its stimulating presentation, especially to those concerned with the teaching of any of this material. Further editions or teaching notes using other programming languages might be useful.

Peter WALLIS
University of Bath
Bath, United Kingdom

Introduction to Functional Programming. By R. Bird and P. Wadler. Prentice-Hall International, Hemel Hempstead, United Kingdom, 1988, Price £15.95, ISBN 0-13-484197-2.

The significance of *Introduction to Functional Programming*, by Richard Bird and Philip Wadler is that with its publication, the necessary revolution in Computer Science/Software Engineering education, long overdue, at last becomes feasible. That the revolution is overdue is testified to by the long-standing gap between correct programming practices revealed by decades of research into programming methodology on the one hand, and the (at best) half-hearted support they receive in the typical curriculum on the other. To verify that claim, consider the extent to which, for example, formal methods of specification, derivation and verification are practised in final-year software projects. Moreover, this new feasibility (which derives from the book's considerable intrinsic merits) poses a significant ethical challenge to those of us who remain compelled to teach according to the rubrics now rendered patently obsolete. But more of this below.

Pedagogy

To begin however, let's consider the context that mandates such radicalism. For some two decades now, software engineers have become ever more aware that formal (mathematical) methods of software development are the only means by which adequate (i.e. both correct *as well as* efficient) solutions can be crafted. How should the curriculum foster the adoption of such methods by its students?

To this reviewer, the following general pedagogical observations seem self-evident.

(1) If there is a "better" way to do something, then some things should be done in that better way from the outset.

(2) It is curious to require that of some course of study, ostensibly designed to sustain its eventual graduates for another forty-or-so professional years, the precise content of the introductory component should be determined by contemporary technological fads. Rather, what the student eventually needs to understand is lasting truths, plus how these truths may need to be compromised by current circumstances as a pardigm for the different compromises that ongoing changes in circumstances

will inevitably entail. This suggests that more absolute fundamentals be taught first, and subsequently, the skills more "relevant" to the day be introduced with the compromises they entail pointed out.

(3) It is an understandable yet unjustified arrogance on the part of educators to assume that their lucky students retain all or even most of what they are explicitly taught, especially from their introduction to a topic. Introductions are surely primarily for learning methods of working, with factual accretion secondary and incidental (other than the necessary acquisition of the vocabulary of the field). In these circumstances, what instructors *do*, even when opposed to what they may say, is of supreme importance.

Let's now instantiate the three points with the specifics of Computer Programming (the essence of Computer Science/Software Engineering to which this journal is devoted).

(1) Formal methods are not trivial. They do not come easily or "naturally", but have to be consciously learned, and with effort. The more time available for this (e.g., as from the start of the curriculum), the better.

(2) The *complete* rigour that formal methods conceivably allow may not be feasible in the context of the languages and tools with which software engineers currently must work. Nevertheless, the greater rigour that can be economically applied, the better. If there is one "lasting truth" of Software Engineering that has been discovered in two decades of research, this is it. (At the same time, the extent to which rigour is possible need not be understated.)

(3) Because programming knowledge is a "process" skill, it can only be learnt from observation and by practice. The introductory programming course must allow students to appreciate the potential of the abovementioned fundamental truth of software engineering, by experiencing rigour at the outset. As well as controlling the (software) environment so that rigour is possible, the particular facet of programming being addressed rigorously must not demand too much in the way of incidental factual knowledge (e.g. the supporting mathematical vocabulary).

As regards the development of a vocabulary of programming and programming language issues, it is important not to overconstrain the environment so that simple concepts, omitted at first, seem exceptional and complicated when eventually encountered.

Critique

The typical curriculum of today, founded on Pascal or some derivative (Modula-2, Ada, etc.) cannot satisfy these requirements. If the incompatibility between nontrivial procedural languages (in the sense that a comprehensive set of control, data and information-hiding structures is provided) and fully rigorous methodology (at least insofar as a beginning student could cope) is recognised by abandoning formalism, the cause is lost from the outset. When (if ever) students meet formal methods, will their attitude not be one along the lines of "I've done OK so far without them"? Of course, some will realise that there *must* be a better way than online debugging etc., but the "code first, think never" attitude will be hard to wear down.

If, on the other hand, a valiant effort is made to marry Pascal etc. with formal methods from the outset, either the range of constructs actually covered and available to students will have to be unduly restricted as to mislead about the wide applicability of rigour, or the formalism will have to be so diluted as to divest the exercise of credibility. The latter approach actually does a dis-service, with its implication that formal methods aren't all that helpful.

Finally, Pascal etc. are not pure pedagogical instruments. In spite of contrary claims, they represent conscious compromises between design elegance and the prospects of efficient implementation, with the objective balance moving increasingly and unsatisfactorily more towards the latter as the constraints under which the language designers operated vanish as better implementation techniques emerge. (Who can really believe that the requirement that a Pascal program terminate with a "." is anything other than a nuisance?) Entire dimensions of experience in software construction (higher-order functions, polymorphic typing) are ignored. From experience, the mind-set imposed by this ignorance is very difficult to overcome.

The alternative is to adopt for introductory programming teaching a broad-spectrum software environment, which mirrors "real" languages, but which is controlled to make formalism credible. Functional languages provide such an environment. They have simple mathematical structures (especially referential transparency) which make rigorous methods easy to employ. They (usually) have flexible structuring mechanisms which allow interesting motivating examples to be addressed, and a wide range of linguistic constructs exposed. Moreover, they are of current practical utility in Software Engineering practice, in prototyping, and should be treated somewhere in the curriculum—there's no intrinsic harm in doing so at the start! Of course, the concept of assignable store, undoubtedly one of ultimate significance, is not immediately presented. (The celebrated "naturalness" of the von Neumann architecture would, if true, imply that it doesn't need much explicit teaching at all!) When eventually introduced, as part of the "computer systems" stream of the curriculum, it could be grafted onto a functional language using the expository style of denotational semantics, providing as well both an excellent case study of the use of functional programming in prototyping interpreters, and an introduction to the study of programming language theory.

The book

Having said all that, there is little to add but that Bird and Wadler is most satisfactory, in the strict sense of the word. A particular advantage accrues from the employment of Miranda[1] (or language so close thereto to make distinction almost pointless) as the language of illustration. Every once in a while in every field of endeavour there appears an artefact of advanced design that happily avoids ill-judged speculation. In programming language design, twenty years ago, it was Pascal (compared to Algol-68). Today, it's Miranda. As well as showing the cleanness of design that results from the cumulative effort of a capable (to say the least) individual for more than a decade, it has good implementations on many of the systems available to academic users (e.g. SUNs, VAXes, maybe Macintoshes before too long).

Early chapters give the basic vocabulary of function and constant definitions, data types and structures. Because of Miranda's simplicity (compared to Pascal etc.) yet expressiveness, these chapters are both brief and interesting. The presentation of computation as an extension of the pocket-calculator style makes the introduction very painless. The definition of interesting functions without the complications of recursion is accomplished with the help of a library of higher-order functions, analogous to Backus' "combining forms".

The chapter entitled "Recursion and Induction" forms the core where the reader learns to appreciate how formal methods contribute to the development of correct programs. Inductive proof is the paradigm to which the authors choose to attend. The examples are presented in an appealing sequence, from functions over single numbers to those over multiple lists. Note that not all steps are explicated—there is still some work for the instructor to do in support. Supporting exercises match the case studies, but for solutions the instructor again has a role to play. Maybe the authors will produce an instructor's handbook for the mass market?

Remaining chapters: consider the little operational knowledge needed to believe that infinite structures can be programmed safely; prove and derive programs involving infinite lists; introduce Miranda's type definition mechanisms; and use them to prove and derive programs that process trees. The reader is left with the clear impression that formal methods work, plus a wide knowledge of basic concepts of programming languages and their implementation. What more could be wanted?

No alternatives

To date, none of the books on Functional Programming known to this reviewer are viable competition for Bird and Wadler. Like, for example, *Functional Programming* by A. Field and P. Harrison (Addison-Wesley, 1988), most cover less programming and more language implementation, as well as being pitched at more advanced students. Even *The Structure and Interpretation of Computer Programs* by H. Abelson and G. Sussman (MIT Press, 1985), which is used for introductory teaching at MIT, does not compare. Its emphasis is not so much on formal methods of software

[1] "Miranda" is a trademark of Research Software Ltd.

development, but on language technology (which it does well). However, in this regard it is more akin to Field and Harrison than to Bird and Wadler. Moreover, its employment of Scheme (a lexically scoped LISP variant) as expository vehicle is a distracting idiosyncracy.

No sustainable objections

Some of the objections to the adoption of our teaching policy (and hence Bird and Wadler as text), with the refutations that expose their ill-foundednesses, follow.

"It's too hard!"

To be sure, the intellectual level of Bird and Wadler is quite above that of other introductory programming texts. However, given that its mathematical sophistication is nothing much beyond proof by induction, as found in any elementary Analysis text/course, surely it lies within the grasp of students of Programming, as exemplified by those at Oxford University already being taught so? To the objection that this experience is not universally valid, that Oxford undergraduates are clever enough to cope with material that others are not, the reply is that Bird and Wadler provides the simplest approach yet to formal methods. If *that* is regarded as to hard for one's own students, they might as well be given up on forthwith. Also, several other institutions, even as far away as the University of New South Wales, have decided to follow suit.

Another facet of this objection is that not just students but instructors will be overextended. It's true that computer science departments embrace a range of talents, but surely departmental heads who pride themselves on their administrative talents should be able to arrange duties so that staff teach the courses to which they are suited. If there aren't enough of these, and moreover if the others are to be revitalised, the example of MIT (in mounting staff training programs to accompany the adoption of Abelson and Sussman) can be followed. Another tack is to hire pure mathematicians. Nothing better demonstrates the intellectual bankruptcy of the current Computer Science/Software Engineering curriculum than the ease with which pure mathematicians with negligible Computer Science training (the term is deliberately used instead of "education", because that seems about the level at which the typical curriculum operates) are able to contribute to teaching and research in the serious side of Computer Science, almost at will.

"It's too expensive!"

Imagine the following. "Functional programming language implementations as available today are not always cheap to use. The advent of implementations using the latest compilation techniques *may* be soon, but this review advocates adoption of functional prgramming *now*! Why not wait a while, until David Turner puts Miranda on a TRS-80?"

In response, the initiative facilitated by Bird and Wadler is only one of a number of potential resource demands facing institutions wishing to keep up in Computer Science/Software Engineering education. If we (as a society) want good software, we've got to have good practitioners, and to be prepared to pay for it. In this reviewer's own environment, funding Computer Science departments at the same level as Engineering departments would provide enough cash to replace the current first-year Macintosh equipment (still running Pascal. I'm ashamed to admit) with SUN workstations for Miranda, one-for-one.

"*What about Prolog?*"

The basis for this seems to be "If you're going to use a fancy language to teach introductory programming, why not use something which allows really clever (i.e., AI) examples?" The response is that our embrace of functional programming is not on account of a desire for something, anything, different. Rather, it is the result of calculation of pedagogical needs and their implementation. The cleverness of the examples that logic programming facilitates is of no import by comparison.

Prolog itself incorporates many features that make it definitely *un*suitable as an introductory vehicle: lack of sophisticated data structuring; the need to employ nondeclarative constructs in nontrivial programming; an evaluation mechanism that needs to be understood in great detail in order to explain the behaviour of simple programs.

Action

For those with authority over curriculum developments, your responsibility is clear. First, contact your local Prentice-Hall representative for a copy of this marvellous book. Second, buy some Miranda licences and get rid of Pascal today. Reform!

For the rest of us in education, the dual courses of persuasion and resistance beckon. No opportunity to put the case for curriculum reform can be let pass. Don't get discouraged—if when future generations wonder why we didn't do a better job, are we happy to have it said that after being told by our unenlightened colleagues to keep quiet, that's all we did? Likewise, we need to behave like professionals, not prostitutes. As professionals, we have wider responsibilities than to the organisations that currently pay our salaries. When assigned teaching duties in support of dangerously obsolete curricula, should not refusal be a legitimate ethical response? Certainly, meek complicity is not!

Conclusions

The length of this review measures the significance attached by this reviewer to the challenge posed by the appearance of this book. The attempt has been made to identify curriculum requirements through analysis, not habit/prejudice.

History shows that revolutions stem from good ideas, the implementation of which has suddenly become possible. Only now have Bird and Wadler made possible the realisation of the revolution in programming pending for twenty years, by facilitating the transmission of its central idea, the necessity of formal methods, to the core of the curriculum. A few scholars across the world are already lucky enough to get this "first-class" treatment. Let's hope the rest don't get left behind for too long.

Paul A. Bailes
University of Queensland
Brisbane, Australia

Compiling Functional Languages. By A. Diller. Wiley, Chichester, United Kingdom, 1988, Price £15.95 (paperback). ISBN 0-471-920274.

This text describes a number of different techniques which can be used in the implementation of functional programming languages. The style of writing is intelligible; each chapter has an introductory synopsis and is structured into sections and subsections. Suggestions of suitable books and papers for further reading are scattered liberally throughout.

After a brief introduction to Lispkit we are treated to a lengthy exposition of combinatory logic, followed by a demonstration of the translation of a functional language (Lispkit) into combinators. The lambda calculus and its relationship with combinatory logic is discussed briefly.

The motivation behind the inclusion of chapter of somewhat esoteric bracket abstraction algorithms and another on supercombinator algorithms is not at all obvious. In contrast the section on program transformation, which is based on the work of Darlington, is well written, as is the section on partial evaluation.

The ubiquitous "rule of signs" example is given to illustrate abstract interpretation, followed by a short introduction to domain theory and some simple examples of the use of abstraction rules in strictness analysis. One third of the chapter on type systems is devoted to a section entitled "Motivation". After reading the rest of the chapter the reader may well feel motivated to look elsewhere for a more extensive treatment of this material.

The author includes in an appendix the Pascal source code of a compiler and reducer for Lispkit, prefacing the appendix with excuses for the code's inefficiency and inelegance. The reader is invited to improve upon it, which would certainly not be difficult for most Pascal programmers.