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

the difference between lazy and eager evaluation. A small section on proving the correctness of programs introduces the basic strategies for proving partial correctness: equational reasoning and inductive reasoning, and the use of well-founded orders for proving termination. The section on strategies provides an explanation for CAML's choice of eager evaluation.

Chapter 4 introduces the characteristic that distinguishes the ML-family of languages from the pure functional languages like Haskell: the presence of imperative features. Its use is illustrated in defining side-effecting functions (such as I/O functions), and mutable data structures such as Vectors and Records. CAML introduces 3 subtly different assignment operators (=, :=, and ←), which the authors explain. (Basically, variables in CAML are not variables in the sense of Pascal or C, but are more akin to constants. A statement of the form `x := x + 1` is not legal in CAML. To update a location of type `T` requires a reference to that location of type `T ref`. The equivalent to `x:=x+1` in CAML would be `a_x := !a_x + 1`, where `a_x` is of type `int ref`. The variable `a_x` itself does not change value. Instead the object it refers to (the location) changes. This is similar to the idea of objects and constant references in C++, where given `T* const x`, `x` is a constant reference to the object of type T. However, the object (`*x`) being pointed at can be modified. Arrays and mutable records in CAML are also treated like objects. However, this connection (between CAML data structures and objects) is not discussed.

Part II of the book covers Applications. The applications addressed are in the domains of symbolic computation (pattern matching, unification, rewriting), data structures (binary search trees, AVL trees), graphs, formal language (lexical and syntax analyzers), graphics (drawing shapes, trees, tiling), and computer arithmetic.

Part III deals with implementation. Chapter 11 looks at Evaluation mechanisms, including lazy evaluation, (for writing an interpreter), Chapter 12 at a simple compiler for the language, and Chapter 13 at a type synthesizer for the language.

The book is an introduction to functional programming in a strict language, and particularly in CAML. Unfortunately, the authors do not say at whom the book is targeted, apart from mentioning that it was used as part of a teaching course by one of the authors. My guess would be that the style of the book makes it suitable for an introductory graduate course, or perhaps an upper-division class. For a lower division undergraduate course, I would prefer to see a more gentle development of the functional approach, not introducing advanced concepts like recursion, references, function spaces, semantics, etc until later in the book (*ala* Bird and Wadler's *Introduction to Functional Programming*). The examples in an undergraduate text would also have to be more motivational.

I liked the explanations and snippets of theoretical background interspersed throughout the book. Chapter 12 on a simple compiler for the language was particularly good for letting students know that not only do functional languages support symbolic evaluation (and thereby serve as a prototyping tool) but they can also be efficiently compiled down into directly executable code. This chapter gives students a nice simplified overview of how this might be done, and some of the compilation techniques used to make it practical. The use of functions in Chapter 7 to define graphs that could be infinite was a nice solution to the general problem of how to define potentially infinite structures in a strict language. The book is generally well organized and free of typos and errors.

I suppose my biggest gripe about this book is a gripe I have about many books on functional programming. Namely, the examples used (under the guise of "Applications") only serve to reinforce the stereotype of functional languages as "academic" languages. This is far from the truth as the many industrial applications of functional languages show, but someone new to functional programming would be left with the overall impression that functional languages are confined mainly to abstract mathematical problems such as puzzles, computer algebra, and computing π, or for "canned" problems like sorting and tree traversal. These are certainly important problems, but they are hardly likely to encourage graduates to go out into industry and push for the widespread use of functional languages. Where are the "real world" examples, like a GUI editor, a web server, or a simple payroll application? I was particularly disappointed in that regard with this book, since a wonderful opportunity was lost to show how judicious use of CAML's imperative features could make the writing of such applications not only more efficient but less unwieldy (notwithstanding the examples of linked lists). This is particularly odd, since the CAML web site mentions at least two real applications (an interactive editor and a web browser MMM) written in CAML. A simplified version of these applications could perhaps have been presented and dissected as a case study.

Minor issues:
Chapter 3 ought to have explained that certain theorems are not provable by equational reasoning (the associativity of append being a case in point) to justify introducing proof by induction.

Also, the subsection on How to Deal with Recursion in Chapter 3 left me somewhat confused. Essentially, the authors are trying to explain that defining recursive functions in the λ calculus is not straightforward, and that one needs the fixpoint operator Y, definable only in the untyped l calculus. In the typed λ calculus, upon which CAML is based, Y is not well typed. Hence the claimed need for `let rec`. The alternative (discussed in the next subsection) is to require all function definitions to be named (which is in fact how functional languages get around the problem!). But the `let rec` construct is not used in function expressions (CAML's λ-expressions) but for regular named function definitions. That seems to defeat the claimed purpose of `let rec`. Also, why don't other functional languages need a `let rec`? Is this more of a 1-pass compiler issue than anything else? In any case, the issue has very little to do with evaluation strategies and its appearance in that section seems out of place.

I was left wondering where the definition of `gentree_of_string` used on p.136 was, before remembering that several chapters back, the authors mention they assume the existence of a function `T_of_string` for any type `T`. This is something that is easy to overlook and a reminder to that effect would have helped. The same goes for the `list_it` function on p. 138 that is used without comment.

I would recommend the book for anyone looking for teaching material on CAML, and as a text for a graduate course on functional programming.

Reviewed by Srinivas Nedunuri, Semantic Designs, Austin, Texas – snedunuri@semdesigns.com.

Automated Software Testing - Introduction, Management and Performance

Elfriede Dustin, Jeff Rashka, and John Paul

Automated Software Testing - Introduction, Management and Performance is written by Elfriede Dustin, Jeff Rashka, John Paul, and published by Addison Wesley, 1999 paperback, ISBN 0-201-43287-0, 575 pages, US $44.95

This book is almost the job description or a field handbook of a software test manager. The authors have managed to simplify the whole process as close to the actual way of working as possible - while reading it, you can almost imagine yourself doing it yourself !

Chapter 1 builds-up the case for software testing in general, and introduces the Automated Test Life-Cycle Methodology (ATLM) and its six steps. Surprisingly, there is a discussion on software testing careers - which is strange because it distracts the main theme of the book and also because complete Appendix C is devoted to the same topic in much more details.

The six steps of ATLM are introduced one by one from Chapter 2 through Chapter 10. Every chapter has the same look-and-feel: There is a graphic of the ATLM that specifies which step of the ATLM is being discussed in the following chapter. Whenever required, the chapter describes the work breakdown structure of the task or the activity being discussed, sometimes also accompanied by a flowchart. I guess that makes the whole discussion really deployable. The summaries at the end of each chapter are good.

The Appendices are a real value-addition. Appendix A is a good paper on "How to Test Requirements". Appendix B is on tools that support automated testing lifecycle - a real wealth of information on tools, nicely described phase-wise, available in the market with all details you might want, should you decide to explore more about a particular tool. Appendix C is on career management for a test engineer. I like the authors highlighting the soft-skills required for a test engineer in molding his / her career. Appendix D is Sample Test Plan - I am almost sure no one will actually write such a detailed plan. However, it does bring out the most minute details one must be careful of while planning a test activity (though a few of these things might happen implicitly and hence might not be required to be put on paper in such detail). One thing is for sure, though, you could not land up in a surprise if you create a test plan with this much details. Appendix E is for on Best Practices, but it is not treated well.

A CD comes along with the book. It contains chapters from the book on Sample Test Plan, etc. in the PDF format. I would have expected the authors to put the document on "Test Tool Evaluation" and "Sample Test Plan" as a template in some editable format (Word, RTF, etc.) so that it could add value to the readers looking for a jumpstart in deploying some uniformity in their evaluation or planning process. Nevertheless, the files are still not bad for a quick, on-line reference, and with some effort, one can create the template on his own.

References are good in general, and abundantly point to sources on the web.

This book is timed well - in the world of internet-cycle projects, there is a double-edged pressure on a software test manager to improve test quality in the ever-shrinking test schedules. The book makes its point that there is no cowboy substitute to carefully planning the whole aspect of automating the software testing process

The authors rightly highlight the correct audience for this book - software test manager. The book is not meant for entry-level engineers or even people with limited experience in complete software lifecycle.

I liked the discussion on "Test Effort Sizing" (chapter 5.3) and examples of causal analysis and corrective actions in chapter 10. In my experience, I have seen most people uncomfortable in these two important aspects and need some handholding. This book manages to at least accomplish the later one.

Reviewed by Tathagat Varma, Digital Networks - Home Access Solutions, Philips Software Centre, Bangalore, India tathagat.varma@philips.com

Object-Oriented Software Engineering: Conquering Complex and Changing Systems.

Bernd Bruegge and Allen H. Dutoit

Object-Oriented Software Engineering: Conquering Complex and Changing Systems. was written by Bernd Bruegge and Allen H. Dutoit and published by Prentice Hall, New Jersey, 2000. ISBN 0-13-489725-0

This is the book I have been looking for! Twelve chapters covering the important highlights of software engineering, using UML when a design notation is needed and Java when a snippet of code will make things clear. The book does not go into all the gory details of UML notation semantics and