JFP **14** (1): 113–123, January 2004.
DOI: 10.1017/S0956796803004805 Printed in the United Kingdom

EDUCATIONAL PEARL

The risks and benefits of teaching purely functional programming in first year

MANUEL M. T. CHAKRAVARTY and GABRIELE KELLER

School of Computer Science and Engineering,
University of New South Wales, NSW, Australia
(*e-mail:* {chak,keller}@cse.unsw.edu.au)

Abstract

We argue that teaching purely functional programming *as such* in freshman courses is detrimental to both the curriculum as well as to promoting the paradigm. Instead, we need to focus on the more general aims of teaching elementary techniques of programming and essential concepts of computing. We support this viewpoint with experience gained during several semesters of teaching large first-year classes (up to 600 students) in Haskell. These classes consisted of computer science students as well as students from other disciplines. We have systematically gathered student feedback by conducting surveys after each semester. This article contributes an approach to the use of modern functional languages in first year courses and, based on this, advocates the use of functional languages in this setting.

1 Introduction

Let us start with a controversial thesis: we should not teach purely functional programming in freshman courses! In fact, we should not teach procedural, object-oriented or logic programming either. Instead, we should concentrate on teaching the elementary techniques of programming and the essential concepts of computing as a scientific discipline as well as foster analytic thinking and problem solving skills. In contrast to the first statement, little argument will probably arise over the last. Nevertheless, when we, for example, have a closer look at two of the most popular textbooks for teaching Haskell to freshmen (Bird, 1998; Thompson, 1999), we observe that (a) they concentrate on teaching functional programming as such and (b) they relegate some important topics of general interest to the backstage.[1] The most obvious example of the latter is probably the treatment of I/O. Given the pivotal role that I/O has in programming, we would surely expect it to be a central theme in any introductory programming class; in fact, the infamous "Hello World!" program is often enough the first example that one finds in a programming

[1] These observations are by no means intended to belittle the referenced textbooks. They merely illustrate what we believe are assumptions that frequently go unchallenged in our community.

textbook. Nevertheless, Bird (1998) discusses I/O in Chapter 10 (out of 12)[2] and Thompson (1999) discusses it in Chapter 18 (out of 20). In addition, when I/O is covered, it is encumbered by advanced functional programming notions, such as monads, which to freshmen probably does little in the way of promoting the practicability of Haskell. In comparison, Hudak (2000) takes a different approach; he faces I/O early on, in Chapter 3 (out of 24), and uses it throughout the book. As is already apparent from the title of the book, Hudak (2000) still has a distinct emphasis on functional programming as such, but his text undoubtedly lends itself better to the style of teaching that we advocate in this article.

In the end, we have to justify the use of a functional language in an introductory course to both our students and colleagues. In particular, the former tend to have their heads filled with buzzwords and have little appreciation of the difference between the latest marketing hype and foundational knowledge. In such a context, an argument concluding that a language like Haskell happens to be perfectly suited to convey the elementary techniques of programming and the essential concepts of computing has much more hope of success than blind insistence on the superiority of the paradigm of functional programming.

The central thesis of this article is that purely functional languages are ideally suited for introductory computing classes, but only if the focus is on general concepts rather than the specifics of functional programming. We support this thesis by presenting a concrete approach to an introductory computing class, which we repeatedly taught at the University of New South Wales to between 400 and 600 students per semester. We will detail our position and method as well as summarise the feedback that we received in the form of student surveys, which we conducted after each semester and that contained both multiple-choice and free-form questions. Our focus is on modern, strongly typed functional languages and, in particular, on Haskell (Peyton Jones S., 2003). We are aware that, at least some of the arguments that we present have been made before. In particular, Felleisen *et al.* (2002b) take a position similar to ours; moreover, Wadler's (1987) critique highlights important points. However, especially when it comes to pure, strongly-typed languages, we are not aware of any other work that presents the same set of arguments as this article.

2 Functional languages and the foundations of computing

To justify our approach in more detail, we will first state our goals. Our concerns lie with introductory computing courses and, in particular, those courses where university students are first exposed to programming. We believe that such courses should have three principal aims:

1. Convey the elementary techniques of programming (the practical aspect).
2. Introduce the essential concepts of computing (the theoretical aspect).
3. Foster the development of analytic thinking and problem solving skills (the methodological aspect).

[2] There is a brief mention of `putStr` on Page 51, though.

https://doi.org/10.1017/S0956796803004805 Published online by Cambridge University Press

Some foundational topics, such as basic algorithms, exhibit elements of all three aspects: they may involve skillful coding, often require formal arguments about complexity, and demonstrate typical approaches to solving programming problems. In fact, a good course should integrate all three aspects as neatly as possible.

We propose the use of functional languages in introductory programming courses as they support these three aims well and, in particular, lead to maximum integration of these aims. The clean semantic underpinnings of functional languages help with the first two aims: (1) elementary programming techniques can be discussed clearly, without much clutter; and (2) formal reasoning about programs is simple and natural. Moreover, the tight interplay of theory and practice in functional languages naturally integrates the first two aims in the classroom. With respect to the third aim, the high level of expressiveness of functional languages is an essential factor. It means that complicated programming problems can be tackled early on in the course, which automatically moves the focus from the tedious mechanics of programming to analytical and problem solving challenges. For example, programming a game tree to implement a computer player is, after a couple of weeks, feasible in a functional language, but certainly out of scope in a vanilla imperative, and probably also in an object-oriented language.

Survey results. When being asked what the best aspect of the course was, many students named the emphasis on problem solving and analytical thinking as well as the fact that the assignments were interesting. In fact, one student wrote, "[t]he assignments were actually entertaining & interesting. What an anomaly." Another student mentioned that "completing the assignment[s] gave me a real sense of achievement, they were not just toy problems." The multiple choice question that asked the students to rate the assignments as to how interesting they perceived them led to the following distribution:[3]

Year	Very interesting		Average		Rather boring
2000	16%	32%	30%	11%	11%
2001	26%	31%	25%	10%	8%
2002	21%	30%	26%	9%	14%

Students found the assignments, especially later stages, also challenging, as the survey results from 2000 show, which are summarised in Table 1; results from other years are comparable.

2.1 Technical advantages of functional languages

A rigorous static type discipline is one of the most defining features of modern functional languages, such as ML and Haskell. Although it may be argued that types

[3] In reading the numbers, it might be helpful to take into account that usually about 30% of the enrolled students fail this course.

https://doi.org/10.1017/S0956796803004805 Published online by Cambridge University Press

Table 1. *Survey results regarding the perceived difficulty of assignments (2000)*

Assignment	Very difficult		Average		Very easy
Assignment 1, Stage I	15%	25%	35%	20%	7.1%
Assignment 1, Stage II	19%	37%	31%	11%	2.6%
Assignment 2, Stage I	28%	38%	23%	8.3%	4.3%
Assignment 2, Stage II	40%	35%	15%	7.9%	3.3%
Assignment 2, Bonus Task	85%	8.1%	5.0%	0.7%	2.9%

are less important in mainstream languages, we put strong emphasis on types as (1) they structure the problem solving process, (2) their value for software engineering is beyond doubt, and (3) they become increasingly important in mainstream languages (e.g. in Java). Moreover, they are a fundamental concept in computing. We introduce types right from the start and emphasise them throughout the course. We also introduce the use (not the definition) of Haskell type classes early on. Their inclusion in the course is justified as overloading is a common concept in programming languages. We introduce them early as they are pervasive in type error messages. Overall, the treatment of types meets all three of our aims: they are an elementary concept in programming, they are foundational, and they structure analytical thinking about programs.

The lightweight and orthogonal syntax of Haskell helps to relegate syntactic issues to the background and to concentrate on general programming concepts. This obviously assists in getting quickly to interesting topics and problems. In particular, side-effects, memory allocation, and other implementation details of program execution need not be introduced initially when a functional language is used. Instead, these topics can be covered at a later point in isolation when students have already grasped the more general concepts. The concise syntax and high level of expressiveness of modern functional languages is of particular advantage in the treatment of data structures. Algebraic data structure definitions and pattern matching put complex structures, such as expression trees and memory tries, within reach of beginners. This is difficult to achieve in more conventional languages and we believe that it contributes significantly towards the students perceiving the course as interesting.

As already mentioned, the clean semantics of functional languages leads to a good integration of the teaching of programming techniques with computing concepts and theory. For example, we encourage students from the start to get a feeling for what a program does by way of stepwise evaluation of expressions on a piece of paper. This neatly provides a starting point for the introduction of equational reasoning by performing stepwise evaluation on expressions that are not closed, which brings us to correctness proofs and program derivation. In our opinion, this is significantly easier to motivate and implement than the calculus of weakest preconditions or the Hoare calculus that would be the corresponding theory for imperative languages. We can also cover proofs of larger example programs, which motivates students to regard the whole approach as more practical. On the downside, complexity analysis

requires, from the outset, recurrence relations, whereas in an imperative language the discussion of the complexity of loop programs requires less advanced mathematical tools.

Overall, functional languages support the move from a language-centric to a concept-centric teaching style, where the programming language is merely a tool that allows the lecturer to illustrate the fundamental principles of computing. This is in stark contrast to the old-fashioned approach to teaching programming, where the various syntactic forms and features of a programming language guide the development.

2.2 Coincidental advantages of functional languages

As we all know, functional languages can, by no stretch of imagination, be called mainstream. Incidentally, this apparent disadvantage turns out to be one of the big selling points for using Haskell: as almost no first year student has any prior knowledge of Haskell, it acts as an equaliser between students who bring existing programming skills into the course and those who do not.

In computer science, we are faced with the problem that we have a significant number of students in a first year course who have already mastered one or more programming languages and students who have hardly ever used a computer before. The latter easily lose their motivation and self confidence when they struggle with seemingly trivial syntax problems, while they perceive the other students to be miles ahead of them. On the other hand, students with programming skills often do not pay attention or may skip lectures altogether since they are initially bored by the presentation of material that is already familiar to them. They receive the impression that the course is merely an introduction to programming and tend to miss the point when the course starts to deal with the more challenging, language-independent concepts. These problems can be significantly reduced by using a language that is little known to first year students and by focusing on general computing concepts from the outset.

Female students often approach computing coming from mathematics and, on average, have significantly less prior experience in programming, as reported by a study conducted at Carnegie Mellon University (Margolis *et al.*, 2000). A modern functional language makes it easier to appeal to the mathematical background of these students and serves to neutralise some of the advantage that their peers have due to prior programming experience. The latter is a crucial factor as female students are reported to have a high likelihood of being discouraged by the boasting of their male peers to the point were they change majors. In fact, the study points out that, "[t]o create gender equity at the undergraduate level, computer science programs must address the question of the unlevel playing field in terms of prior experience." (Margolis *et al.*, 2000). In our experience, functional languages successfully serve as a leveller. The same has been said about courses based on the language Scheme, after Abelson & Sussman's (1984) book became popular.

Survey results. We had no questions specifically addressing the choice of programming language in the survey. However, in the free feedback section, some students

https://doi.org/10.1017/S0956796803004805 Published online by Cambridge University Press

commented on the issues discussed above. While some students would have preferred a language that is more buzzword-compliant on their CV and some complained that they could not apply their existing knowledge of Java or C in the course, beginners also perceived the latter as an advantage, which is reflected in the following comments: "do not scrap Haskell as a starting language....speaking from the point of view of a novice it is the perfect start to programming and computer science" and "[Using Haskell is] a relative[ly] easy way to approach programing for a person who has never done any programing just like me." However, also a number of students who were already proficient in a programming language appreciated looking at new concepts: "[One of the best things about this course was] learning a new programming language and a new way of thinking." Students also appreciate the tight integration of theory and practice.

2.3 Environments and online program development

As a practical matter, the concise syntax and high-level of expressiveness of a functional language turns out to be a great advantage. It allows us to develop fairly complex example programs interactively in the lecture; that is, attach a laptop to a data projector and actually develop programs in class. According to student feedback, this is perceived as very helpful; much more so than a purely blackboard or slides-based presentation. Instead of simply presenting the solution to a problem, we can in this way demonstrate the complete process, including common errors and an explanation of the resulting error messages. Online program development also turned out to be well suited to encourage students to actively participate in lectures.

We found that by using this technique, first year students are more likely to imitate the development process and programming style of the lecturer. We conjecture from that observation that the difference between a blackboard or slides-based presentation and the student's own programming experience is too wide a gap for many students to bridge. Through online program development, students learn the whole process more effectively. The fact that programs in modern functional languages are very concise, and that interactive execution environments are readily available, supports this style of lecturing.

Another advantage of functional languages is the availability of integrated interactive and batch development tools. An interactive environment provides students with the means to gently take their first steps, experiment with library functions, obtain type information, and test and debug their own programs. Later in the course, we introduce the use of the batch component. We found that in those years where we used the batch component in addition to the interactive environment, students more easily understood the concepts of I/O programming (see also Section 4). Moreover, it helps students to establish a connection between the environment of their first programming course and those of subsequent courses.

Survey results. To evaluate the effectiveness of various presentation methods in lectures, we incorporated a corresponding question into the surveys. The results for 2001, which are close to that for 2000 and 2002, are displayed in Table 2. It is

https://doi.org/10.1017/S0956796803004805 Published online by Cambridge University Press

Table 2. *Survey results regarding lecture presentation techniques* (2001)

Technique	Very helpful		Don't care		Confusing boring	N/A
Computer projected slides	33%	43%	16%	4%	3%	1%
Use of the blackboard	5%	26%	35%	11%	8%	15%
Stepwise program development in Emacs	44%	36%	12%	3%	4%	1%
Running examples in GHCi	60%	28%	7%	3%	2%	0%
Questions asked to the class	25%	40%	24%	6%	3%	2%

NB: GHCi is the interactive interpreter of the Glasgow Haskell Compiler.

interesting to observe that the interaction with the interactive Haskell interpreter is perceived to be the single most helpful presentation technique. This was also reflected in the free form comments, where student feedback with respect to online program development was very positive.

3 The pitfalls of functional programming

There is never enough time in a freshman course to discuss everything one would like to teach. Therefore, the most difficult decisions are often about what to leave out in order to keep the course manageable and to maintain focus. Our approach establishes the three principal aims from Section 2 as the basis for inclusion of topics. This implies that we drop topics specific to functional programming in favour of more general topics. In particular, we omit the following four topics, which one would expect to cover in a course on functional programming: (1) list comprehensions (in Haskell); (2) currying; (3) sophisticated use of higher-order functions; and (4) lambda expressions.

List comprehensions provide an elegant, expressive means to concisely formulate certain patterns of calculation on lists. But although they are rather similar to set comprehensions, we found that students generally need time to absorb the idea. Given the lack of list comprehensions in most languages, we believe that spending the time required to cover them in lectures and exercises is not justified.

A probably more controversial issue is currying. It is even more specific to functional programming and, in our experience, frequently perceived as an obstacle by students. It may be argued that the essence of currying is an important, general concept in computing, but to make this connection is beyond the capabilities of students without a strong mathematical background (at this point in their studies). This immediately takes us to the general area of higher-order functions. It is central to functional programming, but a rather advanced concept in all other programming paradigms. Hence, we restrict ourselves to very basic use of higher-order functions, such as applications of the functions `map` and `filter`, to illustrate the general concept and motivate the use of higher-order functions for modularity. It might be argued that a concept that is generally regarded as being advanced should be

https://doi.org/10.1017/S0956796803004805 Published online by Cambridge University Press

dropped altogether, but we found that students deal quite naturally with the two mentioned higher-order functions, and it gives them an indication of things to come. However, the treatment of more advanced higher-order functions – even `folds` – is usually counterproductive, as it confuses even average students, and detracts from the main aims of the course.

Lambda expressions, or unnamed functions, again, are central to the functional programming paradigm, but do not contribute to our motivating aims, as they are not essential in most languages.

All of the above mentioned topics can be left out without compromising the consistency of the course. However, there is one topic that we have to tackle early on because we are using a functional language and that students have a tendency to perceive as a significant obstacle: recursion. In imperative languages, iteration can be introduced first as a simpler, more specialised form of repetition, and recursion can be delayed until trees are covered.

The biggest challenge in using a functional language for first-year teaching is probably to counter arguments along the lines of "we're learning Haskell, which isn't really used out in the real world" (cited from a survey).[4] We found that this objection was best countered in three ways: (1) explain to students that foundational knowledge is more important than individual languages; (2) explicitly discuss "real world" applications of functional languages; and (3) emphasise practical aspects, such as I/O programming and explicitly highlight connections to other languages and courses. As a result, the surveys contained comments like "Haskell is more advanced than C and Java in certain areas" and "good to learn about functional programming, I had never done it before." Although, such statements may simply be repeated from the lectures, they indicate that it is useful to explicitly discuss these issues.

4 I/O programming

In the introduction, we mentioned I/O programming as a victim of an overemphasis on functional programming at the expense of general programming concepts. In fact, we have heard teachers argue that purely functional languages are unsuitable for teaching I/O. We beg to differ. We even claim that purely functional languages allow for an especially precise treatment of the nature of I/O and stateful programming.

Covering I/O is important for three reasons: (1) I/O is central to all programming; (2) functional programming without I/O is easily perceived as impractical; and (3) I/O allows us to gently introduce students to imperative programming, which improves integration with later courses. Here we shall substantiate this claim by describing how we believe I/O should be taught using Haskell. This description is also intended to serve as a more technical example of our approach.

As mentioned previously, we combine the teaching of I/O with the move from an interactive to a batch interface of the development environment. Before we

[4] Interestingly, this argument is sometimes also invoked by colleagues who, one would expect, should know better.

https://doi.org/10.1017/S0956796803004805 Published online by Cambridge University Press

introduce I/O as such, students learn the concept of a standalone executable and we discuss how an executing program can be regarded as a process interacting with its environment. Then, we use Haskell's `print` function to obtain the same effects as entering an expression in the eval-print loop of the interactive environment. It requires little discussion to establish that it is unsatisfactory to be forced to modify and possibly recompile a program whenever the user wants to evaluate a new expression.

At this point, we introduce expressions of type `IO a` as *actions* that specify the interaction of a process with its environment (eventually, returning a value of type `a`). To further emphasise the essential difference between purely functional computations and I/O actions, we explain that a well structured program consists of an outer *interaction shell* that defines the communication of a process with its environment and a *computational kernel* that performs computations internal to the process. Haskell's type system enforces this structure by way of the `IO` type, which fits well with our earlier motivation of types as a mechanism to structure programs. Imperative languages permit the programmer to lump everything into one unstructured entity, but, from a software engineering point of view, this can hardly be considered an advantage.

Further, we explain that, for pure computations, evaluation order is only determined by data dependencies, whereas for I/O actions, this is not true. This can be demonstrated by a simple ask question/read answer example, from whence, it is a small step to motivating Haskell's `do` notation as a programming construct that explicitly sequentialises imperative actions. These concepts, in combination with a batch development system, are well suited to draw a connection between our course and our university's second programming-related course, which is currently taught in C and emphasises concepts, such as pointers, explicit memory management, more advanced I/O, and so on.

In the whole process of teaching I/O to freshmen, it is imperative to avoid the monad-based heritage of I/O in Haskell. Moreover, we use flow diagrams to conceptualise the control flow in a program's interaction shell and have found graphics programming, as proposed by Hudak (Hudak, 2000), a good motivation for students to master the challenges of I/O programming.

Survey results. When we asked students in surveys about the relative difficulty of the various topics in the course, I/O programming was rated at above average difficulty. However, topics such as trees, work complexity, and shell scripting were rated as much more difficult. It is also interesting to note that students found I/O easier to understand in 2001 than in 2000. The main difference between the two years was that in 2001 we used a combined interactive/batch environment (as outlined above), whereas in 2000 we used a purely interactive environment.

5 Conclusions

First-year programming courses need to be language agnostic; or as phrased by Felleisen *et al.* (2002b), they need to be liberated from the tyranny of syntax. In

https://doi.org/10.1017/S0956796803004805 Published online by Cambridge University Press

this article, we have argued that this goal is facilitated by the light syntax and clean semantics, as well as the high-level of abstraction, of purely functional languages. Functional languages help us to replace the tyranny of syntax by a principled approach that (1) conveys elementary techniques of programming, (2) introduces essential concepts of computing, and (3) fosters the development of analytic thinking and problem solving skills.

The potential of functional languages for introductory computing courses was already tapped by Abelson & Sussman's (1984) influential text *The Structure and Interpretation of Computer Programs*. Unfortunately, their approach suffered from a number of shortcomings, which were eloquently identified by Wadler (1987) and Felleisen *et al*. (2002b). Among these are a lack of explicit instructions regarding program design and dependence on sophisticated domain knowledge, which is beyond the average computing freshman. However, this should not distract from the fundamentally beneficial approach of concentrating on essential principles, rather than on the specifics of a single language. The textbook of Felleisen *et al*. (Felleisen *et al*., 2002a) demonstrates how the shortcomings of Abelson & Sussman can be avoided. In addition, we developed a textbook (Chakravarty & Keller, 2002) for the introductory course, the design of which was outlined in this article. Let us exploit the strength of functional programming for first year teaching by focusing on paradigm-transcending, essential principles.

Acknowledgements

We would like to thank Richard Buckland for our discussions about first year teaching and Matthias Felleisen for his helpful suggestions for improving this article. Moreover, we are grateful to the attendees of the *International Workshop on Functional and Declarative Programming in Education (FDPE 2002)*, and in particular, to Shriram Krishnamurthi and Simon Thompson for their feedback on an earlier version of this article. Last, but not least, we are indebted to Ingrid Slembek for proofreading this article.

References

Abelson, H., Sussman, G. J. and Sussman, J. (1984) *Structure and Interpretation of computer programs*. MIT Press/McGraw-Hill.

Bird, R. (1998) *Introduction to Functional Programming using Haskell, 2nd ed*. Prentice Hall.

Chakravarty, M. M. T. and Keller, G. C. (2002) *An Introduction to Computing*. Pearson Education Australia.

Felleisen, M., Findler, R. B., Flatt, M. and Krishnamurthi, S. (2002a) *How to Design Programs*. MIT Press.

Felleisen, M., Findler, R. B., Flatt, M. and Krishnamurthi, S. (2002b) The structure and interpretation of the computer science curriculum. In: Hanus, M., Krishnamurthi, S. and Thompson, S., editors, *International Workshop on Functional and Declarative Programming in Education (FDPE 2002)*. Bericht, no. 0210. Christian-Albrechts Universität Kiel, Institut für Informatik und praktische Mathematik. `http://www.informatik.uni-kiel.de/reports/2002/0210.html`.

Hudak, P. (2000) *The Haskell School of Expression: Learning functional programming through multimedia*. Cambridge University Press.

Margolis, J., Fisher, A. and Miller, F. (2000) The anatomy of interest: Women in undergraduate computer science. *Women's Stud. Quart*. Spring/Summer.

Peyton Jones, S. (2003) Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, **13**(1).

Thompson, S. (1999) *Haskell: The craft of functional programming, 2nd ed*. Addison-Wesley.

Wadler, P. (1987) A critique of Abelson and Sussman, or, why calculating is better than scheming. *SIGPLAN Notices*, **22**(3).