

Streaming algorithms



Jeremy Gibbons
University of Oxford
APPSEM II, April 2004

1. Origami programming

In a *compact category* (where initial algebras and final coalgebras coincide), *recursive datatype*

$$T = \text{fix } \mathcal{F}$$

induces *morphisms* for common patterns of computation:

$$\text{fold}_{\mathcal{F}} \quad :: \quad (\mathcal{F} A \rightarrow A) \rightarrow (T \rightarrow A)$$

$$\text{unfold}_{\mathcal{F}} \quad :: \quad (A \rightarrow \mathcal{F} A) \rightarrow (A \rightarrow T)$$

These compose to form *hylomorphisms*:

$$\text{hylo}_{\mathcal{F}} (f, g) = \text{fold}_{\mathcal{F}} f \circ \text{unfold}_{\mathcal{F}} g$$

Under certain strictness conditions, these two *fuse* and the intermediate datatype $\text{fix } \mathcal{F}$ may be *deforested*.

2. Metamorphisms

What about the opposite composition?

$$\begin{aligned} \mathit{meta}_{\mathcal{F}, \mathcal{G}} &:: (A \rightarrow \mathcal{F} A, \mathcal{G} A \rightarrow A) \rightarrow (\mathit{fix} \mathcal{G} \rightarrow \mathit{fix} \mathcal{F}) \\ \mathit{meta}_{\mathcal{F}, \mathcal{G}} (f, g) &= \mathit{unfold}_{\mathcal{F}} f \circ \mathit{fold}_{\mathcal{G}} g \end{aligned}$$

This pattern captures many *changes of representation*.

$$\begin{aligned} \mathit{regroup} \ n &= \mathit{group} \ n \circ \mathit{concat} \\ \mathit{heapsort} &= \mathit{flattenHeap} \circ \mathit{buildHeap} \\ \mathit{baseConv} \ (b, c) &= \mathit{toBase} \ b \circ \mathit{fromBase} \ c \\ \mathit{arithCode} &= \mathit{toBits} \circ \mathit{narrow} \end{aligned}$$

3. Streaming

In general, metamorphisms are less interesting than hylomorphisms: there is no analogue of deforestation.

However, under certain conditions, there is a kind of fusion. Some of the work of the unfold can be done before all of the work of the fold is complete.

We call this *streaming*. It allows *infinite* representations to be processed.

4. Streaming for lists

Recall from Haskell libraries:

```
> foldl :: (b -> a -> b) -> b -> [a] -> b
> unfoldr :: (b -> Maybe (c,b)) -> b -> [c]
```

Define

```
> stream :: (b->Maybe (c,b)) -> (b->a->b) -> b -> [a] -> [c]
> stream f g b as =
>   case f b of
>     Just (c, b') -> c : stream f g b' as
>     Nothing      ->
>       case as of
>         (a:as') -> stream f g (g b a) as'
>         []      -> []
```

4.1. Streaming Theorem (Bird and Gibbons, 2003)

The *streaming condition* for f and g is that whenever

$$f\ b = \text{Just}\ (c, b')$$

then, for any a ,

$$f\ (g\ b\ a) = \text{Just}\ (c, g\ b'\ a)$$

It's a kind of invariant property.

Theorem: if the *streaming condition* holds for f and g , then

$$\text{stream}\ f\ g\ b\ as = \text{unfoldr}\ f\ (\text{foldl}\ g\ b\ as)$$

for all *finite* lists as .

4.2. Example of streaming

First, a simple example. The streaming condition holds for `unCons` and `snoc`, where

```
> unCons [] = Nothing
> unCons (x:xs) = Just (x, xs)

> snoc xs x = xs ++ [x]
```

Therefore the two-stage copying process

```
unfoldr unCons . foldl snoc []
```

agrees with the one-stage process

```
stream unCons snoc []
```

on finite lists (but not infinite ones!).

4.3. Flushing streams

More generally, a streaming process will switch into a *flushing* state when the input is exhausted.

```
> fstream :: (b->Maybe (c,b)) -> (b->a->b) -> (b->[c]) ->
>          b -> [a] -> [c]
> fstream f g h b as =
>   case f b of
>     Just (c, b') -> c : fstream f g h b' as
>     Nothing      ->
>       case as of
>         (a:as') -> fstream f g h (g b a) as'
>         []      -> h b
```


4.4. Flushing Streams Theorem

Vene and Uustalu's *apomorphism*:

```
> apo :: (b -> Either (c,b) [c]) -> b -> [c]
> apo f b = case f b of
>     Left (c,b') -> c : apo f b'
>     Right cs    -> cs
```

Theorem: if the *streaming condition* holds for f and g , then

$$\text{fstream } f \ g \ h \ b \ as = \text{apo } (\text{alt } f \ h) \ (\text{foldl } g \ b \ as)$$

for all *finite* lists as , where

```
> alt f h b = case f b of Just (c,b') -> Left (c,b')
>                Nothing    -> Right (h b)
```

(Typically, the unfold part has to be somewhat cautious, delaying an output that might be invalidated later. With no input remaining, it can become more aggressive.)

5. Generic streaming?

- restricted to fold over lists
- that fold must be a `foldl`
- perhaps those constraints are connected: don't know how to do a generic version of `foldl`
- I have given a generic `scanl`
(improved by Alberto Pardo at WCGP 2002)
- the unfold could be generalized; then a generic invariant property would be involved
- other applications?

6. Example of flushing

Consider converting a fraction from base m to base n .

```
> fromBase m = foldr (stepr m) 0
>   where stepr m d x = (d+x)/m

> toBase n = unfoldr (split n)
>   where split n 0 = Nothing
>           split n x = Just (floor y, y - floor y)
>           where y=n*x
```

(coercions between numeric types omitted for brevity).

Of course, this only works for finite input (because `stepr m d x` is strict in `x`). The result will be finite iff the value is finitely representable in base n .

6.1. Invert order of input

The fold is of the wrong kind; refactor to

```
> fromBase m = extract . foldl (step1 m) (0,1)
>   where step1 m (u,v) d = (d+u*m,v/m)
>   extract (u,v) = v*u
```

The state (u, v) here is a defunctionalization of $(v^*) . (u+)$.

6.2. Unfold after a fold

We now have an unfold after an abstraction function after a fold.

Fortunately, the abstraction function fuses with the unfold:

```
> toBase' n = toBase n . extract
>           = unfoldr (split' n)
>   where split' n (0,v) = Nothing
>           split' n (u,v) = Just (y, (u-y/(v*n), v*n))
>                               where y = floor (n*u*v)
```

6.3. Streaming condition

The streaming condition does not hold for `step1 m` and `split' n`.
For example,

```
split' 7 (1, 1/3)           = Just (2, (1/7, 7/3))
split' 7 (step1 3 (1,1/3) 1) = split' 7 (4, 1/9)
                             = Just (3, (1/7, 7/9))
```

(That is, $0.1_3 \approx 0.222222_7$, but $0.11_3 \approx 0.305305_7$.)

We must be more cautious while input remains.

```
> toBase' n = apo (alt (splitS n) (unfoldr (split' n)))
>   where
>     splitS n (u,v)
>       | floor (u*v*n) == floor ((u+1)*v*n) = split' n (u,v)
>       | otherwise                          = Nothing
```

The streaming condition holds for `step1 m` and `splitS n`.

6.4. The complete program

```
> baseConv (n,m) = fstream (splitS n)
>                               (step1 m)
>                               (unfoldr (split' n))
>                               (0,1)
```

This works for finite or infinite input; it produces a finite output iff the value is finitely representable in the output base *and* finitely represented in the input.

Output digits are produced whenever possible (that is, whenever completely determined). Input digits are consumed when output is not possible. The state is flushed if and when the input is exhausted.

7. An application: computing π

Here is one of many elegant series for π :

$$\pi = 2 + \frac{1}{3} \left(2 + \frac{2}{5} \left(2 + \frac{3}{7} \left(2 + \frac{4}{9} \left(2 + \dots \right) \right) \right) \right)$$

Rabinowitz and Wagon use this series as the basis for a *spigot algorithm* for the digits of π .

```
a[52514], b, c=52514, d, e, f=1e4, g, h;
main(){for(;b=c-=14;h=printf("%04d", e+d/f))
for(e=d%=f;g=--b*2;d/=g)
d=d*b+f*(h?a[b]:f/5), a[b]=d%--g;}
```

(this version due to Dik Winter and Achim Flammenkamp)

7.1. Linear fractional transformations

The series above can be seen as an infinite composition of *linear fractional transformations*:

$$\pi = \left(2 + \frac{1}{3} \times\right) \left(2 + \frac{2}{5} \times\right) \left(2 + \frac{3}{7} \times\right) \cdots \left(2 + \frac{i}{2i+1} \times\right) \cdots$$

(Each such LFT is a contraction on the interval $(3, 4)$; the value represented is the limit of the intersections of compositions of finite prefixes of this infinite composition.)

The decimal representation of π is another such composition:

$$\pi = \left(3 + \frac{1}{10} \times\right) \left(1 + \frac{1}{10} \times\right) \left(4 + \frac{1}{10} \times\right) \left(1 + \frac{1}{10} \times\right) \cdots$$

(contractions on $[0, 10]$) in which there is no regular pattern in the terms.

Computing the digits of π is therefore a matter of converting from the one representation to the other: a *metamorphism*.

7.2. Representing LFTs

The general form of a LFT is to take x to $(qx + r)/(sx + t)$.

It can be represented as a two-by-two matrix

$$\begin{pmatrix} q & r \\ s & t \end{pmatrix}$$

Then composition of transformations corresponds to matrix multiplication.

As x ranges from 0 to ∞ , the transformation of x ranges between $\frac{r}{t}$ and $\frac{q}{s}$ (provided that s and t have the same sign).

In fact, any tail of our infinite composition represents a value in the interval $(3, 4)$:

$$\begin{aligned}
 & 3 \\
 = & x \text{ where } x = 2 + \frac{1}{3}x \\
 = & 2 + \frac{1}{3} \left(2 + \frac{1}{3} \left(2 + \dots \right) \right) \\
 < & 2 + \frac{i}{2i+1} \left(2 + \frac{i+1}{2i+3} \left(2 + \dots \right) \right) \\
 < & 2 + \frac{1}{2} \left(2 + \frac{1}{2} \left(2 + \dots \right) \right) \\
 = & x \text{ where } x = 2 + \frac{1}{2}x \\
 = & 4
 \end{aligned}$$

7.3. Streaming π

The streaming process maintains a LFT as state.

The invariant is that the composition of the LFTs produced with the LFT as state equals the composition of the LFTs consumed (or equivalently...).

If the state LFT $\begin{pmatrix} q & r \\ s & t \end{pmatrix}$ completely determines the next digit (that is, if $(3q + r)/(3s + t)$ and $(4q + r)/(4s + t)$ have the same integer part), that term can be produced; otherwise, another term must be consumed.

Since the input is infinite, flushing streams are not needed.

7.4. Complete program

```

> piStream = stream f mm init lfts where
>   init = (1,0,0,1)
>   lfts = [(k, 4*k+2, 0, 2*k+1) | k<-[1..]]
>   f qrst | floor (ext qrst 4) /= n = Nothing
>           | otherwise = Just (n, mm (10, -10*n, 0, 1) qrst)
>   where n = floor (ext qrst 3)
>   ext (q,r,s,t) x = (q*x+r) / (s*x+t)
>   mm (q,r,s,t) (u,v,w,x) = (q*u+r*w, q*v+r*x, s*u+t*w, s*v+t*x)

```

This can be compressed (obfuscated) to:

```

> pi = g(1,0,1,1,3,3) where
>   g(q,r,t,k,n,l) = if 4*q+r-t<n*t
>     then n:g(10*q,10*(r-n*t),t,k,div(10*(3*q+r))t-10*n,l)
>     else g(q*k,(2*q+r)*l,t*l,k+1,div(q*(7*k+2)+r*l)(t*l),l+2)

```