International Journal of Mathematical Education in Science and Technology

Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713736815

Some experiences in teaching functional programming

Mike Joy [a]; Steve Matthews [a]
[a] Department of Computer Science, University of Warwick, Coventry, CV4 7AL, England

Online Publication Date: 01 April 1994
To cite this Article: Joy, Mike and Matthews, Steve (1994) 'Some experiences in teaching functional programming', International Journal of Mathematical Education in Science and Technology, 25:2, 165 - 172
To link to this article: DOI: 10.1080/0020739940250202
URL: http://dx.doi.org/10.1080/0020739940250202

INT. J. MATH. EDUC. SCI. TECHNOL., 1994, VOL. 25, NO. 2, 165–172

Some experiences in teaching functional programming

by MIKE JOY and STEVE MATTHEWS
Department of Computer Science, University of Warwick,
Coventry, CV4 7AL, England

(*Received 3 November 1992*)

The authors have been developing functional programming units within the computer science BSc degree course at Warwick. In recent years there has been a shift away from a principally theoretical approach to the subject; functional programming is now taught using a 'hands on' experiential approach. This paper outlines the reasons for this change of emphasis and discusses some of the pedagogical issues raised.

1. Introduction

Until recently computer languages have relied on imperative constructs, whereby a program is a sequence of instructions describing how to manipulate the storage and i/o devices of a machine in order to produce a defined behaviour. It is now easily possible to implement languages which are not defined in terms of a machine architecture, thus allowing a programmer to concentrate on *what* is to be solved, rather than the details of *how* it is to be solved. A major class of language which allow such a *declarative* style of programming is the class of *functional languages*.

A program written in a functional language takes the form of a sequence of definitions and expressions. The 'output' from such a program will be those expressions, evaluated with names occurring within them associated with the given definitions. Each name has a *unique* definition, and cannot be redefined. There is no concept of a *variable* or of a *storage location*.

Since a name in a function program has a unique definition, and all names therein referred to must also have unique definitions, the value to which a name evaluates must also be unique. Functional programs are thus said to be *referentially transparent*. Since the value of a name is unique, it is of little interest to the programmer how it is computed, rather its relationship with the other defined names. Functional programming becomes an exercise in algebra.

By writing programs in a functional style it is possible to reason formally about them with ease, and consequently development time is reduced. However, since such languages are not defined with reference to the hardware on which they are executed, this comes at a price, namely efficiency. A functional program will typically run much slower than an equivalent imperative program, and it is not uncommon for the storage used by such a program to become unexpectedly large.

We aim to square the circle of teaching students the highest standards of programming discipline while writing efficient programs. Efficiency can be adequately discussed in traditional languages using devices such as global variables and side effects. A pure functional programming language allows us to enforce the

highest standards, but it is our experience that any attempt to suggest that such standards automatically imply faster programs will be rejected by students. We can make a convincing case for efficiency for functional programs by arguing that they are speedier to write and debug. As long as we neither claim nor imply that functional programs must be faster and more space-saving than their imperative counterparts our reasoning is acceptable.

We have found that a major challenge is how to get the message across that functional programming is an economic route to good software design regardless of the eventual target language. The computer experience of the vast majority of students prior to commencing our course must be seen in the context of an imperative culture. Microcomputers running various dialects of BASIC still form the mainstay of computer provision in British schools and colleges. Only a minority of institutions teach other languages, such as Pascal, and virtually none give students exposure to a declarative style of programming. Thus students come to us with the concepts of 'variable' and 'assignment' already hard-wired into their perception of programming.

In this paper we address issues raised by this task, and argue that by approaching functional programming from a pragmatic angle students find the concepts underlying functional programming easier to digest.

2. Functional programming

'Functional programming is a good example of a specification oriented program development tool, and so worth teaching, as such tools are becoming increasingly important in Computer Science'. So might a lecture on functional programming commence. Some detailed arguments in favour of the relevance of functional programming will motivate our discussions below.

- Increasingly important parallel algorithms, distributed programming architectures and techniques can be studied easily using functional programming.
- Problem specification (i.e. functional specification) is facilitated.
- Problem analysis (for example, examining a problem to produce a divide-and-conquer algorithm) is facilitated, since program transformation can be placed on a sound formal footing.
- Top-down program design (for example, turning divide-and-conquer algorithms into networks of communicating processes) is also made simpler, for the same reason.
- Information hiding becomes a natural process; the use of algorithmic and data abstraction is a normal functional programming skill, as opposed to explicit data representation.
- Data specification is placed on a similar footing to problem specification.
- *Levels* of abstraction can be introduced naturally into a program.

In summary, these aspects are all applications of various forms of 'abstraction' in the software design and development process. A major aim of the computer science course at Warwick was to communicate both the theory and some experience of using abstraction in software design and development.

These goals do not seem to be achievable in (for instance) Pascal. Our experience of programming courses which used Pascal made it clear that students were reluctant

to design their own data structures appropriate for their parsing exercises, preferring instead to do everything in much less structured ways using only Pascal's built-in data types. Students have a mental barrier against abstraction; it is understandable that an instinct for security tells them to stick with the more concrete built-in data structures that they have already conceptualized. This barrier should be attacked from day 1.

Once a student has understood an imperative language such as Pascal, their experience of that language does not provide a sound foundation for understanding the concepts underlying a functional language. Imperative programming and functional programming are different skills. To teach a student to program well in a functional language one must somehow cause that student to understand clearly the foundations of functional programming. We would argue that this comprehension is best facilitated by the students themselves [1] and that substantial hands-on experience with a functional language is a prerequisite.

2.1. *Teaching functional programming at Warwick*

The teaching of functional programming was instituted at Warwick by the late David Park in the 1970s. This tradition has been carried on by Bill Wadge, Steve Matthews, Meurig Beynon, and Mike Joy using WAFL (Warwick Functional Language), PIFL [2], Lucid [3], Standard ML and Miranda. However, the emphasis has changed dramatically from a purely examined course on the theoretical foundations of Kahn Data Flow or the lambda calculus to an experiential partly assessed and partly examined course. This was partly due to improved software availability and a local shift in opinion that it was actually necessary to get students to write functional programs. The result of this evolutionary process is now for the first time this year a 50% assessed course in Miranda for both first-year computer science students and for first-year joint computer science and electronic engineering honours students.

The utopian dream of understanding functional programming by first understanding the foundations of constructive types and Tarski's fixed point theory has apparently 'bitten the dust' at Warwick. This should not be misinterpreted as saying that we are totally disregarding essential theoretical topics in our approach to teaching software engineering with functional languages. Our approach has become more pragmatically based in trying to get students to experience the science of declarative programming. The course this year has been a successful experiment in providing students with such experience. Ironically, as will be shown below, the result has been to highlight the necessity of a more rigorous approach to teaching high-level programming based upon the very methods which we have 'let slip' in favour of 'pragmatism'. Instead of trying (in vain) to convince computer science students that a logic of functions can be applied to the programming of realistic problems we are now taking the approach of letting them convince themselves by giving them the opportunity to write medium size programs. We are thus faced with a challenge of how sucessfully to introduce topics to students such as types, higher-order functions, and streams in step with their appreciation of functional programming. To take such an approach now seems prudent for non-academic reasons. Students in the UK are fast becoming 'consumers' of education, and so will become increasingly intolerant of courses which appear 'irrelevant' to themselves. Today a

'theory first without experience' approach appears to be a certain way to invite extinction for functional programming on a computer science degree.

2.2. *Functional languages*

Functional languages have been available for several years now. Many came into existence as research tools, were not designed with teaching in mind, and certainly could not be described as 'user-friendly'. With the development of sophisticated languages such as Standard ML [4] and Miranda [5]†, the opportunities for using a functional language as a teaching tool have increased dramatically.

For the purposes of this paper, we can divide functional languages into two principal categories. Firstly, there are the *strict* languages, such as Standard ML, whose semantics are based on 'call-by-value'. A function defined in such a language requires all its arguments to be passed to it fully evaluated. Secondly there are the *lazy* languages, such as Miranda, whose semantics are 'call-by-need'. In such a language an expression is evaluated *only* if it is actually required.

The programming skills necessary for languages in these two categories are significantly different. We therefore felt that a conscious decision was required as to which to offer our students. To make that decision one has to trade off the potential efficiency of strict language implementations against the arguably richer constructs (such as infinite data structures) available in the latter. We adopted a language in the latter category; a major reason for this decision was that one can model strict semantics in a lazy language, but the converse is difficult. There are other criteria we are also able to use in order to identify a suitable language for the task.

A major feature of a functional language is its type system; a language may be weakly or strongly typed, possibly with polymorphic typing. From the viewpoint of the programmer, strong and polymorphic typing is beneficial as programming errors can be detected early on by a compiler. In this category we can cite *inter alia* Miranda, Haskell [6] and Hope [7].

Another important consideration when choosing a language for teaching purposes is ease of use. It is undesirable to let students be bogged down with the mechanics of getting a simple program to run. The user-interface is therefore important, and an area which has regrettably been developed in few functional languages.

Also of importance are the machine resources consumed by a program. If a program takes up too much memory to run, the size of class that can work on such a program at any one time will be limited. Similarly if the time required for compilation and execution is excessive then progress will be hampered. A number of language implementations were regarded as unacceptable for our course for precisely these reasons, and for the time being we have found that Miranda suits our purposes well.

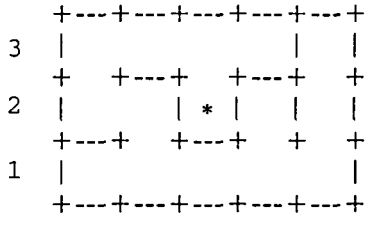
3. The maze problem

For the 50% assessed component of our first year course, students were required to write a Miranda script which would then be tested. We now look in detail at this particular exercise.

† Miranda is a trademark of Research Software Ltd.

3.1. *Description of the maze problem*

A 'Maze' is a rectangular grid of paths such as



typically found in the garden of an English stately home. The first part of the exercise was to produce a script to draw a picture on a dumb terminal (such as an ADM3E) for any given specification of a maze. The second part of the exercise was to write an interactive script which would allow the user to move around the maze. The maze exercise has some definitely novel features for an assignment in functional programming at Warwick. To teach first years was an experiment, but to give them an interactive dumb terminal graphics exercise appeared bold at the time. Lessons learnt here were not only that such exercises were quite within the reach of the vast majority of first years, but that this could be achieved with the minimal amount of programming theory. 90% of students submitted scripts which could, by and large, draw the correct maze and move around it.

The maze problem was chosen as it is an exercise in which the programmer must think of a maze as both a 'data structure' and as an arrangement of characters on a screen. As a data structure a maze has logical qualities such as those which define horizontal and vertical paths. Also, the conditions which determine whether the walker can move in a given direction in a given position are logical properties of this data structure. The arrangement of characters on the screen is a representation of the data structure. It was clear from the marking of the assignment that many students were not aware of this distinction, preferring to think of the maze as solely a problem in character manipulation. The drawback with the maze problem was that the logical properties of a maze can be programmed at a lower character level. The lesson here was that whatever wise words were given in lectures on data structures students will try to avoid them if they can 'hack' their way around high-level concepts using low-level structures such as characters.

A maze can be described as a sequence of line segments, each segment being defined by the coordinates of its endpoints. A Miranda specification for a maze is of the type

```
maze :: [ ((num, num), (num, num)) ]
```

where num is the data type for a number (integer or floating point), (num, num) is a pair of numbers (which thus can represent an (x, y) coordinate) and ((num, num)), (num, num)) a pair of coordinates (thus denoting a line segment). The square brackets in the type specification indicate a list of objects of the type specified between them, so here the data type `maze` does indeed denote a list of line segments.

For example, the above maze is defined by

```
maze = [((1,3),(4,3)),
        ((1,2),(2,2)),
        ((1,1),(5,1)),
        ((1,2),(1,3)),
        ((2,1),(2,2)),
        ((3,2),(3,3)),
        ((4,1),(4,2)),
        ((5,1),(5,3))  ]
```

The set of valid maze specifications is a proper subset of the set of values of the Miranda type necessitating error handling for maze specifications. Despite being told expectations on error handling 47% failed to complete this stage of the problem. Although there is no way of knowing for sure it is most probable that students did not use a sufficiently varied range of example mazes upon which to test their work. It is doubtful whether students at this stage are aware of their dependency upon an empirical debugging method which cannot be exhaustively completed. After marking this error handling section it was unfortunately too late to use this example to impress upon the class that other more rigorous methods were needed to correctly implement the error handling.

Many students did not recognize the need to return informative error messages, so failing to emphasize with the user of their software. Using tools from predicate logic and arithmetic it is quite easy to identify the most informative error for a given maze specification. These problems with error handling highlighted the difficulty some students found in discussing the properties of a data structure such as a maze in first-order predicate logic. For example, how do I express that the width of a maze must be a non-negative odd integer?

```
1<=width & width mod 2=1 & width=entier width
```

It was not always clear that the centre square (marked with a '*') could be defined by,

```
((width+1)/2, (height+1)/2)
```

The essential interplay between predicate logic and program development in a functional language needs to be developed more in future courses.

Drawing mazes was, by and large, quite successful. One recurring problem did highlight the inability of people to appreciate a maze as a grid of squares. A 'path' is a sequence either horizontal or vertical of squares, thus allowing for the possibility that a path may be of length one. However, 44% of people failing to understand unit length paths interpreted such isolated squares as solid hedges. A more serious problem arising from an incomplete understanding of a maze as a data structure was the following. Interactive commands to be allowed for movement in the maze were 'u' (up), 'd' (down), 'l' (left), and 'r' (right). Any one sequence of such commands which track out a path to the centre of a given maze in one's student's work should thus do likewise for any other student. However, 27% interpreted these commands as movement by character positions, thus not appreciating the intended homeomorphism between two different solutions to the maze problem. Much time was spent labouring the message that the exercise was a commission for them to meet a specification, however this message does not appear to have sunk in. The four above mentioned commands for movement around the maze were specified as the only

permitted commands along with their intended behaviour in the maze data structure. However, 23% of students insisted on including additional commands such as 'rr' (move right twice) undoubtedly conceived by students as a natural extension worthy of extra marks. According to the precise written specification of the maze problem it is an illegal command and so its existence resulted in the deduction of marks. Marks were thus deducted for an extension to the required script inconsistent with the original specification. An extension made by 4% of the students which was warmly welcomed was the use of dumb terminal graphics when only the use of textual characters was required. Only one student managed to reconcile both the original specification with an extension into graphics characters.

After the initial culture shock of facing a functional language there was only really one area of significant concern for the students outside of program design. The interleaving of a sequence of inputs with a sequence of outputs has to be explicitly programmed in a Pascal-like language. In a lazy functional language such as Miranda it was all too easy for the teacher to forget that demands for outputs generated by the evaluator are interleaved with demands for inputs also generated by the evaluator. This led to considerable initial confusion. Another related problem in Miranda was the use of the pre-defined input variable '`$-`', which at run-time is associated with the UNIX® 'standard input' stream. It took many people a long while to realise that `$-` has a type `[char]` in exactly the same way that any other `[char]` variable has. Having grasped this it was even more difficult to get many people to use the conversion function

```
lines :: [char] -> [[char]]
```

for 'abstracting' character input into a list of strings.

4. Conclusions

Teaching a functional language today in a British university is far from uncommon. As a vehicle for introducing formal notions into programming we too conclude that functional languages should play an important role in the computer science undergraduate curriculum. We cannot escape, however, the political realities of a changing British university system in which students are fast becoming consumers of educational products. No longer is it possible to preach the virtues of the lambda calculus to an élite captive audience, instead functional languages are now just another product to be sold to students. Our experience suggests that the traditionally difficult task of convincing students of the merits of programming using executable specifications has now been complicated by political changes far beyond our control. We will have to fight even harder in the future to prevent student misconceptions of functional languages as competitors to the more mainstream languages such as Pascal and C. At Warwick we have chosen to confront this challenge by replacing our traditional theoretical approach by a hands-on approach.

Concern over the effect of national curriculum changes on core mathematical skills of incoming home undergraduates is growing. Our experience suggests that if such concern is well founded functional languages will become much harder to teach. Ironically a functional language is an ideal vehicle for teaching both essential mathematical and logical skills in programming. It will be interesting to see whether functional languages become a teaching aid for enhancing mathematical and logical skills or whether they remain a serious component of the undergraduate programming agenda.

References

[1] VON GLASERSFELD, E., 1983, *Learning as a constructive activity* in *Proceedings of the 5th Annual Meeting of the North American Chapter of the International Group for the Psychology of Mathematics Education*, Montreal, edited by J. Bergeron and N. Herscovics (Montreal, Canada: Faculté de Sciences et de l'Education, Université de Montreal), pp. 41–70.

[2] BERRY, D., 1981, The Pifl programmer's manual, Department of Computer Science, University of Warwick, Coventry.

[3] ASHCROFT, E. A., and WADGE, W., 1985, *Lucid the Dataflow Programming Language* (Reading, MA: Addison-Wesley).

[4] WIKSTRÖM, Å., 1986, *Functional Programming Using Standard ML* (London: Prentice-Hall).

[5] HOLYER, I., 1991, *Functional Programming with Miranda* (London: Pitman).

[6] DAVIE, A. J. T., 1992, *An Introduction to Functional Programming Using Haskell* (Cambridge, UK: Cambridge University Press).

[7] BAILEY, R., 1990, *Functional Programming with Hope* (Chichester, West Sussex: Ellis Horwood).