The Risks and Benefits of Teaching Purely Functional Programming in First Year

Manuel M. T. Chakravarty   Gabriele Keller

School of Computer Science and Engineering
University of New South Wales
{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 more general aims. We support our viewpoint by experience gathered during repeatedly teaching large first-year classes (up to 600 students) in Haskell. The students in these classes have been both computer science majors and students from other disciplines. We have systematically gathered student feedback by conducting surveys after each semester. This paper contributes an approach to the use of modern functional languages in first year courses and, based on this, advocacy for the use of functional languages.

1 Introduction

Let us start with a controversial thesis: We should stop teaching purely functional programming in freshman courses! In fact, we should not be teaching object-oriented or logic programming either. Instead, we should concentrate on teaching the essential concepts of programming and the fundamentals of computing as a scientific discipline. In contrast to the first statement, little argument will probably arise over the last one. Nevertheless, when we, for example, have a closer look at two of the most popular textbooks for teaching Haskell to freshman [1, 6], 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; and, in fact, the infamous "Hello World!" program is often enough the first example that one finds in a programming textbook. Nevertheless, Bird [1] discusses I/O in Chapter 10 (out of 12)[2] and Thompson [6] 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 freshman probably does little in the way of promoting the practicability of Haskell. In comparison, Hudak's book [3] takes a different approach as it faces I/O early on, in Chapter 3 (out of 24), and uses it throughout the book. As is already apparent from its title, the book still has a distinct emphasis on functional programming as such, but it undoubtedly lends itself significantly better to the style of teaching that we advocate in this paper.

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 little appreciation of the difference between the latest marketing hype and foundational knowledge. In such a context, an argument that a language like Haskell happens to be perfectly suited to convey the essential concepts of programming and the fundamentals of computing has much more hope of success than insistence on the superiority of the paradigm of functional programming.

[1] These observations are by no means intended to belittle the quality of the referenced textbooks. They merely illustrate what we believe are seldom challenged assumptions by our community.

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

In the rest of this paper, we will report on an implementation of our approach in the introductory computing class at the University of New South Wales. 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 [5].

The contributions of this paper are (1) an approach to the use of pure functional languages in first year and, based on this, (2) arguments to advocate the use of functional languages. We are aware that, at least, some of the arguments that we present have been made before. However, especially when it comes to pure, strongly-typed languages, we are not aware of any other summary like the present one.

2 Functional Languages and the Foundations of Computing

To justify our approach in more detail, we first need to specify our goals. Our concern are introductory computing courses and, in particular, those courses where university students are first exposed to programming. We believe that such a course 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).

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.

Our motivation for using a functional language in an introductory programming course is based on the observation that it supports these three aims well and, in particular, that it leads 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 simpler and more natural. Moreover, the tight interplay of theory and practice in functional languages carries naturally over to integrating 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 more complicated programming problems can be tackled 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 languages, 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 put the fact that the assignments were actually interesting very high on their list. In fact, one student wrote, "[t]he assignments were actually entertaining & interesting. What an anomaly." Another students mentioned that "completing the assignment gave me a real sense of achievement, they were not just toy problems". The multiple choice question that asked as how interesting students perceived the assignments led to the following distribution:[3]

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

Nevertheless, students found the assignments, especially later stages, also challenging, as the survey results from 2000 show, which are summarised in Figure 1 (next page); results from other years are comparable.

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

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%

Figure 1: Survey results regarding the perceived difficulty of assignments (2000)

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 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. 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, into the reach of beginners. This is difficult to achieve in more conventional languages and contributes significantly towards the students perceiving the course as interesting and challenging.

As already mentioned, the clean semantics of functional languages leads to a good integration of the aims 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 to introduce 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 from the outset requires the use of recurrence relations, whereas in an imperative language the discussion of the complexity of loop programs requires less advanced mathematical tools.

2.2 Coincidental Advantages of Functional Languages

As we all know, functional languages can, by no stretch of imagination, be called mainstream. Funnily enough, 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 already bring some 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 group easily looses their motivation and self confidence when they struggle with seemingly trivial syntax problems, while other students are perceived to be miles ahead of them. On the other hand, students with programming skills often do not follow the course since they are initially bored by the presentation of material that is already familiar to them. They get 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.

Female students often approach computing coming from mathematics and, on average, have significantly less prior experience of programming, as reported by a study conducted at Carnegie Mellon University [4]. 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 discourage 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." [4]. In our experience, functional languages successfully serve as an equaliser.

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

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 even 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 gap between a blackboard or slides-based presentation to their own programming experience seems to be too wide for many students to bridge effectively. By online program development, students more effectively learn the whole process. The fact that programs in modern functional languages are very concise and that interpreter-based environments are readily available supports this style of lecturing.

Another advantage of functional languages over purely compiled languages is the availability of integrated interpreter and compiler development tools. The interpreter 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 compiler. We found that in years were we did this, students more easily understood the concepts of I/O programming (see also Section 4). Moreover, it helps students to establish a connection between the material in their first programming course with the courses that they take later.

Survey results. To evaluate the effectiveness of various presentation methods in lectures, we included a corresponding question into the surveys. The result for 2001, which is close to that for 2000 is displayed in Figure 2. It is interesting to observe that the interaction with the Haskell interpreter is perceived to be the single most helpful presentation technique. This was also reflected in the free form comments: the feedback from students with respect to online program development was extremely positive.

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 interpreter of the Haskell system GHC.

Figure 2: Survey results regarding lecture presentation techniques (2001)

3 The Pitfalls of Functional Programming

There is never enough room in a freshman course to discuss everything one would like to teach. Therefore, the most difficult decisions are often about what to leave out to keep the course manageable and to maintain focus. Our approach makes the three principal aims from Section 2 the basis for inclusion or exclusion of topics. Note that 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 comprehension provide an elegant, expressive means to concisely formulate certain patterns of calculation on lists. But although they are rather similar to set comprehension, we found that students in general need a while to absorb the idea. Given the lack of lists comprehensions in most languages, we believe that it is not justified to spend the required time in lectures and exercises.

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 the very basic use of higher-order functions, such as 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 dropped altogether, but we found that students deal quite naturally with the two mentioned higher-order functions and it gives them a perspective 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 distracts 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 due to using a functional language and that students have a tendency to perceive as a significant obstacle: recursion. In imperative languages, iteration can be introduced early on as a simpler form of repetition and recursion be moved further back in the course, 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 is best countered in three ways: (1) explain to students that foundational knowledge is more important than individual languages, (2) explicitly discuss existing "real world" applications of functional languages, and (3) emphasis practical aspects, such as, I/O programming and explicitly highlight connections to other languages and courses. As a result, the surveys also 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."

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

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. In the following, 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 before, we combine the teaching of I/O with the move from an interpreter-based to a compiler environment. Before we discuss I/O as such, students learn the concept of compiling a program and how an executing program can be regarded as a process interacting with its environment. Initially, we use Haskell's `print` function to obtain the same effects as entering an expression in the interpreters eval-print loop. It requires little discussion to establish that it is unsatisfactory to have to recompile a program any time the user likes 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 motivate the differentiation 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 motivation of types as a mechanism to structure programs early on in the course. Imperative languages permit 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 and that, in contrast, for I/O actions, this is not true. This can be demonstrated by a simple ask question/read answer example. From there 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 the use of a compiler are well suited to draw a connection between our course and the second programming-related course, which at the University of New South Wales 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 freshman, 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 [3], 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 interpreter/compiler environment, whereas in 2000 we had only an interpreter.

5 Conclusions

In summary, we have presented strong reasons for the use of a modern, strongly-typed functional language in first-year programming classes. However, we found it advantageous to avoid techniques specific to functional programming, such as extensive use of higher-order functions, and instead focus on the aims of (1) conveying elementary techniques of programming, (2) introduction of essential concepts of computing, and (3) fostering the development of analytic thinking and problem solving skills.

Despite our focus on statically typed languages in this paper, the paper can barely be considered complete without mentioning Abelson, Sussman & Sussman's influential text [2] on the subject. They surely follow similar aims as those that we outlined in Section 2. However, their text is pitched at a more advanced audience than ours as can be seen from that more than three quarters of their students already have programming experience[5] and that they cover a fair share of language implementation techniques. (We do cover an interpreter for a simple functional language in the last week, though.) Hence, the present paper may be regarded as a call to, in particular, the Haskell community to return to the programming-paradigm transcending approach of Abelson, Sussman & Sussman.

References

[1] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Press, second edition, 1998.

[2] G.J. Sussman H. Abelson and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press and McGraw-Hill, 1984.

[3] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

[4] Allan Fisher Jane Margolis and Faye Miller. The anatomy of interest: Women in undergraduate computer science. *Women's Studies Quarterly*, Spring/Summer, 2000.

[5] Haskell 98: A non-strict, purely functional language. `http://haskell.org/definition/`, February 1999.

[6] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, second edition, 1999.

[5] See, `http://www-mitpress.mit.edu/sicp/course.html`