J. Functional Programming **8** (4): 445–446, July 1998. Printed in the United Kingdom

Book review

Applications of Functional Programming edited by Colin Runciman and David Wakeling, UCL Press Limited, 1995.

Runciman, Wakeling and the other contributors to this book set out to demonstrate how Haskell can be used in realistic computing applications. They succeeded admirably. Anyone interested in building software in Haskell should read this book. Anyone interested in understanding the practical usefulness of functional languages should read this book. Anyone who wants to stay on the forefront of modern approaches to software construction should read this book. It provides a sound and engaging introduction for people with or without experience in Haskell. It provides well-documented points of reference for researchers in the field of functional program development. It supplies worthwhile ideas to functional language implementors. And, it does all this in a compact 240 pages.

Chapter 1 explains the salient features of Haskell. This is one of the most cogent introductions to the language available anywhere. It is followed by a chapter on some of the tools supporting Haskell development, in particular the Chalmers and Glasgow compilers and the profiling tools associated with them.

Seven applications appear in the second part of the book, all of which exhibit significant portions of their code and include explanations of roles played by various Haskell features and analyses of performance trade-offs. Readers can benefit from the exemplary coding practices exhibited in many of these programs.

For example, an application addressing a job scheduling problem contains instructive uses of Haskell classes, inheritance, and overriding of operations and two excellent examples of how lazy evaluation can reduce program complexity and lead to software formulated as compositions of small, testable units. A text compression problem is another application using lazy evaluation as an essential design element. This application uses a circular definition in the output stream to extend the domain of the program while maintaining a separation between its data encoding and input/output components.

The job scheduling application makes use of memoization and includes an analysis of the trade-offs between the time savings afforded by memoization and the garbage collection it triggers. An implementation of a geometric modelling system investigates memoization in even greater detail. Its authors managed to save space by sharing memos and observe that lazy evaluation is especially important in such contexts because it means fewer memos will be generated.

The geometric modelling system implements essentially all of the functionality of a 16,000-line C program in about one thousand lines of Haskell. This was accomplished primarily through the extensive re-use of certain higher-order functions, with some interplay with lazy evaluation. Other codes presented in the book were also concise, compared to implementations in conventional languages, but usually more on the order of factors of three to five. The impressive factor-of-sixteen reduction in the geometric modelling system shows what can be done with the right combination of higher-order functions and algebraic data types.

Space leaks seem to be the most common problem in Haskell programs. Almost all of the applications made use of heap profiling to chart patterns of storage utilization. Those patterns were often surprising, and in most cases led to the discovery of ways to reduce storage demands. It is encouraging that space problems in these applications were usually

solved, through the use of profiling tools, without negative effects on program structure or correctness.

The third part of the book discusses parallelism, a longtime promise of functional programming that has not yet been fulfilled. The authors report some progress: in the GRIP project, garbage collection is down to 5% time consumption, the bus seems adequate to the communication needs, statistics gathering degrades performance by only 2%, and speed-up approximates half the number of processors. Global memory access seems to be the main problem. But, parallelism is easy to express and performance concerns can be separated from correctness concerns through the use of semantics-preserving annotations directing parallel computation. Control of granularity remains problematic.

Applications in the book cover a broad range: software in which graphical user interfaces play a big role (graphical design, proof assistance, geometric modelling), applications where performance counts: including numeric (computational fluid dynamics); non-numeric (text compression, job scheduling) and system-level programs (terminal emulation). This may not be comprehensive, but there is enough breadth to support the conclusion that functional programming is, in practice as well as theory, a general purpose paradigm.

The book's value does not rest entirely on functional programming. For example, the computational fluid dynamics application contains an excellent description of the essentials of Jacobi iteration and the Choleski method and an insightful analysis of several methods of sparse matrix representation. This is characteristic of the book's approach: it is mostly self-contained. It lists 125 references, but these are mostly for archival and follow-up purposes. Most of the essential material can be gleaned from reading one or another section of the book.

The authors conclude that the main problems to be solved in Haskell development environments are improvements in interfaces to existing systems (for example, the lack of an interrupt capability limits interactive applications), tools for runtime fault analysis (applications resort to tricks such as placing extra tokens on the output stream, but run into problems when there is no convenient output stream on which to place the tokens), performance improvement (Haskell programs run, on the average, about ten times slower than equivalent programs in conventional languages – arrays may help solve some of this, but good implementations had not arrived at the time this book was written), and tools for analysing space faults (a good start has been made in this area). One application in the book, a virtual terminal model, exposed a deficiency in Haskell's polymorphic type system; but for most applications the type system provided advantages in terms of conciseness and program correctness, as one would expect.

There may be an intrinsic price to be paid for the expressive power of lazy evaluation and higher order functions, but, as Runciman and Wakeling point out in their conclusion, Haskell compilers are still in their infancy. The real price of its features cannot be estimated until the effort invested in compilers for lazy functional languages begins to approach the effort that has been invested in compilers for conventional languages. The concluding chapter also notes that Haskell shares some of its greatest advantages with other declarative programming languages: automatic memory management, independently testable units, program correctness, and conciseness of expression. Haskell does have some advantages of its own, however. First among these are its type classes, which, along with higher order functions, lazy evaluation and polymorphism, facilitate re-use.

In this book, Runciman, Wakeling and their co-authors present convincing evidence that functional programming has earned a place in conventional practice and that its promise justifies a great investment of future research and development effort.

Rex Page

https://doi.org/10.1017/S0956796898003232 Published online by Cambridge University Press