JFP **25**, e4, 9 pages, 2015. © Cambridge University Press 2015
doi:10.1017/S0956796815000064

FUNCTIONAL PEARL

How to mingle streams

RICHARD S. BIRD

Department of Computer Science, Oxford University, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK

Where it shall mingle with the state of floods,
And flow henceforth in formal majesty.
Shakespeare's Henry IV, part 2

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 consider streams and in particular the problem of designing a function `mingle` of type

```
mingle :: Stream (Stream a) -> Stream a
```

for mingling a stream of streams together as a single stream. For simplicity you can think of `Stream` as a type synonym

```
type Stream a = [a]
```

except that by a stream we always mean an infinite list, not a finite or partial one.

Jack: Since streams are lists, cannot we just take `mingle` to be the standard list function `concat :: [[a]] -> [a]`?

Mary: Well, that's not really mingling. Applying `concat` to a stream of streams just returns the first stream. The other streams never get a look in. Formally, if `xs` is an infinite list, then `concat (xs:xss) = xs`.

Theo: You can concatenate a stream of finite lists though. That is,

```
concat :: Stream [a] -> Stream a
```

is a perfectly valid function for mingling all the (finite) lists in a stream. If we could design a function with type

```
Stream (Stream a) -> Stream [a]
```

that does not ignore any element of any stream, or duplicate them, then we could just concatenate the finite lists.

https://doi.org/10.1017/S0956796815000064 Published online by Cambridge University Press

Anne: Yes, that seems a sensible idea. In particular, we could construct the diagonals as a stream of finite lists. For example, the diagonals of

```
11 12 13 14 15 ...
21 22 23 24 25 ...
31 32 33 34 35 ...
41 42 43 44 45 ...
...
```

are the lists `[11]`, `[12,21]`, `[13,22,31]`, `[14,23,32,41]` and so on. Concatenating the diagonals gives us a genuinely fair mingling: each original stream appears as a substream of the result. So, if we can construct `diags`, then

```
mingle = concat . diags
```

Mary: There is another way of mingling but it does not give the same result. The Haskell library `Data.Stream` contains the function `interleave`:

```
interleave :: Stream a -> Stream a -> Stream a
interleave (x:xs) ys = x:interleave ys xs
```

This function interleaves elements of the first stream with elements of the second. Now we can define

```
mingle = foldr1 interleave
```

That also gives a fair mingling in that every stream appears as a substream of the result. But it is not quite as fair as Anne's diagonal method because half of the elements in the mingled stream come from the first stream, a quarter from the second stream, and so on. The nth stream does not get a look in until position 2^n. With Anne's approach the delay is quadratic rather than exponential.

Theo: I don't like the use of `foldr1` in your definition, Mary. Better is a more general version of fold tailored to streams:

```
fold :: (a -> b -> b) -> Stream a -> b
fold f (x:xs) = f x (fold f xs)
```

Of course, the function `f` had better be nonstrict in its second argument for this definition to produce any defined result. The function `interleave` has this property, so `fold interleave` is fine.

But `Stream` is really a co-datatype rather than a datatype, so the basic pattern for computations on streams is unfolding, not folding:

```
unfold :: (b -> (a,b)) -> b -> Stream a
unfold f y = x:unfold f y'  where (x,y') = f y
```

I imagine this might come in useful.

https://doi.org/10.1017/S0956796815000064 Published online by Cambridge University Press

Anne: Let's get back to diagonals. One way to compute them is to divide the streams into two: a finite prefix and an infinite suffix. The first elements of the finite prefix, which grows by one stream at each step, give the next diagonal. We can define

```
diags :: Stream (Stream a) -> Stream [a]
diags (xs:xss)    = crop [xs] xss
crop xss (ys:yss) = map head xss:crop (map tail xss ++ [ys]) yss
```

You see? At each step the heads of the finite list of streams are output, and a new stream is included to continue the computation. Using Theo's unfold we can write

```
diags (xs:xss) = unfold f ([xs],xss)
 where f (xss,ys:yss) =
                (map head xss, (map tail xss ++ [ys],yss))
```

But I don't know if writing it this way this adds anything.

Teacher: Well done Anne. You have rediscovered a method for mingling that goes back to the 1980s, at least. I recall that it was used to inspire functional programmers by demonstrating that many interesting functions can be defined holistically, that is, without recourse to indexing, using nifty applications of `(++)`, `map`, `head` and `tail`.

Can you think of any other way to define `diags`?

Jack: Yes, I think so. Consider the first few diagonals of Anne's infinite matrix, which I will write as a triangle:

```
      11
    12  21
  13  22  31
14  23  32  41
```

Suppose we add a new first row `01 02 03 04 ...` to the matrix. Then the new diagonals are given by

```
      01
    02  11
  03  12  21
04  13  22  31
```

The old diagonals are shunted one place right and down, and the new row becomes the new first elements of the new diagonals. That suggests

```
diags ((x:xs):xss) = [x]:zipWith (:) xs (diags xss)
```

If you prefer, I can also write this in the alternative form

```
diags (xs:xss) = zipWith (:) xs ([]:diags xss)
```

We get the first from the second in one reduction step:

```
diags ((x:xs):xss) = zipWith (:) (x:xs) ([]:diags xss)
                   = (x:[]):zipWith (:) xs (diags xss)
```

Either of these one-liners seem more attractive than Anne's definition.

https://doi.org/10.1017/S0956796815000064 Published online by Cambridge University Press

Teacher: Excellent. I don't think this version was known in the 1980s, probably because `zipWith` was not part of the standard toolbox at that time. But maybe I am wrong.

In a moment I will suggest you try and prove that the two definitions of `diags` are equivalent. But first, can you think of any other way to define the same function `mingle`, one that does not invoke `diags`? Here is a hint: think of how to mingle what is left after taking the very first element.

Mary: The first element of the mingling of Anne's matrix is the element `11`. What is left is a matrix with a hole:

```
   12 13 14 15 ...
21 22 23 24 25 ...
31 32 33 34 35 ...
41 42 43 44 45 ...
...
```

We can't continue by mingling these streams. Ah, but we can continue by mingling the streams

```
12 21 22 23 24 25 ...
13 31 32 33 34 35 ...
14 41 42 43 44 45 ...
15 51 52 53 54 55
...
```

This is the first matrix minus its first row, which now becomes the first column. The diagonals of the new matrix are different than before, but concatenating them, and putting `11` at the front, gives the same stream. In symbols,

```
mingle ((x:xs):xss) = x:mingle (zipWith (:) xs xss)
```

This one-liner for the full mingling process is really cool.

Theo: Not so cool, Mary, because your version of mingling is far less efficient than the other two. The first two versions produce the first n elements in $O(n)$ steps, while yours, pretty as it is, takes $O(n^2)$ steps. It takes n cons operations to produce the nth element.

Teacher: You have all been exceedingly clever in getting to these three algorithms, but now comes the hard part. Can you prove that they all give the same result?

Jack: Do we really have to? Does not Anne's matrix and the two triangles say it all? The example seems to me to be completely general.

Teacher: No, because it just amounts to a look–and–see argument. You might be able to convince me that three polymorphic functions on streams are identical because they do the same thing on streams of pairs of positive integers, but even so

https://doi.org/10.1017/S0956796815000064 Published online by Cambridge University Press

that does not amount to a finite test. How many outputs do we need to see? There are results like the zero-one principle of sorting networks, which says that a sorting network is correct if it can sort all 2^n sequences of 0s and 1s, but I know of no similar meta-results about streams.

Also, no example can address the question: *why* are two things the same. If I gave you two definitions of a function that produced, say, the digits of π, you might convince me they were equivalent by displaying their results to a suitable number of digits. But how much more satisfactory would be the argument that their equivalence follows from the facts that multiplication is associative and distributes over addition.

Anne: Enough philosophy. I want to prove that the two ways of defining `diags` are the same. Here they are again:

```
diags1 (xs:xss) = crop [xs] xss
crop xss (ys:yss) = map head xss:crop (map tail xss ++ [ys]) yss
diags2 (xs:xss) = zipWith (:) xs ([]:diags2 xss)
```

That means we have to show

```
crop [xs] yss = zipWith (:) xs ([]:diags2 yss)
```

How do we do that?

Theo: I suspect you would have to generalise the claim first. That singleton `[xs]` as the first argument bothers me. I would like to see a claim involving `crop xss yss` for an arbitrary finite list `xss` of streams.

Mary: Yes, and I can see what the generalisation is. I claim that `crop xss yss` concatenates the columns of `xss` with the diagonals of `yss` prefixed with an empty diagonal. In symbols,

```
crop xss yss = zipWith (++) (cols xss) ([]:diags2 yss)
```

The function `cols` transposes a matrix, more precisely a $m \times \infty$ matrix. Here is the definition:

```
cols :: [Stream a] -> Stream [a]
cols xss = map head xss:cols (map tail xss)
```

In particular, since `cols [xs] = [[x] | x <- xs]` we have

```
zipWith (++) (cols [xs]) xss = zipWith (:) xs xss
```

and so we get our desired result, namely

```
crop [xs] xss = zipWith (:) xs ([]:diags2 xss)
```

as a special case. I knew matrix transposition would come in somewhere!

https://doi.org/10.1017/S0956796815000064 Published online by Cambridge University Press

Theo: Your definition of `cols` is not the standard definition of matrix transposition because it is specific to $m \times \infty$ matrices. But there are nice consequences. In particular,

```
cols (xss ++ yss) = cols xss <++> cols yss
```

where I have taken the liberty of writing `zipWith (++)` as an infix operator `<++>`. And the reason I did that is because `<++>` is an associative operation, and we have been taught that such operations should usually be described using infix operators. That is a small but not unimportant consequence of thinking about the algebraic properties of functions we define.

Anne: Let me summarise. Given

```
crop xss (ys:yss) = map head xss:crop (map tail xss ++ [ys]) yss
diags (xs:xss)    = zipWith (:) xs ([]:diags xss)
cols xss          = map head xss:cols (map tail xss)
```

we have to show that

```
crop xss yss = cols xss <++> ([]:diags yss)
```

Can we do this by induction, proving it true for `ys:yss` assuming it's true for `yss`?

Theo: No, not without a base case for the induction. Consider the assertion that the length of `yss` is finite. The induction step obviously goes through, but the conclusion is false: no stream has finite length. The missing case is when `yss` is the undefined list. If that can be established, and it cannot for the finiteness assertion, then you can conclude that the assertion holds for all partial lists. And because our assertion takes the form of an equation, you can conclude that it holds for all infinite lists too.

Anne: I don't think the base case causes any problems. Take `yss = undefined` in

```
crop xss yss = cols xss <++> ([]:diags yss)
```

and use the fact that `diags undefined = undefined`. The right-hand side gives

```
  cols xss <++> ([]:undefined)
= (map head xss:cols (map tail xss)) <++> ([]:undefined)
= map head xss ++ []:(cols (map tail xss) <++> undefined)
= map head xss:(cols (map tail xss) <++> undefined)
= map head xss:undefined
```

The last step follows because `zipWith` uses pattern matching on its last two arguments, so is strict in its third argument.

The left-hand side gives the same result. Oh, no, it does not! The function `crop` is defined by pattern matching on its second argument, so

```
crop xss undefined = undefined
```

Worse, I cannot redefine `crop` to use an irrefutable pattern, as in

https://doi.org/10.1017/S0956796815000064 Published online by Cambridge University Press

```
crop xss ~(ys:yss) =
     map head xss:crop (map tail xss ++ [ys]) yss
```

because under this definition

```
crop xss undefined =
     map head xss: map (head . tail) xss ++ undefined
```

That is not the same as the value for the right-hand side. Maybe Mary is wrong and her claim is false.

Theo: No, it is only that the claim is not true for partial lists. The problem is that you are trying to prove too much. Induction on `yss` is not the right way to prove the claim. Instead you have to use coinduction.

Jack: Drat. I have never really understood coinduction. It seems to involve something called bisimularity and a lot of algebraic machinery.

Theo: It is not as bad as you think, Jack, at least when the task is to prove equations on streams. It comes down to this: in order to prove `f x1 ... xn = g x1 ... xn` it is sufficient to prove

```
f x1 ... xn = e:f a1 ... an
g x1 ... xn = e:g a1 ... an
```

where `e, a1 ... an` are values that depend on `x1 ... xn`. In other words, we have to show that both `f` and `g` satisfy an equation of the form

```
h x1 ... xn = e:h a1 ... an
```

This proof technique is nicely explained in (Turner, 2004).

Anne: OK, let me try it. Our coinduction hypothesis is

```
crop xss yss = cols xss <++> ([]:diags yss)
```

By definition of `crop` the left-hand side is equal to

```
map head xss:crop (map tail xss ++ [head ys]) (tail yss)
```

By definition of `cols` the right-hand side is equal to

```
(map head xss:cols (map tail xss)) <++> ([]:diags yss)
```

By definition of `<++>` this simplifies to

```
map head xss:cols (map tail xss)) <++> diags yss
```

So we are left with showing

```
crop (map tail xss ++ [head yss]) (tail yss)
  = cols (map tail xss)) <++> diags yss
```

Using `zipWith (:) xs xss = cols [xs] <++> xss` we have

https://doi.org/10.1017/S0956796815000064 Published online by Cambridge University Press

```
diags yss = zipWith (:) (head yss) ([]:diags (tail yss))
          = cols [head yss] <++> ([]:diags (tail yss))
```

So we have to prove

```
crop (map tail xss ++ [head yss]) (tail yss)
 = cols (map tail xss) <++>
   cols [head yss] <++> ([]:diags (tail yss))
```

We are there! By Theo's identity we have

```
cols (map tail xss) <++> cols [ys] = cols (map tail xss ++ [ys])
```

And the result follows by the coinduction hypothesis. It all comes down really to the associativity of `<++>` and Theo's identity about the distributivity of `cols` over concatenation.

Teacher: Well done. It is not uncommon that equational proofs involve the associativity of one operation and the fact that it distributes over another.

You still have to prove that `mingle = concat . diags`, where

```
mingle ((x:xs):xss) = x:mingle (zipWith (:) xs xss)
```

Jack: Let me try that one. It is sufficient to show that

```
(concat . diags) ((x:xs):xss)
 = x:(concat . diags) (zipWith (:) xs xss)
```

We work on the left-hand side:

```
 (concat . diags) ((x:xs):xss)
=  {definition of diags}
 concat (zipWith (:) (x:xs) ([]:diags xss))
=  {definition of zipWith}
 concat ([x]:zipWith (:) xs (diags xss))
=  {definition of concat}
 x:concat (zipWith (:) xs (diags xss))
```

Thus we have to show

```
concat (zipWith (:) xs (diags xss))
 = concat (diags (zipWith (:) xs xss))
```

I can sort of see how to rewrite the left-hand side, which is to use

```
concat (zipWith (:) xs yss) = concat (zipWith snoc xs ([]:yss))
```

where `snoc x xs = xs ++ [x]`. And my proof of that identity is

```
([x1] ++ ys1) ++ ([x2] ++ ys2) ++ ([x3] ++ ys3) ++ ... =
 [x1] ++ (ys1 ++ [x2]) ++ (ys2 ++ [x3]) ++ ...
```

Of course, now I have written that I appreciate that

```
concat ((xs:xss) <++> yss) = xs ++ concat (yss <++> xss)
```

is a more general fact. You see, I have learnt how to generalise.

https://doi.org/10.1017/S0956796815000064 Published online by Cambridge University Press

Theo: Go the extra step, Jack, and generalise even more. Your claim is an instance of

```
fold (*) (x:xs <*> ys) = x * fold (*) (ys <*> xs)
```

for any associative operator `(*)`, where `<*> = zipWith (*)`. Here, `xs` and `ys` have to be streams.

Jack: Whatever. I am still going to leave it as a look–and–see argument. My second claim is that

```
diags (zipwith (:) xs xss) = zipwith snoc xs ([]:diags xss)
```

And my proof of this is another look–and–see argument:

```
01 11 21 31 41 51              01
02 12 22 32 42 52           11  02
03 13 23 33 43 53        21  12  03
04 14 24 34 44 54     31  22  13  04
05 15 25 35 45 55  41  32  23  14  05
```

Now I am done because, putting my claims together, we have

```
  concat (zipWith (:) xs (diags xss))
=  {first claim}
  concat (zipWith snoc xs ([]:diags xss))
=  {second claim}
= concat (diags (zipWith (:) xs xss))
```

I have split the claims into two because the first depends on `concat` but not `diags`, while the second depends on `diags` but not on `concat`. I suppose, dear teacher, that you want formal proofs of my claims.

Teacher: Ideally, yes, but we have gone on for long enough and I do not want you all to drown in the streams. So let us leave them for another time.

Afterword

The real story behind this pearl was that Jeremy Gibbons knew about Anne's original definition of `diags` and had recently heard about the one-liner for `mingle` from Shin-Cheng Mu on a trip to Taiwan. He tried without success to prove them equivalent on the plane home (but then he was flying over the Ukraine at the time). He therefore posed it as a problem during a meeting of the Algebra of Programming problem solving club one Friday morning. We did not make as much progress as Anne, Jack, Mary and Theo did, but then they are very clever students.

Finally, thanks to a referee who pointed out David Turner's paper.

Reference

Turner, D. A. (2004) Total functional programming. *J. Universal Comput. Sci.* **10**(7), 751–768.

https://doi.org/10.1017/S0956796815000064 Published online by Cambridge University Press