Book reviews

An Introduction to Computing with Haskell by Manuel M. T. Chakravarty and Gabrielle C. Keller, Pearson SprintPrint, 2002, ISBN 1 74009 404 2. Price $AU 39.95, Pp 150. DOI: 10.1017/S0956796803215033

The soft cover text *An Introduction to Computing with Haskell* with thirteen chapters, two appendices and an index on 145 pages of A4 is published on a 'print-on-demand' basis and available directly from the publishers in Sydney (at least in the sense that neither of the two large University bookshops I rang in Brisbane was able to find the book on their databases).

Derived from lecture notes used in an introductory course at the *University of New South Wales* this book is attractively priced by Australian standards. At $AU 39.95 it retails for just slightly more than half the price of its nearest competitors. The relatively low price and a steering theme which embraces the elements and process of mathematical program analysis and reasoning are its distinguishing features. By the time they are finished, attentive readers should have soaked up a strong basis for a rational approach to computer programming and software design.

In the words of the authors, the book sets out to introduce an 'absolute beginner' to computing including 'fundamental concepts of programming and simple forms of reasoning about programs'. That introduction mainly takes place in the computer programming language Haskell running under the Unix operating system. In pursuit of their aim Chakravarty and Keller eschew the usual heavy theory and dogma surrounding programming models in favour of a pedagogical framework composed of basic definitions built on informal discussion of some important ideas in programming.

For example, the novice computer programmer is introduced to the idea and benefits of types as categories of values but doesn't have to grapple with category theory. Likewise, input and output are introduced as sequenceable actions expressed within the lazy paradigm of Haskell by the *do* notational idiom, but the theory of monoids and monads is ignored. Science does assume a fundamental role, however, as the student is quickly shown how to reason mathematically about program correctness and complexity through tools such as structural induction, the O notation and recurrence relations.

Discussion around the main theme of this book is supported by a series of toy example programs, the most substantial of which are a video store database, a supermarket pricing program, a computer programming assignment auto-tester and an arithmetic expression evaluator. The auto-tester is a Unix shell script; the remaining examples being presented in Haskell. Exercises at the end of each chapter build on those examples and other issues. The *GHCi* compiler is also quickly introduced to the reader as a vehicle for Haskell programming.

Let's turn now to a lecturer's check-list of the material covered (not in chapter order).

Apart from the obligatory introduction and two chapters briefing the reader on Unix operating system basics and history, there are chapters covering types, control structures, recursion and input and output. Lists, higher order functions, user-defined data types, and tree structures fill out the remaining bread and butter chapters. There are also more advanced chapters on modularization and program decomposition, formal reasoning and work complexity.

The chapter on trees covers sorted binary trees, AVL trees and tries. Another chapter introduces complexity analysis by developing the use of O notation through a discussion of that classic trio of sorting algorithms – insertion, merge and quicksort.

You won't find brightly coloured multimedia experiences here.

Regarding more down-to-earth issues, the prose is clear and easy to read. Newly defined terms are introduced in italics, which I found to enhance readability.

Having said that, I am concerned by the lack of references (apart from a few web page addresses amongst the footnotes) and, in particular, I would have expected a reference to the Haskell 98 report at least. Even in a book for beginners, you need to provide an "out" for students who want to go further. One of the defining moments in my computer science education occurred as a seventeen year old on the day I read one of the references from the set text for our course. *The Pascal User Manual and Report* contained my first BNF diagram and I suddenly realized that it must be possible to write a Pascal compiler in Pascal itself. That book gave me deeper insights into computer languages than either my long ago introductory course in computer science or the primary text itself.

While dwelling on the negatives, there are a few more typographical errors than I would like to see in a couple of sections (including some leftover references to the classes accompanying the notes from which the book was derived). These do not substantially detract from readability although the odd source code error may induce questions to tutors.

Given the method of publication, I expect that it should be quite easy for the authors to sort out these problems quickly.

Returning to the strengths of the book and another personal moment of revelation; a particularly rewarding insight is that Haskell naturally falls into the role of a medium for teaching beginning students the art of reasoning about software design. I suppose that fact should be obvious given the genesis of Haskell but for me, somehow, Haskell's elegance often seems to fall through the cracks when in other forums and books I bog down in the theoretical and experimental aspects of the language itself and of functional programming in general.

Here, functional programming in its Haskelly guise provides a crystal clear teaching environment, rather than being an end in itself or a heavy duty research tool.

There is some strong competition with respect to introductory programming with Haskell, notably Hudak's *The Haskell School of Expression* and Thompson's *The Craft of Functional Programming*.

Each of those books covers substantially more ground (416 and 507 pages respectively) but also costs more. The best prices I could get in Brisbane were $AU 75 and $AU 79.95 respectively. You will have to decide on which book to use based on the needs of your students; particularly on whether they are likely to be computer science majors who will study functional programming in more detail later on. If they are then they may need one of the more substantial books. Otherwise *An Introduction to Computing with Haskell* should be a nice fit.

Leaving the choice of language for introductory programming courses aside, there are countless introductory programming books in non-functional languages, but I have not read one that provides the same kind of background as the subject of this review.

I think that Chakravarty and Keller deserve applause for providing a cost-effective platform for setting new programmers on the road to software engineering enlightenment. The unusual combination of price point and content should push the book into the hands of "toe-in-the-water" non-computer science majors such as accountancy and engineering students who are likely to swim in the waters of management after they graduate. I hope that happens, as it may encourage a more methodical approach to corporate management of software planning and development in the future. Of course, we may need a special magic potion, which infuses course coordinators and curriculum planners alike with a sense of adventure.

References

Hudak, P. (2000) *The Haskell School of Expression: Learning Functional Programming Through Multimedia.* Cambridge University Press.

Jensen, K. and Wirth, N. (1974) *The Pascal User Manual and Report*. Springer-Verlag.

Thompson, S. (1999) *Haskell: The Craft of Functional Programming*. Second Edition. Addison-Wesley.

Mike Thomas
Paradigm Geophysical
Brisbane, Australia

Practical Aspects of Declarative Languages, Shriram Krishnamurthi and C. R. Ramakrishnan (eds.), LNCS 2257, Springer-Verlag, 2002. ISBN 3-540-43092-X. Price £31.50, pp. 359. DOI: 10.1017/S095679680322503X

Introduction

The International Symposium on the Practical Aspects of Declarative Languages (PADL) provides a venue for researchers and application builders to discuss recent advances in the implementation and use of declarative programming techniques. The 4th PADL symposium took place as a part of the array of events surrounding POPL 2002 at Portland, Oregon, USA.

Categorizing interesting research at a symposium of this sort into a set of clean categories is next to impossible, as the most interesting work bridges several categories. The typical PADL paper might offer a technical improvement in language implementation, show how to apply this improvement to a task area such as web processing, and also describe an application in a particular domain such as medical informatics. However, one can identify several overarching themes to the work presented at PADL'02, themes that reflect important trends in declarative programming and in computing more generally.

The Papers

One timely theme involves the **integration** of declarative techniques with object-oriented programming to build on their respective strengths. Jayaraman and Tambay's *Modeling Engineering Structures with Constrained Objects* describes the use of constraints as an essential part of the internal definition of objects. This integration allows the authors to build engineering models that both provide the continuity of object-oriented decompositions and support constraint-based reasoning about the interactions of objects. In a different vein, Torgersson's *Declarative Programming and Clinical Medicine* shows how to build a practical medical informatics system by combining definition-based programming of the data model in the language Gisela with object-oriented programming of the application and GUI in Objective C.

Several papers focus on a second timely theme, that of **web-centric applications** and especially XML processing. My favorite paper of this sort was Kiselyov's *A Better XML Parser through Functional Programming*. Kiselyov treats XML parsing as a folding over XML documents, which allows his SSAX program to do both DOM parsing and stream-based parsing completely and efficiently. Gupta et al.'s *Semantic Filtering: Logic Programming's Killer App?* gives a technique for reliably translating between two notations based on denotational semantics that can be applied to many problems, most notably to the resolution of XML DTDs. Thiemann addresses a different element of web-based computing in his *WASH/CGI: Server-Side Scripting with Sessions and Typed, Compositional Forms*, which details a Haskell library that allows programmers to define CGI-style programs by describing layouts and data relationships declaratively.

A third timely theme involves **program transformation**. This topic has always been important in the programming languages and functional programming communities, but it has now also become a point of focus in the larger software development community, in the form of refactoring. Lammel and Visser's *Typed Combinators for Generic Traversal* describes an important advance in the use of combinators to define traversal strategies that are both generic (in that they operate over any type) and non-generic (in that they can provide type-specific behavior for subtypes). This technique scales well to transformations of over large code bases, which makes industrial-strength refactoring possible. But it also exploits functional genericity in order to simplify maintenance of transformation code in the face of changes in the application domain. Rhiger's *Compiling Embedded Programs to Byte Code* also defines a library of combinators, showing how such a library can be used to implement just-in-time compilation of embedded programs. Finally, *Compiler Construction in Higher Order Logic Programming* by Liang discusses the benefits of using higher-order languages across the spectrum of program Transformations that constitute a compiler.

The largest category of papers in these proceedings deals with various means for improving the **implementation** of declarative language systems. Several of these papers focus on compiler-level concerns. Johansson and Sagers explore the use of linear scan algorithms for register allocation in an Erlang compiler. Their empirical results show the effects of choosing different methods for implementing the linear scan, including instruction ordering, liveness analysis, renaming, and spilling heuristics. Vandeginste et al. describe improvements to Prolog garbage collectors in two areas. First, they give a technique for preserving heap segment order in a garbage collector that does copying; second, they show techniques for maintaining generation lines in a multigenerational collector. Castro et al. compare different implementations of the SLGWAM architecture for evaluating Prolog programs, which uses tabling to manage multiple computation paths. They focus on how the various implementations store suspended computations and decide when to resume them and consider the effects of these decisions. Finally, Yang and Gregory propose the use of nested domain variables as a mechanism for implementing more efficient lookahead in non-determinate logic programs. They give preliminary results showing how nested domains can allow an interpreter to prune inconsistent branches of computation earlier.

Several implementation-oriented papers deal with **higher-level issues**. Shen et al. describe the choices they made in implementing the external language interface in their language ECLiPSe. The central goal of their approach is to define a generic interface in a way that allows their language to interact with multiple other languages without having to reimplement any common components of the interface. This sort of approach bodes well for making declarative languages more useful as the glue that binds together programs written in a variety of other languages, a purpose for which functional programming is well suited and which is of growing importance in web-based applications. Bunus and Fritzson present their framework for supporting the debugging of programs written in equation-based languages, which uses static analysis to find errors and to annotate programs with information that can help the debugger provide meaningful feedback to the programmer. Gallagher and Puebla show how abstract interpretation can be used to do program analysis over non-deterministic types, yielding some of the precision of set constraints for this class of programs.

Finally, a few papers in the collection seem driven by their **application area**. This is not to say that the papers have less theoretical importance, but that they demonstrate something valuable about the use of declarative programming in a particular domain. I was most intrigued by Erwig and Burnett's paper, *Adding Apples and Oranges*, which treats spreadsheet programming formally, using the concept of units in place of types. This application area offers great opportunities to the declarative programming community, given the growing number of spreadsheet users in the world and the empirical data on spreadsheet errors. Erwig and Burnett develop a system of units evocative of type theory but with many benefits inherent in the concrete relationships among units and values. This work offers the potential for supporting spreadsheet programmers with better debugging tools, better error prevention, and better understanding of their domain models. The paper *Event-Driven FRP* by Wan et al. extends work on functional reactive programming into domains characterized by multiple external events, such as autonomous robotics. The authors describe a small language, E-FRP, and show how this language can be compiled into efficient imperative code. Karczmarczuk's *Functional Approach to Texture Generation* describes some marvellous applications of functional programming in the realm of image synthesis, in particular for generating textures for images. These applications are built on a library of texture combinators and transformers, which enable the generation of relatively complex textures from relatively small and straightforward pieces of code.

As someone who was unable to attend PADL'02, I greatly appreciate that the proceedings include useful extended abstracts – and even a full paper for the symposium's three **invited talks**. So often, the contents of invited talks are lost to all but those in attendance, because the proceedings contain nothing more than a one-paragraph abstract of the talk that barely expands on its title. But Catherine Meadows's extended abstract, *Using a Declarative Language to Build an Experimental Analysis Tool*, and Veronica Dahl's extended abstract *How to Talk to Your Computer So that It Will Listen* gave me a good sense of their talks and provided ample citations to the Literature so that I can learn more. J. Strother Moore even provided a full-length paper, Boyer and Moore's *Single-Threaded Objects in ACL2*, which gives a complete treatment of how ACL2 can efficiently execute models of multiprocessors using declarative representations of single-threaded objects.

Conclusion

This simple volume presents an engaging snapshot of the state of declarative programming languages and their practical implementation and application. The editors and symposium program committee have done a nice job selecting high-quality papers that represent the broad spectrum of work being done in this field. As described above, editors are often hard-pressed to organize a proceedings, especially one with only eighteen papers, as anything more than a collection of papers. I could not tell if the Krishnamurthi and Ramakrishnan had organized this collection in any particular way, though the ordering of the papers occasionally resembled by own organization.

As I studied this book, I was impressed with the diversity of the ideas being explored by the declarative programming community. This style of programming offers many benefits to the builders of large programs, yet we still have only scratched the surface of how we can apply declarative programming in an efficient way. I recommend this book to anyone interested in the possibilities available in using declarative programming to build large systems, and to anyone interested in how these languages can be processed more efficiently.

Eugene Wallingford
University of Northern Iowa
USA