Where do I begin? A problem solving approach in teaching functional programming

Simon Thompson

Computing Laboratory
University of Kent at Canterbury
S.J.Thompson@ukc.ac.uk

Abstract. This paper introduces a problem solving method for teaching functional programming, based on Polya's *How To Solve It*, an introductory investigation of mathematical method. We first present the language independent version, and then show in particular how it applies to the development of programs in Haskell. The method is illustrated by a sequence of examples and a larger case study.

Keywords. Functional programming, Haskell, palindrome recognition, Polya, problem solving.

1 Introduction

Many students take easily to functional programming, whilst others experience difficulties of one sort or another. The work reported here is the result of attempts to advise students on how to use problem solving ideas to help them design and develop programs.

Some students come to a computer science degree with considerable experience of programming in an imperative language such as Pascal or C. For these students, a functional approach forces them to look afresh at the process of programming; it is no longer possible to construct programs 'from the middle out'; instead design has to be confronted from the start. Other students come to a CS programme with no prior programming experience, and so with no 'baggage' which might encumber them. Many of these students prefer a functional approach to the imperative, but lacking the background of the experienced students need encouragement and advice about how to build programs.[1]

In this paper we report on how we try to answer our students' question '*Where do I begin?*' by talking explicitly about problem solving and what it means in programming. Beyond enabling students to program more effectively a problem solving approach has a number of other important consequences. The approach is not only beneficial in a functional programming context, as we are able to use the approach across our introductory curriculum, as reported in [1], reinforcing ideas in a disparate set of courses including imperative programming and systems analysis. It is also striking that the cycle of problem solving is very

[1] Further reports on instructors' experience of teaching functional programming were given at the recent Workshop in the UK [6].

close to the 'understand, plan, write and review' scheme which is recommended to students experiencing difficulties in writing essays, emphasising the fact that problem solving ability is a transferable skill.

In this paper we first review our general problem solving strategy, modelled on Polya's epoch-making *How To Solve It*, [5], which brought these ideas to prominence in mathematics some fifty years ago. This material is largely language-independent. We then go on to explore how to take these ideas into the functional domain by describing '*How to program it in Haskell*'. After looking at a sequence of examples we examine the case study of palindrome recognition, and the lessons to be learned from this example. We conclude by reviewing the future role of problem solving in functional programming and across the computer science curriculum, since the material on problem solving can also be seen as the first stage in learning software engineering, 'in the small' as it were; more details are given in [1].

I am very grateful to David Barnes and Sally Fincher with whom the cross-curricular ideas were developed, and to Jan Sellers of the Rutherford Study Centre at the University of Kent who provided support for workshops in problem solving, as well as pointing out the overlap with essay writing techniques. The Alumni Fund of the University of Kent provided funding for Jan to work with us. Finally I would like to acknowledge all the colleagues at UKC with whom I have taught functional programming, and from whom I have learned an immense amount.

2 How To Program It

Polya's *How To Solve It*, [5], contains a wealth of material about how to approach mathematical problems of various kinds. This ranges from specific hints which can be used in particular circumstances to general methodological ideas. The latter are summarised in a two-page table giving a four-stage process (or more strictly a cycle) for solving problems. In helping students to program, we have specified a similar summary of method – *How To Program It* – which is presented in Figures 1 and 2. The stages of our cycle are: understanding the problem; designing the program; writing the program and finally looking back (or 'reflection').

The table is largely self-explanatory, so we will not paraphrase it here; instead we will make some comments about its structure and how it has been used.

How To Program It has been written in a language-independent way (at least as much as the terminology of modern computing allows). In Section 3 we look at how it can be specialised for the lazy functional programming language Haskell, [4, 7]. Plainly it can also be used with other programming languages, and at the University of Kent we have used it in teaching Modula-3, [1], for instance.

Our approach emphasizes that a novice can make substantial progress in completing a programming task *before* beginning to write any program code. This is very important in demystifying the programming process for those who find it difficult. As the title of this paper suggests, getting started in the task

UNDERSTANDING THE PROBLEM

First understand the problem.

Name the program or function.

What is its type?

What are the inputs (or arguments)? What are the outputs (or results)? What is the specification of the problem?

Can the specification be satisfied? Is it insufficient? or redundant? or contradictory? What special conditions are there on the inputs and outputs?

Does the problem break into parts? It can help to draw diagrams and to write things down in pseudo-code or plain English.

DESIGNING THE PROGRAM

In designing the program you need to think about the connections between the input and the output.

If there is no immediate connection, you might have to think of auxiliary problems which would help in the solution.

You want to give yourself some sort of plan of how to write the program.

Have you seen the problem before? In a slightly different form?

Do you know a related problem? Do you know any programs or functions which could be useful?

Look at the specification. Try to find a familiar problem with the same or similar specification.

Here is a problem related to yours and solved before. Could you use it? Could you use its results? Could you use its methods? Should you introduce some auxiliary parts to the program?

If you cannot solve the proposed problem try to solve a related one. Can you imagine a more accessible related one? A more general one? A more specific one? An analogous problem?

Can you solve part of the problem? Can you get something useful from the inputs? Can you think of information which would help you to calculate the outputs? How could you change the inputs/outputs so that they were *closer* to each other?

Did you use all the inputs? Did you use the special conditions on the inputs? Have you taken into account all that the specification requires?

Fig. 1. How To Program It, Part I

WRITING YOUR PROGRAM

Writing the program means taking your design into a particular programming language.

In writing your program, make sure that you check each step of the design. Can you see clearly that each step does what it should?

Think about how you can build programs in the language. How do you deal with different cases? With doing things in sequence? With doing things repeatedly or recursively?

You can write the program in stages. Think about the different cases into which the problem divides; in particular think about the different cases for the inputs. You can also think about computing parts of the result separately, and how to put the parts together to get the final results.

You can think of solving the problem by solving it for a smaller input and using the result to get your result; this is recursion.

You also need to know the programs you have already written, and the functions built into the language or library.

Your design may call on you to solve a more general or more specific problem. Write the solutions to these; they may guide how you write the solution itself, or may indeed be used in that solution.

You should also draw on other programs you have written. Can they be used? Can they be modified? Can they guide how to build the solution?

LOOKING BACK

Examine your solution: how can it be improved?

Can you test that the program works, on a variety of arguments?

Can you think of how you might write the program differently if you had to start again?

Can you see how you might use the program or its method to build another program?

Fig. 2. How To Program It, Part II

can be a block for many students. For example, in the first stage of the process a student will have to clarify the problem in two complementary ways. First, the informal statement has to be clarified, and perhaps restated, giving a clear informal goal. Secondly, this should mean that the student is able to write down the name of a program or function and more importantly give a type to this artifact at this stage. While this may seem a small step, it means that misconceptions can be spotted at an early stage, and avoid a student going off in a mistaken direction.

The last observation is an example of a general point. Although we have made reflection (or 'looking back') the final stage of the process, it should permeate the whole process. At the first stage, once a type for a function has been given, it is sensible to reflect on this choice: giving some typical inputs and corresponding outputs, does the type specified actually reflect the problem? This means that a student is forced to check both their understanding of the problem and of the types of the target language.

At the design stage, students are encouraged to think about the context of the problem, and the ways in which this can help the solution of the problem itself. We emphasise that programs can be re-used either by calling them or by modifying their definitions, as well as the ideas of specialisation and generalisation. Generalisation is particularly apt in the modern functional context, in which polymorphism and higher-order functions allow libraries of general functions to be written with little overhead (in contrast to the C++ Standard Template Library, say).

Implementation ideas can be discussed in a more concrete way in the context of a particular language. The ideas of this section are next discussed in the context of Haskell by means of a succession of examples in Section 3 and by a lengthier case study in Section 4. Note that the design stage of the case study is essentially language independent.

Students are encouraged to reflect on what they have achieved throughout the problem solving cycle. As well as testing their finished programs, pencil and paper evaluation of Haskell programs is particularly effective, and we expect students to use this as a way of discovering how their programs work.

3 Programming it in Haskell

As we saw in the previous section, it is more difficult to give useful language-independent advice about how to write programs than it is about how to design them. It is also easier to understand the generalities of *How To Program It* in the context of particular examples. We therefore provide students with particular language-specific advice in tabular form. These tables allow us to

- give examples to illustrate the design and programming stages of the process, and
- discuss the programming process in a much more specific way.

The full text of *Programming it in Haskell* is available on the World Wide Web, [9]. Rather than reproduce it here, in the rest of this section we look at some of the examples and the points in the process which they illustrate.

Problem: find the maximum of three integers

A first example is to find the maximum of three integers. In our discussion we link various points in the exposition to the four stages of *How To Program It*.

Understanding the problem Even in a problem of this simplicity there can be some discussion of the specification: what is to be done in the case when two (or three) of the integers are maximal? This is usually resolved by saying that the common value should be returned, but the important learning point here is that the discussion takes place. Also one can state the name and type, beginning the solution:

```
maxThree :: Int -> Int -> Int -> Int
```

Designing and writing the program More interesting points can be made in the design stage. Given a function `max` to find the maximum of two integers,

```
max :: Int -> Int -> Int
max a b
  | a>=b         = a
  | otherwise    = b
```

this can be used in two ways. It can form a model for the solution of the problem:

```
maxThree a b c
  | a>=b && a>=c      = a
  |     ....
```

or it can itself be used in a solution

```
maxThree a b c = max (max a b) c
```

It is almost universally the case that novices produce the first solution rather than the second, so this provides a useful first lesson in the existence of design choices, guided by the resources available (in this case the function `max`). Although it is difficult to interpret exactly why this is the case, it can be taken as an indication that novice students find it more natural to tackle a problem in a single step, rather than stepping back from the problem and looking at it more strategically. This lends support to introducing these problem solving ideas explicitly, rather than hoping that they will be absorbed 'osmotically'.

We also point out that given `maxThree` it is straightforward to generalise to cases of finding the minimum of three numbers, the maximum of four, and so on.

Looking back Finally, this is a non-trivial example for program testing. A not uncommon student error here is to make the inequalities strict, thus

```
maxThreeErr a b c
  | a>b && a>c      = a
  | b>c && b>a      = b
  | otherwise       = c
```

This provides a discussion point in how test data are chosen; the vast majority of student test data sets do not reveal the error. A systematic approach should produce the data which indicate the error – `a` and `b` jointly maximal – and indeed the cause of error links back to the initial clarification of the specification.

Problem: add the positive numbers in a list

We use this example to show how to break down the process of *designing* and *writing* a program – stages two and three of our four-step process – into a number of simpler steps. The function we require is

```
addPos :: [Int] -> Int
```

We first consider the design of the equations which describe the function. A paradigm here if we are to define the function from scratch is *primitive recursion* (or structural recursion) over the list argument. In doing this we adopt the general scheme

```
addPos []    = ...
addPos (a:x) = ... addPos x ...
```

in which we have do define the value at `[]` outright and the value at `(a:x)` from the value at `x`. Completing the first equation gives

```
addPos []    = 0
```

The `(a:x)` case requires more thought. Guidance can often come from looking at examples. Here we take lists

```
[-4,3,2,-1]
[2,3,2,-1]
```

which respectively give sums `0` and `6`. In the first case the head does not contribute to the sum; in the second it does. This suggests the case analysis

```
addPos (a:x)
  | a>0         = ...
  | otherwise   = ...
```

from which point in development the answer can be seen. The point of this example is less to develop the particular function than to illustrate how the process works.

The example is also enlightening for the other design possibilities it offers by way of *looking back* at the problem. In particularly when students are acquainted with `filter` and `foldr` the explicit definition

```
addPos = foldr (+) 0 . filter (>0)
```

is possible. The definition here reflects very clearly its top-down design.

Further examples

Other examples we have used include

Maximum of a list This is similar to `addPos`, but revisits the questions raised by the `maxThree` example. In particular, will the `max` function be used in the definition?

Counting how many times a maximum occurs among three numbers This gives a reasonable example in which local definitions (in a `where` clause) naturally structure a definition with a number of parts.

Deciding whether one list is a sublist of another This example naturally gives rise to an auxiliary function during its development.

Summing integers up to `n` This can give rise to the generalisation of summing numbers from `m` to `n`.

The discussions thus far have been about algorithms; there is a wealth of material which addresses data and object design, the former of which we address in [9].

4 Case study: palindromes

The problem is to recognise palindromes, such as

```
"Madam I'm Adam"
```

It is chosen as an example since even for a more confident student it requires some thought before implementation can begin. Once the specification is clarified it presents a non-trivial design space in which we can illustrate how choices between alternative designs can take place. Indeed, it is a useful example for small-group work since it is likely that different groups will produce substantially different initial design ideas. It is also an example in which a variety of standard functions can be used.

We address the main ideas in this section; further details are available on the World Wide Web [8].

Understanding the problem

The problem is stated in a deliberately vague way. A palindrome can be identified as a string which is the same read forwards or backwards, so long as

(1) we disregard the punctuation (punctuation marks and spaces) in the string;
(2) we disregard the case (upper or lower: that is capital or small) of the letters in the string.

Requirement (2) is plainly unambiguous, whilst (1) will need to be revisited at the implementation stage.

Overall design

The palindrome example lends itself to a wide choice of designs. The **simpler problem** in which there is no punctuation and all letters in lower case can be helpful in two ways. It can either form a *guide* about how to write the full solution, or be *used* as a part of that solution. The choice here provides a useful discussion point.

Design: the simpler problem

Possible designs which can emerge here may be classified in two different ways.

- Is the string handled as a single entity, or split into two parts?
- Is comparison made between strings, or between individual characters?

These choices generate these outline designs:

- The string is reversed and compared with itself;
- the string is split, one part reversed and the result compared with the other part;
- the first and last characters are compared, and if equal are removed and an iteration or a recursion is performed;
- the string is split, one part reversed and the strings are then compared one character at a time.

Again, it is important for students to be able both to see the possibilities available, and to discuss their relative merits (in the context of the implementation language). Naturally, too, there needs to be a comparison of the different ways in which the string is represented.

Design: the full problem

Assuming we are to use the solution to the simpler problem in solving the full problem, we reach our goal by writing a function which removes punctuation and changes all upper case letters to lower case. Here again we can see an opportunity to split the task in two, and also to discuss the order in which the two operations are performed: do we remove punctuation before or after converting letters to lower case? This allows a discussion of relative efficiency.

Writing the program

At this point we need to revisit the specification and to make plain what is meant by punctuation. This is not clear from the example given in the specification, and we can choose either to be proscriptive and disallow everything but letters and digits, or to be permissive and to say that punctuation consists of a particular set of characters.

There are more specific implementation decisions to be taken here; these reinforce the discussions in Section 3. In particular there is substantial scope for using built-in or library functions.

We give a full implementation of the palindrome recognition problem in Figure 3.

```haskell
palin :: String -> Bool

palin st = simplePalin (disregard st)

simplePalin :: String -> Bool

simplePalin st = (rev st == st)

rev :: String -> String

rev []     = []
rev (a:st) = rev st ++ [a]

disregard :: String -> String

disregard st = change (remove st)

remove :: String -> String
change :: String -> String

remove [] = []
remove (a:st)
  | notPunct a  = a : remove st
  | otherwise   =     remove st

notPunct ch = isAlpha ch || isDigit ch

change [] = []
change (a:st) = convert a : change st

convert :: Char -> Char

convert ch
  | isCap ch      = toEnum (fromEnum ch + offset)
  | otherwise     = ch
    where
    offset = fromEnum 'a' - fromEnum 'A'

isCap :: Char -> Bool

isCap ch = 'A' <= ch && ch <= 'Z'
```

Fig. 3. Recognising palindromes in Haskell

Looking back

Using the approach suggested here, students see that the solution which they have chosen represents one branch in a tree of choices. Their solution can be evaluated against other possibilities, including those written by other students. There is also ample scope for discussion of testing in this problem.

For instance, the solution given in Figure 3 can give rise to numerous discussion points.

- No higher order functions are used in the solution; we would expect to revisit the example after covering HOFs to reveal that `change` is `map convert` and that `remove` is `filter notPunct`.
- In a similar way we would expect to revisit the solution and discuss incorporating function-level definitions such as

  ```
  palin = simplePalin . disregard
  ```

  This would also apply to `disregard` itself.
- Some library functions have been used; digits and letters are recognised by `isDigit` and `isAlpha`.
- An alternative definition of `disregard` is given by

  ```
  disregard st = remove (change st)
  ```

  and other solutions are provided by implementing the two operations in a single function definition, rather than as a composition of two separate pieces of functionality.
- We have chosen the proscriptive definition of punctuation, considering only letters and digits to be significant.

5 Conclusion

In this paper we have given an explicit problem solving method for beginning (functional) programmers, motivated by the desire to equip them with tools to enable them to write complex programs in a disciplined way. The method also gives weaker students the confidence to proceed by showing them the ways in which a seemingly intractable problem can be broken down into simpler parts which can be solved separately. As well as providing a general method we think it crucial to illustrate the method by examples and case studies – this latter approach is not new, see [2] for a very effective account of using case studies in teaching Pascal.

To conclude, it is worth noting that numerous investigations into mathematical method were stimulated by Polya's work. Most prominent are Lakatos' investigations of the roles of proof and counterexample, [3], which we believe have useful parallels for teachers and students of computer science. We intend to develop this correspondence further in the future.

References

1. David Barnes, Sally Fincher, and Simon Thompson. Introductory problem solving in computer science. In *CTC97, Dublin*, 1997.
2. Michael Clancy and Marcia Linn. *Designing Pascal Solutions: Case studies using data structures*. Computer Science Press, W. H. Freeman and Co., 1996.
3. Imre Lakatos. *Proofs and Refuations: The Logic of Mathematical Discovery*. Cambridge University Press, 1976. Edited by John Worrall and Elie Zahar.
4. John Peterson and Kevin Hammond, editors. *Report on the Programming Language Haskell, Version 1.3*. `http://haskell.cs.yale.edu/haskell-report/haskell-report.html`, 1996.
5. G. Polya. *How To Solve It*. Princeton University Press, second edition, 1957.
6. Teaching functional programming: Opportunities & difficulties. `http://www.ukc.ac.uk/CSDN/conference/96/Report.html`, September 1996.
7. Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.
8. Simon Thompson. Problem solving: recognising palindromes. `http://www.ukc.ac.uk/computer_science/Haskell_craft/palindrome.html`, 1996.
9. Simon Thompson. Programming it in Haskell. `http://www.ukc.ac.uk/computer_science/Haskell_craft/ProgInHaskell.html`, 1996.