Book reviews

Advanced Topics in Term Rewriting by Enno Ohlebusch, Springer Verlag, 2002. doi:10.1017/S0956796805215812

To a functional computer scientist the concepts of normalization, Church–Rosser, confluence, and strong normalization are associated with Church's λ-calculus, either in its typed or untyped form. However these concepts have a wider application that can be found in the field of term rewriting. Although it is arguable whether the motivation for term rewriting originated in the *word problem* investigated by Hilbert's student Max Dehn (Dehn, 1911) or was initiated in the pioneering work of abstract reduction by Newman (Newman, 1942), the book *Advanced Topics in Term Rewriting* (ATITR) begins with an Abstract Reduction System ARS.

An $ARS = (A, \rightarrow)$ is a mathematical structure consisting of a set of objects and a finite set of relations,[1] such that $l \rightarrow r$ has the intuitive meaning that the object $l \in A$ is reduced (rewritten, replaced, etc.) to $r \in A$. Given $\rightarrow$, the reflexive, transitive closure relation $\xrightarrow{*}$ and the equivalence (the reflexive, symmetric, transitive closure) relation $\xleftrightarrow{*}$ induced by $\rightarrow$ can be computed. An object $a \in A$ is in *normal form* when it can not be reduced ($\neg\exists a' \in A.\ a \rightarrow a' \wedge a \neq a'$).

The four interesting system properties of *normalizing* (all objects $a \in A$ have a normal form), *strong normalization (or terminating)* (for all $a, a' \in A$, all the path lengths computed using $a \xrightarrow{*} a'$ are finite), *confluence* (for strongly normalizing ARSs, the normal form can be reached by any reduction sequence), and *Church-Rosser* (objects $a, a' \in A$ that are 'equivalent' under $\xleftrightarrow{*}$ can be reduced to the same object) can be studied. Dehn's word problem is that "given a set of objects (i.e. a finitely presented group), is it possible to find an algorithm that will say which objects are equivalent under a given finite set of equations?" can also be formulated and answered.

Having studied ARSs, the structure $(A, \rightarrow)$ can be changed by replacing the set A with a collection of *terms*. Here, the notion of term comes from Universal Algebra, where $T(\mathscr{F}, \mathscr{V})$ is the set of terms inductively constructed from the set of function (or operator) symbols $\mathscr{F}$ (together with an *arity* function, $arity(f) : \mathscr{F} \rightarrow \mathbb{N}_0$) and a set of variables $\mathscr{V}$ (disjoint from $\mathscr{F}$). Again the relevant properties of normalization, strong normalization, confluence, and Church-Rosser can be formulated and studied.

In ATITR, Chapters 2 and 3 introduce the basic mathematical machinery of ARSs and term rewriting systems TRSs (viz. $(\mathscr{F}, \rightarrow)$) as sketched out above. This is done after a very brief motivation/introduction chapter. Given what has already been said about ARSs and TRSs, the word problem can be solved by demanding that the structure be strongly normalizing and confluent. In this case, two terms on opposite sides of an $=$ sign can be proved equal by reducing both terms to their normal forms (using a well-founded ordering on $\rightarrow$) and testing for syntactic equivalence (maybe in the presence of an additional equivalence relation $\approx$).

Although it is taken for granted that the typed λ-calculus is strongly normalizing, confluent, and equal λ-terms are equivalent under α conversion $\approx_\alpha$, this is clearly not the case for all ARSs and TRSs. In fact, the two key questions – to decide whether a structure is confluent and to decide whether it is strongly normalizing – are both undecidable in general, and so

[1] Specifically, the ARS should be defined as $(A, \{\rightarrow_i\}_{i \in I})$ for some index set I, but it is common practice to use the shorthand $\rightarrow = \{\cup_{i \in I} \rightarrow_i\}$.

the word problem is itself undecidable. It is undecidability of these properties that pushes the development of research in term rewriting.

Chapter 4 deals with the research concerning results on confluence in both terminating and non-terminating systems, while Chapter 5 deals with termination issues. It deals first with the standard techniques and then introduces newer proof techniques such as dependency pairs, semantic labelling, type introduction, and innermost termination. Up to this point (the end of Chapter 5), the material presented in the book is at an advanced undergraduate/beginning graduate level, but it is the remaining chapters that qualify it for an advanced text in term rewriting systems.

Chapters 7, 8, and 9 form the core chapters that a graduate student would be interested in, having mastered the earlier chapters. Conditional rewriting systems are systems where $l \rightarrow r$ is replaced by a conditional rewrite $l \rightarrow r \Leftarrow s_1 = t_1, \dots, s_n = t_n$, with $l, r, s_i, t_i \in T(\mathscr{F}, \mathscr{V})$; in this case l is rewritten to r only when each of the equations $s_i = t_i$ is valid in the TRS. In this case, the concept of termination needs to be changed to effective termination as the evaluation of the conditionals might not terminate. For a functional programmer, conditional rewriting comes into play when the arguments to a function come from an algebraic datatype. Chapter 7 deals with the issue of conditional rewriting while Chapter 8 deals with the question of how the properties of a combined rewriting system $(F_1 \cup F_2\,, \rightarrow_1 \cup \rightarrow_2)$ are related to the properties of the individual TRSs $(\mathscr{F}_1, \rightarrow_1)$ and $(\mathscr{F}_2, \rightarrow_2)$. The author of ATITR has written a number of articles concerning modularity and from the theoretical viewpoint, this chapter is useful as a divide-and-conquer approach to dealing with the properties of large *TRS*s.

Chapter 9 is the only 'practical' chapter in the book as it is in this chapter that the aspects of implementing term rewriting systems is considered. Noting that graph rewriting is different from term rewriting, the book introduces a new framework called *marked rewriting* that bridges the gap. Both conditional and non-conditional term rewriting are covered; but the treatment is still mathematical rather than implementation based. The remaining two chapters of the book deal with cutting-edge research.

Chapter 6 expands on the Termination Hierarchy introduced in Chapter 5. Although termination is in general an undecidable property of *TRS*s, there are certain instances of a *TRS* where termination is decidable. Therefore, instead of asking whether a property Y of a *TRS* is decidable, Chapter 6 introduces the notion of relative undecidability, that is deciding whether a TRS has property Y given the fact that the *TRS* has property X. Finally, Chapter 10 deals with the idea that the termination properties of a logic program can be examined by transforming it into a conditional rewrite system.

In conclusion, there are not many textbooks written in English on term rewriting. Two textbooks that are available are the books by Terese (Terese, 2003) and Baader and Nipkow (1998). Terese is similar in scope to ATITR, and is a collection of articles written by the **TE**rm **RE**writing **SE**minar group in Amsterdam. In contrast to ATITR, Baader and Nipkow's book is frequently populated with exercises and fragments of Standard ML programs illustrating the concepts under discussion, but this is lacking in ATITR. Other things that are missing from the book are any references to concrete term rewriting systems, a chapter on the complexity of the algorithms alluded to by the theorems (usually derived from the number of steps carried out by a Turing Machine (Book, 1993)), or a chapter on higher-order term rewriting. However, it is not possible to address everything in a single book, and if you like mathematics, already know the basics of term rewriting and you are a researcher or a postgraduate then this book is definitely recommended. Otherwise, and from a functional programming perspective, Baader and Nipkow is a better buy.

References

Baader, F. and Nipkow, T. (1998) *Term Rewriting and All That*. Cambridge University Press.

Book, R. V. and Otto, F. (1993) *String-Rewriting Systems*. Texts and Monographs in Computer Science. Springer-Verlag.

Dehn, M. (1911) Über unendliche diskoontinuierliche gruppen. *Mathematische Annalen*, **71**, 116–144.

Newman, M. (1942) On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, **43**(2), 223–243.

Terese (2003) *Term Rewriting Systems.* Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Nimish Shah

The Haskell Road to Logic, Maths and Programming by Kees Doets and Jan van Eijck, King's College Publications, 2004, ISBN 0-9543006-9-6.

doi:10.1017/S0956796805225819

It should come as no surprise that a textbook has been written that attempts to teach foundational math and introduce programming. Indeed, several undergraduate Computer Science departments schedule both courses side by side. This book, unlike those programmes, attempts to teach the subjects as complementary, using one to learn about the other. Requiring only a secondary education in math, this textbook's goal is to teach the reader Haskell programming and theorem proving.

The first chapter serves as a short crash-course in Haskell and the hugs interpreter. Subsequent chapters are each about a mathematical topic and use both text and Haskell code to illustrate these concepts. Chapters two through seven analyze foundational math constructs including sets, relations, functions, and induction. The final four chapters touch on more difficult concepts such as corecursion and infinite sets.

Throughout the book, programs are used to illustrate the text. This is one area where the book really excels. As well as illustrating the text, the programs incite further thinking, ensuring a deeper understanding of the mathematical concepts. For example, when teaching recursive proofs over the natural numbers, they solidify understanding by defining, in Haskell, the natural numbers in terms of successors of zero. This style of integrating programs within the text gave the math an applicable feel, absent from most pure math books.

The coverage of proofs, sets, relations, functions, and induction is gentle and effective. New notation is always accompanied by easily understood explanations. Those explanations are supplemented with common mistakes and a variety of examples. This methodology allows one to learn the math with very little background and external support.

In some instances, because the book focuses primarily on mathematical concepts, the reader is deprived of necessary and basic programming knowledge. An example of this is chapter one, where Haskell is introduced. Although someone familiar with programming could read through the chapter with ease, a beginner would likely stumble on some of the undefined words, like stack-overflow and floating point numbers. Furthermore, additional material about how Haskell relates to programming languages in general could have better set the stage.

The Haskell programs in the book are very concise and directly connected to the math they demonstrate. The high-level math constructs which are applied enhance the reader's ability to write more compact and conceptually elegant programs. Moreover, the reader is forced to think about programming from a declarative point of view, which encourages using higher-level constructs for problem solving. By contrast, most introductory programming books concentrate mostly on successfully solving the problem.

Instead of locating exercises at the end of the each chapter, the authors mixed them within the text and examples. Designed to flow with the text, most of them are quick and have an easy to medium difficulty. The integration of exercises and text gives the book a enjoyable, hands-on feel. Some difficult proof problems, marked with a '*', are scattered throughout the text. Although few in number, they provide excellent preparation for upper level math classes.

On the whole, I think this book would be very successful for use in an introductory college level math course or self learning. The integration of programming provides a unique and enjoyable way to learn math.

David Sankel

The Standard ML Basis Library by Emden R. Gansner and John H. Reppy, editors, Cambridge University Press, 2004, 406pp.
doi:10.1017/S0956796805235815

The Standard ML Basis Library is both the title of this book, and the library which the book documents. This library, "concerns itself with the fundamentals; primitive types such as integers and floating-point numbers, operations requiring runtime system or compiler support, such as I/O and arrays; and ubiquitous utility types such as booleans and lists."

The book solidly targets experienced SML developers, as it is neither a tutorial nor a textbook. It is more than a simple reference, however: it provides context and examples of how to use the library. It is also targeted at implementors of the language, and seems to be the definition of this aspect of the SML language.

The Standard ML Basis Library is a very complete and clear reference for this library. The first quarter of the book is perhaps the most valuable to those not already familiar with the library, as it contains exposition, examples, and idiomatic uses of various parts of the library. The rest of the book contains what the authors describe as the "meat" of the text; the manual pages describing each and every structure and function call in the library. While it is nice to have the manual pages in book format, they are available online in a convenient hyper-linked format.

ML was standardized in 1990 and 1997, and this book represents the 1997 standardization. It was published in 2004 and should be relevant for as long as this is the current ML standard. This book is written in an authoritative style, giving justification for design decisions for both language implementors and users interested in some of the finer details of how the library works.

The exposition section, roughly the first 100 pages, is very well organized and presented. It digs into various aspects of the library like Text, Numerics, and Input/Output. The description of the I/O model, for instance, explains what the I/O subsystem provides, how it is organized, and gives clear, brief code examples for using it.

The code examples are a strong point throughout this section. The authors manage to construct very compact examples, usually between five and ten lines of code, which are nevertheless precise and illuminating. However, the type signatures of example functions are not provided alongside their definitions. I felt that this made them somewhat harder to read.

Even in this section, one must have a working knowledge of SML, since this book makes no attempt to teach SML, or even provide a certain amount of context for explanations. This was probably an explicit choice on the part of the authors, but I felt that they could have given more types, definitions, and some background on items rather than relying on the reader to look up the definitions in the manual pages.

For instance, the description of option type in Section 4.2 does not give its very brief definition (which is, of course in the manual page, and again earlier in the chapter). It does not mention the SOME constructor until pointing out that the "valOf" function can strip it away. In fact it says that "OS.Process.getEnv" returns a string value unless the given name is undefined, rather than saying that it returns a string value wrapped in the SOME constructor. It also does not give the common pattern-matching usage of this type and presents only the "valOf" and "getOpt" styles of using it. It points out that option is type-safe compared to C's NULL pointer convention, but does not tie that into the "getOpt" or pattern-matching usages.

I understand the authors' decision not to include such details in a reference manual, but I felt that the exposition section could have been stronger if it could stand alone. The lack of context gave the impression of sacrificing this section for the sake of the manual pages.

The next 300 pages are manual pages describing each structure and function in the library. As mentioned above, this is available online. Each structure has a brief explanation of its purpose and the type signatures of its interface. It then provides English explanations of each function (repeating its type), and each type (repeating its definition).

For instance, the description of the function "nth" from the List structure reads as follows:

```
val nth : 'a list * int -> 'a
    nth (l, i) returns the ith element of the list l, counting from 0.
    It raises Subscript if i ¡ 0 or i <= length l. We have
    nth (l, 0) = hd l, ignoring exceptions.
```

Overall I found this book to be very well written. The code examples are clear and concise, and the English explanations are likewise very clear. The exposition section is excellent, and could have been expanded. The inclusion of the manual pages make for a complete reference for The Standard ML Basis Library.

Isaac Jones