J. Functional Programming **9** (1): 105–111, January 1999. Printed in the United Kingdom

Book reviews

Advanced Functional Programming edited by John Lauchbury, Erik Meijer and Tim Sheard, *Lecture Notes in Computer Science 1129*, Springer-Verlag, 1996.

This collection of tutorial papers from the *Second International School in Advanced Functional Programming* follows a similar format to the Båstad school (Jeuring and Meijer, 1995), reviewed in this journal (Warren Burton, to appear). These schools had the aim of widening the audience for a mature functional paradigm, which has moved from the original innovations of the late 1970s and 1980s to larger-scale applications, language extensions and more advanced programming techniques. As in Jeuring and Meijer (1995), the intended audience consists of people with a first course in functional programming as well as wider computing experience. The authors use Haskell and Standard ML (SML) as vehicles, but many of their insights are non-language specific. We now review the individual contributions of the authors and conclude by making some general remarks.

Finne and Peyton Jones: Composing the User Interface with Haggis

Historically one of the problematic issues for the designer and user of functional languages has been in handling input/output; a comprehensive survey is given by Gordon (1994). The monads of the later versions of Haskell[1] represent an emerging consensus of how to manage not only I/O but also mutable state, exceptions and so forth; Jeuring and Meijer (1995) contains substantial amounts of tutorial material on monadic programming. Also amenable to monadic treatment is the programming of Graphical User Interfaces (GUIs); the Haggis approach to GUI programming is the subject of this article. Various strands of advanced functional programming come together in Haggis, thus.

- The system uses a declarative view of pictures. Representations of pictures are built from a small number of primitives, together with various picture combinators. This view is independent of the particular way that the pictures are rendered, an approach it shares with Haskore (see the discussion of the article by Hudak below) and other systems.
- Concurrency is used to support the realization of the GUI as a virtual device; this also assists in separating the user interface of an application from the application itself.
- A component-based system of GUI construction, similar to the approach to Pictures.

The authors give copious numbers of examples and exercises, including the well-known model-view-controller design pattern. Haggis is implemented in the Glasgow Haskell Compiler.

Hudak: Haskore Music Tutorial

Haskore is a Haskell-based system for describing music. At its heart is an algebraic type, `Music`, which provides a concrete representation of pieces of music. A user of the system can compose by describing objects of the `Music` type as well as giving general transformations

[1] Hammond, K. and Peterson, J. (eds.), *Report on the Programming Language Haskell 1.4*, available at www.haskell.org.

over the type. Equalities between different representations can also be proved using simple equational reasoning over the `Music` type. From `Music` other artifacts can be produced: Haskore allows the modelling of `Performances` on `Instruments` which can themselves be realized in MIDI and other sound formats as well as being notated. This modelling approach, also exemplified by Haggis and Fran (1997), is elegant and flexible and is characteristic of modern functional programming. It also appears to show one way in which functional languages out-perform their object-oriented rivals in giving a programming device that naturally reflects the nature of the system being modelled.

Jeuring and Jansson: Polytypic Programming

The essential idea of polytypic programming is that some functions can be defined uniformly for all algebraic data types. A canonical example of this is a measurement of the size of a member of an algebraic type: at a node, one sums the sizes of the recursive components and adds one, for instance. The authors' achievement is to show how to construct a programming system (Polyp) in which such general 'polytypic' functions can be expressed by induction over the construction of the data type involved. Their approach is built upon a categorical view of functional programming, more details of which can be found in Bird and de Moor (1997). This view combines a datatype-independent expression of structural recursions (or catamorphisms) with a relatively low-level collection of categorical 'plumbing' combinators. After outlining their approach the authors give some 'theorems for free' for polytypic functions and illustrate their work by developing in Polyp case studies of term rewriting and unification.

Lee: Implementing Threads in Standard ML

This chapter shows how the standard functional constructs of higher-order functions and continuations as implemented in SML of New Jersey are used to implement 'lightweight' threads of concurrent control. The author illustrates the value of a functional approach by the arguing that the simple thread implementation he has given will leak space. The clarity of the code for threads and the simplicity of the functional execution model surely make this kind of argument easier to justify, and the article is completed with a refined, 'safe-for-space', thread system.

Okasaki: Functional Data Structures

Okasaki gives a tutorial on his and others' work on designing efficient data structures in a functional setting. In the author's words: "Some imperative data structures can easily be adapted to functional languages, but most cannot. Leftist heaps are essentially the same in SML as in C, FIFO queues and catenable lists must be redesigned from scratch; the usual imperative solutions are completely unsuitable for functional implementations". Functional languages provide a convenient framework for designing persistent data structures – which can then be implemented in imperative languages. In fact for some problems the best-known persistent solutions were designed in this way.

It is interesting to observe that although the implementation is in the strict language SML, Okasaki introduces primitives for laziness, since these are the key to integrating the amortization of costs of computation over repeated functions over that data with the potential persistence of the data structure itself.

Runciman and Rojemo: Heap Profiling for Space Efficiency

One of the most difficult problems facing a programmer in a lazy language is to predict the space behaviour of his or her programs. It is all too easy to make a small change to a program which does not affect the results of the computation but which does have drastic effects on the efficiency of its evaluation. The authors describe an extension to the NHC Haskell compiler that delivers profiles of the heap usage of programs. In their tutorial the authors follow an introduction to their system by giving a taxonomy of many of the causes of space leaks together with a sequence of illustrative examples.

Swierstra and Duponcheel: Deterministic, Error-Correcting Combinator Parsers

This article builds on Fokker's (1995) introduction to combinator-based parsing from the earlier AFP Spring School (Jeuring and Meijer, 1995). Fokker shows how to construct parsers from a small set of higher-order combinators, which Swierstra and Duponcheel point out is almost the classical example of such a style of programming. As they also argue, this approach is inefficient, since it has inherent overheads due to its non-determinism, and limited in being unable to provide facilities for error reporting and recovery. The authors give a new set of combinators for LL(1) grammars, which are commonly used to give the syntax of programming languages, and show how their combinators avoid the difficulties of the naive approach. They conclude their discussion with an informative account of the possibilities for partial evaluation in this context and the difficulty of applying this analysis for the monadic version of the parser combinators (Hutton and Meijer, to appear).

Tofte: Essentials of Standard ML Modules

Tofte summarizes the SML module system in which a distinction is drawn between interfaces (ML signatures) and their implementations (ML structures), as well as containing parametric structures (ML functors) which are dependent on arguments which need to conform to a given signature. The type system is illustrated by means of a series of practical exercises to build a polymorphic type-checker for the idealized language MiniML.

Conclusion

The text reviewed here continues themes begun in Jeuring and Meijer (1995), showing how the powerful abstraction mechanisms of functional languages can be used to provide elegant and powerful solutions to programming problems such as GUI and thread programming, modelling musical structures and general modular programming. A different theme is the need for efficiency in functional systems. Okasaki aids a programmer by providing efficient implementations of data structures which can be used 'off the shelf', while Runciman and Rojemo show how programmers can analyze the space behaviour of their programs for themselves. Finally, Jeuring and Jansson show that research into functional languages themselves still has useful results to deliver by describing a new definition mechanism for polymorphic functions which uses induction over the construction of the domain types to give functions defined over all algebraic types in a uniform way.

References

Bird, R and de Moor, O. (1997) *Algebra of Programming*, Prentice-Hall.

Elliott, C. and Hudak, P. (1997) Functional Reactive Animation, *International Conference on Functional Programming*, ACM.

Fokker, J. (1995) Functional Parsers, in Jeuring, J. and Meijer, E. (eds.), *Advanced Functional Programming: Lecture Notes in Computer Science 925*, Springer-Verlag.

Gordon, A. (1994) *Functional Programming and Input/Output*, Cambridge University Press.

Hutton,G. and Meijer,E. (to appear) Monadic Parser Combinators, *J. Functional Programming*.

Jeuring, J. and Meijer, E. (eds.) (1995) *Advanced Functional Programming: Lecture Notes in Computer Science 925*, Springer-Verlag.

Warren Burton, J. (to appear) Review of *Advanced Functional Programming: Lecture Notes in Computer Science 925*, *J. Functional Programming*.

Simon Thompson

Advanced Functional Programming: Lecture Notes in Computer Science 925
edited by Johan Jeuring and Erik Meijer, Springer-Verlag, 1995.

Advanced Functional Programming contains the lecture notes used at the *First International Spring School on Advanced Functional Programming Techniques* in May 1995, in Båstad, Sweden. *Advanced* Lazy *Functional Programming* perhaps would have been a better title. Many of the ideas presented do not apply to strict functional languages such as *Standard ML*. For example, most of the chapters at least mention monads, and type classes are widely used.

Most books on functional programming, or introductory books on programming in other languages, usually focus on programming in the small and use small toy examples. This book focuses on larger problems and more advanced topics. Because of the power and simplicity of functional languages, this book contains a number of examples that simply would be too large and complex for even an advanced book of similar size based on a different style of language. Many of the concepts considered in the book, while useful with toy problems, don't really show their worth until applied to more complex problems, such as these.

The book consists of nine chapters, written by different authors. The chapters tend to relate to some aspect of the authors' research, usually containing some of the authors own results. However, the contents of the book do not appear to have been dictated by the interests of the authors. Instead, it appears that the contents of the book were carefully selected and appropriate authors were chosen for the different topics. In general, the authors are known not only for the quality of their research, but also for the quality of their presentations, both oral and written. Clearly they invested a significant amount of effort in making this book a pleasure to read.

While the chapters are independent, and in principle can be read in any order, they fit together nicely, reference each other occasionally, and are ordered so that later chapters often build on ideas introduced in earlier ones. However, each chapter is sufficiently self-contained that reading of previous chapters is not required.

As the title suggests, this book assumes some prior knowledge of functional programming. I covered selected chapters in the later part of a graduate course that introduced functional programming to graduate students who, for the most part, were not primarily interested in functional programming or programming languages. The book would be ideal for a graduate course for students with a previous knowledge of functional programming, or for self-study by anybody interested in extending their knowledge of functional programming techniques. Individual chapters are a good starting point for learning about particular topics.

The power and simplicity of functional programming is illustrated throughout the book. The modularity and potential for code reuse that results when functional program components can be combined in various ways is frequently demonstrated. Hence, chapters may be worth reading as examples of functional programming techniques and styles even if the official topic of the chapter is not of interest to the reader. Most chapters contain exercises, and all have a good balance between the presentation of concepts and examples (often nontrivial).

Most, if not all, of the chapters contain minor errors. These did not seriously impede my reading, so I will leave finding and correcting them as an exercise.

Brief comments on the individual chapters follow:

Functional Parsers by Jeroen Fokker. Fokker gets the proceedings off to a good start with an interesting and nontrivial application. Lazy evaluation and higher-order functions are put to good use in functional parsers, and backtracking using the 'list of successes' method is presented.

Monads for functional programming by Philip Wadler. Monads, which are used in various ways in the following chapters, are introduced by Wadler. A number of practical examples are given, including updatable state, arrays, output, exceptions and parsers. The laws that a monad should satisfy are also given. The presentation is clear and the examples demonstrate the power of monads. This chapter should be read before later chapters by anyone who is not already comfortable with monads.

The Design of a Pretty-printing Library by John Hughes. The topic of the chapter by Hughes is really program derivation and transformation. Using a pretty-printing library as an example, an efficient program is derived from a simple specification, with a number of variations of the program appearing along the way. Extensive use is made of higher-order functions. Experiments are used to check the efficiency of the various programs, so the chapter has a practical as well as theoretical flavour. It is unlikely the final program could have been produced without recourse to formal transformations. Near the end of the chapter the functional pretty-printing library is compared to a classic pretty-printing library.

Functional Programming with Overloading and Higher-Order Polymorphism by Mark P. Jones. Type classes and constructor classes can now be found in several functional programming languages. Mark Jones presents the basic concepts behind classes and presents a variety of examples showing how classes can be put to good use in a functional programming language such as *Haskell*. Constructor classes are a fairly natural generalization of type classes as originally proposed, but significantly add to the power of the class system. In particular, `monad` is a constructor class, since a monad (as defined in a functional language) is a type constructor. While monads are only one of many classes considered by Jones, the relation of monads to constructor classes nicely complements Wadler's earlier presentation.

As Jones demonstrates, classes contribute significantly to the ability to write modular and reusable code.

Programming with Fudgets by Thomas Hallgren and Magnus Carlsson. The book takes a more applied turn with the chapter on fudgets. Fudgets are 'functional widgets' where a 'widget' is a 'window gadget'.

One of the areas where functional languages have tended to be weak is input and output. Early functional programs were functions from input streams to output streams. File I/O, if supported at all, tended to not be purely functional. Dialogues provided a more general, yet purely functional, I/O mechanism. But dialogues are not pretty. Monadic I/O is a big improvement, but does not address the increasingly important problem of supporting a graphical user interface.

Fudgets support graphical user interfaces in a simple and almost purely functional manner. Actually, fudgets are slightly nonfunctional because some mechanism is required to allow a program to respond to input from different sources arriving at different rates, but a carefully considered practical compromise maintains the spirit of functional programming and allows graphical user interfaces to be written in a surprisingly functional style.

Fudgets can also be used for communication between different computers, using sockets.

Constructing Medium Sized Efficient Functional Programs in Clean by Marko C.J.D. van Eekelen and Rinus (M.) J. Plasmeijer. This chapter discusses the implementation of a spreadsheet in Clean, version 0.8. The resulting program was about 30,000 lines long, with about two thirds of that consisting of code reused from a previously written I/O library and a text editor.

While the focus of the chapter is on software engineering, and the suitability of a functional language for code reuse and programming in the large, the chapter also describes many of the key features of Clean, a pure lazy functional programming language. The most novel feature is unique types. If a value has a unique type, then there can be only one reference to it. Unique types provide a viable alternative to monads for allowing destructive updating of data and supporting nice input and output. Clean is able to support graphical user interfaces through the use of unique types. Clean uses a class system similar to that of *Haskell* or *Gofer*, but with some interesting differences.

While the use of Clean for writing a medium sized program was successful, experience with writing the spreadsheet in Clean 0.8 has resulted in a number of changes to Clean 1.0.

Merging Monads and Folds for Functional Programming by Erik Meijer and Johan Jeuring. As someone who probably should know more category theory than I do, I expected this chapter to be heavy going. However, I was pleasantly surprised by the clarity of the presentation.

The `fold` (or `foldr`) function takes three arguments, the last of which is a list, and effectively replaces each `cons` (or `:`) used in the construction of a list with one of the arguments and replaces the `nil` (or `[]`) at the end of the list with the other argument. For example:

```
foldr (+) 0 [1,2,3] = foldr (+) 0 (1:(2:(3:[]))) = 1+(2+(3+0))
```

Fold functions, called catamorphisms, can be written for other types of data structures. With a fold function, each data constructor is replaced with one of the arguments to the fold. That is, a fold function is a higher-order function encapsulating structural recursion for a particular datatype.

Meijer and Jeuring give a number of examples of fold functions for different types, and show that many computations can be defined in terms of these. However, they then go on and show how fold functions can be modified to monadic fold functions, which incorporate many of the advantages of both monads and folds. A few further examples are sufficient to convince the reader that monadic folds are a useful addition to one's bag of programming techniques.

They close the chapter with an example showing how an efficient interpreter for a toy language can be derived from a simple specification using monadic folds. Thomas Johnsson had previously performed the same derivation using fold and unfold operations, but the authors claim that their approach results in a derivation that is about two thirds the lengths of Johnsson's, and easier to understand.

Programming with Algebras by Richard B. Kieburtz and Jeffrey Lewis. As someone who probably should know more category theory than I do, I found this chapter to be a bit hard to understand in places. Part of the problem is that the notation used in this chapter is different than that used for the same concepts earlier in the book. For example, a variation of the monadic `bind` operation is used, but it has a different name and different argument order, and no mention of its relation to the `bind` operator is made. Some of the mathematical concepts are introduced quickly and not in enough detail to be easily understood. However, the fundamental concepts of the chapter are clear.

Kieburtz and Lewis discuss *ADL*, an *Algebraic Design Language*. *ADL* is a more formal functional language than those considered in previous chapters, and is oriented towards proving programs correct.

In a language like *Haskell* it is necessary for a programmer to define a separate fold operator for each data structure, if a fold operator is desired (as it is likely to be, by any programmer who has read the previous chapter). In *ADL* there is a general fold operator that works with any type of data structure. This construct is extended to facilitate the writing of many functions for which the fold operator by itself is not sufficient. Near the end of the chapter, monads are considered.

Graph Algorithms with a Functional Flavour by John Launchbury. As mentioned above, input and output has been one of the traditional trouble spots for functional programming. Another has been updatable state. Graph algorithms are an ideal example of a problem area where updatable state is at least very useful, and probably necessary for efficient programs.

Of course monads provide updatable state of the type needed for graph algorithms. However, if, using monads, one directly translates a standard graph algorithm to a functional language, one really hasn't gained much.

The really nice thing about this chapter is that the use of state is encapsulated in a small bit of code, and a more functional style is used for most of the program. This makes it relatively easy to use the standard techniques for reasoning about functional programs. Launchbury illustrates this by doing a calculational proof of correctness for an algorithm for finding the strongly connected components of a directed graph, and argues that his proof is simpler and more straight forward than other proofs of correctness for this algorithm.

F. Warren Burton

Modern Compiler Implementation in ML: Basic Techniques by Andrew W. Appel, Cambridge University Press, 1997, ISBN 0521587751.

Andrew Appel's *Modern Compiler Information in ML* is an in-depth study of modern compilers. Although it uses the functional language ML as an implementation language and a later chapter covers the implementation of functional programming languages it is not a 'functional' book. Rather it gives a very detailed account of implementing conventional languages.

This is achieved by taking a made-up language 'Tiger' and working through each part of the compilation process in sequence. There are chapters on lexical analysis, parsing, abstract syntax, semantic analysis, intermediate code generation, basic blocks, instruction selection, liveness analysis and register allocation.

Appel claims his goal was to describe a good compiler that is 'as simple as possible – but no simpler', and he has designed the Tiger language with this in mind. This does not mean he does not cover topics not required by the simple language but they are then covered at a higher level of abstraction. Specific ML code as is provided for much of the Tiger language. It is this high level of detail that is both the strength and weakness of this book. On the positive side it means that all the nitty-gritty details are explained and good practice code is given. However, the downside is that the detail also tends to obscure the principles that may be more appropriate in a lecture course. However, as there are a number of excellent texts that provide a higher level view of compilation (for example, the 'Dragon' book of Aho *et al.* (1986)), Appel's text provides a welcome addition to the literature. Although the use of ML as an implementation language would somewhat limit it as a suitable textbook for many courses, there are also versions of the book available that use the languages C and Java. I have not studied these alternative versions in detail, but a quick comparison of the ML and Java texts suggested that apart from the fragments of example code and small changes of emphasis, the majority of the text, including the language Tiger used as a running example, was unchanged.

The reason for the subtitle 'Basic Techniques' in the title is that these three books are preliminary versions of books to be published in 1998. These contain four extra chapters on loop optimizations, static single-assignment form, pipelining and scheduling and finally the memory hierarchy, as well as revising some of the other material in the books. However, even the 'basic' versions cover topics that are not found in many other compiler books. Each version is supported by WWW resources, including the code required for the numerous programming exercises, sample Tiger programs and pointers on how to obtain additional software such as the scanner and parser generators used.

In conclusion this book is a valuable addition to the bookshelves of programmers interested in reading about or implementing modern compiler techniques and would make an excellent book to support a university compiler implementation course.

Reference

Aho, A. V., Sethi, R. and Ullman, J. D. (1986) *Compilers, Principles, Techniques and Tools*, Addison-Wesley.

Bruce McKenzie