J. Functional Programming **10** (5): 501–508, September 2000. Printed in the United Kingdom
© 2000 Cambridge University Press

Book reviews

The Haskell School of Expression: Learning Functional Programming Through Multimedia by Paul Hudak, Cambridge University Press, 2000, 363pp, ISBN 0-521-64408-9.

This textbook offers a novel and intuitively appealing approach to teaching functional programming (and Haskell in particular): that is to focus on some interesting application area(s) and develop a sequence of libraries for this area, introducing language concepts as needed. This approach is particularly attractive for Haskell, since one of the major strengths of the language is its power as host language for domain-specific languages.

The theme of the book is multimedia applications. The major part of the book is devoted to the development of a series of increasingly sophisticated tools for graphics and animation. There are also passages on robotics and computer music, but these are considerably shorter and less developed.

The example-driven presentation actually introduces new Haskell features on two different scales. On a larger scale, chapters alternate between three categories:

- an important Haskell (or more generally, functional programming) concept is introduced and discussed;
- a high-level, declarative data type for an application is introduced and a collection of useful application functions developed;
- a low-level, implementation-oriented data type is introduced, typically from a non-standard library, and a translation from the high-level type developed.

In the following overview of contents, we emphasise this alternation.

Chapter 1 introduces programming with functions through examples and computation as rewriting of definitions. Chapter 2 then introduces a simple data type of geometrical shapes together with a function to compute the area of shapes and uses this function to exemplify equational reasoning. This is followed in Chapter 3 by a discussion of a tailored graphics library (available from the book's website), which supplies a `Graphic` data type, functions to build `Graphic` values and a function to render these on screen. Finally, in Chapter 4, a function to translate shapes into `Graphic` values is developed, and the first cycle of chapters is complete.

Several such cycles follow:

- Chapter 5 introduces higher order functions (including map and folds), and Chapter 6 develops a function to compute perimeters of shapes, shifting to a higher-order style of writing function definitions.
- Chapter 7 introduces trees and Chapter 8 a particular kind of trees: geometric regions built from atomic shapes with constructors for union, intersection, translation and scaling. Chapter 9 adds further insight into higher-order functions and Chapter 10 discusses rendering of regions (using a corresponding low-level type from the graphics library and translation into this).
- Chapter 12 introduces type classes (Chapter 11 is a somewhat independent chapter on proof by induction). Chapter 13 starts the discussion of animation, based on the elegant idea that an animation of values of some type (such as shapes or regions) is a function from (real) time to that type. Type classes are put to use, both by instantiating

animations into standard type classes and by introducing application-specific classes. A new function from the graphics library is introduced, permitting the use of a timer to periodically sample the animation function and render it on screen.

- Chapter 14 discusses streams, which are used in Chapter 15 in the design of a high-level data type for reactive animations, i.e. animations that can react to user input. Chapter 16 introduces first-class channels and concurrency, needed for the brief discussion of implementation of reactive animations in Chapter 17.

The development so far has taken some 250 pages, and the author now turns to two other fields. Chapter 19 presents a simple imperative language for controlling a turtle-like robot. This is, following the same spirit, preceded in Chapter 18 by a discussion of higher-order types and monads, in particular state monads. Chapters 20 to 22 cover computer music, presenting a simplified version of the author's Haskore library. The pattern continues: Chapter 20 presents a high-level data type for music structure, this is translated in Chapter 21 into a more low-level type of 'presentations' (a temporally ordered version of the earlier type) and finally in Chapter 22 the structure is separated into individual instruments and translated into a MIDI file, using Haskore. The book concludes with two chapters giving an overview of the list processing functions in the standard prelude and of the standard type classes.

The author's selection of topics for the 'just in time' chapters on Haskell concepts is admirably well thought out. On this scale, the repertoire of language concepts gradually expands in an organized way and additions are immediately put to use. I am more hesitant about the fine-scale level. Throughout the text one finds numerous 'Detail' boxes, which introduce syntactic details, Prelude functions and minor language features on a 'call by need' basis. These boxes are typically a few lines long and appear in a rather random order. This makes the book very difficult to consult for the learning programmer, in need of discussion of some particular topic. The author suggests having the Haskell Report at hand for such consultation; in my experience this source is not very accessible for students.

In the preface the author stresses that the goal of this book is to teach how to solve problems in a functional language such as Haskell. Indeed, there are numerous good examples of this throughout the book, both in the text and in the exercises. I do, however, have two reservations. First, in many places an intriguing problem is posed, and then solved by pulling a new, and previously unknown, tool out of the author's toolbox. This is naturally true for many of the implementation problems, where types and functions from non-standard libraries are used, but also elsewhere. For instance, we are faced with the problem of computing the perimeters of shapes and come to ellipses. This is no easy problem, of course, but the infinite sum involving the radii that solves the problem is probably unknown to most readers. Elsewhere, we are faced with the problem of representing the grid world of the turtle robot and we are told about arrays.

Another perhaps slightly disappointing thing is the lack of really impressive applications. The two major examples of animations are a kaleidoscope program (like some screen savers) and a rudimentary version of paddleball. In a book about programming multimedia applications one could have hoped for more.

Web resources for the book include source code for all programs in the book, the non-standard libraries used (Linux and Windows versions), errata and PowerPoint slides (the latter still under construction at the time of this writing).

The writing style of the author is attractive, with nice discussion and well chosen examples and exercises. I believe that this book could profitably be used for an advanced undergraduate course focusing on domain-specific languages in this area, which certainly is motivating for students. I think that the book should be complemented by some larger programming examples to show the potential of the approach and for further motivation. For students without previous knowledge of functional programming I also believe that supplementary material is needed in the beginning of the course; Chapter 1 and its exercises seems to me a bit too brief to get new students (especially those who are familiar with imperative programming) into the right habits.

Björn von Sydow

The Optimal Implementation of Functional Programming Languages by A. Asperti and S. Guerrini, Cambridge University Press, 1998, 392 pp.

Given that computer science is a minority taste within what might be broadly called IT, and functional programming is a minority taste within computer science, it is curious to discover a minority taste within functional programming. The signifier is that weasel word 'optimal' in this book's title. Now, a functional programmer from the world of LISP and SML and Haskell might think that 'optimal' indicates a 'fastest' or 'smallest' implementation, given the bad press functional languages get in the imperative world. In fact, 'optimal' has a highly specialised meaning here, to do with the sharing properties of a literal lambda-calculus basis for implementing functional languages.

It is well known that normal order beta-reduction leads to duplication of terms and hence of reduction sequences. This can often be prevented through sharing of common terms, typically in some form of graph reduction scheme. However, it is hard to avoid duplication of functions, as sharing will result in a function body being changed when its application to some argument is first reduced. Levy (1978) formalised this 'optimality problem' in the late 1970s, specifying precisely the requirements for its solution. Lamping's (1990) algorithm was the first to meet Levy's requirements, and was subsequently shown to have close resonances with Girard's Linear Logic (Girard, 1987).

As Levy says in his generous introduction: 'This book covers the whole story'. The main body of the book provides a thorough account of the optimality problem, especially the authors' own contributions in refining Lamping's algorithm and implementing it in their Bologna Optimal Higher-Order Machine (BOHM). Throughout the book, detailed theoretical presentations are well complemented with diagrams and informal discussion.

For me, the main interest lies in the evaluation of BOHM, which is compared with Caml Light and Haskell, through pure lambda-calculus arithmetic based on Church numerals. For small computations, such as factorial five or Fibonacci thirteen, the prototype BOHM 0.0 implementation is considerably slower and tends to 'explode' on larger problems. This is due to the accumulation of redundant control operators, which appear to proliferate as do K's and I's in SKI-combinator based implementations. However, the introduction of safe rules for eliminating unnecessary control operators from graphs in BOHM 1.1 leads to a considerable improvement in performance. The modified implementation is actually faster than Haskell and comparable to Caml Light on the Church numeral examples. Nonetheless, as the authors acknowledge, typical 'real-world' functional programs do not display such low granularity of higher-order functional use and they estimate that their approach is about an order of magnitude worse than a typical applicative order implementation.

Indeed, in 'real-world' functional programming no one would actually go about implementing a functional language in this way. As Plasmeijer and van Eekelen (1993) noted:

[Lamping's] algorithm is very complicated. A lot of administration is needed to always avoid duplication of work. In practice, for most cases it may be more efficient to duplicate the work instead. (p. 109)

The authors recognise a complementary problem in the reduction of shared global expressions, where far more space may be needed for a partially evaluated term than for the original term. Thus, in BOHM they introduce explicit copying of global terms.

Overall, this book imputes a strange sense of *deja vu*. It was quickly recognised back in the 1980s that the apparent promise of literal implementations of normal order beta-reduction through lazy graph reduction techniques actually led to gross inefficiencies compared with strict implementations. Since then considerable effort has gone into trying to avoid laziness where it isn't needed through strictness analysis and into maximising the granularity of reduction through super-combinator techniques. Sadly, the authors of this book seem barely aware of such research and their sole reference to the world of 'real' functional language implementation is to Peyton-Jones' (1987) work.

References

Girard, J.-Y. (1987) Linear logic. *Theoretical Computer Science*, **50**(1), 1–102.

Lamping, J. (1990) An algorithm for optimal lambda-calculus evaluation. *Proceedings 1st Annual ACM Symposium on Principles of Programming Languages*, pp. 16–30. San Francisco, CA.

Levy, J.-J. (1978) *Reductions Correctes et Optimales dans le lambda-calcul*. PhD Thesis, Universite Paris VII.

Peyton-Jones, S. (1987) *The Implementation of Functional Programming Languages*. Prentice-Hall.

Plasmeijer, R. and M. van Eekelen, M. (1993) *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley.

Greg Michaelson

Research Directions in Parallel Functional Programming by Kevin Hammond and Greg Michaelson, editors, Springer-Verlag, 1999.

Parallelism and functional programming have been together for a quarter of century. The history of parallel functional programming can be divided into two periods (Lins, 1997). The first spans from 1975 to 1987, and opened when Burge suggested the technique of evaluating function arguments in parallel, and perhaps also exploiting speculative evaluation. This is the period in which functional languages sought in parallelism a solution to their inefficient implementations. The second period was inaugurated by the introduction of the G-machine and the consequent possibility of having efficient functional compilers for sequential hardware. In the second period, functional programming and parallelism are together trying to offer a solution to the big challenges in computer science.

The book edited by Hammond and Michaelson looks at 24 years of parallel functional programming, emphasising the progress made in the latter phase (from 1987 to today), building a background for further developments. It is aimed at practitioners and researchers, providing a good source of reference about recent activity and trends. This book goes far beyond a collection of separate papers on Parallel Functional Programming. 'The editors have sought contributions from a wide variety of established international researchers on their current activities and situated these within a common context'. Chapters point at each other linking together similar or related topics forming a whole.

The Foreword by Simon Peyton Jones colourfully presents the area of Parallel Functional Programming and the book. He beautifully justifies the significant editorial decision to omit chapters on completely-automatic parallelisation and special purpose hardware, areas in which a lot of research work has been done but which have become unfashionable nowadays. The former area has turned out to be technically extremely difficult, whilst the latter has an economic basis: it is impossible to compete with the billion-dollar investments that now go into mainstream microprocessor designs.

The material is split into three parts: Part I – Fundamentals, Part II – Current Research Areas, and Part III – Conclusion.

Part I of the book comprises six chapters. The first is an Introduction by the editors, which presents a taxonomy used for describing parallel functional languages and architectures. The second chapter, entitled 'Foundations', is authored by Greg Michaelson, Kevin Hammond and Chris Clack, and intends to provide a background on FP. Rita Loogen, in the third chapter of the book (Programming Language Constructs), explains the difference between implicit, controlled and explicit parallelism in their different 'flavours'.

The claim that functional programs are easier to prove correct than their imperative counterparts is an old one. In the fourth chapter of the book, Simon Thompson uses the divide-and-conquer approach to this difficult, but fundamental, issue by splitting the problem into the proof of the sequential and parallel part of programs. The strategies to prove the correctness of programs are elegantly described. Pointers are given to formalisms that may help in proving the correctness of the parallel parts of programs. Implementation of parallel functional languages is presented in Chapter 5 by Werner Kluge ('Realisations for Strict Languages') and Chapter 6 by Chris Clack ('Realisations for Non-Strict Languages'). Both chapters set an overview of the area without implementation details, and provide useful pointers to the reader in the literature.

Part II addresses the 'Current Research Areas', and comprises eleven chapters. John O'Donnell opens the second part of the book in Chapter 7 with 'Data Parallelism' advocating the better suitability of Haskell to obtain data parallelism than either Fortran or C. His point is that some combinators, such as *map,* are naturally data-parallel. 'Cost Modelling' is the title and subject of Chapter 8 by David Skillicorn. His attention is restricted to 'total execution time'. According to him, cost modelling is critical to successful parallel software construction precisely because intuition is not to be relied on in the parallel context; instead of relying on intuition, he presents an algebraic cost calculus. Chapter 9, 'Shaping Distributions', by C. Barry Jay, asserts that knowledge of the shapes of data structures can be used to improve memory management, which in turn has an impact on programming style, error detection and optimisation. In 'Performance Monitoring' (Chapter 10), Nathan Charles and Colin Runciman discuss the design and use of the parallel profiling tools for lazy functional languages, through layered conceptual data models for concurrent graph reduction.

Chapter 11, 'Memory Performance of Dataflow Programs', by A. P. Willem Böhm and Jeffery P. Hammes, reports on their experience of writing numeric, scientific algorithms in a mostly functional style in the programming language Id for the Monsoon Dataflow Machine. They claim that there continue to be serious problems in the space performance of functional languages. Jonathan Hill analyses the 'Portability of Performance in the Bulk Synchronous Parallel (BSP) Model' in Chapter 12. He argues for the use of the BSP cost calculus in enabling provability of performance of scalable parallel applications. In Chapter 13, Murray Cole introduces the central concept involved in 'Algorithmic Skeletons', outlines the research issues that arise, and investigates the two projects that have made most progress towards practical realisation. 'Co-ordination Languages' is the title of Chapter 14 by Paul Kelly and Frank Taylor, in which a critical view of the design and implementation of Paul Kelly's Caliban is given.

In Chapter 15, Rinus Plasmeijer, Marko van Eekelen, Marco Pil and Pascal Serrarens present the constructors for 'Parallel and Distributed Programming in Concurrent Clean'. Ali Abdallah, in Chapter 16, 'Functional Process Modelling', presents a calculus for parallel program derivation based on Bird–Meertens formalism. In Chapter 17, Flemming Nielson presents in 'Validating Programs in Concurrent ML' a way to extract the flow of control in a more readable and succinct form, so as to validate the overall structure of communication taking place in the program. In ‘Explicit Parallelism’ (Chapter 18), Jocelyn Sérot discusses the design and implementation of an interface to MPI for Objective CaML.

The third part of the book has one single chapter entitled ‘Large Scale Functional Applications’, written by Philip W. Trinder, Hans-Wolfgang Loidl and Kevin Hammond. The chapter describes applications that have been written using parallel functional programming language technology presented in the previous chapters of the book. The range of applications spans from numerical programming, symbolic computing, computer vision, to telephony and data mining. While the majority of these applications come from the academic community, others such as Erlang (Ericsson) and Compaq’s Smart Kiosk represent serious trial of the technology by commercial organisations. The authors identified several research directions and room to find a killer application in parallel functional programming:

- *Irregular and Symbolic Applications*: new applications in computer science, such as computer vision, make use of complex algorithms and data structures exhibiting irregular parallelism, in which intensive symbolic manipulation is needed.
- *Implementations*: some key implementation issues revealed by applications are the need for improved data locality and for the reduction of heap usage.
- *Architecture Independence*: it appears that the high-level of abstraction in functional programming has the potential to ease the movement between parallel platforms.
- *Metrics for Parallelism*: assessment of parallel systems and applications would be greatly facilitated by a widespread use of a standard set of programs and standard performance metrics.
- *Programming Environments*: both the development environments and methodologies currently used for parallel functional programming are at an early stage, and require improvement.

Garbage Collection and FP have been together since the very beginning. One can say they are twins, as both were born with LISP. The greatest absence I felt in the book is a chapter devoted to Parallel Garbage Collection, a complex and challenging topic which is addressed only briefly in Jones and Lins (1996).

Hammond, Michaelson and the authors in this book give a great contribution to the area of by setting a starting point to whoever wants to learn about parallel functional programming. Aiming to foster the area, the editors and the authors have waived all royalties from the sale of the book, which will form a fund for a prize at the annual *International Workshop on the Implementation of Functional Languages (IFL)*.

I leave the reader quoting Simon Peyton Jones from the Foreword:

Functional programming, including parallel functional programming, is a place where theory and practice meet particularly fruitfully. That alone makes it a fascinating territory. The early euphoria of parallel functional programming has worn off, but the fun has not. This book presents an up-to-date picture of the state-of-the-art in a field that is still advancing in many fronts. Enjoy.

References

Lins, R. D. (1997) Functional programming and parallel pocessing. In: J. L. Palma and J. Dongarra, editors, *Proceeding of VECPAR’96 – Vector and Parallel Processing: Lecture Notes in Computer Science 1215*, pp. 429–457. Springer-Verlag.

Jones, R. E. and Lins, R. (1996) *Garbage Collection: Algorithms for Dynamic Memory Management*. Wiley.

Rafael D. Lins

Theories of Programming Languages by John C. Reynolds, Cambridge University Press, 1998.

The author

John Reynolds is a well-known and highly regarded researcher who has made significant and fundamental contributions to both theory and practice of programming and programming languages, combining computational insight with mathematical rigour. Some of his most notable contributions are: the programming languages Gedanken and Forsythe, the polymorphic lambda calculus, possible-world semantics of block structure, the theory of sub-types, the concept of relational parametricity, and the defunctionalization transformation.

The audience

This book is intended to be a graduate or advanced-undergraduate level survey of fundamental principles and issues in programming languages. It originated from course notes for a course taught by Reynolds at Carnegie-Mellon University to all doctoral students in Computer Science. Reynolds argues that such a course is needed to help bridge the gap between language theory and software-systems practice, to support mathematics-based programming practices, and to ensure that fundamental principles of this area of computer science are understood by all graduates, and not just specialists.

An overview

After a preliminary chapter on abstract syntax and the semantics of predicate logic, Chapters 2–9 discuss a simple imperative language and various extensions, including arrays, jumps, input/output, non-determinism, concurrency and communication. Both compositional and operational semantic methods are treated, as well as program-correctness (Hoare) logic.

Chapters 10–14 discuss untyped functional languages, starting with the lambda calculus, but also treating eager evaluation, control operators and continuations, references, and normal-order and lazy evaluation.

Chapters 15–18 discuss types, starting with the simple type system and then sub-types, intersection types, polymorphism, and existential types. Both ‘extrinsic’ (Curry-style) and ‘intrinsic’ (Church-style) semantics are described.

Finally, Chapter 19 discusses Algol-like languages and the possible-worlds treatment of the stack discipline.

The presentation uses only ‘elementary’ mathematics; nevertheless, readers will find that considerable mathematical maturity will be necessary. Category theory is *not* used; in fact, even the definition of ‘function’ (given in the Appendix) is pre-categorical.

Some topics that are *not* treated (except perhaps for brief mentions) are: fixed-point induction and admissibility, solutions to domain equations, logical relations and relational parametricity, Hindley-Milner type inference, abstract interpretation, dependent types, computational monads, temporal and linear logics, the π calculus, and propositions as types.

Appraisal

This is clearly an important book and, to a large extent, it achieves its aims. It is thorough and well organized, and the explanations are very clear. I found the technical content to be in almost every respect flawless, and the book seems to be virtually typo-free: a few very minor ones are noted at

`http://www.cs.cmu.edu/~jcr/tpl-errata.ps`.

The index is very comprehensive. Some readers may find the font size to be unpleasantly large.

The material will be found challenging by many, perhaps even most, students; but surely no more so than comparable topics of foundational significance, such as computability theory or complexity theory.

Although written to be a student text, I believe that most researchers (and many practitioners) in programming languages (and related fields) would find it worth reading. This includes *beginning* research students, of course, but with an important caveat: this book is not intended to be a *comprehensive* source of background knowledge. In particular, the minimal or non-existent treatments of categorical, domain-theoretic and type-theoretic material and many important or current topics mean that much supplementary reading will be necessary. Reynolds has provided good references in bibliographic notes for each chapter and a very extensive bibliography.

The one reservation I have in regard to this book is that, despite the welcome inclusion of numerous examples of semantic peculiarities, programming techniques and program verifications, there is insufficient motivational material for readers who, *a priori*, might not have much interest in theories of programming languages.

This problem is particularly apparent at the very beginning of the book. Most books on semantics skip over the concept of abstract syntax as lightly and quickly as possible, so experts will welcome the detailed and rigorous treatment given here; however, many readers will find it hard to see the practical significance of this rather technical material, at least initially. Instructors using this as a text for a 'general' computing audience will often need to provide additional motivation.

Comparable books

Books which overlap significantly with this one are:

- my own *Semantics of Programming Languages* (Prentice Hall International, 1991);
- Carl Gunter's *Semantics of Programming Languages* (MIT Press, 1992);
- Glynn Winskel's *The Formal Semantics of Programming Languages* (MIT Press, 1993);
- David Schmidt's *The Structure of Typed Programming Languages* (MIT Press, 1994); and
- John Mitchell's *Foundations for Programming Languages* (MIT Press, 1996).

However, in each case, the emphasis or coverage is slightly different. For example, Mitchell doesn't cover imperative aspects as well as Reynolds, but is much more thorough on lambda calculus models, and the three books on semantics do not cover many aspects of types as well as Reynolds.

Concluding remarks

This book is certainly a welcome addition to the available literature on programming languages, but its *commercial* success is not assured. How many schools currently offer a course that could be supported by this book? I'm somewhat reminded of another ground-breaking, important and challenging book that similarly arose out of a somewhat non-standard course (in program specification and verification techniques for professional programmers). That text is Reynolds's earlier work, *The Craft of Programming* (Prentice Hall International, 1981), now, sadly, out of print. It is perhaps up to all of us to ensure that this book does not share that fate.

R. D. Tennent