JFP **16** (1): 13–20, 2006.
doi:10.1017/S0956796805005678 Printed in the United Kingdom

FUNCTIONAL PEARL

Finding celebrities: A lesson in functional programming

RICHARD BIRD

Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK
(*e-mail:* Richard.Bird@comlab.ox.ac.uk)

SHARON CURTIS

Department of Computing, Oxford Brookes University, Wheatley Campus, Oxford OX33 1HX, UK

The setting is a class on functional programming. There are four students, Anne, Jack, Mary and Theo.

***Teacher*:** Good morning class. Today I would like you to solve the following problem. Given is a nonempty list *xs* of people at a party. By definition, a nonempty sublist *ys* of *xs* forms a *celebrity clique* if everybody at the party knows every member of *ys*, but members of *ys* know only each other. Assuming there is such a clique at the party, the problem is to write a functional program to find it. As data for the problem you can assume that the predicate *knows* is given, so *knows x y* is true just when *x* knows *y*.

***Jack*:** Just to be clear, does every member of a celebrity clique actually know everyone else in the clique? And does everyone know themselves?

***Teacher*:** As to the first question, yes, it follows from the definition: everyone in the clique is known by everyone at the party. As to the second question, ask a philosopher.

***Theo*:** This is going to be a hard problem, isn't it? I mean, the problem of determining whether there is a clique of size k in a party of n people will take $\Omega(n^k)$ steps, so we are looking at an exponential time algorithm.

***Anne*:** That doesn't follow since being a celebrity clique is a much stronger property than being a clique. In a directed graph, a clique is a set of nodes in which each pair of nodes has an arc in both directions between them, but a celebrity clique also requires an arc from every node in the graph to every node in the clique, and no arcs from the clique to nodes outside the clique.

***Mary*:** Yes, while there can be many cliques in a graph, there is at most one celebrity clique. Suppose that ys and zs are two celebrity cliques. Pick any y in ys. We have that z knows y for every z in zs by the fact that everybody in the clique ys is known by everybody at the party. Since zs is a celebrity clique, and clique members know only other members of the clique, we have that y has to be an element of zs. But y was arbitray, so ys is a subset of zs and hence, by symmetry, equal to zs.

***Theo*:** Agreed, they are different problems. Anyway, here is a straightforward exponential-time algorithm. Let $subseqs\ xs$ return the subsequences of a list xs and $test\ xs\ ys$ return true just when ys is a clique of xs. Then we can define

$$\begin{array}{lcl} find\ xs & = & head\ (filter\ (test\ xs)\ (subseqs\ xs)) \\ test\ xs\ ys & = & and\ [knows\ x\ y \mid x \leftarrow xs, y \leftarrow ys, x \neq y] \wedge \\ & & and\ [knows\ y\ x \Rightarrow elem\ x\ ys \mid x \leftarrow xs, y \leftarrow ys, x \neq y] \end{array}$$

The definition of *test* is a direct translation of the clique conditions into Haskell. I have included the guard $x \neq y$ in both list comprehensions simply to avoid the issue of whether everyone knows themselves or not.

***Mary*:** Well, your definition of *test* allows the empty sequence to be a celebrity clique of any party, so you have to be careful to ensure either that *subseqs* returns only the nonempty subsequences of a list, or that the empty sequence comes last in the enumeration. With the first choice, $find\ xs$ will return $\bot$ if xs does not contain a celebrity clique; with the second choice $find\ xs$ will return $[\,]$.

***Theo*:** I prefer total functions to partial ones, so let me take the second option and define

$$\begin{array}{lcl} subseqs\ [\,] & = & [[\,]] \\ subseqs\ (x : xs) & = & map\ (x :)\ (subseqs\ xs) +\!\!+ subseqs\ xs \end{array}$$

***Jack*:** Theo's generate-and-test program seems a reasonable place to start I would say. Clearly, the way to achieve greater efficiency is to fuse the filtering with the generation of subsequences. Let me abbreviate $filter\ (test\ xs)$ to $ft\ xs$. We have

$$ft\ [\,]\ (subseqs\ [\,]) \quad = \quad [[\,]]$$

For the inductive case we can reason:

$$\begin{array}{ll} & ft\ (x : xs)\ (subseqs\ (x : xs)) \\ = & \quad \{\text{definition of } subseqs\} \\ & ft\ (x : xs)\ (map\ (x :)\ (subseqs\ xs) +\!\!+ subseqs\ xs) \\ = & \quad \{\text{since } filter \text{ distributes over } +\!\!+\} \\ & ft\ (x : xs)\ (map\ (x :)\ (subseqs\ xs)) +\!\!+ ft\ (x : xs)\ (subseqs\ xs) \end{array}$$

What next?

***Anne*:** We have to simplify $test\ (x : xs)\ (x : ys)$ and $test\ (x : xs)\ ys$ when ys is a subsequence of xs. We can assume that x is not in xs since the list of people at

https://doi.org/10.1017/S0956796805005678 Published online by Cambridge University Press

the party is presumably given without duplicates. So, x is not in ys either. Clearly, $test\ (x\ :xs)\ ys$ holds just in the case that ys is a clique of xs, nobody in ys knows x, and x knows everyone in the clique ys. In symbols,

$$test\ (x\ :xs)\ ys \quad = \quad nonceleb\ x\ ys \wedge test\ xs\ ys$$

where

$$nonceleb\ x\ ys \quad = \quad and\ [knows\ x\ y \wedge not\ (knows\ y\ x) \mid y \leftarrow ys]$$

Now we can reason:

$$\begin{array}{ll} & ft\ (x\ :xs)\ (subseqs\ xs) \\ = & \quad \{\text{expanding abbreviation } ft\} \\ & filter\ (test\ (x\ :xs))\ (subseqs\ xs) \\ = & \quad \{\text{since } filter\ (p \wedge q) = filter\ p \cdot filter\ q\} \\ & filter\ (nonceleb\ x)\ (filter\ (test\ xs)\ (subseqs\ xs)) \\ = & \quad \{\text{re-introducing abbreviation } ft\} \\ & filter\ (nonceleb\ x)\ (ft\ (subseqs\ xs)) \end{array}$$

Secondly, $test\ (x\ :xs)\ (x\ :ys)$ holds just in the case that ys is a clique of xs and x is a new celebrity, meaning that everyone knows x and x knows all and only members of ys. In symbols,

$$test\ (x\ :xs)\ (x\ :ys) \quad = \quad celeb\ x\ xs\ ys \wedge test\ xs\ ys$$

where

$$celeb\ x\ xs\ ys \quad = \quad and\ [knows\ x'\ x \wedge (knows\ x\ x' \equiv elem\ x'\ ys) \mid x' \leftarrow xs]$$

A similar calculation to the one above now gives

$$\begin{array}{ll} & ft\ (x\ :xs)\ (map\ (x\ :)\ (subseqs\ xs) \\ = & \quad map\ (x\ :)\ (filter\ (celeb\ x\ xs)\ (ft\ (subseqs\ xs))) \end{array}$$

Summarising, $find = head \cdot solns$, where

$$\begin{array}{lll} solns\ [\,] & = & [[\,]] \\ solns\ (x\ :xs) & = & map\ (x\ :)\ (filter\ (celeb\ x\ xs))\ yss) +\!\!+ filter\ (nonceleb\ x)\ yss) \\ & & \textbf{where}\ yss = solns\ xs \end{array}$$

The predicates *celeb* and *nonceleb* can be evaluated in linear time and, as *solns* returns at most two lists, a proper clique and an empty one, we have reduced an exponential algorithm to a quadratic one.

***Theo*:** Well, you can't do better than a quadratic algorithm. Suppose there was a sub-quadratic one, so at least one entry in the *knows* matrix is not inspected. Suppose furthermore that all entries are true, so everyone knows everyone else and the clique is the whole party. Now change the non-inspected entry, $knows\ x\ y$ say, to false. Then y is no longer a celebrity, that is, a member of the celebrity clique. But everyone apart from x still knows y so they can't be celebrities. That leaves x

https://doi.org/10.1017/S0956796805005678 Published online by Cambridge University Press

as the only possible celebrity, but unless x and y are the only people at the party, there is some non-celebrity that x knows, so x isn't a celebrity either. That means there is no celebrity clique at the modified party, and the sub-quadratic algorithm returns the wrong answer. So, in the worst case, every element of the matrix has to be inspected to arrive at the correct answer.

***Teacher*:** True, but the problem was not to determine whether or not there was a celebrity clique. In your scenario, Theo, the answer xs will suffice for both cases: in the first case it is the correct answer, and in the second case there is no celebrity clique, so any answer will do.

There is a pause while the class digests this information.

***Mary*:** I have an idea. Anne's reasoning shows in effect that

$$test\ (x : xs)\ ys \quad \Rightarrow \quad test\ xs\ (after\ x\ ys) \tag{1}$$

where *after* is defined by

$$\begin{array}{lcl} after\ x\ [\,] & = & [\,] \\ after\ x\ (y : ys) & = & \textbf{if}\ x = y\ \textbf{then}\ ys\ \textbf{else}\ y : ys \end{array}$$

Doesn't this give us another way of solving the problem? I mean, suppose $ys = find\ xs$ and consider the value of $find\ (x : xs)$. Firstly, if $ys = [\,]$, so xs does not contain a clique, then by (1) the only possible nonempty clique of $x : xs$ is $[x]$. So

$$find\ xs = [\,] \wedge find\ (x : xs) \neq [\,] \quad \Rightarrow \quad find\ (x : xs) = [x]$$

Secondly, suppose $ys = find\ xs$ and ys is not empty. Let y be some element of ys, say the first. If x and y know each other and $x : xs$ contains a nonempty clique, then (1) shows that it can only be $x : ys$. So we conclude

$$\begin{array}{ll} & find\ xs = y : ys \wedge knows\ x\ y \wedge knows\ y\ x \wedge find\ (x : xs) \neq [\,] \\ \Rightarrow & find\ (x : xs) = x : y : ys \end{array}$$

***Jack*:** Sorry, that was a bit fast for me. By (1) if $find\ xs = y : ys$ and $x : xs$ has a nonempty clique zs, then $after\ x\ zs = y : ys$. We know that because xs has only one nonempty clique, namely $y : ys$. If $zs = [x]$, then $after\ x\ zs = [\,]$, a contradiction. The only remaining possibility is $zs = x : y : ys$. Yes, Mary's reasoning is correct.

***Mary*:** The other cases are handled similarly. Again, if $find\ xs = y : ys$ and if x knows y but y does not know x, then the only possible nonempty clique of $x : xs$ is $y : ys$. Hence

$$\begin{array}{ll} & find\ xs = y : ys \wedge knows\ x\ y \wedge not\ (knows\ y\ x) \wedge find\ (x : xs) \neq [\,] \\ \Rightarrow & find\ (x : xs) = y : ys \end{array}$$

If x does not know y, then no element of ys can be in a clique of $x : xs$ because every such element knows y and y isn't a celebrity. So if $x : xs$ has a nonempty

https://doi.org/10.1017/S0956796805005678 Published online by Cambridge University Press

clique, it can only be $[x]$. In symbols,

$$\begin{array}{ll} & \mathit{find}\ xs = y : ys \wedge \mathit{not}\ (\mathit{knows}\ x\ y) \wedge \mathit{find}\ (x : xs) \neq [\,] \\ \Rightarrow & \mathit{find}\ (x : xs) = [x] \end{array}$$

These four cases are exhaustive.

***Theo*:** While I agree that your reasoning is correct, Mary, I don't see how it leads to a solution. All you have shown is that if we know the value of *find xs* and if $x : xs$ has a nonempty clique, then we can quickly determine it. But how do we know the value of *find xs* in the first place? You seem to be suggesting that if we define *soln* by

$$\begin{array}{lcl} \mathit{soln} & = & \mathit{foldr}\ \mathit{op}\ [\,] \\ \mathit{op}\ x\ [\,] & = & [x] \\ \mathit{op}\ x\ (y : ys) & = & \textbf{if}\ \mathit{knows}\ x\ y\ \textbf{then} \\ & & \quad \textbf{if}\ \mathit{knows}\ y\ x\ \textbf{then}\ x : y : ys\ \textbf{else}\ y : ys \\ & & \textbf{else}\ [x] \end{array}$$

then

$$\mathit{find}\ xs \neq [\,] \quad \Rightarrow \quad \mathit{find}\ xs = \mathit{soln}\ xs \tag{2}$$

But I don't see how your reasoning proves (2).

***Mary*:** Let me try again then, proving (2) by induction on xs. I think I want three cases.
Case $[\,]$: We have $\mathit{find}\ [\,] = [\,] = \mathit{soln}\ [\,]$, establishing the case.
Case $[x]$: Here we have

$$\mathit{find}\ [x] \neq [\,] \quad \Rightarrow \quad \mathit{find}\ [x] = [x] = \mathit{soln}\ [x]$$

establishing the case.
Case $x : xs$ *where* $xs \neq [\,]$: Let $\mathit{soln}\ xs = y : ys$. By induction, if $\mathit{find}\ xs \neq [\,]$, then $\mathit{find}\ xs = y : ys$. Assuming $\mathit{find}\ (x : xs) \neq [\,]$ we have

$$\begin{array}{ll} & \mathit{find}\ (x : xs) \\ = & \quad \{\text{my reasoning above, and induction}\} \\ & \mathit{op}\ x\ (y : ys) \\ = & \quad \{\text{definition of } \mathit{soln}\} \\ & \mathit{soln}\ (x : xs) \end{array}$$

The remaining case is when $\mathit{soln}\ xs = y : ys$ and $\mathit{find}\ xs = [\,]$. Once again assuming $\mathit{find}\ (x : xs) \neq [\,]$ we have

$$\begin{array}{ll} & \mathit{find}\ (x : xs) \\ = & \quad \{\text{my reasoning above}\} \\ & [x] \\ = & \quad \{\text{since } \mathit{knows}\ y\ x \text{ and } \mathit{not}\ (\mathit{knows}\ x\ y)\} \\ & \mathit{op}\ x\ (y : ys) \end{array}$$

https://doi.org/10.1017/S0956796805005678 Published online by Cambridge University Press

$$
\begin{array}{ll}
= & \{\text{definition of } soln\} \\
& soln\,(x : xs)
\end{array}
$$

This establishes the case and the induction.

***Anne*:** That's amazing, a simple linear-time algorithm! But we have only arrived at the solution because of Mary's cleverness. I still want a formal derivation of *soln* from some suitable fusion law.

***Teacher*:** Thank you, Anne, its good to have you in the class.

***Jack*:** It occurs to me that if Theo had taken the other option, and allowed $find\ xs = \bot$ if xs does not contain a nonempty clique, the relationship between *find* and *soln* would have been $find\ xs \sqsubseteq soln\ xs$, where $\sqsubseteq$ is the approximation ordering $x \sqsubseteq y \equiv (x = \bot \vee x = y)$.

***Anne*:** We can write *subseqs* using *foldr*:

$$
\begin{array}{lcl}
subseqs & = & foldr\ add\ [[\,]] \\
add\ x\ xss & = & map\ (x :)\ xss +\!\!+ xss
\end{array}
$$

So it appears that we are appealing to some fusion law of *foldr*. The textbook statement of the fusion law for *foldr* –see, for example, (Bird, 1998)– says that $f \cdot foldr\ g\ a = foldr\ h\ b$ provided that f is strict, $f\ a = b$, and $f\ (g\ x\ y) = h\ x\ (f\ y)$ for all x and y. The strictness condition is not needed if we want only to assert that $f\ (foldr\ g\ a\ xs) = foldr\ h\ b\ xs$ for all finite lists xs. This fusion rule does not apply directly to the clique problem, namely $filter\ (clique\ xs)\ (subseqs\ xs)$ firstly because $filter\ (clique\ xs)$ has xs as a parameter, and secondly because we want something more general than the equality of both sides.

***Theo*:** The first restriction is not really a problem. We can define a version of *subseqs* that returns both the subsequences of a list and the list itself. Suppose we define

$$
\begin{array}{lcl}
subseqs2 & = & foldr\ step\ ([[\,]], [\,]) \\
step\ x\ (xss, xs) & = & (map\ (x :)\ xss +\!\!+ xss, x : xs)
\end{array}
$$

Then $find = f \cdot subseqs2$, where $f\ (xss, xs) = head\ [ys \mid ys \leftarrow xss, test\ xs\ ys]$. In this way we can avoid the additional parameter.

***Anne*:** The second restriction isn't too serious either: interpreting $\sqsubseteq$ as the approximation ordering, we have a more general statement of the fusion law, namely $f \cdot foldr\ g\ a \sqsubseteq foldr\ h\ b$ if f is strict, $f\ a \sqsubseteq b$ and $f\ (g\ x\ y) \sqsubseteq h\ x\ (f\ y)$ for all x and y.

***Jack*:** Yes, but you also need $y \sqsubseteq z \Rightarrow h\ x\ y \sqsubseteq h\ x\ z$, which is trivial when $\sqsubseteq$ is the approximation ordering. When $\sqsubseteq$ is the ordering $xs \sqsubseteq ys \equiv xs = [\,] \vee xs = ys$, we need $h\ x\ [\,] = [\,]$ or $h\ x\ [\,] = h\ x\ zs$ for any zs. Neither condition holds when $h = op$, as can be seen directly from the definition of op. We definitely want the approximation ordering here, so I think Theo chose the wrong option.

https://doi.org/10.1017/S0956796805005678 Published online by Cambridge University Press

***Mary*:** Your more general statement of the fusion law won't work, Anne. Applied to the clique problem it requires us to prove

$$find\ (x\ :xs) \quad \sqsubseteq \quad op\ x\ (find\ xs)$$

Suppose $find\ (x\ :xs) = [x]$, so xs does not have a clique and $find\ xs = \bot$. We require $[x] = op\ x\ \bot$, which implies $op\ x\ ys = [x]$ for all ys. This isn't the case, as can be seen from the definition of op.

***Theo*:** That is because the textbook statement of the fusion rule, or the generalisation proposed by Anne is still not general enough. Let $\trianglelefteq$ be some relation on values, I don't care what. Then it is easy to show by induction that

$$f\ (foldr\ g\ a\ xs) \trianglelefteq foldr\ h\ b\ xs$$

for all finite lists xs provided $f\ a \trianglelefteq b$ and $f\ y \trianglelefteq z \Rightarrow f\ (g\ x\ y) \trianglelefteq h\ x\ z$ for all x, y and z. It works both when $\trianglelefteq$ is intepreted as the approximation ordering and when $\trianglelefteq$ is interpreted as the relation $xs \trianglelefteq ys \equiv xs = [\,] \vee xs = ys$ that Jack defined. The last condition above is the one we need for fusion to be established. In the celebrity clique problem, and choosing Jack's ordering, the condition we want is

$$\begin{array}{l} f\ (xss, xs) \neq [\,] \wedge f\ (step\ x\ (xss, xs)) \neq [\,] \\ \quad \Rightarrow \quad f\ (step\ x\ (xss, xs)) = op\ x\ (f\ (xss, xs)) \end{array}$$

Mary's reasoning establishes this property.

***Teacher*:** Yes. The more general statement of fusion is the one provided by parametricity in Wadler's *Theorems For Free!* paper (Wadler, 1989) It is nice to see an example where the more general statement is needed. What is interesting about the problem is that it is the first example I have seen in which it is asymptotically more efficient to find a solution assuming one exists than to check that it actually is a solution. A similar problem is the *majority voting* problem –see, for example, (Morgan, 1994), Chapter 18– in which one is given a list xs and it is required to determine whether there is a value in xs that occurs strictly greater than $\lfloor length\ xs/2 \rfloor$ times. It is easier to first compute a putative majority, assuming one exists, and then check whether it is actually a majority afterwards. But checking for a majority takes linear time rather than quadratic time, so there is no asymptotic gap.

***Afterword*:** The true story of the celebrity clique problem was as follows. Richard Bird was giving a course of lectures on Formal Program Design (in a procedural rather than functional framework) and thought of the problem as a generalisation of the one in Kaldewaij's book (Kaldewaij, 1990). But despite a day of effort of reasoning about loop invariants he couldn't produce a sufficiently simple solution, so he set it as a challenge to the class. He also talked about it at a research meeting the following Friday. Over the weekend, Sharon Curtis produced a simple linear-time algorithm, though the reasoning was still somwehat complicated. Meanwhile, Julian Tibble, a second-year undergraduate, developed essentially the same solution.

https://doi.org/10.1017/S0956796805005678 Published online by Cambridge University Press

In the belief that whatever can be reasoned about with loops and invariants can be reasoned about at least as easily using the laws of functional program derivation, the problem was translated into a functional setting and the dialogue above was composed. Afterwards, the problem was tried out at a WG2.1 meeting in Nottingham in September, 2004. Gratifyingly, the actual discussion followed the early part of the dialogue quite closely. On repeatedly being urged to try harder, Andres Löh and Johan Jeuring came up a day later with the linear-time solution.

References

Bird, R. (1998) *Introduction to Functional Programming using Haskell*. International Series in Computer Science, Prentice Hall.

Kaldewaij, A. (1990) *Programming: The Derivation of Algorithms*. International Series in Computer Science, Prentice Hall.

Morgan, C. (1994) *Programming from Specifiactions* (2nd edition). International Series in Computer Science, Prentice Hall.

Wadler, P. (1989) Theorems For Free! *Fourth International Symposium on Functional Programming Languages and Computer Architecture*, pp. 347–359. ACM Press.