© 2001 Cambridge University Press

Book reviews

Concurrent Programming in ML by John H. Reppy, Cambridge University Press, 1999, ISBN 0-521-48089-2, xv+308pp.

Concurrent ML (CML) is an extension to Standard ML that supports programming with independent threads. Its main contribution lies in combining concurrency with the 'hot' (higher order, typed) capabilities of SML. The language uses synchronous message passing, giving it a distinctive flavour quite different from languages where processes communicate by shared variables.

This book is partly a monograph summarising the results of a long line of research, partly an advanced tutorial on concurrent programming technique, and partly a user manual for CML. The author says the primary purpose of the book is "to promote the use of CML as a concurrent language," and this seems like an accurate description. The target audience of the book is primarily the programming language research community; the reader needs substantial background both in functional programming (particularly in Standard ML) and in concurrent programming.

Although the book was not designed explicitly for teaching, it should make an excellent auxiliary text for an advanced course in concurrent programming or operating systems, as long as the students already have some expertise with Standard ML and know the basics of concurrency. It is clearly too advanced for an introduction to concurrency.

The first major theme of the book is a critical survey and tutorial on concurrent programming, motivating and leading to a presentation of the Concurrent ML language. The book begins with a motivation of the use of concurrency for structuring programs, followed by a high speed tour through a variety of concurrent algorithms. In the course of this discussion, the same problem – a producer/consumer buffer – is implemented with several techniques, all expressed in CML notation, including semaphores, locks and message passing (both asynchronous and synchronous).

The book's second theme is a discussion of the tradeoffs between alternative choices of primitives and a rationale behind the design of Concurrent ML. Two of the most interesting chapters for programming languages researchers are The Rationale for CML and Implementing Concurrency in SML/JJ (Chapters 6 and 10, respectively). These explain how some of the decisions behind CML were made, in particular the choice of message passing and simple synchronous rendezvous.

The third theme of the book, and one of its most valuable features, is a collection of three substantial case studies. These serve as concrete evidence to support the claims made for CML, and they also provide plenty of good examples of how to use the language.

The first case study is the simplest: a make program that analyses the data dependencies among a set of tasks and performs them concurrently when possible. This is a useful application, as it can introduce parallelism (at a very coarse granularity) on an ordinary network of workstations. The second program is a concurrent window system. The window manager divides neatly into one function for each topic; for example, one function handles frames, another buttons, and so on. This is a much cleaner and more lucid approach than an event loop. The final case study implements the Linda tuple space. A major issue here is bridging the gap between the synchronous CML message passing primitives and the asynchronous behaviour of distributed systems.

These case studies are toy programs in a sense, as they don't include all the details and refinements that are found in modern software products. They are nonetheless complete programs. The fact that three interesting projects can be presented concisely using CML is a convincing advertisement for the language. The window manager and Linda system are particularly interesting; I learned a lot about those topics, and found myself thinking more about the application than about the language.

There are two appendices, which give a reference manual for CML and present its semantics, and an extensive bibliography.

John O'Donnell

Structure and Interpretation of Computer Programs, 2nd Ed by Abelson and Sussman, with Sussman, MIT Press, 1996, ISBN 0-262-51087-1, 657pp.

The first edition of *Structure and Interpretation of Computer Programs* (*SICP*) was published in 1986, and very quickly became a classic – some might say a *cult* classic. The emergence of a strongly favourable group of acolytes inevitably produced balancing critical groups, leading to a fairly strong polarisation of opinion about the book. Interestingly, the main criticisms came from two more or less opposite camps: the 'pragmatist' objection was that the book used an old-fashioned programming language, which was based on obscure features (such as higher-level functions, and lambda) and data structures (lists) that nobody uses in the real world; the 'idealist' objection was that the book used an old-fashioned programming language that, despite having some good features and data types (first-class functions and lists), included operations (such as assignment) which were there merely for practical reasons, and which destroyed the underlying purity. However, very few of the criticisms were directed at the book itself, only the choice of programming language: Scheme.

People get confused about what this book is about. Some think that it is about Scheme. Many see it as a book about programming using Scheme. The more enlightened see it as a book about programming, which incidentally uses Scheme as its language of discourse. Once you become familiar with it, however, you come to see it as a book about abstraction, using programming as its source of examples. This is perhaps its main contribution.

1 Content

The overall content of the second edition – chapters, conceptual threads, . . . the story, in fact – follows that of the first very closely, although there is extensive modification of the details which address the few awkwardnesses of presentation in the previous version.

The first two chapters introduce the dual abstractions of procedures and data, respectively, covering all the material that you would expect. The examples are both standard (numerical calculations, lists and trees, etc.) as well as the less common for a first-level programming book (symbolic differentiation, implementing generics, etc.). Of course, since a 'proper' language is being used, issues such as first-class functions and recursion are introduced as easily as the basic data structures. In addition, important topics such as complexity, and the distinctions between names, representations and values, are developed early and naturally. In passing, Scheme is described by introducing any new features as they are required – since there aren't many, this doesn't take long, and the majority of Scheme is dealt with in Chapter 1.

An addition to the first edition is the development and use of a picture language (based on Peter Henderson's 1982 example) to illustrate the use of higher order functions, such as map, as means for abstracting patterns over data.

The third chapter brings together many of the threads of the first two by developing the idea of 'object'. To do this it is necessary to introduce that most contentious of topics (at least to FP aficionados): assignment. This enables the idea of state to be addressed, and hence enables programs which model state-full entities (as exist in the 'real' world) to be written in a natural way. It is worth noting that assignment, or state mutation, is regarded as a difficult advanced topic (over 200 pages pass by before it's introduced), that it is presented as something to be used in a highly controlled way (mutation of *local* state variables using object-style encapsulation), and that the downsides of assignment (a more complex and complicated computational model) are firmly presented.

The additional material in Chapter 3 consists principally of a 20-page section on Concurrency. In essence this section introduces the basic features of concurrent systems, and demonstrates how techniques such as mutexes can be used to control the problems of concurrent programming. The purpose of this section appears to be to introduce *time* as another 'material' that the programmer can use, and must control... indeed, one sub-section is called 'Time is of the Essence'. This leads on to the presentation of *streams* (lazy lists) as an alternative way of dealing with state – more particularly as another way of encapsulating, and hence controlling, the use of state. Finally the chapter compares the object, and functional views of modularity.

This is a book of two halves. The first half consists of the first three chapters, and corresponds to the 'Structure' part of the book's title. The final two chapters constitute the 'Interpretation' of programs. In fact, this division is rather too naive, as both structure and interpretation are themes throughout the book, but there is a fairly clean change of emphasis at this point. There has been only one significant change from the first edition: chapter four now adds a non-deterministic (*amb* plus backtracking) interpreter, to go with the previous three (scheme, lazy scheme, and logic programming). These four cover most of the evaluators that a reader is likely to encounter, and provide an absorbing account of the differences in philosophy and technology between these paradigms. That it is possible to describe and implement all these in one language *naturally* is support for the use of Scheme as the language discourse for this book.

The final chapter is where the nuts and bolts appear – it deals with implementing all the previous interpreters by means of register machines. There has been little change from the first edition, apart from some re-organisation of the topics (which is interesting in itself from a didactic point of view). Included are sections on compilation and garbage collection.

2 Conclusion

Given that the first edition has been extensively reviewed previously, and is well known, it is perhaps best to concentrate on the changes when trying to summarise the qualities of this new edition.

The new material on concurrency is rather a disappointment since such a large and fundamental topic is relegated to little more than a convenient link item between lists and streams. Although it's clear that the authors see the topic as important, the original edition was conceived without concurrency in mind and, since the original structure is so very strong and coherent, it is not surprising that it was difficult to integrate it into the book. A radical rewrite would be required – one can hope that this might be done.

The book has been updated so that the flavour of Scheme used now conforms to the 1990 IEEE standard (in essence, the R4Rs version). At least, *most* of the code is standard – it is not clear that the concurrency code will run on "any Scheme implementation conforming to" the standard, and the graphics additions required to run the picture language examples are MIT-specific. Both problems can be overcome (either by judicious use of Scheme macros and continuations, and/or local implementation extensions), but this does make those sections of the book more or less out of scope for the general reader.

The major omission, and it *is* major, is that types are never fully, or even adequately discussed. This can be seen as the only explicit disadvantage in using Scheme. However, on the positive side, the lack of static types gives educators an excellent way of motivating the need for careful thought at design time when their students' programs fail through type errors.

So, who is this book for? It is for anyone who has a need to understand computer science.

It doesn't tell the whole story – it would be unfair to expect that. However, it does cover most of the essential parts of the story with a clarity and to a greater depth than any other text. Reading this book draws you in quite inexorably, reminding you of things you had forgotten, clarifying things you thought you understood, and showing you how to do things that you considered too difficult.

If you're new to SICP, it's still a 'must read' book. However, there is no real *need* to read the second edition if you're familiar with the first, unless you're going to use it for teaching. In fact the best feature of the book is that there has been little fundamental change. The writing style is still exemplary... despite the silly jokes; the story is still gripping, epic, and inspirational; and the examples (and problems) are eclectic. The joy is still apparent. The new material is interesting, but not outstanding when compared to the 'core' of the book. That said, it's still one of the best books on computer science... it's the book I would most like to have written.

Finally, the binding of the softback is terrible! It is cheap and nasty, and nowhere near robust enough for the sort of use to which the book should be subjected. I already have pages falling out of the second edition where the first is still going strong!

Alan Wood

Introduction to Programming Using SML by M. R. Hansen and H. Rischel, Addison Wesley, 1999, ISBN 0-201-39820-6.

The material contained in this text is used to teach a first programming course for computer science and electrical engineering students at the Technical University of Denmark. The course is intended to introduce basic programming concepts in an informal but nevertheless precise way, using almost exclusively small example programs and exercises. Though the programming techniques are claimed to be language independent (whatever this may mean), Standard ML is used as the programming language of choice for its data structuring facilities, its resemblance to mathematical notation, its well-defined semantics, its interactive interface, and for the availability of various ML implementations on different platforms.

The text sets out, in the first two chapters, with the basic notions of values, types, declarations of identifiers for values and functions, declarations of recursive functions by alternative pattern clauses, and with stepwise evaluation by systematic expansion of function calls with instantiated clauses. It then continues with the notion of expressions and their environment-based evaluation.

Structured data in the form of *n*-tuples and records enter the scene in Chapter 3, which also makes parameter passing through tuple (record) patterns a little more explicit. Next on the agenda, in Chapters 5–8, are (binary) lists and list processing functions, list-specific pattern matching, tagged values, constructor patterns, `case`-expressions composed of sets of pattern matching functions, tree structures, pattern matching on trees, and symbolic differentiation as an interesting application of tree transformations.

Other issues briefly touched upon in between include the declaration of infix functions on argument pairs, type declarations, exception handling, `let`-/`local`-expressions, (polymorphic) types and type inference, partially defined functions, mutually recursive data type and function declarations, and abstract data types.

After 135 pages, the text is finally getting around, in Chapter 9, to giving a more thorough, though not really complete, account of the concept of functions in ML, specifically of functions being first class objects (values). Here it is explained how functions may be represented as `let` or `fn` (value) expressions to pass them as function arguments, and what it takes to declare recursive functions as values. After some more pages of going through the obligatory `map` and `fold` motions one can finally learn something (on roughly three pages) about closures, static binding, and about the strict semantics of ML versus the non-strict evaluation of `if_then_else` clauses.

The following chapters describe, largely by examples as before, how to implement finite sets and tables, how to use ML modules and the stream-based input/output facilities, and how to program dialogs. Finally, in Chapter 18 the line is being crossed between a purely expression-oriented (functional) and an imperative programming style by introducing the notion of a store. It outlines how values in the store may be created and changed by assignments, and how the store differs from the environment in which expressions are evaluated. Some pages are specifically spent on arrays and on how in-place array updating may improve the efficiency of program execution.

In between, the text includes four problem solving chapters. Using exercises of increasing complexity and sophistication, they highlight three phases of systematic program development: problem analysis and formal specification, turning the specification into a working program, and validating it by means of selected test cases.

A complete specification of the syntax and semantics of SML, of the ML module system (including modules for sets and tables), and of selected parts of the ML library is given in appendices.

This is undoubtedly a good text for an SML-based programming course for beginners, as it neither assumes nor requires much prior knowledge of computer programming, let alone of the underlying theory. Students may simply work their way step-by-step through a series of well chosen example programs and programming exercises which should enable them to develop fairly quickly some hands-on experience and competence in writing compact programs in an elegant recursive style. It is also worth studying as a complementary text for an undergraduate course on basic programming (language) concepts.

The emphasis in this text clearly is on ways and means of massaging structured data, obviously in an effort to acquaint students right from the beginning with the essentials of real life application programmming. The most important lesson to be learned from this approach may be that the construction of functions closely follows (in fact, is dictated by) the data structures they are to be applied to, and that program complexity and efficiency of execution are intimately related to the choice of a suitable data model.

Unfortunately, the text is not of a comparable quality in the conceptual department, as one might expect after having read the preface. Fundamental concepts such as functions, pattern matching, variable bindings or environments are introduced piecemeal, more or less casually as need arises, and explained only matter-of-factly, but the full picture of the underlying ideas (why can or should it be done the way it is done) is hardly ever given, not even informally.

To begin with, it is nowhere explicitely stated that SML in large parts is a functional language (not even when, after 263 pages, the text turns to imperative programming), and what exactly this means, e.g. absence of side-effects, referential transparency (context-free substitution of equals by equals), and the role of variables (identifiers) and of variable scoping in this game. Also, it takes 157 pages until, in a subsection explaining how `if_then_else` clauses need to be correctly evaluated (which, strangely enough, is entitled ‘lazy evaluation’), some five lines of text can be found stating that SML has a strict (eager) semantics.

The functions used throughout the text are almost all closed – a point that is never explicitely mentioned, even though this seems to be an essential prerequisite for the correct workings of the rather simple environments introduced in Chapter 2. When in section 3.7 `let`-expressions enter the game, it simply says that bindings for `let`-defined (local) identifiers are added to the environment in which the entire `let`-expression is being evaluated. This, in turn, is supposed to define the environment for evaluating the `let`-body. At this point, the conscious reader might ask what happens with identically named identifiers bound both in the `let`-expression and in its environment? Likewise, what happens with identifiers that occur free in local (`let`-defined) functions and are bound higher up? Answers to these questions may finally be found in sections 9.9 and 9.10, where closures are introduced as internal representations of functions. However, rather than giving a full account of what closures are needed for, the text just describes in a few words how they look like and the mechanics of evaluating them, using as an example the factorial function which, by all means, is too trivial to get the complete message across. It is then briefly outlined how closures can be used to make bindings invariant against re-declaration of identifiers, which is referred to as stating binding. However, it says nowhere that static binding as implied can only be had if variables (identifiers), in contrast to values and functions, are not first class citizens (can neither be passed as function arguments nor returned as function values). It also means that variables must be bound to values before the expressions in which they occur can be evaluated (which is taken care of by the strict evaluation regime), and that partial function applications cannot be evaluated, but instead must be kept around as closures. The latter would also explain why functions returned as function values, other than giving their types, cannot be represented in intelligible form as output – a point that should have been made in the chapter on higher-order functions.

Computer science students should know about these things right from the beginning, and it doesn't require a full excursion into λ-calculus or other heavy theoretical stuff to explain them comprehensibly in an 'informal but precise way'.

Another problem concerns pattern matching. The reader is first faced with patterns composed of (tuples of) variables/identifiers and constant values to define alternative function clauses, then she/he learns about tuple and record patterns, then about list patterns, then about constructor patterns, then about pattern matches as components of `case`-expressions, and finally, about pattern matching on trees (fortunately, one might say, there is no syntactic sugar for pattern matches on sets and tables). Notwithstanding the diversity of notations for ML data structures and the ensuing syntactical diversity of patterns, it would have been a good idea to explain, in some suitable place, the basic concept of pattern matching as an operation that extracts (sub-)structures from some given structural context and substitutes them for free occurrences of pattern variables in other contexts, instead of going repeatedly through essentially the same stuff, adding every time just a tiny little extra twist to it. All the structures that need to be dealt with are essentially trees, or degenerate forms thereof, so why not start from there?

Thus, the problems with this text seem to be in large parts related to the way the material is organized and to the idea of explaining just about everything by means of example programs, not so much to its substance since all there needs to be known about ML programming is essentially said. It may be a suitable form of trying to introduce beginners step-by-step to the intricacies of good program design, but it certainly takes another ordering of subjects and a little more than just a few simple examples to precisely and completely (!) explain concepts.

Everything considered, the text can be recommended as a programming course of undisputable qualities, particularly since it is a rich source of interesting exercises, but it offers not exactly the best possible way of teaching programming concepts, not even to first year students. Ullman's (1994) and Paulson's (1991) books on ML programming are more thorough in this respect, and present the material in a more appropriate order.

References

Ullman, J. D. (1994) *Elements of ML Programming*. Prentice Hall.

Paulson, L. C. (1991) *ML for the Working Programmer*. Cambridge University Press.

Werner Kluge

Programming with Constraints: An Introduction by Kim Marriott and Peter J. Stuckey, MIT Press, 1998.

This book is the first comprehensive textbook on constraint programming, one of the most fascinating application areas of declarative programming. Constraint programming is the result of the amalgamation of techniques developed in areas like Artificial Intelligence, Operations Research and Logic Programming. So, why could it be of interest for functional programmers?

Constraint programming, logic programming as well as functional programming can be seen as different facets of a single idea, namely *declarative programming*. This approach to programming is driven by the idea to specify properties of the problem domain in a high-level language rather than describing the sequence of manipulations of memory cells as in imperative languages. Functional languages use functions as their building blocks to describe problem-specific properties, logic languages use predicates, and constraint languages use constraints over specific domains. In this sense, constraint programming can be seen as the most abstract approach to declarative programming, since constraint programming consists of specifying a set of constraints that each solution to a given problem must satisfy (rather than describing functions to assign to each given input an output). On the other hand, the idea of constraint programming can be combined with other programming paradigms (even with imperative languages) where logic and functional languages seem to be the most natural 'host languages' due to their declarative nature.

A further interesting point for functional programmers is the fact that constraint programming is one of the success stories of declarative programming in the real world. Constraint programming has been successfully applied in many industrial domains, e.g. optimization problems from operations reasearch. Due to its declarative style of programming, constraint programming shows that this programming style can have an impact on solving industrial and commercial problems. Thus, constraint programming is a field where functional programming might also have an impact on real world applications. Although a big part of this book is devoted to constraint *logic* programming, it contains a language-independent introduction and discusses also other constraint programming languages, including extensions of functional languages by constraints.

Although there exists a lot of material about constraint programming in different articles, surveys, and books, there was no comprehensive textbook on constraint programming. The intended audience for this book are fourth year level students but it is accessible to everybody with some knowledge in programming since it does not assume familiarity with logic programming. Due to its comprehensive collection of material about constraint programming, it is likely that this book will be a standard reference for this field.

The book is divided into three parts. The first part provides a general language-independent introduction to constraints and computation with constraints. The second part discusses programming techniques and implementation aspects related to constraint programming by the use of constraint logic programming for concrete examples. The third part surveys the combination of constraints with other programming paradigms.

Part I. This part introduces the notion of constraints and the use of them for problem modelling. Constraints are relationships between variables of a particular domain. Examples are real arithmetic constraints, where the domain is the set of real numbers and primitive constraints are equalities and inequalities between arithmetic expressions, or finite domain constraints, where each variable can take a value in a finite set of possible values (usually natural numbers) and primitive constraints are equalities and inequalities. This part explains basic notions like solvability or satisfiability, and discusses the heart of any constraint language, namely the constraint solver. Algorithms for constraint solving and optimization are provided for various domains, like Gauss–Jordan elimination and the simplex algorithm for real arithmetic constraints, classical unification for tree constraints, or node and arc consistency for finite domain constraints.

Part II. This part introduces Constraint Logic Programming (CLP) as a language to provide rule-based descriptions of constraint problems. Knowledge about logic programming is not assumed since the authors introduce classical logic programming as constraint logic programming with tree constraints, i.e. constraints over algebraic data terms. The operational behavior is explained by the construction of derivation trees wrt the set of program rules. This section provides also a good introduction into programming techniques for constraint programming by various larger examples, like the computation of tree layouts, designing bridges, or solving scheduling problems. Since the efficiency of a constraint program heavily depends on the built-in constraint solver and is, therefore, difficult to determine, the authors discuss in detail different modellings for a given problem. This is not only a collection of good hints for the constraint programmer but a necessary technique in practice since the authors admit in § 8.4 that “given the difficulty of understanding interaction between the constraints and the solver it is often useful to empirically evaluate the different models.” This part of the book finishes with a discussion of advanced programming techniques, features and implementation aspects of CLP systems.

Part III. This part provides a survey of constraint programming with other languages than CLP. This includes constraint databases, where bottom-up evaluation techniques for constraint programming are presented, as well as the addition of constraints to concurrent, functional, or imperative languages, or libraries for constraint programming. Unfortunately, the authors did not include a discussion or references to (constraint) functional logic languages (e.g. see (Arenas-Sánchez *et al*., 1999; Hanus, 1994; López Fraguas, 1992)) which are a conservative extension of the CLP framework and combine the advantages of both functional and logic programming in a single framework.

This book provides an introduction to the practice of constraint logic programming rather than a detailed explanation of the constraint logic programming paradigm and its foundations. The practical orientation of the book is shown by numerous programming examples and a number of algorithms to explain the implementation of constraint solvers for various domains. The underlying logical theory of constraint logic programs and the correctness of the presented algorithms is not shown. This is not necessarily a weak point of the book since including both aspects of constraint logic programming might be too much material for a single book. Thus, it is a book written for people interested in the practice of constraint programming and fills an important gap in the literature in this area. Nevertheless, each section in this book concludes with relevant bibliographic notes so that the reader interested in more background material finds appropriate references.

To make the book a useful introduction to the practice of constraint programming, each section has not only a subsection with exercises but also a subsection with ‘practical exercises’. Since there is no standard constraint logic programming system or language, the authors provide in these subsections many hints about obtaining and using different CLP systems and how the constructs introduced in each section are implemented in the different systems. These parts make the book a helpful source to the practice of constraint programming.

The developments of the recent years have shown that constraint programming is a new fascinating programming paradigm with a big potential for practical applications. Unfortunately, there was no good textbook about this area available and the current book fills this gap. Thus, I can recommend it for everyone who is interested in an area where the idea of declarative programming has made its way into practice and where there is no question about the ‘killer applications’ for this programming style.

References

Arenas-Sánchez, P., López-Fraguas, F. J. and Rodríguez-Artalejo, M. (1999) Functional plus logic programming with built-in and symbolic constraints. *Proc. International Conference on Principles and Practice of Declarative Programming (PPDP’99): Lecture Notes in Computer Science 1702*, pp. 152–169. Springer-Verlag.

Hanus, M. (1994) The Integration of Functions into Logic Programming: From Theory to Practice. *J. Logic Programming*, V**19 & 20**, 583–628.

López Fraguas, F. J. (1992) A general scheme for constraint functional logic programming. *Proc. 3rd International Conference on Algebraic and Logic Programming: Lecture Notes in Computer Science 632*, pp. 213–227. Springer-Verlag.

Michael Hanus

Term Rewriting and All That by Franz Baader and Tobias Nipkow, Cambridge University Press, 1998, ISBN 0-521-45520-0 (hardback), 301pp.

The subject of term rewriting has rapidly matured over the last few decades, as witnessed by the international RTA and TLCA conferences and the recent emergence of IFIP Working Group 1.6 on Term Rewriting.

However, until a few years ago there were no comprehensive text books on the subject. The first and very valuable monograph on term rewriting was published by Avenhaus (1995). The present book covers more ground, and contains many recent results. It is aimed towards students as well as researchers. The exterior and layout of the book is pleasant, and the playful title (suggested by CUP's David Tranah) is in analogy with an amusing history book: *1066 and All That*, that is familiar to many British readers. The material in the book is classroom tested, and that shows: the style of exposition is clear, and the presentation is well structured. Many exercises are included, though solutions are not provided.

The contents are as follows: 1. Motivating examples; 2. Abstract reduction systems; 3. Universal algebra; 4. Equational problems; 5. Termination; 6. Confluence; 7. Completion; 8. Gröbner bases and Buchberger's algorithm; 9. Combination problems; 10. Equational unification; 11. Extensions; Appendix 1, Ordered sets; Appendix 2, A Bluffer's Guide to ML; Bibliography; Index.

A nice feature of the book is the presence of many ML programs, serving to make the student familiar with notions and algorithms. As the authors put it, the book contains a little ML-based term rewrite laboratory: it implements terms, substitutions, unification, matching and term rewriting. Interesting and useful is that the authors treat unification not only on terms, but also on (acyclic) term graphs. In an exercise (4.27), even the unification of cyclic term graphs is briefly touched upon. In general, the book is strong in its treatment of various aspects of unification, including several discussions of complexity issues, and including a chapter on the fundamental issue of equational unification.

The treatment of termination also does credit to that subject, and is certainly complete enough for classroom use. After the well known reduction of the undecidability of termination for term rewriting systems to the (uniform) halting problem for Turing machines, the chapter treats reduction orders, the interpretation method, polynomial orders, simplification orders, the beautiful Kruskal tree theorem and well-partial orders, and concludes with recursive path orders.

The chapters on confluence and completion are also complete and to the point. The confluence chapter treats not only the classical notion of orthogonality, but includes more recent theory in a section 'Beyond orthogonality', on parallel closure and weak orthogonality. The chapter on completion treats the important notion of proof orders.

The book gives most of the proofs, but omits the proofs of some theorems such as Toyama's theorem on modularity of confluence. Certainly for classroom use this is a good policy.

Chapter 8 is very welcome: it treats the basics of Gröbner bases and Buchberger's algorithm, which is very prominent in computer algebra. This makes the book interesting to mathematicians too.

The authors have put some sensible restrictions to themselves ("In der Beschränkung zeigt sich der Meister".) They have not tried to give complete coverage to the state of the art in term rewriting, and thus have excluded some topics that are also of some importance in term rewriting today. The notable restrictions are: higher-order rewriting (where Nipkow is one of the main contributors with his HRSs, Higher-order Rewrite Systems); infinitary rewriting;

and term graph rewriting. Some other topics that are only briefly mentioned in the concluding chapter on Extensions include rewriting modulo an equational theory, strategies for rewriting, and conditional rewriting. The authors are being very realistic here, and clearly state that their discussion of these subjects is only meant to whet the reader's appetite, and provide pointers to the literature.

Summing up: this is a highly welcome addition to the literature on term rewriting, and an important contribution to putting term rewriting 'on the map'. It is a very readable, well written and likeable book. It should be of great value to students and researchers alike.

Reference

Avenhaus, J. (1995) *Reduktionssysteme*. Springer-Verlag.

Jan Willem Klop