Dynamic Technology Holdings

meet the λambdas

Workshop

A Taste of Algebra of Functions

Notes & Study Guide

Andrea Falconi
andrea.falconi@gmail.com

November, 2011

Odd-One-Out Challenge

Consider the following four problems.

P1. *Finding one odd occurrence.* Given a list of positive integers containing exactly one integer which occurs an odd number of times in the list, write a program to find that integer.[1] E.g. if you're given $[1, 2, 3, 1, 2, 3, 3]$, then your program should return 3.

P2. *Finding arbitrary odd occurrences.* How would you generalise the program so that it would still work for an arbitrary number of odd elements in the list? Specifically, your program should return the list (possibly empty!) of all integers that occur an odd number of times in the input list. Examples:

- given $[1, 2, 1, 2]$, you should return the empty list;
- given $[1, 2, 3, 1, 2, 3, 3]$, you should return the singleton list $[3]$;
- given $[1, 2, 3, 3, 2, 2]$, you should return $[1, 2]$.

P3. *Doing it for strings.* Would you be able to modify the program so that it worked on lists of strings too? E.g. given ["aa","bb","aa","cc"], your program should output ["bb","cc"].

P4. *Employees who like Haskell.* Now consider the problem of finding the teams in your company with at least two members whose favourite programming language is Haskell. You are given a list of employees and each employee has an associated team name and favourite programming language.

Once you're comfortable you have a solution for all these problems, you may want to try something more challenging. The questions below are meant to make you think about some central themes in programming. Don't worry if you don't come up with a solution at this stage; sometimes it's just good enough to try!

Challenge. *Abstraction and reuse.* How does problem 4 relate to the previous ones? Can you come up with a more general problem statement which would encompass all of the above? (i.e. problems 1 to 4 should become special cases of your generalised problem statement.) If you could do that, then could you also, accordingly, write a very generic and reusable program? Your core algorithm should be generic enough that it doesn't need to distinguish among all the above cases—i.e. can your algorithm be used to solve P1, P2, P3, and P4?

Hard Nuts. *Correctness and performance.* How would you go about convincing somebody else that your program solves the problem? What kind of evidence can you provide to prove your claim? Can you predict the performance of your program? That is, can you come up with some sort of formula which would allow you to estimate performance without having to run the program on a computer?

[1] This one seems to be a popular interview question and has a very elegant solution, see [3].

Solution: A Window into the Future

The solution we present below requires good knowledge of list processing theory and elementary abstract algebra.[2] It is more or less how you would solve the challenge by the end of Workshop IV—i.e. the kind of argument you should easily be able to put together at that stage. As such, you should think of it as a window into the future and come back here at the end of Workshop IV to understand it fully. In the mean time, you can have a quick read through to get a feel for the algebraic approach to programming.

Generalised Problem Statement. Problems P1, P2, P3, and P4 can be brought back to the problem of finding the classes of an equivalence kernel that satisfy some given criteria. Specifically, given a function $\mathsf{f} : [\alpha] \to \beta$ and a predicate $\mathsf{p} : [\alpha] \to \mathsf{Bool}$,[3] we want to find a function ϕ which is a solution of the equation

$$\boxed{\phi\ \mathsf{f}\ \mathsf{p} = (\mathsf{filter}\ \mathsf{p}) \circ (\mathsf{ker}\ \mathsf{f})} \tag{1}$$

In the above equation, filter is the standard list processing function which lists all the elements of an input list that satisfy p; $\mathsf{ker}\ \mathsf{f}$ is the function $[\alpha] \to [[\alpha]]$ which lists the classes of the equivalence kernel of f.

We see that P1, P2, and P3 can be readily restated in terms of the above. In fact, take the identity for f and, for the predicate p, the function that checks if an equivalence class has an odd number of members; then P1, P2, and P3 become the problem of computing the function $\mathsf{solveOdd}$ defined by the equation

$$\boxed{\mathsf{solveOdd} = (\mathsf{map}\ \mathsf{head}) \circ (\phi\ \mathsf{id}\ \mathsf{p})} \tag{2}$$

where $\mathsf{map}\ \mathsf{head}$ is the function $[[\alpha]] \to [\alpha]$ that lists the first member of each equivalence class. (Note that map and head are standard list processing functions.)

As for P4, call Empl, Team, and Lang, respectively, the type of employees, that of teams, and that of programming languages. The problem statement says that we are given a list of employees and two functions $\mathsf{team} : \mathsf{Empl} \to \mathsf{Team}$ and $\mathsf{lang} : \mathsf{Empl} \to \mathsf{Lang}$ that tell us, respectively, what team an employee belongs in and what is their favourite programming language. P4 becomes the problem of computing the function $\mathsf{solveEmpl}$ defined by the equation

$$\boxed{\mathsf{solveEmpl} = (\mathsf{map}\ (\mathsf{team} \circ \mathsf{head})) \circ (\phi\ \langle \mathsf{team}, \mathsf{lang} \rangle\ \mathsf{p})} \tag{3}$$

where $\langle \mathsf{team}, \mathsf{lang} \rangle\ \mathsf{e} = (\mathsf{team}\ \mathsf{e}, \mathsf{lang}\ \mathsf{e})$ and p checks if the equivalence class at hand has at least two members and their favourite language is Haskell.

[2] For a mathematical approach to list processing see e.g.[1]; for the algebra bits you may want to have a read through §1 of [2].

[3] $[\alpha]$ stands for the lists of elements of an arbitrary type α—i.e. α could stand for integers, strings, or employees; it doesn't matter. Ditto for β, whereas Bool is your run of the mill boolean type.

Program Construction and Verification. We can think of the three equations (1), (2), and (3) as a formal specification of P1, P2, P3, and P4. But we see that if we "plug" a computable definition of ker into equation (1), we get a working definition of ϕ which means we also get a solution of the generalised problem statement and hence a generic solution for all the other (more specific) problems too. This way the spec becomes a working program and the correctness of the program boils down to being able to prove that our definition of $\mathsf{ker}\ f$ actually computes the classes of the equivalence kernel of f.

The induction principle gives us an easy way to kill two birds with one stone: define $\mathsf{ker}\ f$ and prove, at the same time, that what we're defining actually computes the equivalence classes. The base case is when we're given an empty list; then there are no equivalence classes in the kernel and so we have to return the empty list. We can now make the assumption that $\mathsf{ker}\ f$ returns the classes (as lists) of the equivalence kernel of f when given an arbitrary list xs as input and give a suitable definition on $(x : xs)$—i.e. the list obtained from xs by adding an arbitrary element x to the front. We can achieve this by introducing a new function $\triangleright$ which given x and the classes ys computed out of xs (i.e. $ys = \mathsf{ker}\ f\ xs$) adds x to the class it belongs in. Here's the definition of ker then

$$\mathsf{ker}\ f\ [\,] = [\,]$$
$$\mathsf{ker}\ f\ (x : xs) = x \triangleright (\mathsf{ker}\ f\ xs)$$

And how do we define $\triangleright$? By induction, you say!

$$x \triangleright [\,] = [[x]]$$
$$x \triangleright (y : ys) = \begin{cases} (x : y)\ :\ ys & \text{if } f\ x = (f \circ \mathit{head})\ y \\ y\ :\ (x \triangleright ys) & \text{otherwise} \end{cases}$$

This is a routine definition as is the verification that $\triangleright$ actually adds x to the class it belongs in.

Finishing Touches. What have we got so far? We've come up with a generalised problem statement good enough to encompass all the other problems (P1 to P4); it is made up of the equations (1), (2), and (3) that served us as a formal spec from which we derived a program by providing a suitable definition of ker. In so doing, we've also proved that the program is correct. In conclusion, our (correct!) program consists of equations (1), (2), (3) along with those that define ker.

All is left to tackle is performance, but this is easy because of the recursive definition of ker we've given above. In fact, if τ_n is the maximum number of steps ker can take to compute its result given a list of length n, then from the definition of ker follows immediately the recurrence

$$\tau_0 = 0$$
$$\tau_{n+1} = \gamma_n + \tau_n$$

where γ_n is how many steps at most $\triangleright$ can take to complete. But the worst case scenario for $\triangleright$ is when it has to traverse the entire list $(y : ys)$, which can have n

elements at most;[4] then $\gamma_n = n$ is a fair estimate and so

$$\tau_{n+1} = n + \tau_n = n + (n-1) + \tau_{n-1} = \cdots =$$
$$n + (n-1) + \cdots + 1 + 0 = \frac{n^2+n}{2}$$

That is, ker is $O(n^2)$. The other functions that appear in (1), (2), and (3) are either standard list processing functions that run in linear time or predicates that are $O(n)$ too; we conclude that our program must run in $O(n^2)$ time.

Yet Another Challenge! So we've worked out a complete solution of the odd-one-out challenge. Are you ready for the next one? What about improving performance? Can we do better than $O(n^2)$? Here's an idea. Why not try a divide and conquer approach to defining ker? If we could somehow come up with an equation of the form ($\oplus$ stands for the list concatenation operator):

$$\mathsf{ker}\ f\ (xs \oplus ys) = \vartheta\ (\mathsf{ker}\ f\ xs)\ (\mathsf{ker}\ f\ ys)$$

and prove that a solution function ϑ does exist, then we could hope to bring down the running time to $O(n \cdot \log_2 n)$, or perhaps even better. For an alternative approach, have a look at how Bird[1] goes about partitioning a list.

[4]We would hit the worst case scenario precisely when f is injective.

Show me the Code!

At this point you may be wondering how to translate our program into something a computer can actually digest. Well, it turns out that if you pick a decent functional programming language, the translation into code is immediate; here's what the program looks like in Haskell.

```
module Main where
import Control.Arrow

ker f []     = []
ker f (x:xs) = x |> (ker f xs)
    where
    x |> []     = [[x]]
    x |> (y:ys) | f x == (f . head) y  = (x:y) : ys
                | otherwise            = y : (x |> ys)

solve f p = filter p . ker f

solveOdd :: Eq a => [a] -> [a]
solveOdd = map head . solve id p
    where p = (== 1) . (`mod` 2) . length

data Employee = E { name::String, team::Int, lang::String } deriving Show

solveEmpl = map (team . head) . solve (team &&& lang) p
    where
    p (e:t) = lang e == "Haskell" && length t > 0


main = do
    print . solveOdd $ [1, 2, 3, 1, 2, 3, 3]
    print . solveOdd $ [1, 2, 3, 3, 2, 2]
    print . solveOdd $ ["aa","bb","aa","cc"]
    print . solveEmpl $ employees

employees = [ E "John" 1 "Haskell"
            , E "Jane" 1 "Haskell"
            , E "Mark" 1 "Java"
            , E "Mary" 2 "Haskell"
            , E "Rob"  2 "Java"
            , E "Toby" 3 "Python"
            ]
```

Note how close the Haskell code is to the equations we used earlier to define the program in algebraic terms; basically a one-to-one translation, except for the main function and some test data below it which have been added as a convenience. You can try to run the program yourself at http://codepad.org/—copy & paste the code above into the form, select Haskell, and hit Submit.

References

[1] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987. NATO ASI Series F Volume 36. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.

[2] Saunders Mac Lane and Garret Birkhoff. *Algebra*. American Mathematical Society, 3 edition, June 1999.

[3] Technical Interview Questions. Array: Find the number with odd number of occurrences. http://www.technicalinterviewquestions.net/. Answers to frequently asked programming interview questions and puzzles asked at Google, Microsoft, Amazon, Yahoo, Facebook, MySpace, and such for SDE/Developer and SDET positions. Retrieved on 12 November 2011.