morphic function test. Good definitions help one to readily recognize which pattern to apply. There is also a well written section on inheritance related bugs, carried through to chapter 11 on testing reusable components. Table 11.1 shows the six languages compared with respect to performance across abstract and generic class. The advantage of Java over the others is apparent. Chapters 13 and 14 build subsystem test into system tests. Keep in mind that subsystem and system are defined by the author for his purpose. Chapter 15 is devoted to regression where regression is defined as *once having passed a baseline test but now has a modified component.* The system with modified component is said to be in need of regression testing.

Section 4 Test Models is composed of chapters 16 through 19. The purpose of this section is to help the reader develop application specific test automation. Since it is application specific, it may be of limited value for those that fall outside the authors application set. The book is a little quirky in the way chapters are developed. Chapter 16 is only five pages long compared to chapter 10s 175 pages. The author takes literary license here , but perhaps allowance should be made for the complexity of the topic. Chapter 17 introduces a interesting concept of building in test code into the application. The author says that a assertion is a Boolean expression that defines the necessary conditions for correct execution. Basically these are checks and double checks, and they make good sense in some applications such as an operating system. To bolster this view the author gives some amazing statistics on IBMs OS/400 operating system. Chapter 18 is on test oracles. A test oracle is a code fragment of a test code system that produces the results of a test case. An oracle is something that can be trusted to deliver the expected results. Several patterns of oracles are discussed from parametric to gold. He foes on to tackle automated test result evaluation. Some of these concepts make sense for a trusted operating system but not for an application running under a trusted OS but the author does not make that distinction. Chapter 19 is on the topic of test harness design. A test harness is the nomenclature of a set that contains test case, test suites, stubs, drivers and control systems. This is all the elements of a test system. When the system is said to support effective and repeatable automated testing it becomes a harness. Funny how hardware terminology weans its way into software engineering.

A software designer will most likely be fluent in more than one language but probably not all six as used here. Given the power of Java, it may be a designers second language. References are made to Java and high speed RDB on clustered systems. The reader is left with the impression that Java is the most powerful O-O language in use today, but the other five are still powerful. Thus all six are deserving of the full test and verify treatment. This book belongs on every serious software designers library where it should be used as reference.

Reviewed by Claude Caci, Lockheed Martin – claude.caci@lmco.com

Theories of Programming Languages

John C. Reynolds

Theories of Programming Languages is written by John C. Reynolds, and published by Cambridge University Press, 1998, ISBN 0-521-59414-6, 500 pp. $49.95

To be very clear, this is not a book about computer programming languages. Rather, the primary topic of this book is how to reason about computer programming languages.

I was expecting a text similar to most programming language texts I've read; texts focusing on syntax, structure and implementation of different historical and currently popular computer languages. I had expected a comparative languages text. In hindsight the title should have warned me.

This book is devoted to programming language theory. In the course of the entire text, I can remember only one example using a code fragment in a real world programming language; most of the ideas are progressively developed through creation of an entirely artificial language.

Although the cover notes suggest this is a advanced undergrad/beginning graduate student course text, the author makes it clear in the preface that the book's primary audience is doctoral level students. The back cover also states that the book assumes only elementary programming and mathematics; this is true, in that there is an appendix that provides an introduction to the mathematic theories and notations used throughout the text.

However, if you are not comfortable reading formal notations (or like me, just rusty) this book is something of a struggle to follow. The formulas and proofs get quite busy, as the author adds the ideas (and corresponding notations) for state, environment and continuations to the semantics presented.

Will you benefit from reading this book? If you're a graduate student, the answer is most likely yes. This book expands theoretical concepts that were only touched in my graduate programming languages course.

As a practicing programmer, I can't say the concepts presented are going to find their way into my code, no more than previous study of formal logic and proofs have found their way into my code (occasionally I use predicate logic to restructure a complex `if` construct; frequently I'm tempted to formally prove the bug *cannot* be in my algorithm!). I suspect that if I were working on a tool to prove formal program correctness I would turn to this as a resource.

More important, at least in my world, this book seems to focus on functional programming languages, while we are primarily an object-oriented C++ shop. In a previous lifetime I spent a lot of time with LISP and Scheme, so this book was comfortable; I was able to fill in gaps in my knowledge of the lambda calculus. I went back to SICP[3] and found a lot of overlap in topics (for example Sec 4.2.2 in SICP *An*

[3]Structure and Interpretation of Computer Programs, Abelson, Sussman and Sussman, The MIT Press

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.