Language*).

In contrast, I didn't find much comparable material in *The Design and Evolution of C++*[4]. Languages discusses types and polymorphism but these seem just a little differently the than corresponding concepts in C++ OO programming. While the book's discussion on imperative languages can apply to portions of my daily work programming objects in C++, the most interesting behaviors of our programs derive from interactions between objects, so I would have welcomed rules for reasoning about object correctness.

This book starts with an introduction to the predicate logic used throughout the text, the follows with the base semantics for a simple imperative language.

From this, the author develops methods for specifying and proving programs in the simple imperative language. Following chapters (4 and 5) add arrays, failure, I/O and continuations to the simple imperative language. The transition semantics of this language is considered in chapter 6.

The next several chapters (7-9) cover concurrent programming concepts, starting with a discussion of non-determinism and ending with chapters on the two main types of concurrency: shared variable concurrency and message-passing concurrency.

Chapters 10-14 cover topics in functional programming. Chapter 10 is a review of the lambda calculus, 11 introduces an eager-evaluation. Chapter 12 introduces continuations to the eager-evaluation language.

Chapter 13 describes the Iswim [5] approach to combining functional and imperative approaches. Chapter 14 introduces returns to functional languages by introducing a normal order functional languages.

Chapters 15-17 cover type systems, including simple types, subtypes and type polymorphism( functional polymorphic programming, which is somewhat different than (C++) object-oriented programming).

Finally, chapter 19 covers Algol-like languages. As stated above, the final section of the book includes an appendix summarize basic formal logic and methods.

Reviewed by Peter Claussen, Daktronics, Inc. – pclauss@daktronics.com.

[4] Bjarne Stroustrup, Addison-Wesley

[5] for C++ programmers, like myself, unfamiliar with Iswim, *Iswim-like* languages incorporate imperative features into an eager-evaluation functional language. This is contrasted with Algol-like languages, including C++, where different imperative features are added to a normal order language.

## Concurrent Programming in ML

John H. Reppy

*Concurrent Programming in ML* is written by John H. Reppy, and published by Cambridge University Press, 1999, ISBN 0-521-48089-2, 308 pp., $44.95

Concurrent ML is a set of concurrency primitives added to the **SML/NJ** [6] implementation of Standard ML.

The open sentence of this book states quite simply that "this book is about the union of two important programming paradigms in programming languages, namely *higher-order*[7] languages and *concurrent* languages." If you are currently an ML user, specifically **SML/NJ**, and wish to use the concurrent features available through Concurrent ML (**CML**), this is your user manual. 'Nuff Said.

For the rest of us, this book is a very good introduction to concurrent programming if you've had some experience with ML or at least functional programming[8] and is a pretty acceptable introduction to functional ideas, if you've had concurrent experience.

I wouldn't recommend this book if you're a primarily imperative (i.e. C/C++) programmer; the combination of functional constructs with concurrent primitives will give you just a little to much cognitive dissonance for comfort, plus the examples aren't all that useful unless you intend to learn functional progrmamming.

This book covers the fundamental concepts of concurrent programming, such as processes, shared-memory concurrency, message-passing concurrency and parallel programming (the

[6] **SML/NJ**, including **CML**, can be obtained at
**http://cm.bell-labs.com/cm/cs/what/smlnj/index.html**
More information on **CML** itself can be found at:
**http://cm.bell-labs.com/ jhr/sml/cml/index.html**

[7] higher-order meaning functions are first-class; commonly this translates to *functional*.

[8] Personally I'm something of an ML neophyte but have had enough experience with Scheme to follow most of the examples in the book. For instance, the sample code

```
fun insert (BUF{data, mu, dataAvail, dataEmpty}, v)= let
      fun waitLp NONE = data := SOME v; signal dataAvail)
          | waitLp (SOME v) = (wait dataEmpty; waitLp(!data))
      in
          withLock mu waitLp(!data)
      end
```

can be roughly translated into the Scheme code

```
(define insert
    (lambda (data mu dataAvail dataEmpty v)
       (let ((waitLp
            (lambda (data)
            (cond ((null?  data) ((signal dataAvail)(SOME v)))
            (#t ((wait dataEmpty) (waitLp (not data)))))))
            )
       (withLock mu (waitLp (not data)))
)))
```

Note to Schemers: I don't vouch for the accuracy of the above translation; if you can indentify errors you should have no trouble with the examples presented in this book.

author makes a distinction between *concurrency*: operations executed in software, and *para llelism* : temporally overlapping operations in hardware). Concepts are extensively illustrated with **CML** code fragments.

**CML** programming style is demonstrated using merge sort, power series multiplication and client-server mechanisms as examples. There are also examples implementing synchronous programming mechanisms in **CML**.

One chapter is devoted to the design of the **CML** primitives. This is complemented by a later chapter that shows how concurrent mechanisms can be implemented in standard ML (specifically **SML/NJ**).

In addition code fragments are used through most of the book, the author demonstrates the scalability of **CML** to larger systems by implementing a 'parallel' UNIX/**C** build system, a parallel windowing framework and **Linda**-style[9] extension to **CML**.

Bottom line, if you have an available **SML/NJ** this book will get you started with concurrent programming. If you're a student of either functional or concurrent programming, this is an excellent resource. However, if you're a 'mainstream' applications programmer, this book will be of limited use.

Reviewed by Peter Claussen, Daktronics, Inc. pclauss@daktronics.com.

## The Optimal Implementation of Functional Programming Languages

Andrea Asperti and Stefano Guerrini

*The Optimal Implementation of Functional Programming Languages* is written by Andrea Asperti and Stefano Guerrini and published by Cambridge University Press ISBN 0-521-62112-7

This book is quite complex, mainly dealing with lambda calculus and sharing. The authors even provided a road map to follow so the reader doesn't have to become an expert before moving on from one chapter to the next. This book is aimed at grad students that want to improve the optimization techniques in computer languages like lisp.

It consists of 382 pages of well-written material. I especially enjoyed the graphs and figures without them, I would have been completely lost. About a third of the book is focused on an Optimal Reduction algorithm.

The book just kind of fades out at the end. I was expecting a source listing of the language that was being built. The authors did make reference to an FTP site, However, I couldn't access the referenced file.

Reviewed by Ron Dinishak – Ron.Dinishak@fema.gov.

[9]**Linda** provides tuple-space extensions to standard imperative languages. The tuple space is shared between processes and provides the mechanism for inter-process communication.

## Term Rewriting and All That

Franz Baader and Tobias Nipkow

*Term Rewriting and All That* is written by Franz Baader and Tobias Nipkow, and published by Cambridge University Press, 1998 (paperback), 0-521-77920-0, 301 pp., $27.95.

*Term Rewriting and All That* is a self-contained introduction to the field of term rewriting. The book starts with a simple motivating example and covers all the basic material including abstract reduction systems, termination, confluence, completion, and combination problems. Some closely connected subjects, such as universal algebra, unification theory, Grobner bases, and Buchberger's algorithm, are also covered.

For those who want to get an understanding of the field of term rewriting but do not want to go through the literature and deal with different notations, this book is a great start point. For those who are only interested in one or two of the subjects covered in this book and do not mind dealing with different notations, however, it might be a better idea to directly dive into the literature. Even in this case, the bibliography of this book, which contains a list of 252 publications, could justify the price of the whole book. Most chapters close with a "Bibliographic notes" section. These brief guides to the literature can help readers stay focused.

The book contains many examples. Algorithms are presented both informally and as ML programs. In addition, over 170 exercises make this book sufficient to serve as a textbook.

Besides the consistent style of presentations of various subjects, I also like the text that is printed in easy-to-read big fonts and the clearly presented figures.

Reviewed by Chang Liu, University of California, Irvine – liu@ics.uci.edu.

## The Functional Approach to Programming

Guy Cousineau and Michel Mauny

*The Functional Approach to Programming* is written by Guy Cousineau and Michel Mauny, and published by Cambridge University Press, 1998. ISBN 0-521-57681-4(paperback), $39.95, ISBN 0-521-57183-9(hardback), $85.00 445 pp.,

This book is an introduction to functional programming, using CAML (a dialect of ML) as the teaching language

The first 2 chapters cover the usual introductory material on functional programming: builtin types, function definitions, pattern matching, higher order functions, polymorphism, lists, Cartesian products, etc. and are as good an introduction as can be found anywhere. They are also useful reading for the reader unfamiliar with the syntax of CAML.

Chapter 3 (Semantics) introduces rewriting as a mechanism for evaluating expressions, briefly discusses operational (rewrite) semantics and denotational semantics, and explains