Exploring Languages with Interpreters and Functional Programming Chapter 20

H. Conrad Cunningham

6 August 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of August 2018 is a recent version of Firefox from Mozilla.

20 Problem Solving

20.1 Chapter Introduction

This Chapter is incomplete.

- Add intro.
- Give better or more detailed examples.
- Add What Next? and Exercises

20.2 Problem Solving Philosophy

I approach computing science with the following philosophy:

- Programming is the essence of computing science.
- Problem solving is the essence of programming.

Here I consider programming as the process of analyzing a problem and formulating a solution suitable for execution on a computer. The solution should be correct, elegant, efficient, and robust. It should be expressed in a manner that is understandable, maintainable, and reusable. The solution should balance generality and specificity, abstraction and concreteness.

In my view, programming is far more than just coding. It subsumes the concerns of algorithms, data structures, and software engineering. It uses programming languages and software development tools. It uses the intellectual tools of mathematics, logic, linguistics, and computing science theory. Etc.

20.3 Polya's Insights

The mathematician George Polya (1887–1985), a Professor of Mathematics at Stanford University, said the following in the preface to his book *Mathematical Discovery: On Understanding, Learning and Teaching Problem Solving* [Polya 1981].

> Solving a problem means finding a way out of a difficulty, a way around an obstacle, attaining an aim which was not immediately attainable. Solving problems is the specific achievement of intelligence, and intelligence is the specific gift of mankind: solving problems can be regarded as the most characteristically human activity. ...
>
> Solving problems is a practical art, like swimming, or skiing, or playing the piano: you learn it only by imitation and practice. ... if you wish to learn swimming you have to go into the water, and if you wish to become a problem solver you have to solve problems.

> If you wish to derive the most profit from your effort, look out for such features of a problem at hand as may be useful in handling the problems to come. A solution that you have obtained by your own effort or one that you have read or heard, but have followed with real interest and insight, may become a *pattern* for you, a model that you can imitate with advantage in solving similar problems. ...
>
> Our knowledge about any subject consists of *information* and *know-how.* If you have genuine *bonafide* experience of mathematical work on any level, elementary or advanced, there will be no doubt in your mind that, in mathematics, know-how is much more important than mere possession of information. ...
>
> What is know-how in mathematics? The ability to solve problems—not merely routine problems but problems requiring some degree of independence, judgment, originality, creativity. Therefore, the first and foremost duty ... in teaching mathematics is to emphasize *methodical work in problem solving.*

What Polya says for mathematics holds just as much for computing science.

In the book *How to Solve It* [Polya 1945], Polya states four phases of problem solving. These steps are important for programming as well.

1. Understand the problem.
2. Devise a plan.
3. Carry out the plan, checking each step.
4. Reexamine and reconsider the solution. (And, of course, reexamine the understanding of the problem, the plan, and the way the plan was carried out.)

20.4 Problem-Solving Strategies

There are many problem-solving strategies applicable to programming in general and functional programming in particular. We have seen some of these in the earlier chapters and will see others in later chapters. In this section, we highlight some of the general techniques.

20.4.1 Solve a more general problem first

The first strategy is to *solve a more general problem first.* That is, we solve a "harder" problem than the specific problem at hand, then use the solution of the "harder" problem to get the specific solution desired.

Sometimes a solution of the more general problem is actually easier to find because the problem is simpler to state or more symmetrical or less obscured by special conditions. The general solution can often be used to solve other related problems.

Often the solution of the more general problem can actually lead to a more efficient solution of the specific problem.

We have already seen one example of this technique: finding the first occurrence of an item in a list.

First, we devised a program to find all occurrences in a list. Then we selected the first occurrence from the set of all occurrences. (Lazy evaluation of Haskell programs means that this use of a more general solution differs very little in efficiency from a specialized version.)

We have also seen several cases where we have generalized problems by adding one or more *accumulating parameters*. These "harder" problems can lead to more efficient tail recursive solutions.

For example, consider the tail recursive Fibonacci program we developed in a previous chapter. We added two extra arguments to the function.

```
fib2 :: Int -> Int
fib2 n | n >= 0 = fibIter n 0 1
    where
        fibIter 0 p q         = p
        fibIter m p q | m > 0 = fibIter (m-1) q (p+q)
```

Another approach is to use the *tupling* technique. Instead of adding extra arguments, we add extra results.

For example, in the Fibonacci program `fastfib` below, we compute `(fib n, fib (n+1))` instead of just `fib n`. This is a harder problem, but it actually gives us more information to work with and, hence, provides more opportunity for optimization. (We formally derive this solution in a later chapter.)

```
fastfib :: Int -> Int
fastfib n | n >= 0 = fst (twofib n)

twofib :: Int -> (Int,Int)
twofib 0 = (0,1)
twofib n = (b,a+b)
    where (a,b) = twofib (n-1)
```

20.4.2 Solve a simpler problem first

The second strategy is to *solve a simpler problem first.* After solving the simpler problem, we then adapt or extend the solution to solve the original problem.

Often the mass of details in a problem description makes seeing a solution difficult. In the previous technique we made the problem easier by finding a more general problem to solve. In this technique, we move in the other direction: we find a more specific problem that is similar and solve it.

At worst, by solving the simpler problem we should get a better understanding of the problem we really want to solve. The more familiar we are with a problem, the more information we have about it, and, hence, the more likely we will be able to solve it.

At best, by solving the simpler problem we will find a solution that can be easily extended to build a solution to the original problem.

Consider a program to convert a positive integer of up to six digits to a string consisting of the English words for that number. For example, `369027` yields the string:

```
three hundred and sixty-nine thousand and twenty-seven
```

To deal with the complexity of this problem, we can work as follows:

a. Solve the problem of converting a two-digit number to words. (The single digit numbers and numbers in teens are special cases.)
b. Then extend the two-digit solution to three digits.
c. Then extend three-digit solution to to six digits.

See Section 4.1 of the classic Bird and Wadler textbook [Bird 1988] for the details of this problem and a solution.

The process of generalizing first-order functions into higher-order functions is another example of this "solve a simpler problem first" strategy. Recall how we motivated the development of the higher-order library functions such as `map`, `filter`, and `foldr`. Also consider the function generalization approach used in the cosequential processing case study.

20.4.2.1 Reuse off-the-shelf solutions to standard subproblems

The third strategy is to *reuse an off-the-shelf solutions to a standard subproblem.*

We have been doing this all during this semester, especially since we began began studying polymorphism and higher-order functions.

The basic idea is to identify standard patterns of computation (e.g. standard prelude functions such as `length`, `take`, `zip`, `map`, `filter`, `foldr`) that will solve some aspects of the problem and then combine (e.g. using function composition) these standard patterns with your own specialized functions to construct a solution to the problem.

We have seen several examples of this technique in this textbook and its exercises.

See section 4.2 of the classic Bird and Wadler textbook [Bird 1988] for a case study that develops a package of functions to do arithmetic on variable length integers. The functions take advantage of several of the standard prelude functions.

20.4.3 Solve a related problem

The fourth strategy is to *solve a related problem.* After solving the related problem, we then transform the solution of the related problem to get a solution to the original problem.

Perhaps we can find an entirely different problem formulation (i.e. stated in different terms) for which we can readily find a solution. Then that solution can be converted into a solution to the problem at hand.

For example, we can recast a problem in terms of mathematical or logical frameworks (e.g. sets, relations, graphs, finite state machines, grammars, or algebraic structures), solve the corresponding problem in those terms, and then interpret the result for the original problem. The simplification provided by the frameworks may reveal solutions that are obscured in the details of the problem. We can also take advantage of the theory and techniques that have been found previously for the mathematical frameworks.

Consider the problem of breaking a string of text into the list of its component lines.

This is not trivial. However, the "inverse" problem is trivial. All that is needed to convert a list of lines to a string of text is to insert linefeed characters between the lines.

We can first solve the inverse problem (line-folding) and then use it to calculate what the line-breaking program must be. (See Section 4.3 of the Bird and Wadler textbook [Bird 1988] and a Chapter 27 in this textbook.)

20.4.4 Separate concerns

The fifth strategy is to *separate concerns.* That is, we partition the problem into logically separate problems, solve each problem separately, then combine the solutions to the subproblems to construct a solution to the problem at hand.

As we have seen in the above strategies, when a problem is complex and difficult to attack directly, we search for simpler, but related, problems to solve, then build a solution to the complex problem from the simpler problems.

We have seen examples of this approach in earlier chapters and homework assignments. We separated concerns when we used stepwise refinement to develop a square root function, data abstraction in the rational number case study, and function pipelines.

Consider the development of a program to print a calendar for any year in various formats. We can approach this problem by first separating it into two independent subproblems:

a. building a calendar

b. formatting the output

After solving each of these simpler problems, the more complex problem can be solved easily by combining the two solutions. (See Section 4.5 of the classic Bird and Wadler textbook [Bird 1988] for the details of this problem and a solution.)

20.4.5 Divide and conquer

The sixth strategy is *divide and conquer*. This is a special case of the "solve a simpler problem first" strategy. In this technique, we must divide the problem into subproblems that are the same as the original problem except that the size of the input is smaller.

This process of division continues recursively until we get a problem that can be solved trivially, then we combined we reverse the process by combining the solutions to subproblems to form solutions to larger problems.

Examples of divide and conquer from earlier chapters include the logarithmic exponentiation function `expt3` and the merge sort function `msort`.

Another common example of the divide and conquer approach is binary search. (See Section 6.4.3 of the classic Bird and Wadler textbook [Bird 1988].)

Chapter 29 of this textbook examines divide and conquer algorithms in terms of a higher order function that captures the pattern.

There are, of course, other strategies that can be used to approach problem solving.

20.5 What Next?

TODO

20.6 Exercises

TODO

20.7 Acknowledgements

In 2016 and 2017, I adapted and revised my previous notes to form Chapter 7, More List Processing and Problem Solving, in the 2017 version of this textbook. In particular, I drew the information on Problem Solving from:

- chapter 10 of my *Notes on Functional Programming with Haskell* for discussion of problem-solving techniques in section 7.4

The Notes drew on chapters 4 and 6 of [Bird 1988], chapter 4 of [Thompson 2011], [Polya 1945], and [Polya 1981].

- part of chapter 12 of *Notes on Functional Programming with Haskell* for discussion of the tupling technique incorporated into subsection 7.4.2.1

In Summer 2018, I divided the previous More List Processing and Problem Solving chapter back into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Previous sections 7.2-7.3 became the basis for new Chapter 18, More List Processing, and previous section 7.4 (essentially the two items above) became the basis for new Chapter 20 (this chapter), Problem Solving.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

20.8 References

[Bird 1988]: Richard Bird and Philip Wadler. *Introduction to Functional Programming*, First Edition, Prentice Hall, 1988.

[Bird 1998]: Richard Bird. *Introduction to Functional Programming using Haskell*, Second Edition, Prentice Hall, 1998.

[Bird 2015]: Richard Bird. *Thinking Functionally with Haskell*, Second Edition, Cambridge University Press, 2015.

[Cunningham 2014]: H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1993-2014.

[Polya 1945]: George Polya, *How to Solve It*, Princeton University Press, 1945.

[Polya 1981]: George Polya, *Mathematical Discovery: On Understanding, Learning and Teaching Problem Solving*, Wiley, 1981.

[Thompson 2011]: Simon Thompson. *Haskell: The Craft of Programming*, First Edition, Addison Wesley, 1996; Second Edition, 1999; Third Edition, Pearson, 2011.

20.9 Terms and Concepts

Problem solving, Polya, information, know-how, bonafide experience, problem-solving strategies, solve a more general (harder) problem first, accumulating parameters, tupling, solve a simpler problem first, reuse an off-the-shelf solution, higher-order functions, stepwise refinement, data abstraction, solve a related problem, separate concerns, divide and conquer.