JFP **14** (2): 129–135, March 2004. © 2004 Cambridge University Press
DOI: 10.1017/S0956796803004817 Printed in the United Kingdom

EDUCATIONAL PEARL

The Functional "C" experience

PIETER HARTEL
Department of Computer Science, University of Twente, The Netherlands
(*e-mail:* pieter@cs.utwente.nl)

HENK MULLER
Department of Computer Science, University of Bristol, Bristol, UK
(*e-mail:* henkm@cs.bris.ac.uk)

HUGH GLASER
Department of Electronics and Computer Science, University of Southampton, Southampton, UK
(*e-mail:* hg@ecs.soton.ac.uk)

Abstract

A functional programming language can be taught successfully as a first language, but if there is no follow up the students do not appreciate the functional approach. Following discussions concerning this issue at the 1995 FPLE conference (Hartel & Plasmeijer, 1995), we decided to develop such a follow up by writing a book that teaches C to students who can write simple functional programs. This paper summarises the essence of our approach, which is based on program transformation, and presents our experience teaching functional C at the Universities of Southampton and Bristol.

1 Introduction

At many universities students are taught declarative programming (Thompson & Wadler, 1993; Glaser *et al.*, 2001) as a first language. The reasons behind this are not only to show students a different programming language, but often to change the way students think about programming. Indeed, we can often observe that C programmers change programming style once they are well acquainted with other programming paradigms. For example, exposing C programmers to an object-oriented language often inspires them to make better use of abstract data types. Once students have become competent declarative programmers, they often use fewer global variables in C, and make only use of side effects when it is clearly beneficial.

In this paper, we report on our experience with functional and imperative languages in the first programming course at the Universities of Southampton and Bristol in the UK. Our approach is different from others in two respects. Firstly we wrote a text book based on program transformation to support the course. Secondly, we developed a methodology (programming by numbers) used to help

students with the minutiae of programming. The programming by numbers method is not explicitly mentioned in the book, because the two were developed roughly at the same time.

Section 2 describes the philosophy of our text book, and Section 3 summarises the programming by numbers method. Section 4 discusses some of the main differences between functional and imperative programming, and in particular the tension arising from thinking recursively and programming iteratively. Section 5 gives some anecdotal reports from student feedback. The reviews on the book may be found in Section 6, and the last section concludes.

2 Functional C

Functional C (Hartel & Muller, 1997) was intended as a book to teach people to program competently in C, while applying a functional style throughout. This does not imply that side effects or pointers aren't taught; we teach them while emphasising in which contexts they are useful, and when they are to be avoided.

From the outset, we wanted to be able to teach students programming using a single methodology. For this we choose informal refinement, from a specification via a declarative algorithm into an imperative implementation.

Some problems, such as the factorial, or greatest common divisor, are easy to write as a mathematical specification. Other problems are best specified in natural language, such as "print a table of primes". Once the specification has been written, the search for a suitable algorithm begins. Sometimes it is possible to prove formally that an algorithm satisfies the specification; often this is too difficult for beginning programmers, so we have to content ourselves with convincing arguments. This completes the difficult part of the task for the functional programmer. The imperative programmer needs to make one further step and should turn the algorithm into a suitable C program. This last step is the main topic of the book. However, we often refer back to the first two steps, i.e. the specification and the algorithm.

In the rest of this paper we first visit the programming method that we advocate in the course, and then we visit the order in which we teach imperative programming concepts. Some "difficult" concepts are taught early on, whereas the "easy" concepts (such as global variables) are taught much later in the course. Section 5 discusses the experiences we had with teaching students programming using this programming method. Section 6 quotes some of the reviews of the book and the final section concludes.

3 Programming by numbers

In our course (but not explicitly in the book), we use a methodology called programming by numbers (Glaser *et al.*, 2000), which we designed particularly for the benefit of novice programmers. As the name suggests, programming is a step-by-step process. We will comment upon each of these steps, indicating the strengths and weaknesses of the book in the light of the programming methodology

Step 1: Name The first step is to state the name of a function, which provides a useful starting point for the development. There is no difference in naming functions in mathematics, a functional language or C.

Step 2: Types The second step of Programming by Numbers requires students to write the type of the function. This step also carries over from the functional domain to the imperative domain. The student who is used to writing types of functions in a functional language can do so in C considering that the basic types available in C are not too different from those available in the functional language (i.e. `int` → `int`, `real` → `float` etc). There is a syntactic hurdle to be taken, which is that of having to write the names of formal arguments interspersed with the types in a C prototype. Once this hurdle is overcome, we believe that students can feel confident about writing function headers.

Steps 3–6: Case Analysis All non-trivial functions recur over some inductively defined data structure. A function would normally have one or more base cases, and one or more general cases (though variations are possible). The majority of the steps identified by the programming by numbers method relate to deciding which cases to use and how to decide what the functions results should be.

Deciding what the cases should be is largely driven by the types to be used for the functions arguments and result. Having decided on the case it is often easy to decide what the function result should be for the base case(s). The general case(s) is(are) often more difficult to solve, because this is where the recursive 'solution' of a sub-problem is used. We encourage our students to practice the analysis of problems, discovering the right cases and solving the sub problems defined by the cases. This requires a significant investment on the part of the lecturer and the students, which in the case of functional programming is well spent. In the case of imperative programming, we had to struggle with loops (See Section 4.1).

Step 7: Think This last step is a reflection on the activities that the student has engaged in as well as the results produced.

4 Imperative programming

The imperative part of the course discusses almost all elements of C (the only aspects excluded are bit-fields in structures, and other aspects of the language that are not widely used.). In Table 1 we list the elements in the order that we teach them, with the place in which they occur in our material, and the place they occur in two well established books on C: The C programming Language (Kernighan & Ritchie, 1978), and C by Dissection (Kelley & Pohl, 1996).

As one may expect, functions and recursion are discussed earlier in Functional C, whereas global variables and assignment are discussed later. There are however a couple of other, more interesting, elements that stand out.

- The *struct* is discussed early on in Functional C, as the first user defined data type. Both other books discuss arrays before structures.
- The *union* follows directly from Algebraic Data Types, and it is discussed together with a *struct* and *switch* in Functional C. In the C Programming

https://doi.org/10.1017/S0956796803004817 Published online by Cambridge University Press

Table 1. *Table showing where in each book C concepts are taught. We have excluded the introductory chapters from the books, and list the start page of the main explanation of each topic. A dash (–) indicates that this subject is not discussed in the main text*

Element	Functional C	C programming	by Dissection
Earlier than C books			
Functions	11 (1%)	69 (25%)	139 (20%)
Recursion	11 (1%)	86 (38%)	386 (73%)
Type definitions	23 (5%)	146 (83%)	265 (47%)
If-then-else	26 (6%)	55 (15%)	97 (10%)
Abort/exit	30 (7%)	163 (96%)	172 (27%)
Structs	114 (34%)	128 (69%)	412 (79%)
Dynamic memory	175 (54%)	167 (99%)	343 (64%)
Same time as C books			
Numbers	11 (1%)	37 (1%)	55 (1%)
While loops	66 (19%)	60 (18%)	102 (11%)
Breaks	85 (25%)	64 (21%)	115 (14%)
For loops	93 (27%)	60 (18%)	106 (12%)
Enumerated types	24 (5%)	39 (3%)	263 (46%)
Arrays	157 (48%)	98 (47%)	328 (61%)
Strings	162 (50%)	104 (51%)	362 (68%)
Multi dimensional arrays	199 (62%)	111 (57%)	338 (63%)
Streams (I/O)	249 (78%)	151 (87%)	455 (88%)
Later than C books			
Local variables	61 (17%)	40 (3%)	49 (0%)
Global variables	302 (95%)	40 (3%)	299 (54%)
Assignments	62 (17%)	42 (5%)	51 (0%)
Switch	122 (37%)	58 (17%)	117 (15%)
Pointers	127 (38%)	94 (44%)	288 (52%)
Lifetime	135 (41%)	73 (28%)	299 (54%)
Pointer arithmetic	187 (58%)	101 (49%)	335 (62%)
#include, ifdef, define	280 (88%)	88 (39%)	158 (24%)
Recursive data types	205 (64%)	140 (78%)	428 (82%)
Unusual			
Unions	119 (36%)	148 (84%)	–
Higher order functions	40 (10%)	119 (63%)	–
Partial application (void*)	139 (42%)	–	–
Varargs	231 (72%)	155 (90%)	–
Opaque structures	307 (97%)	–	–
Polymorphism (void*)	310 (98%)	120 (63%)	–

Language the *union* is not discussed until right at the end, and it is not mentioned at all (except in the reference section) in C by Dissection.

- Arrays, pointers, strings are all discussed roughly in the same place in all three texts.

https://doi.org/10.1017/S0956796803004817 Published online by Cambridge University Press

- Advanced subjects such as partial applications and polymorphism (essential to proper development) follow naturally from a functional context, and do not appear in C by Dissection at all.
- Error handling, and stopping programs using `abort()` follows naturally from partial functions.
- Functional C does not introduce global variables until page 300. This is where we need a random number generator. Functional programmers often find themselves passing the current seed of the random number generator from function to function to make sure that it can be refreshed when the next random number is drawn. A monad may make this somewhat easier to do but definitely does not contribute to insight. In C it is natural to pin the seed value down in a global variable and to pick it up when needed again.

4.1 Loops

Functional programmers use recursion all the time, simply because it is a natural and fundamental tool. Imperative programmers, and certainly inexperienced imperative programmers, shy away from recursion; instead they use loops wherever possible. Recursion is perceived to be more difficult to understand than loops, and also believed to be inefficient. However, using recursive solutions can often be a less error-prone methodology. To persuade students to use recursion as much in their C code as in their functional programs simply does not work. Therefore, we devised an approach that transliterates functional programs into C. In the book we provide what we believe to be straightforward 'recipes' that turn tail-recursive functions into C code with loops. The recipes were designed so that they could be followed almost mechanically. In retrospect this was a mistake. We could not persuade students to follow the steps meticulously (requiring non-negligible amounts of work) only to produce small fragments of programs. The upshot of all this is, of course, that there is a gap between functional and imperative programming, which we knew we could bridge in theory, using elementary program transformations. In practice, many of our students did not wish to cross our bridge.

Those with some, or significant programming experience would write the loops directly; students without programming experience had difficulty with getting the loops to work.

5 Experience

As expected with an experiment on this scale we have had a lot of feedback, both positive and negative. We have used the material during the academic years 96/97 (136 students), 97/98 (115 students) and 98/99 (120 students).

- Using transformation schemas to translate SML into C did not work well. Schemas are useful in showing the relation between, for example, recursion and a while-loop, or the use of an algebraic data structure and a switch, but after that, students want to get on with it and avoid using the schemas directly.

https://doi.org/10.1017/S0956796803004817 Published online by Cambridge University Press

- The biggest difficulty that we face is that even though one can teach any imperative language after any functional language, our materials have to be fixed to one particular functional language, and one particular imperative language; similar to programming textbooks being fixed on one language. This means that students think that the methodology is particular to those two languages, and it reduces the interest of any methodology to a small group of people.
- Many students dislike the use of programs that do not require the use of read and print functions or even graphical user dialogues. Interestingly, the methodology that we advocate prolongs this torture so that students do not get to write windows user interfaces during their assignments.
- On the bright side, the strong students on the course seem to pick it up well, and are writing programs, which are well structured, avoiding global state, and using dynamic data types wherever required.

During those three years we taught the course, the majority of complaints of students was levied at the use of SML in teaching C. Students did not appreciate this, in their view, contortious route because of the use of 'non-real' examples, such as `factorial` and `map`. Students would have preferred to print tables of prime numbers, or to write simple bookkeeping programs, with input-output.

In addition, even when we started in 1996, students felt that they should be taught Java. One student put it succinctly thus: "C is very SMeLly".

6 Reviews

This section presents excerpts from the reviews on the book that we have been able to find on the web. The first is from the Association of C and Unix Users (`www.accu.org`), which is a non-profit organisation devoted to professionalism at all levels in C, C++ and Java. Brian Bramer (Department of Computing Science, DeMontfort University, Leicester) writes (in March 1998).

> With so many books already written on C, one wonders why anyone would want to write another one and what makes this one different enough to make people buy it. This book is designed for a fairly select 'niche' audience – those who have been taught a functional programming language as their first programming language and want to learn an imperative language. It does not try to teach the reader how to program, but assumes they can program in SML, though more or less any functional programming language would do. Given this precondition, the book fulfils its aim.
> ...
> Overall, if you know how to program in a functional language and want or need to learn C, this book is worth considering.

The second review by Francis Glassborow (Editor of ACCU – C. Vu) adds:

> The caveat I have is that while you will learn to express functional programming in C, you will be less likely to understand how to write 'good C'. Think of all that documentation written in Japanese English and you will get the idea. The author's motive is to make the transition to C easy, but it is exactly those that find the transition hardest who will find the greatest difficulty in progressing further.

https://doi.org/10.1017/S0956796803004817 Published online by Cambridge University Press

The third review is by Josef Eschgfäller (Department of Maths, University of Ferrara, Italy):

This is a very beautiful book. The title is somewhat misleading; it is about teaching imperative programming using C as a second language to students who had functional language as the first programming language. But the algorithms start from a functional way of thinking and the authors give a very transparent presentation of the differences between C and functional programming and how their paradigms can be combined, for example in the chapter on lists, where append, filter and map are programmed in C. The book is exceptionally clear, on a good and interesting level and didactically I think very efficient, and nice and comfortable to read. Good appendices (a brief review of SML and the C standard libraries) and many interesting examples.

The opinion of the reviewers coincides with our experience, in the sense that lecturers appreciate the principle of the book, but students do not appreciate the practice.

7 Conclusions

We wrote *Functional C* as the basis of an extensive educational experiment, which aims to teach imperative programming as a logical follow up to functional programming. The experiment was a success because *Functional C* helps good students to develop a good programming style. The book has a major disadvantage too, because it alienates the less able students. As lecturers we enjoyed teaching the course more than we would have using a cut and dried method.

Acknowledgements

We thank three generations of students at Southampton and Bristol for their patience, help and feedback on our courses and the course material.

References

Glaser, H., Hartel, P. H. and Garratt, P. W. (2000) Programming by mumbers – a programming method for complete novices. *Comput. J.* **43**(4), 252–265.

Glaser, H., Hartel, P. H., Leuschel, M. and Martin, A. (2001) Declarative languages in education. *Encyclopaedia of Microcomputers*, vol. 27, pp. 79–102. Marcel Dekker.

Hartel, P. H. and Muller, H. L. (1997) *Functional C*. Addison-Wesley Longman.

Hartel, P. H. and Plasmeijer, M. J. (eds.) (1995) *1st Functional Programming languages in Education (FPLE)*, volume LNCS 1022. Springer-Verlag.

Kelley, A. and Pohl, I. (1996) *C by Dissection: the Essentials of C Programming*. Addison Wesley, Reading, Massachusetts, 1996.

Kernighan, B. W. and Ritchie, D. W. (1978) *The C programming language*. Prentice Hall.

Thompson, S. and Wadler, P. L. (1993) Special issue: Functional programming in education. *J. Functional Progra.* **3**(1), 1–3.