JFP **14** (5): 588–600, September 2004.
Printed in the United Kingdom

Book reviews

Knowledge Representation, Reasoning and Declarative Problem Solving by C. Baral, Cambridge University Press, 2003.
DOI: 10.1017/S0956796804215325

This book describes theoretical results about AnsProlog* that have been obtained over the past decade. AnsProlog* or *Prolog with Answer Sets*[1] is a variation of the Prolog programming language, and extends the language by allowing clauses of the form:

$$(C) \qquad L_1 \texttt{ or} \ldots \texttt{ or } L_k \leftarrow L_1, \ldots, L_m, \texttt{not } L_{m+1}, \ldots, \texttt{not } L_n$$

in the program. The L_i's are the literals (or atoms) of the Prolog language and may be supplied with a prefix $\neg$ sign, indicating the negation of a literal, while the prefix of `not` indicates negation as failure. Hence the semantics of the clause (C) may be read as follows: *if all the literals* $L1, \ldots, L_m$ *are true and all the literals* $L_{m+1}, \ldots, L_n$ *can be safely assumed false then at least one of the literals* $L_1, \ldots, L_k$ *is true.* (The actual semantics of each AnsProlog* program will be defined in terms of the Herbrand Universe of ground terms and the Herbrand Base of ground atoms.)

The book takes the approach that the clause (C) is the most general form of a clause in the AnsProlog* language and so various subclasses of AnsProlog* can be defined by restricting this clause. For example: an AnsProlog$^{\neg\texttt{not}}$ program is when none of the clauses of a program contain the prefix `not`. In this respect, the book discusses the tractability, the complexity, the expressibility of the various subclasses of AnsProlog* based on the premise that AnsProlog* is both an excellent knowledge representation language and that it has a number of advantages over the Prolog language implementations based on SLDNF. For example, the ordering of goals within a clause and the ordering of clauses within a Prolog program affects whether a solution can or cannot be found (i.e. the program might get into an infinite loop); but not this is not the case within an AnsProlog* program. The reason being is that the semantics of an implementation of the AnsProlog* language can be thought of as allowing all models of the program to exist and then by using the clauses within the program, to impose restrictions on these models. The actual model(s) produced can then be interpreted in either a bi-valent (where a ground atom is either true or false) fashion or a tri-valent (where a ground atom is either true, false or unknown) fashion. The implementation algorithms describing how to restrict the models are described in Chapter 7 and two systems implementing the AnsProlog* language (and various subclasses), viz: (i) `lparse+smodels` and (ii) `dlv` are discussed in Chapter 8. The `lparse+smodels` program produces the *stable models* (or bi-valent) implementation, while the `dlv` produces the *well-founded models* (or tri-valent) implementation. Baral does note that both systems are under development and so implying that Chapter 8 may be out of date within a few years. However this aspect is compensated by Baral having a website `www.baral.us/bookone` where hypertext links to both the two systems and an errata/additional notes for the book are presented.

On the application side, the book is peppered with many examples and simple programs illustrating the current point being made in the text. For example: how various forms of the

[1] AnsProlog* is sometimes called A-Prolog in the literature.

clause (C) may be used to solve constraints satisfaction problems. On the whole this is a good textbook writing style, but the style is compromised by having Chapter 8 (the chapter providing the hand-on experience using the AnsProlog* language) situated towards the end of the book.

For a book having the title of *Knowledge Representation*, and arguing that AnsProlog* is an excellent knowledge representation language, it is a pity that the only area of application covered in depth is Planning (in the Artificial Intelligence (AI) sense). There are other areas that have emerged on the AI scene over the last decade, (such as: qualitative reasoning, data mining, natural language processing, machine learning, inductive logic, expert system shells, etc.) and as such, all these areas have an association with implementations in Prolog but none of them are covered in any depth in the book. Seen this way, the book presents a lop-sided impression of being too theoretical. In terms of using the AnsProlog* language to encode and solve Planning problems, it is refreshing to see Planning presented with temporal side of it (when to do something) being treated on equal par with the action side (what to do), a fact that is even neglected in books on AI Planning. Also, the fact that the Planning Domain Definition Language (PDDL) is discussed indicates that Baral has spent some time talking the AI Planning Community, since the standardization of using PDDL to encode Planning problems has only be recently adopted.

In conclusion, Baral presents the book as a textbook that can be used both at an undergraduate level and a graduate level; indicating that the whole of Chapters 1, 2, 8 and part of Chapters 3–5 are to be used at the former level and the whole of Chapters 1–8 at the latter level. At the undergraduate level, comparing this book with a classic such as Bratko (Bratko, 2001), indicates the text should be pitched at an advanced undergraduate/MSc level. On further reflection this conclusion is not surprising since Baral assumes that the reader is knowledgeable about the Prolog language syntax, the semantics of the Prolog language, set theory, basic model theory, propositional and predicate logic. Therefore, for a typical undergraduate using this book, the learning curve will be steep. For example: the four pages of appendixes summarizing ordinals, lattices, fix-point theory, and Turing Machines will be barely comprehensible to an undergraduate. Matters are different at a graduate level and it is at this level that the book is best used, and certainly will provide a excellent start to researchers of the AnsProlog* language.

References

Bratko, I. (2001) *PROLOG: Programming for Artificial Intelligence, 3rd edn.* Addison-Wesley.

Nimish Shah
Nimish_Shah@onetel.com

Reasoning About Program Transformations: Imperative Programming and Flow of Data by Jean-Francois Collard, Springer-Verlag, 2003, ISBN 0-387-95391-4.
DOI: 10.1017/S0956796804225321

This 237 page A5 format work of eleven chapters also includes a list of 90 references and a small index supplemented by an impressive array of nearly 160 figures of which most are fragments of code. The chapters are separated into three parts; namely Basic Concepts, Analyses and Data Flow, and Expansion. It is well written and both instructive and (even) enjoyable to read.

An impression of Jean-Francois Collard's aims is provided by the following quotes. He tells us that his book is not "a survey of compilation techniques" but rather, about "analyses which

extract the flow of data which imperative programming hides through its use and reuse of memory."

With this in mind, the principal theme deals with increasing by means of static analysis the amount of information available to those parts of a compiler which transform and reorder execution pathways. At the heart of that theme is the distinction between what Collard calls *classical* statement analysis and *statement instance* analysis.

Statement instance analysis uses each determinable execution instance of a statement to build a detailed model of possible execution and data pathways for a program at the level of the ordering of memory reads and writes to specific locations. That model may then be used to plan transformations of those pathways to maximise the potential for parallelism or other kinds of efficiency gain.

A mathematical framework supporting this model is sketched which is able to label instances in loops and recursive function calls and to symbolically describe and manipulate execution order, statement instance and storage relationships. The heart of this framework is a relatively simple algebraic methodology based on sets and logic. The book does go substantially beyond that simple level in some places but it is not necessary to be especially well read at a mathematical level to benefit from reading it. Automata (statement instance labelling), polytopes (array dataflow) and equivalence classes (maximal static expansion) are also used to build the framework.

Collard applies the framework to numerous snippets of example code in the context of some standard static analysis techniques with special emphasis on reaching definition analysis. He analyses statement instance relationships under various kinds of iteration constructs (including recursion) and also array dataflow analysis, demonstrating the effect of decision/branch points within those constructs on the amount of information potentially available. Single-assignment form, static single assignment, maximal static expansion and parallel languages are all examined.

I found myself drawn into practising the techniques I had just digested by some excellent worked examples and exercises. W. Pugh's research group gets a nod through the use of their program *Omega* to symbolically solve many of those examples. There is a strong influence throughout the book from the work of Pugh and also P. Feautrier, with each person named as an author of ten of the references independently of each other. The reference list provided in this book is a nice starting point for some serious reading on developments in high performance compiler optimisations over the last decade.

As an aside, the *Omega* download site provides only source code or a Linux binary. However, after minor changes I was able to build and use the program under Cygwin on Windows.

You might think from the title that the only people who might benefit from this book in the functional programming community would be ML and Lisp compiler writers, but that would be a mistake because the deeper aspects of the book should be widely applicable, particularly for students. The historical links between such functional programming cornerstones as referential transparency and parallel computation lurks in a silent way beneath much of this book and is another reason for student functional programmers to take a look.

These links become apparent in the third part of the book which examines the preservation of the meaning of programs under transformation, parallelism and conversion to single assignment forms and, importantly, how to do it economically. You may have noticed that I used the word 'determinable' earlier on. The methods set out by Collard are not magical, and do come with a computational and developmental price tag – the usual compiler writer's constraints. For example, transformation of imperative programs to single assignment forms can lead to an uneconomical blow-out of the amount of computation required to evaluate so-called *oracle functions*. Instancewise analysis helps to reduce the number of oracle functions which might otherwise have been needed. However it can't remove oracles caused, for example, by indeterminate branches.

So how do we work around that problem? One way of maximising parallelism without the added cost of oracle functions is by *maximal static expansion* to which Collard devotes Chapter 9. This construct sets a theoretical upper bound on the amount of parallelism present in a program free of oracle functions and in which the results of reaching definitions are given. The theoretical development is supported by an algorithm and some exercises.

Having considered parallel languages, Collard rounds out his program transformation theme by speculating on a future in which compilers use derived information on dataflow to recognise complex algorithms and potentially substitute better ones. There is a place for such capability in translating 'legacy libraries' from one language to another. Such tools (without algorithm recognition of course) have been quite successful in converting Fortran numerical libraries. The application of *f2c* and *f2j* in translating the popular LAPACK library into C (CLAPACK) and Java (JLAPACK) is more widely recognised than the less well known use of f2l on various Fortran numerical routines (to Common Lisp) for the recently revitalised open source symbolic algebra system Maxima. I personally use the results of two of those projects regularly. Imagine extracting the fundamental algorithms from LAPACK to form a new easy to read and maintain Haskell or SML equivalent! There is, of course, a long way to go here.

I think this book is a great example of the rewards available to those prepared to think mathematically about problems in computer science. No engineer would consider building a bridge without resort to a mathematical model and if the research at the heart of this book is correct then likewise, apparently, for compiler writers. On the other hand, a former university lecturer recently told me he felt that functional programming is probably left behind in universities by information technology graduates because it is too mathematical. It does not take too much of a leap of the imagination to see that the same could apply to compiler writers! This is an interesting bind for educators and employers to work through.

Putting aside the principal theme for the moment, I would like to point out that there are other strands running through the book which appeal to me and which make it more interesting than the average textbook.

Collard not only summarises a useful collection of research papers, but provides a teaching framework for the main results from that work and some beautiful applications of certain elementary abstract mathematical concepts (already mentioned earlier in this review). Those applications remind me of the wonderful practical insights that group theory brings to the field of molecular spectroscopy. I was pleased to see that equivalence classes actually have a use – I always believed that my third year abstract algebra course would find a practical end eventually.

This book is concise. If your mind wanders as you read on the train or you skip some exercises, you may start to lose track quickly. Collard starts off in an easygoing manner, but soon piles on the definitions. Apart from Chapter 9, however, most of the development is easily digested with the author sometimes intervening on your thought processes during examples to ensure that subtle points are not missed. It made me feel, at times, that Collard was standing in the same room as myself!

The text draws heavily on code fragments and diagrams, so one particularly welcome feature is the wealth of figures. I rarely turned more than two leaves to find the figure (usually a code fragment) under discussion. I think that the book could have been improved with the addition of both a list of entries in the index under a new heading *definitions* and a glossary listing those symbols used to represent various sets and other entities during the mathematical development. Good engineering and science texts usually do have such a glossary.

It is hard to say where the research results brought to us so effectively by Collard will take compiler design. I suppose that time will tell as the results of applying them to commercially successful production compilers become known. Whatever the case, the good thing about this book is that even if those results do happen to fall quickly into the weedy garden of forgotten science, readers should nevertheless carry lasting benefits away with them by

learning to deconstruct software complexity with algebraic constructs as simple as sets and logic and by developing better insights into writing code which gets the most out of a compiler.

Mike Thomas

Développement d'applications avec Objective CAML by E. Chailloux, P. Manoury and B. Pagano, O'Reilley, 2003
DOI: 10.1017/S0956796804235328

The Caml language (an acronym for Categorical Abstract Machine Language) has its root in the work of Pierre-Louis Curien on categorical combinators and in the ML (Meta Language) designed by Milner as a part of his LCF (Logic for Computable Functionals) verification system. In 1990, Xavier Leroy designed a new implementation of Caml, known as Caml Light, which was more portable and could be run efficiently on personal computers. This version is in widespread use in research and education.

There are many books on Caml (Weis and Leroy, 1993; Accart-Hardin and Donzeau-Gouge Viguié, 1993; Cousineau and Mauny, 1995, 1998). Apart from Weis and Leroy (1993), none of these contains complete information on the programming environment offered by the Caml Light distribution, but rather they focus on functional programming. In 1996, Objective Caml was released. It extends Caml Light, adding a module system, object-oriented features and a native code compiler. The Objective Caml distribution also contains a number of new tools.

This is the first book about Objective Caml. In the introduction, the authors write:

> Objective Caml has never been the subject of a presentation to the "general public". This is the task which the authors have set themselves, giving this exposition three objectives:
>
> 1. To give an in-depth description of the Objective Caml language, its libraries and its development environment.
> 2. To show and explain the concepts underlying the programming paradigms embodied in Objective Caml.
> 3. To illustrate through numerous examples how Objective Caml can serve as the development language for various classes of applications.

The book is self-contained, and is suitable for the 'general public'. Nevertheless, the first chapters are rather dry for a novice reader. Thus it is better for the reader to have a bachelor degree of computer science, or at least to have read an introduction to the principles of functional programming (Cousineau and Mauny, 1995, 1998). The authors made a balance between writing an introductory text and a reference book on programming in Objective Caml by presenting not only the language features but also the practical use of the programming environment. The indices make the book particularly useful as a reference.

The book is organized into four parts. Two of them present the elements of the programming paradigms embodied in the language and the others present the Objective Caml tools and libraries and applications written using these tools, libraries and paradigms. The first part is devoted to functional and imperative programming; the third part is devoted to modular and object-oriented programming. Each part is divided in four subparts: a presentation of each style of programming, a comparison between the two styles, and applications.

The presentation of functional and imperative programming in Objective Caml is classical. The main strength of this part, and indeed the whole book, is the number and variety of applications used to illustrate the language. This is in stark contrast with the usual books on programming with Caml and makes the book very popular with students. Chapter 5 is devoted to the Graphics module and is used to illustrate the functional and imperative style. It

ends with a program for a calculator with a graphical user interface. The next chapter is about additional applications: queries in a small database, a BASIC interpreter and a minesweeper game using the graphical tools developed in chapter 5. Each chapter also contains exercises whose solutions are given on the CDROM and is ended by a 'To Learn More' section which gives relevant references and web links.

The second part is composed of six chapters about the tools and libraries of the Objective Caml programming environment, together with one chapter of applications. This kind of material is very valuable for working programmers. Chapter 8 about the main libraries and Chapter 11 about lexical and syntactic analysis are relatively mainstream but with some sections which cannot be found elsewhere, such as the section of Chapter 8 about the Marshal module (to serialize and deserialize values) and the section of Chapter 11 about the Str module (rational expressions). Some chapters of this part have no equivalent in other books because they are about features not present in previous versions of Caml. This is the case for the compilation modes (byte-code and native code) in Chapter 7, program analysis (debugging and profiling) in Chapter 10 and interoperability with C in Chapter 12, all of which are very important for practical use of the language.

Chapter 9 presents the Gc and Weak modules: the former gives some information about the stack and allows garbage collection on the stack to be controlled; the latter allows the use of weak pointers. But of course to write something understandable about these modules, the authors spent most of the chapter to explain what is garbage collection and to describe several GC algorithms. This constitutes a very good introduction to garbage collection. The last chapter of this part presents a library of graphical components and a program to find least cost paths through weighted directed graphs which uses the graphical components library and weak pointers.

The rest of the book presents material specific to Objective Caml and not present in Caml Light. The third part contains a chapter about the module system and a chapter about objects. Each chapter is very well written. Notions are illustrated through a substantial collection of code and the classic pitfalls to avoid are explained. A third chapter about the comparison of these two models of organization show the advantages, limitations and drawbacks of each model and the interest to mix the two models. The last chapter presents applications. The first application uses a set of parameterized modules. One module implements the minimax-$\alpha\beta$ algorithm and another implements a graphical interface. Two instantiations of these ideas are given: the Connect Four and Stonehenge games. The second application uses objects to simulate a world of robots.

Concurrency and distributed programming in Objective Caml are the subjects of the last part. Unix tasks, Objective Caml threads and IP-based communication are presented in three chapters. The content of these chapters are applied to the programming of a client-server tool-box which is then used to change the robot application to a distributed application. The last application is a small HTPP server.

Chapter 22 and the conclusion together constitute a review of the language. They give the advantages and drawbacks of the development of applications in Objective Caml and compares Ocaml to several functional languages and Java. If Objective Caml is already an academic success both in research and teaching, it remains unsuccessful in industry. The authors point out three conditions to be fulfilled to promote industrial use of Ocaml: to ensure backward compatibility as the language develops; to specify the language to standardize it; and to have a development environment with interfaces to other systems, such as a portable graphic interface, CORBA and databases.

These conditions seem reasonable and the Caml Hump, the collection of links of Caml-related libraries and tools (http://caml.inria.fr/humps) contains some of the needed software even if it is not yet mature enough. However, I hope that education is also an important part. If our students are convinced that Objective Caml is a useful language and can check that Objective Caml increases productivity and the quality of code but mainly that it can be used to program 'real' applications, maybe they would support the use of Objective Caml

in industry. The book of Emmanuel Chailloux, Pascal Manoury and Bruno Pagano is very helpful to attain this goal.

Thus, if you read French you should buy the book, otherwise

http://caml.inria.fr/oreilly-book

should be in your bookmarks: "Developing Applications with Objective Caml" is the preliminary English translation of the book, with updates incorporated, and Russian and Japanese versions are also planned.

References

Accart-Hardin, T. and Donzeau-Gouge Viguié, V. (1993) *Concepts et outils de programmation, Du fonctionnel à l'impératif*. InterEditions, Paris.

Cousineau, G. and Mauny, M. (1995) *Approche Fonctionnelle de la Programmation*. Ediscience International.

Cousineau, G. and Mauny, M. (1998) *The Functional Approach to Programming*. Cambridge University Press.

Weis, P. and Leroy, X. (1993) *Le Langage Caml*. InterEditions.

Frèdèric Loulergue

The Fun of Programming edited by Jeremy Gibbons and Oege de Moor, Palgrave Macmillan, 2003, ISBN 1-4039-0772-2 (HB), 0-333-99285-7 (SB).
DOI: 10.1017/S0956796804245324

This excellent book was written in celebration of the work of Richard Bird, on his sixtieth birthday. Its title is taken from one of Richard Bird's lectures.

The book is intended as a text-book for a second or advanced course on functional programming, or for a course of self-study. Each of the 12 chapters is written by a leading researcher in functional programming, and is focused on a substantial idea. Most chapters contain exercises that range from verification of assertions in the text to more demanding, open-ended questions that require further exploration of the material. The language Haskell is used throughout as a vehicle for expressing programming ideas. The prerequisites are a basic understanding of programming in a lazy functional language, such as might be obtained from an introductory undergraduate course. There is a website associated with the book (http://web.comlab.ox.ac.uk/oucl/publications/books/fop) that contains pointers to code developed in the various contributions, and to related websites.

What is welcome about this book is that to a high degree the contributors have written a book primarily about programming, and almost incidentally a book about functional programming. In several chapters there is a penetrating conceptual analysis of the matter at hand, on which a superstructure of code is erected. The code happens to be written in a functional language, but this is not the main point. It may be that Haskell is a particularly apt medium for the expression of programming ideas, but (for the reviewer) the real message of the book is not the medium, but the demonstrations in several cases of what it is that programmers actually do. This is not an introverted book, focused just on functional programming, still less on programming with Haskell.

A little less than half of the contributions encompass a wide variety of problems occurring 'in nature' – representation and description of music (Hudak), graphical images (Elliott), hardware (Claessen, Sheeran and Singh), pretty-printing (Wadler), logic programming (Spivey and Seres) and financial contracts (Peyton Jones and Eber). These papers demonstrate in different ways the power of a declarative approach in their various domains.

The paper by Claessen, Sheeran and Singh describes Lava: an experimental tool for hardware description, simulation and verification that has been successfully applied in the development of FPGA cores, and has attracted the interest of a major manufacturer of these devices. Interestingly, Lava plays with a number of other tools used in hardware design and analysis, such as CAD tools, model checkers, automated theorem provers, and device-specific configuration tools. The approach is refreshingly free from functional programming 'religion'. Instead, the power of the declarative approach is demonstrated in a series of examples that show how patterns of interaction expressed by Lava combinators become amenable to algebraic reasoning. The paper is well-signposted for further study.

The paper by Peyton-Jones and Eber on financial contracts is this reviewer's favourite. It proposes a library of combinators defined in Haskell that represents in a precise form and with a precise meaning a bewildering variety of complex contracts that are used in the world of financial trading. A lot of thought has gone into the selection of those combinators, and much urgently needed light has (apparently) been shed on the subject. Programmers quite often find themselves faced with the problem of analysing and gaining some intellectual control over a problem domain that does not present itself in elegant, algebraic chunks but rather in "an immense soup" (to borrow a phrase of the authors). In the case of this paper, the results of the analysis are expressed declaratively in the form of a library of combinators. Moreover, the authors sketch a valuation semantics for contracts that is informed by experience gained in the design, semantics and implementation of programming languages.

The papers by Elliott (on graphics) and Hudak (on music) are delightful for their simplicity. In Hudak's paper, musical structures are represented using a datatype built from primitive notes with constructors for sequential composition, and an interesting form of parallel composition. An element of this datatype is interpreted by a performance, modelled as a temporally ordered sequence of events. The interpretation is given by a function 'perform' that gives rise to what Hudak calls an algebra of music. In Elliott's paper the representation of an image is particularly simple: a function from 2-dimensional point coordinates to 'pixels' of arbitrary type. A small number of ingenious definitions puts into the reader's hands a kit with which to generate and explore a variety of delightful images. The paper serves very effectively as an appetiser to Elliott's system Pan, which is an experimental embedded language and compiler for image synthesis and manipulation.

The paper by Wadler, which is a model of clarity, is concerned with the notoriously tricky problem of pretty printing. Wadler develops a pretty printer library which is exceptionally simple to understand, and use. The code occupies little more than a page. One lesson to learn from this paper is perhaps the value of careful attention to algebraic properties in program development.

Spivey and Seres's paper is concerned with two key features of logic programming, and ways of writing logic programs within Haskell. The first feature they address is that of searching for multiple answers to logic program queries. The starting idea is Wadler's fundamental observation that programs with multiple answers can be represented in lazy languages by functions to list or stream types. However, to obtain a *fair* form of searching, a form of 'diagonalising' product of lists must sometimes be used. Spivey and Seres go further, and show how a search program can be factored into a part that is independent of the search strategy, and a part that determines whether the strategy should be depth-first, diagonalised or breadth-first. The second feature of logic programming addressed is that of 'logical variables' which record the constraints expressed by a logic program.

Other contributions to the book are perhaps concerned less with applications, but rather with more inward-looking technical matters. This is not meant as a criticism, but only as a rough categorisation.

Claessen and Hughes in their contribution give an overview of the tool QuickCheck developed at Chalmers, that provides languages (embedded in Haskell) for writing properties of programs, and for generating data to be used in automated testing. They illustrate the

use of QuickCheck in a number of examples, culminating in a simple theorem-prover for classical propositional calculus. As programmers know all too well, problems arise in the first place because 'first-draft' specifications often fail to express what was intended. The process of testing a program is very often (and most valuably) a matter of deepening and repairing one's understanding (and documentation!) of what it is supposed to do. It commonly takes a few iterations before one begins to find bugs in the program itself. An important point about QuickCheck is that it provides means of monitoring and controlling test coverage. Overall, QuickCheck seems to be a substantial contribution to the viability of Haskell in real applications.

Okasaki's paper gives a number of elegant implementations of priority queues based on binary trees. The broader subject here is functional data structures, and the analysis of their resource usage. The problems of estimating resource usage in lazy functional languages, and the pitfalls of naively transferring intuitions developed in imperative programming are notorious. Okasaki's chief contribution to this difficult subject is the notion of *amortised* analysis of data-structure operations, where one examines the cost of a sequence of operations. He illustrates two techniques, based respectively on metaphors of credits and debits. The paper is clearly written, and a valuable introduction to difficult and under-explored territory.

The paper by Paterson concerns the representation of 'notions of computation' (function-like constructs that may involve for example state-transformation and non-determinism). Paterson expounds a notion called 'arrows' (due to Hughes, and independently to others working in categorical semantics), that generalise Haskell's monads. One lesson to take from the paper is perhaps that there is 'more to life' (or computation) than monads! Using arrows in practice seems to require preprocessor support for a form of arrow notation. (Such a preprocessor can be obtained via a link on the book's web page, given above.)

Gibbons's contribution is a chapter illustrating the pervasive nature of folds and unfolds. A number of algorithms for operating on lists, natural numbers and trees are teased apart to expose the folds and unfolds lying at their core (sometimes in combination). As far as folds go, Gibbons's thesis seems nowadays dangerously close to platitude (though the examples may be sometimes surprising); but part of his agenda is to illustrate that unfolds are as simple and useful as folds.

Sittamplam and de Moor are concerned with prospects for automatically applying optimising transformation steps to a 'active source' program. Roughly speaking, a clear but perhaps inefficient program is annotated with hints that can drive a transformation system towards a more complex but more efficient program that provably meets the same specification. It should be stressed that systems of this kind have for decades been merely a dream. Such a transformation system is MAG, that can be downloaded via the web page for the book.

Hinze's chapter (the last) is unique in that it contemplates a new construct in Haskell – a 'with' construct that allows the expression of per-constructor type constraints on the parameter of a data type definition. The subject here is extensions to the Hindley-Milner type system. In the reviewer's opinion, many of the examples are more naturally treated by use of dependent types arising from the logical tradition. The paper is however delightfully written, and abounds in ingenious examples.

The first 6 words of the preface are "Functional programming has come of age". At any rate, it seems to be emerging from a period of introspective adolescence. There is plenty of evidence in this book. In large part, this is because the problems addressed (problem analysis, specification based testing) are often those of programming, rather than specifically functional programming.

The book is certainly an excellent one for self-study. It may perhaps also be used as the text for an seminar-based course on functional programming. Most of the contributions have an open-ended, introductory character, and seem to work best as invitations or stimulations towards further study or research.

The production of the book seems almost flawless. (The last line of page 109 seems to have suffered a printing glitch.) The index is short but useful. There is a bibliography of 140 items. The paperback is very reasonably priced. I'd certainly buy it myself.

PETER HANCOCK

Programming Methodology A. McIver and C. Morgan, editors, Springer-Verlag, 2002, ISBN 0387953493.
DOI: 10.1017/S0956796804255320

As the title indicates, the contents of this book fall under the general heading of programming methodology, although the description as the "state of the art survey on key foundational topics in programming methodology" is a better reflection of its contents. Indeed, the work contained within grew from the discussions within the IFIP Working Group 2.3 which operates under the title "programming methodology" and seeks to "assess methodologies for creating and improving the quality of software and systems". The membership of WG 2.3 contains many who have been at the forefront of work in this area (including Back, Dijkstra, Gries and Hoare), and the scope and contents of this book reflect this pedigree.

The book is a collection of 20 articles, all falling under the broad theme of the foundations of programming methodology, and written in a variety of styles. Some are introductory, some offer a survey and some contain substantial technical results. They also vary in length from short (about eight pages) to long (30 pages). All chapters are, however, worth reading, each one offering the reader plenty to think about.

The chapters are grouped into nine themes, themselves grouped into three parts. Part 1 concerns models and correctness with themes of:

- concurrency and interaction
- logical approaches to asynchrony, and
- systems and real time.

Jones opens Part 1 with an essay providing an overview of his concerns with respect to concurrency and composition. This is followed by a more technical contribution from Back and von Wright, which presents some of their recent work on contracts undertaken in their action systems framework.

The essence of the second theme in Part 1 is the necessity to reason about concurrent processes which is often achieved by serialising their execution. Misra's contribution here is the definition of a simple programming language whilst that of Cohen is to introduce a new operator that replaces the linear-time "leads-to" with a compositional "achieves" in the context of Unity.

Section C concerns the modelling of real-time systems and offers two contrasting views. The first by Broy overviews his work on streams as a specification paradigm for real-time systems. The second by Hayes extends the refinement calculus to develop a framework for the description and development of real-time programs. The styles are complementary, with Hayes offering a more detailed derivation of some of the properties of his framework than Broy.

The final section in Part 1 has contributions by Jackson, Henderson and Bjørner, all concerning complexity in system specification. The first and third of these are more discursive in style, offering some general insights, whereas the second describes a particular framework for a dynamic architecture.

Parts 2 and 3 are shorter than the first. The second is titled "Programming techniques", although perhaps the foundation of programming would have been a more accurate title, for here we find work on the foundations of OO and type systems. The former includes papers on abstraction dependencies and models of objects and pointers, and the latter includes a very

nice short paper by Pierce that surveys the introduction and use of types in programming languages.

The final part bundles a number of chapters under the heading of "Applications and automated theories". The chapter by Shankar postulates that abstractions hold the key to integrating theorem proving and model checking, while Zave describes her approach to feature interaction in the telecoms domain. There follows a single paper on circuit design, and two on aspects of security. The first of these offers an intriguing insight into how observing the level of power consumption can undermine some of the assumptions made in encryption algorithm correctness. The collection concludes with a paper by the editors themselves on the interplay between non-determinism and probabilities in a guarded command language context.

Overall it can be seen that this book contains a wealth of material. All of it is interesting, and the majority accessible and well written. Whether all of programming methodology is covered is a moot point, but it certainly provides a comprehensive coverage of topics in the foundations of that subject, and to those working in that domain it can be thoroughly recommended.

John Derrick
University of Kent, UK

Program Construction: Calculating Implementations from Specifications by R.C. Backhouse, John Wiley & Sons, 2004.
DOI: 10.1017/S0956796804265327

The 1960s was the era when research in computer science started moving away from automata theory and formal languages towards the understanding of (imperative) programming in general. This was largely brought about by the *software crisis* of the 1960s when deadlines were overrun and faulty programs delivered. The pioneering paper by Hoare (Hoare, 1998) in 1969 (based on ideas by Floyd (Floyd, 1998)) led to the methodology of *axiomatic specifications*. This was further extended by (the late) Dijkstra at Eindhoven University of Technology in a series of papers and books, most notably is his 1976 book (Dijkstra, 1976), where he showed how a program could be developed alongside the proof that the program met a specification. This research theme was taken up by his fellow colleagues at Eindhoven and lead to books by Gries (1981) and Backhouse (1986), among others. The book under review is a complete rewrite by Backhouse of his 1986 book.

The theme of the book has changed, and this is reflected in the elimination of the phrase "and Verification" from the title, and the statement:

> The primary goal of the book is to show how programs are *constructed* to meet their specifications, by means of simple, mathematical calculations. The emphasis on construction is crucial; the fact that the calculations can be formally *verified* is also important, but much less so. (page x)

In abstract terms, the book may be divided into three parts: Chapters 1–8, Chapters 9–13 and Chapters 14–16.

The first part of the book introduces the concepts and philosophy of this style of program specification/construction (as opposed to other styles like model oriented or algebraic specifications) by way of discussing the invariants in the algorithm that Hoare presented in 1971.[1] To contrast this, the author presents a faulty, but published, implementation of the binary search (on a fixed size array) algorithm. Naturally, this leads onto a discussion of the properties of integers under given constraints (say the index i under the constraint $0 \leqslant i < N$).

[1] The problem was: Assume a very large collection of unsorted values; specify a given percentage of the collection; and then identify the smallest values, such that, collectively they equate to the percentage.

Other properties discussed are integer division, the floor and ceiling functions, minimum and maximum values. Intertwined in these discussions, the author introduces Propositional Logic. In comparison to his 1986 book, the author has changed his presentation of logic to that developed by Dijkstra (Dijkstra & Scholten, 1990) in 1990. This presentation is largely unknown and can be explained by examining the mathematical equation $P = Q = R$ (for the boolean variables P, Q, and R) and noting that the associative property of equality is almost never used in formal reasoning. Hence, one can distinguishes between the semantics of $P = Q = R$ [read as the mathematical meaning $(P = Q) \wedge (Q = R)$] and $P \equiv (Q \equiv R)$ [read as the associative meaning of equality] by introducing the operator '$\equiv$' (pronounced *equivales*) as a syntactic replacement for '$=$' in the second case. Backhouse takes the equivales operator as primitive and shows how the familiar algebraic properties (such as symmetry, associativity, etc.) of the boolean operators: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, and $\Leftarrow$ can be deduced.

The second part of the book moves onto the treatment of axiomatic specifications in the format (that has descended from Hoare) of $\{P\}\mathtt{S}\{Q\}$; where P, Q are mathematical statements and `S` is a statement in a programming language. The programming language that Backhouse uses is the Guarded Command Language, which – although is an abstract language – can be easily be mapped onto concrete languages like C, PASCAL, Java, C++, etc. In other presentations of the axiomatic specification/construction style, `S` is typically either an assignment statement, sequential composition, `if...then...else` conditional, iteration(`while`) loop, or the do-nothing (the `skip`) statement.[2] The book covers these five kinds of statements adequately. What makes the presentation interesting, from the reader's point of view, is the emphasis that the author places on reasoning with the mathematical statements P and Q and the concepts underpinning loop invariants. Indeed, two chapters are spent on introducing and using the notation $\langle \oplus bv : range : term \rangle$, where $\oplus \in \{\sum, \Pi, \forall, \exists, \Downarrow, \Uparrow, \equiv, \not\equiv\}$ (viz. summation, product, universal quantifier, existential quantifier, minimum, maximum, equivalence, non-equivalence), *range* is the constraint on the bound variable bv for the term *term* under consideration. (As an example: $\sum_{i=0}^{n} i^2$ is written as $\langle \sum i : 0 \leqslant i \leqslant n : i^2 \rangle$.) This notation and the relationship between the induction hypothesis (from an induction proof) and a loop invariant are then applied to `while` loop examples in Chapter 13.

The third and final part of the book presents problems and examples ranging over sorting and searching algorithms, reminder computation (including the ubiquitous Euclidean Greatest Common Divisor algorithm!), and cyclic codes. Although superficially simplistic, the problems/examples encourage familiarity in program construction as well as providing experience needed to deal with more advanced texts.

Overall, the book is a very pleasing introduction to program construction via axiomatic specifications. On the negative side, the notation for a predicate P taking a single argument i is $P.i$; and this could have been rewritten in the more familiar notation of $P(i)$.[3] The argument that Dijkstra has (Dijkstra, 1996) for using inconsistent logical notation could have been re-examined, as it hinders the advancement of his ideas to a reader taking/having taken a course in Predicate Logic. This and (perhaps) the lack of a chapter on the software tools (e.g. theorem provers, proof checkers, editors, etc.) that can aid this style of program construction are the only minor failings of the book.

The audience for this book are mathematically-shy undergraduates and/or functional programmers wanting to know about this style of program construction. For professional programmers, the ideal candidate would be someone writing small sections of safety critical code in embedded systems. That is systems where the extra features of an imperative language – say,

[2] Other statements have been axiomatized, such as procedure call, array assignment, `repeat...until` loop, etc.

[3] The notation becomes visually awkward in predicates of arity > 1, with unevaluated expressions for arguments.

https://doi.org/10.1017/S0956796804235328 Published online by Cambridge University Press

classes, inheritance, virtual functions, sophisticated I/O, co-routines, etc. – are not needed. In comparison to other similar (and well known) text books dealing with axiomatic specifications, such as Gries (1981) and Kaldewaij (1990), the book sits as a nice introduction to both of them. For example, Kaldewaij presents the "golden rule": $P \wedge Q \equiv P \equiv Q \equiv P \vee Q$ (page 11), gives it a set theoretic semantics (page 5), but neglects to mention how to use the rule. This is something that Backhouse does not do in his book. Like Gries (and unlike Kaldewaij), the book provides answers to numerous and entertaining exercises. The book also cites a website `www.wiley.com/go/backhouse` as a source of further information and further information and exercises can be found on the author's home website: `www.cs.nott.ac.uk/~rcb`

Overall, I would recommend the book. The quality of writing is high, the explanations are in detail and are excellent, while the appendix provides a very useful crib sheet for exams and reference.

References

Backhouse, R. C. (1986) *Program Construction and Verification*. Prentice Hall.

Dijkstra, E. W. (1976) *A Discipline of Programming*. Series in Automatic Computation. Prentice Hall.

Dijkstra, E. W. (1996) A somewhat open letter to David Gries. http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1227.PDF, January.

Dijkstra, E. W. and Scholten, C. S. (1990) *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag.

Floyd, R. W. (1967) Assigning meaning to programs. In: J. T. Schwartz, editor, *Mathematical Aspects of Computer Science: Proceedings American Mathematics Society Symposia*, vol. 19, pp. 19–31. American Mathematical Society.

Gries, D. (1981) *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag.

Hoare, C. A. R. (1969) An axiomatic basis for computer programming. *Comm. ACM*, **12**(10), 576–583.

Kaldewaij, A. (1990) *Programming: The Derivation of Algorithms*. Prentice Hall.

Nimish Shah
Durham University, UK