The Book Review Column[1]

by William Gasarch
Department of Computer Science
University of Maryland at College Park
College Park, MD, 20742
email: gasarch@cs.umd.edu

In this column we review the following books.

1. **Proofs that Really Count: The Art of Combinatorial Proof** by Arthur T. Benjamin and Jennifer J. Quinn. Reviewed by William Gasarch.
2. **Types and Programming Languages** by Benjamin C. Pierce. Review by Mats Kindahl
3. Joint review of **Introduction To Natural Computation** by Dana H. Ballard and **Mathematical Methods in Artificial Intelligence** by Edward A. Bender. Reviews by Lawrence S. Moss.

If you want a FREE copy of one of these books in exchange for a review, then email me at gasarchcs.umd.edu

Reviews need to be in LaTeX, LaTeX2e, or Plaintext.

Books on Algorithms I want Reviewed

1. Graphs, Networks, and Algorithms by Jungnickel.
2. *Algorithms: Sequential, Parallel, and Distributed* by Berman and Paul.
3. *A Course on Computational Algebraic Number Theory* by Henri Cohen.
4. *Algorithms: Design Techniques and Analysis* by Alsuwaiyel.
5. *Computational Techniques of the Simplex Method* by Maros.

Books that are NOT on Algorithms that I want Reviewed

1. *Introduction to Coding Theory* by Juergen Bierbrauer.
2. *Aspects of Combinatorics and Combinatorial Number Theory* by S.D. Adhikari.
3. *Introduction to Information Theory and Data Compression* by Darrel Hankerson, Greg Harris, Peter Johnson.
4. *Block Error-Correcting Codes: A Computational Primer* by Xambo-Descamps.
5. *Combinatorial Designs: Constructions and Analysis* by Stinson.
6. *Dynamic Reconfiguration: Architectures and Algorithms* by Vaidyanathan and Trahan.
7. *Semantic Integration of Heterogenous Software Specifications* by Martin Groβe-Rhode.

Review [2] of

Proofs that Really Count:
The Art of Combinatorial Proof
Author: Arthur T. Benjamin and Jennifer J. Quinn
Publisher: MAA, 2003
$43.95, Hardcover

Reviewer: William Gasarch

Abbott has been teaching Costello combinatorics.

Abbott: Costello, how many subsets are there of $\{1, \ldots, n\}$?

Costello: Oh. You can either choose 0 elements, or choose 1 element, or choose 2 elements, etc. So the answer is $\sum_{i=0}^{n} \binom{n}{i}$.

Abbott: Well ... let me show you a different way to do it. The number 1 is either in the set A or not, so thats 2 choices. Then the number 2 is either in the set A or not, so thats 2 choices, etc. So the final answer is $2 \times \cdots \times 2 = 2^n$. So, Costello, you did the problem your way, I did it my way, and we got different answers. What can you conclude?

Costello: That one of us is wrong?

Abbott: No. We've shown. $\sum_{i=0}^{n} \binom{n}{i} = 2^n$.

Costello: Really! I don't believe that! Prove it!!

Abbott: We did!

Costello: When?

Abbott: Just now.

Costello: What!?

Abbott: Whats on Second.

Costello: Who?

Abbott: Who's on first.

Costello: (Ignoring reference) Usually when I do a math problem two ways and get two answers I assume one of them is wrong and try to find my error. Its better than what a friend of mine did in elementary algebra— do a problem three times and then take the average.

Abbott: In math you can sometimes prove that two things are the same by solving the same problem two different ways.

Costello: No way!

Abbott: Way!

Costello: I'd like to read more about this. Do you have a book to recommend?

Abbott: Yes. You should read **Proofs that Really Count: The Art of Combinatorial Proof** by Arthur T. Benjamin and Jennifer J. Quinn.

Costello: Can you give me some examples from the book?

Abbott: Okay. Here is one involving Fibonacci Numbers. Recall that the Fibonaci numbers are defined by $f_0 = 1$, $f_1 = 1$, and $(\forall n \geq 2)[f_n = f_{n-1} + f_{n-2}]$. I will show that, for all $n \geq 0$, the following holds:

$$\begin{aligned} f_{2n} &= f_n^2 + f_{n-1}^2 \\ f_{2n+1} &= f_n f_{n+1}^2 + f_{n-1} f_n \end{aligned}$$

Costello: That looks interesting. But is it?

Abbott: Yes. Normally to compute f_n you use dynamic programming and compute $f_0, \ldots, f_n$; however, by Theorem 1, you can compute f_n with far less computations of intermediary values.

[2] © William Gasarch, 2005

Costello: Does the book mention this?
Abbott: No. The book is actually a math book. But it should interest computer scientists. Now, onto the proof.

We will interpret the the Fibonacci numbers in a new way. Picture that you have an array of n squares. You want to tile it with 1×1 squares and by 1×2 dominos. Let t_n be the number of ways this can be done. Note that $t_1 = 1$, $t_2 = 2$. We define $t_0 = 1$. Note that you can tile t_n by either having a 1×1 as the last tile (so there are t_{n-1} ways to do that) or by having a 1×2 as the last tile (so there are t_{n-2} ways to do that). Hence $t_n = t_{n-1} + t_{n-2}$. Since the initial conditions match and the recurrence math that of the f_n we have we have $t_n = f_n$.

We show $t_{2n} = t_n^2 + t_{n-1}^2$. Consider a tiling of an array of size $2n$. There are t_{2n} ways to do this. Either there is a 1×2 tile in the middle or there is not. If there is then you still have to tile two $(n-1)$ arrays. This can be done in t_{n-1}^2 ways. If there is no such tile in the middle then you have to tile two n arrays. This can be done in t_n^2 ways. Hence the total number of ways to tile a $2n$-array is $t_n^2 + t_{n-1}^2$. Since $t_n = f_{n+1}$ we have the first result. The second result is proved in a similar way.
Costello: That was fun! Do you have more?
Abbott: (sarcastic) Does a Chicken have lips?
Costello: (serious) Actually, does a chicken have lips?
Abbott: Er, never mind. Here is a nice lemma that will help us count how many elements in the nth row of Pascal's triangle are odd.
Lemma: For all $k \leq n$ the following hold.

1. $\binom{2n}{2k} \equiv \binom{n}{k} \pmod 2$.
2. $\binom{2n+1}{2k+1} \equiv \binom{n}{k} \pmod 2$.
3. $\binom{2n+1}{2k} \equiv \binom{n}{k} \pmod 2$.
4. $\binom{2n}{2k+1} \equiv 0 \pmod 2$.

Hence, mod 2,

$$\binom{n}{k} \equiv \begin{cases} \binom{\lfloor n/2 \rfloor}{\lfloor k/2 \rfloor} & \text{if } n \equiv 0 \text{ and } k \equiv 0 \ ; \\ \binom{\lfloor n/2 \rfloor}{\lfloor k/2 \rfloor} & \text{if } n \equiv 1 \text{ and } k \equiv 0 \ ; \\ \binom{\lfloor n/2 \rfloor}{\lfloor k/2 \rfloor} & \text{if } n \equiv 1 \text{ and } k \equiv 1 \ ; \\ 0 & \text{if } n \equiv 0 \text{ and } k \equiv 1 \ . \end{cases}$$

Note that the operator that maps n to $\lfloor n/2 \rfloor$ just removes the last bit. Note that in three of the four cases the last bit is removed from n and k.
Proof: We prove the first item; the rest are similar.

How many strings are there in $\{0,1\}^{2n}$ that have exactly $2k$ ones? The answer is $\binom{2n}{2k}$. How many of these are palindromes? If you determine the k ones in the first n positions then the rest of the string is determined, so the answer is $\binom{n}{k}$. Let $NONPAL(2n, 2k)$ be the number of nonpalindromes in $\{0,1\}^{2n}$ that have $2k$ ones. Hence

$\binom{2n}{2k} = \binom{n}{k} + NONPAL(2n, 2k)$

Recall that if x is a string then x^R is the string reversed. For every $x \in NONPAL(2n, 2k)$ we have that $x^R \in NONPAL(2n, 2k)$ and that $x \neq x^R$. Hence $NONPAL(2n, 2k)$ is even. Therefore

$\binom{2n}{2k} \equiv \binom{n}{k} \pmod 2$

End of Proof

By the Lemma, the following is true if all we care about is parity of $\binom{n}{k}$:

1. if n ends in 0 and k ends in 0, we can delete both of those bits,

2. if n ends in 1 and k ends in 0, we can delete both of those bits,

3. if n ends in 1 and k ends in 1, we can delete both of those bits,

Lets say I want to know all the numbers i such that $\binom{76}{i}$ is odd. Lets look at $\binom{76}{52}$. In base 2 this is

$$\binom{1001100_2}{0110100_2} \equiv \binom{100110_2}{011010_2} \equiv \binom{10011_2}{01101_2} \equiv \binom{1001_2}{0110_2} \equiv \binom{100_2}{011_2} \pmod 2.$$

We now have the one case where you cannot delete anything; however, by our Lemma we know

$$\binom{100_2}{011_2} \equiv 0 \pmod 2.$$

The only way that $\binom{76}{i}$ will be even is if there is a bit-place j such that 76 in base 2 has as its jth bit 0, and i has as its jth bit, 1. Hence we know that, for any choice of $a, b, c, d \in \{0, 1\}$ the number

$$\binom{1001100_2}{a00bc00_2} \equiv 1 \pmod 2.$$

We also know that any value of i that is not of the form $a00bc00$ will cause $\binom{1001100_2}{i} \equiv 1 \pmod 2$. There are $2^3 = 8$ choices for a, b, c. Hence there are 8 values of i such that $\binom{76}{i}$ is odd.

The above reasoning generalizes to the following theorem.

Theorem: The number of odd elements in the nth row of Pascals triangle (that is, the number of odd elements in $\{\binom{n}{0}, \ldots, \binom{n}{n}\}$) is 2^m where m is the number of 1's in the binary expansion of n.

Costello: Wow! Do you have any that involve the Stirling Numbers?

Abbott: Do I ever!

Recall that the Stirling Number $\left[{n \atop k}\right]$ is the number of ways to place n people at k identical round tables (no table is empty). Also recall that $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$.

Costello: How are you going to relate the Stirling Numbers to the Harmonic Numbers? The Stirling numbers are integers while the Harmonic Numbers are not.

Abbott: Have patience.

Theorem: $\left[{n+1 \atop 2}\right] = n!H_n$.

Proof: How many ways can you put $n + 1$ people at two identical round tables? One answer is $\left[{n+1 \atop 2}\right]$. Another answer is as follows: Each person is a number. We will call the table with the number 1 at it the Left Table. We will represent the left table by $(1, x_1, \ldots, x_m)$. So the question is, how many ways can we fill in the rest of the left table, and all of the right table? What if there were exactly $i \geq 1$ at the right table? (Recall that no table is empty so we need not consider the $i = 0$ case.) Then there would be $n - i$ at the left table (we do not count person 1 who is already there), So we would choose i for the right table, which we can do in $\binom{n}{i}$ ways, arrange them at the table, which we can do in $(i - 1)!$ ways (the answer is not $i!$ since the tables are round), then arrange the $n - i$ at the left table anyway you like, which is $(n - i)!$ ways. So the answer is $\frac{n!}{i!(n-i)!}(i - 1)!(n - i)!$.

$$\sum_{i=1}^{n} \frac{n!}{i!(n-i)!}(i-1)!(n-i)! = \sum_{i=1}^{n} \frac{n!}{i!}(i-1)! = \sum_{i=1}^{n} \frac{n!}{i} = n! \sum_{i=1}^{n} \frac{1}{i} = n!H_n.$$

End of Proof
Costello: Wow! Blows my mind! You should write a review of that book for SIGACT NEWS.
Abbott: I just did.
Costello: When?
Abbott: Just now.
Costello: Who should read this book?
Abbott: Who's on first.
Costello: Never tell the same joke twice.
Abbott: Oh. Right. Anyway, the book contains many identities similar to the ones given above. In all cases they prove combinatorial identities *not* by induction and *not* by algebraic manipulation, but by showing that two quantities solve the same problem. Even though it was written for a math audience the book will profit anyone who has had a basic course in combinatorics, who is happy with proofs, and who wants to see some elegant proofs of unbelievable identities.
Costello: Good, put that in the review.
Abbott: I just did.

Review of **Types and Programming Languages**
Author: Benjamin C. Pierce
The MIT Press, 2002, 623 pages
Review written by Mats Kindahl[3]

Overview

Type systems and type checking are playing an increasingly important role in the development of programming languages. Almost every programming language developed today are developed with type systems as an integral part of the language.

The goals of this book are to cover all important core topics of and give useful examples of how to use and implement type systems for programming languages. The book is separated into several parts, each part with coverage of one important core topic. After the introduction of lambda-calculus and some basic formalism for describing languages, the book presents a simple type system for lambda-calculus. In the following parts, the type system is then extended to cover recursive types, subtypes, polymorphic types, and higher-order polymorphic types (of the form that the purely functional language Haskell uses).

In the course of defining and extending the type systems, the author gives several detailed examples of how to use them. The examples are interesting and to-the-point. Each important topic contain at least one in-depth study of an either an application or an example type system. In the book, there are case studies of both functional and imperative versions for objects (as in object-oriented programming) and also an introduction to *Featherweight Java*, which is a minimal core system for modeling Java's type system.

[3]

Coverage

Untyped Systems

The first part of the book introduces a simple language to express untyped arithmetic expressions. The purpose of this part is to introduce fundamental concepts for the description of syntax and semantics of programs.

The subjects introduced are basic and can be skipped by anybody familiar with the subjects lambda-calculus, functional programming, and basic program theory. I would, however, recommend reading this part: there are several interesting tidbits that you might not be familiar with, unless you are already have experience in the theory and implementation of functional languages.

The first chapter, *Untyped Arithmetic Expressions*, starts off by presenting notations and methods for describing and reasoning about languages. The first notation introduced is the traditional BNF notation. It continues by explaining the nature of BNF as a *metalanguage* for describing languages and also presents some other notations—inductive definitions, definitions by inference rules, and concrete definitions as sets—but do not delve too deep in them. Here is also where the first encounter with structural induction, used to handle the terms of the language.

Several notions for the description of semantics are presented: operational, denotational, and axiomatic. In the sequel, the operational semantics are used to describe the evaluation of the terms of the language. In the next chapter, *An ML implementation of Arithmetic Expressions*, an implementation of the untyped arithmetic is demonstrated.

The part continues with *The Untyped Lambda-Calculus*, introducing the traditional lambda-calculus, including beta-reduction using different evaluation strategies (full, normal order, call by name, and call by value). The chapter continues with introducing Church encodings of booleans and numerals. Finally, the formal definition of the syntax and evaluation of the untyped lambda calculus (called λ) is introduced.

In preparation for the ML implementation of the untyped lambda-calculus, the chapter *Nameless Representation of Terms* introduces *de Bruijn terms*, which is a representation of lambda-terms that eliminates the need for names (and therefore also eliminates scoping problems). The chapter is followed by *An ML Implementation of the Lambda-Calculus* using the above methods for implementing evaluation of lambda-terms.

Simple Types

This part gives reasons for using types and introduces a basic typing systems for lambda-calculus terms. An interesting theoretical results is presented: well-typed terms are *normalizable*—i.e., that every well-typed terms is guaranteed to halt in a finite number of steps.

The part gives a sound and thourogh introduction to the theoretical and practical issues on typing by presenting a simple type system and demonstrating how to implement it in actual code. In addition to giving an informal explanation the description of each extension and/or alteration is given as a scheme with syntax, evaluation, and typing rules; where each change or addition is clearly marked. By starting with simple typing issues, the author manage to give a clear presentation of type systems and their implementation; in addition this gives the reader a gentle introduction to the notation used.

In the chapter *Typed Arithmetic Expressions*, the idea of using typing as a method to achieve safety is presented and Harper's *Safety = Progress + Preservation* slogan is presented and formalized. In the chapter on the *Simply Typed Lambda-Calculus* types and typing rules are added to the previously introduced lambda-calculus, giving a typed lambda-calculus (called $\lambda_{\rightarrow}$). This chapter

also presents the *Curry-Howard Correspondence*, demonstrating the correspondence between logical propositions and types. After discussion of the implementation of $\lambda_{\rightarrow}$ in *An ML Implementation of Simple Types* several simple extensions to $\lambda_{\rightarrow}$ are given in *Simple Extensions*. This chapter demonstrates how to add several interesting extensions to $\lambda_{\rightarrow}$: *base types* and the *unit type*, *ascription*, *let bindings* and its complications, *pairs*, *tuples*, *records*, *sums* and *variants* (as in *variant records*), *general recursion*, and (finally) *lists*.

In the chapter *Normalization* the author proves that every well-typed term is normalizable. After that, some more complicated extensions are demonstrated: in *References* it is shown how to extend the typing system with reference types and in *Exceptions* the effect of adding exceptions are discussed.

Subtyping

With the introduction of various object-oriented languages, the need for type systems able to handle subtyping became vital. In this part, the author describes the intricacies of creating a type system for subtyping. Although many languages use some form of *nominal* subtyping systems—with Ada 95 perhaps having the most rigorous type system—the author have chosen to describe *structural* subtyping systems. In a nominal type system, every type have to be defined prior to use. The definition includes the name and subtype relation, effectively meaning that the *name* of the type defines the type. In a structural type system, the structure of data determines the type. For example (using Pascal notation), in the definitions below, `a` and `b` have different types in nominal type systems—a new type name is created for each structure—but they have the same type in structural type system.

a : **record** x:Integer; y:Real **end**;
b : **record** x:Integer; y:Real **end**;

In the first chapter, *Subtyping*, the author describes the subtyping relation as a *preorder* on the types and demonstrates how to extend $\lambda_{\rightarrow}$ with subtyping $\lambda_{<:}$. The basic $\lambda_{<:}$ system is not very interesting since it does not have any subtype structures. To create an more interesting language, the author defines a subtype relation on records and adds these to the $\lambda_{<:}$. He continues by explaining that the subtype relation can be extended to a lattice by adding a bottom element (the top element is already present in $\lambda_{<:}$), but also demonstrates how this complicates the problem of building a type checker for the language. He the explains how to extend $\lambda_{<:}$ with ascriptions and casting, base types, variants, lists, references, and the new types *arrays* and *channels*. He concludes the chapter by giving an alternative to subset semantics used for describing subtyping above: namely *coercion semantics*, where the compiler subtyping is replaced with run-time coercions changing the representation of types. This method is used in, for example, modern C and C++ compilers.

The chapter *Metatheory of Subtyping* discusses the algorithmic aspects of creating a type-checking algorithm for a type system with subtypes, which the succeeding chapter *An ML Implementation of Subtyping* uses to implement a type-checking system for into $\lambda_{<:}$.

In the chapter *Case Study: Imperative Objects* the author demonstrates how to create and use the type system to implement a functional version of objects with both methods and instance variables and the chapter *Case Study: Featherweight Java* presents and describes a minimal core calculus for modeling Java's type system that was proposed by Igarashi, Pierce, and Wadler in 1999.

Recursive Types

Simple types are sufficient for many purposes and also have the nice property that every well-typed term is normalizable. Unfortunately, a simple (non-recursive) type system fails for some quite trivial examples, such as trying to express the type for a binary tree (in fictious Pascal notation):

```
type Tree = record
    case leaf: (data:String);
    case node: (left,right:Tree)
end;
```

The remedy is to extend the type system with operators to express *recursive types*, which this part explains in detail.

In the first chapter, *Recursive Types*, the advantages of recursive types are discussed and several examples of useful recursive types are presented: lists, streams, processes, and objects. The chapter continues by explaining the difference between iso-recursive and equi-recursive types, but unfortunately a simple, intuitive explanation of the differences is missing. The next chapter, *Metatheory of Recursive Types*, deals with several subjects with regard to recursive types. It starts by introducing the theoretical foundations for recursive types: induction and coinduction in fixed point theory. It continues by introducing the domain of *tree types*, which are then used to represent the subtype relation. The last part of the chapter deals with implementation of iso- and equi-recursive type systems for a recursive type system with subtypes.

Polymorphism

The term *polymorphism* refer to language constructions that allow parts of a program to work with several different types in different contexts. For example, the following procedure 'maparray'[4] (in a fictious Pascal) is an example of using *universal types*.

```
procedure maparray[type T](var a:array[0..10] of T;
                                procedure p(var x:T));
var i:0..10;
begin
    for i := 0 to 10 do p(a[i])
end;
```

The first chapter, *Type Reconstruction*, develop a more powerful *type reconstruction algorithm*. In contrast to the *type checking algorithms* used prior in the book, this algorithm is capable of computing the *principal type* of a term. The algorithm is built around a constraint-based typing algorithm and uses unification to compute solutions to constraints. The chapter also discusses a simpler form of polymorphism known as *let-polymorphism*. In the *Universal Types* chapter, a more general form of polymorphism, known as *System F*, is developed. System F permits abstracting out *types* out of terms in the same fashion as the lambda-calculus permits abstraction of terms out of terms. Since this means that the term is universally defined for any type, the term is typed with a *universal type*, written $\forall A.T$ (where T is a type containing the free *type variable* A). Since terms can have universal types—that is, it is well-typed for *any* supplied type—it is natural to ask if it is meaningful to have terms that are well-typed for *some* type. The chapter *Existential Types*

[4] having the universal type $\forall T.\,(\text{Ref }(\text{Array }T) \times (\text{Ref }T \to \text{Unit})) \to \text{Unit}$.

considers this and shows that it is quite meaningful for use in data abstraction and information hiding. The chapter *An ML Implementation of System F* the author demonstrate how to extend $\lambda_{\rightarrow}$ with universal and existential types.

When combining subtyping and polymorphism, several problems surface. In the chapter *Bounded Quantification* the author demonstrates how to combine subtyping and polymorphism into the calculus $F_{<:}$ and in the chapter *Case Study: Imperative Objects, Redux* he extends on the previous chapter on imperative objects, demonstrating the role of polymorphism and bounded quantification.

The part is concluded with *Metatheory of Bounded Quantification* where the type checking algorithm for $F_{<:}$ is developed.

Higher-Order Systems

Using the polymorphic system above we can express, for example, lists of elements of type X as the type $\forall R.(X \rightarrow R \rightarrow R) \rightarrow R \rightarrow R$ (on page 351)—which is not very readable. This part introduces *type operators* as operators from types to types. For instance, we can define the type operator `List` by

$$\texttt{List}\ X = \forall R.(X \rightarrow R \rightarrow R) \rightarrow R \rightarrow R.$$

In the first chapter, *Type Operators and Kinding*, type operators are discussed in depth. The notion of *kinds* as "types for type operators" are introduced. Both type operators and kinds are added to $\lambda_{\rightarrow}$ giving a system named λ_{ω}. In the next chapter, *Higher-Order Polymorphism*, λ_{ω} and System F is combined into System F_{ω}, which is then, in chapter *Higher-Order Subtyping*, combined with subtyping to yield System $F^{\omega}_{<:}$. The last chapter of the part, *Case Study: Purely Functional Objects*, is a in-depth case study of typing purely functional objects in System $F^{\omega}_{<:}$. The objects permits using all the features described in the book—especially considering subtyping, universal and existential types, and type operators—and also includes *polymorphic update* to handle different representations of the object while retaining the objects original behavior.

General impression

On the general level, the book is well-written and well-structured. The text is easy to follow and, as an extra help, there are small exercises interleaved with the text so that you can test your understanding of the subject. Although the subjects can at times be complicated, no previous experience of type-theory is assumed—you are provided with an understanding of the issues involved before the complicated subjects are introduced.

The intended audience of the book are mature undergraduate and graduate students and also researchers specializing in programming languages and type theory. It is my opinion that the author has managed keep the intended focus. The book is suitable for mature compute scientists but does not require any previous understanding of type theory.

There are not really any bad sides to the book, but there are a few issues that should be mentioned. The ML implementations contain a lot of extraneous pieces of code that can obscure the issue at hand. For example: (1) there are error handling code needed in a real implementation but which makes the code hard to follow and (2) variables used in the actual implementation but which serves no purpose in the code shown. Another issues is that the book is almost entirely in a functional setting (in contrast to an imperative setting). The advantage is that problems with type systems are "out in the open", but the notation is sometimes unwieldy and very hard to follow.

Joint Review of

Introduction To Natural Computation
Dana H. Ballard
ISBN 0-262-52258-6
336 pp.
$40.00 (paperback)
MIT Press March 1997

and

Mathematical Methods in Artificial Intelligence
Edward A. Bender
IEEE Press, 1996
664 pages, 7" x 10" Hardcover
ISBN 0-8186-7200-5,January 1996
$38.21

Reviewed by Lawrence S. Moss[5]

One of the interesting trends in theoretical computer science in recent years is the turn towards new application areas. It is not strange to see career shifts from complexity theory to cryptography, from type theory to security, from formal language theory to computational biology. The trend is reflected in journals as well. For example, the newly-open-for-business electronic journal *Logical Methods in Computer Science* has editors in the "emerging topics" of quantum computation and logic, and in computational systems in biology. *Theoretical Computer Science* recently added to its two sections (algorithms, automata, complexity and games; and logic, semantics and theory of programming) a third section entitled "Natural Computing". This "is devoted to the study of computing occurring in nature and computing inspired by nature. ... it will contain papers dealing with the theoretical issues in evolutionary computing, neural networks, molecular computing, and quantum computing." (Interstingly, SIGACT News seems to be immune from this trend, at least so far.) Finally, we turn from official journals to decidedly unofficial sources: weblogs summaries of panel discussions at conferences. On May 14, 2004, the Columbia/IBM Research/NYU Theory Day held a panel on "The Future of CS Theory." As reported in Lance Fornow's Computational Complexity Web Log, Richard Karp "highlighted three areas of interest for the future: (1) the study of large scale distributed systems such as the Web, incorporating ideas from economics and game theory; (2) connections with areas of natural science, ranging from statistical physics to quantum mechanics to biology; and (3) the 'new face' of AI in which stochastic and graphical models and statistical inference are playing a big role."

The books under review are not addressed to the theoretical computer science community. They are textbooks pertaining to the adjacent areas that Karp mentions as areas of interest for the future. The third area, some newer parts of AI, is of primary interest here, though the books also connect to the first two areas. My suspicion is that some readers of this column will have already gotten interested in these fields, or they might wish to do so later. Also, some readers will eventually teach courses on the background areas. This was my reason for getting involved in these books: I have taught a graduate course in Mathematics and Logic for Cognitive Science twice, each time using one of the books above as a textbook. The course drew students from a number of areas in addition to Cognitive Science, including curious students who otherwise specialize in logic or CS

[5]

theory. In any case, my reviews are partly about the books themselves, and partly about their use in classroom settings.

Natural Computation is a book with a purpose. Ballard aims to present a set of learning models and to relate them to computational theories of the brain. It might be good to list the general topics in the order of the book: minimum description length, discrete and continuous probability, information theory, heuristic search, two-person games, eigenvectors, clustering, linear systems, optimal control, Hopfield and Kanerva memories, perceptrons, competitive learning, Markov models, hidden Markov models, Markov decision processes, genetic algorithms and programming.

In addition to these topics, the connection to the brain is always in the author's mind; it tends to open all of the chapters. For example, part II of the book is called "Memories". The opening pages of this part are about the hierarchical organization of the visual cortex, complete with a table of specific areas and functions in the brain. But quickly, the material turns to models of neurons, and then the real work in that part of the book begins, with a chapter on Hopfield nets and Kanerva memories. Similarly, part III opens with a discussion of "programming" in basal ganglia, and then goes on to Markov models and decision processes without a return to the hippocampus. One wonders whether the discussion on the brain is a motivation or an excuse for the mathematics that follows. To be sure, the mathematical models of memory found in the book are more relevant to cognitive science than what we normally see in theoretical computer science. But I would like to have seen more of a discussion of the remaining distance between neural net models and brain computation. For this, I found the following paper more enlightening: Wolfgang Maass, "Neural computation: a research topic for theoretical computer science? Some thoughts and pointers", *Bulletin of the EATCS*, volume 72, pages 149-158, 2000.

An interesting and suggestive point made by the book is that it pays to examine three kinds of learning: developmental, behavioral, and evolutionary. These exist on very different time scales. The developmental work for humans is mostly done in the first year. Behavioral learning for the most part occurs later, and it is that which is modeled by algorithms from learning theory. The evolutionary scale is associated with genetic algorithms and (to a lesser extent in the book) with evolutionary game theory. A telling remark on p. 68 concerns the relation of all of this with the studies of reasoning that we find in logic-based AI: "This paradigm is sometimes seen as being at odds with the natural computation paradigm, but the explanation is more likely just that the problems are defined at a more abstract level of abstraction [sic]. Harkening back to Newell's organization of behavior into different timescales . . ., the suggestion is that logical inference operates at the 10-second timescale whereas the focal timescales of this text extend to the 100-millisecond and one-second scales." It would be exciting to embed somehow the traditional concerns of logic and theoretical computer science into a larger project that sees reasoning as one process in a group including perception and evolution. One infers from the book that such a project could exist, and there are even hints of what one might do to get started. But the book does no more than hint.

The author has a one-sentence description of the book on the web, "The scope of this book is to introduce the different algorithms that are used to describe behavior together with the mathematics on which these algorithms are based." I am sorry to report that when read as a treatment of algorithms the book is weak, and as an introduction to the underlying mathematics it is not helpful. The overall tone of the book is closer to a set of lecture notes than to a polished presentation. It is full of unexplained notation and shorthand derivations that students will not be able to read. For example, its treatment of probability consists of the definitions concerning discrete probability (an appendix to a chapter on "Fitness"), Bayes' Theorem with an example, and a few pages on continuous distributions leading up to Gaussian approximation to the normal distribution. This

isn't enough for later uses in the book. Another example: the chapter on HMMs is taken from the 1986 survey paper by Rabiner and Juang (as the author notes). I found the older paper clearer to read and better to teach from than Chapter 10 of the book. The same thing happened with the treatment of eigenvectors and their relation to memory and data compression. Other parts of the book also had me scurrying off to the primary sources. Overall, the book is not usable as a textbook.

I hope that my discussion of this book's shortcomings will linger with you only for milliseconds, but that its promise and spirit will rewire some of your circuitry. It is not a book for the ages, and perhaps the author or someone else will write a sequel that has both the sparkle of the original with a much better treatment of all the technical material.

Mathematical Methods in Artificial Intelligence Edward A. Bender has written a rich and engaging book on topics in mathematics that are connected to AI. Although the book has several goals, primarily it sees itself as a textbook. It is a "big book" with lots of detailed examples, discussion, motivational and historical sections. The style is that of a conversationally-presented mathematics book. The book include general remarks on AI and on mathematics, a small bit on complexity theory, trees and search, logic, nonmonotic reasoning, probability theory, Bayesian networks, fuzziness and belief theory, classification, neural nets and minimization, statistics and information theory, and finally, decision trees and search. Typically, the chapters are long. The book could be the text for many different courses because each of the topics listed above could be covered in enough depth that it would take several weeks to do justice to it. It will be less useful as a textbook for survey classes.

It is refreshing to see an AI book that emphasizes the theory underlying many AI topics and not the programming aspects. In fact, the book might have been written to dispel the impression that AI is mostly about programming or data structures. It also has fair and useful comments on the debates going on inside AI. For example, it discusses whether logic and probability are "right" for AI. Arguments on both sides are presented.

I could not hope to use the whole of this book in a course, and so my class ended up mainly using the material on probability, neural nets, Bayesian nets, default reasoning. My feeling is that the book was too hard for my students, despite my own enthusiasm. Overall, it is a wonderful resource for its topics: it sometimes feels like a very perceptive commentary to several numerous fields. It has loads of examples and problems, and as a book on mathematics alone, I think it would be considered excellent. For example, its treatment of discrete probability is very good. Most introductory sources shun provocative conceptual matters like the Monty Hall problem, but Bender dives right in and offers a clear discussion and good exercises. The work on Bayesian nets presents more theory and algorithms than most classes will want to cover (except for classes on this topic). In effect, the author gives the reasoning behind Pearl's propagation algorithms in the singly connected case. Bender's presentation reads as if he has taken the texts of Neapolitan and of Pearl, found a way to present large chunks in as elegant and quick a way as possible. So sometimes the results are not the best known, but they are close. (Again, this has a disadvantage: it's usually too much for beginners, and advanced students will need to go to other books anyway. Fortunately, the book has numerous pointers to the current literature.)

The treatment of logic emphasizes Prolog (though the book does not teach Prolog programming). It covers resolution and Skolemization, for example, and negation as failure. Typical of its informal style, at the end of Chapter 4 it rates first-order logic and Prolog according to a few criteria. They get a "Good" in flexibility, efficiency, and understandability; a "Very Good" in modularity, a "Mixed" in debugging, and an "It depends" in completeness. The logic material also

includes discussion of non-monotonic reasoning, including default reasoning and brief mentions of circumscription. I was less persuaded that those are the "right" topics for the book. I'm also not sure that the treatment of logic via Prolog is the pedagogically best choice, even for AI students. But it's a defensible choice, and the book develops it well.

Concluding remarks It is somewhat awkward to compare the two books under review: they are too different for this to be useful. Overall, both books will be more useful as an instructors' reference in an AI or Cognitive Science class than as classroom textbooks. Bender's book would be a very good secondary textbook in AI courses that use Russell and Norvig's book, for example. As the author notes, much of the mathematics used in AI is simply not covered in standard courses. The book offers a nice view of many different standard topics and will be especially welcomed by researchers desiring to learn or review a new topic. Ballard's book would be useful as a guide to many intersting topics that instructors (and students) would probably have to learn about from other sources. If I were buying books for a library and had to pick one, I would pick Bender's book because it promises to be a reference book that will not be easily superseded. My review didn't get into it, but both books suggest intriguing questions for theorists. Either book could easily convince one that Karp is right in sensing that the future of CS theory lies partly in the direction of applications to AI and Cognitive Science.

Benjamin, Arthur T.Â There are no reviews yet. Be the first one to write a review. 221 Previews. 2 Favorites. DOWNLOAD OPTIONS. download 1 file. ENCRYPTED DAISY download. For print-disabled users. 14 day loan required to access EPUB and PDF files. IN COLLECTIONS. Books to Borrow. Combinatorial Proofs: examples. Combinatorial proof is a perfect way of establishing certain algebraic identities without resorting to any kind of algebra.Â A. T. Benjamin, J. J. Quinn, Proofs That Really Count: The Art of Combinatorial Proof, MAA, 2003. P. Zeitz, The Art and Craft of Problem Solving, John Wiley & Sons, 1999. Combinatorial Proofs. Combinatorial Proofs.