Streaming algorithms

Jeremy Gibbons
University of Oxford
Refactoring Workshop
February 2004

1. Origami programming

In a setting in which initial algebras and final coalgebras coincide (such as Haskell), *recursive datatype*

$$T \quad = \quad \mathsf{fix}\,\mathcal{F}$$

induces *morphisms*

$$\begin{aligned} \mathsf{fold}_{\mathcal{F}} \quad &:: \quad (\mathcal{F}\,A \to A) \to (T \to A) \\ \mathsf{unfold}_{\mathcal{F}} \quad &:: \quad (A \to \mathcal{F}\,A) \to (A \to T) \end{aligned}$$

These compose to form *hylomorphisms*:

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

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

2. Metamorphisms

What about the opposite composition?

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

I reckon these capture many *changes of representation*.

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

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 = Just (c,b')
```

then, for any `a`,

```
f (g b a) = Just (c, g b' a)
```

It's a kind of invariant property.

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

```
stream f g b as = unfoldr f (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!).

5. 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
```

5.1. 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

```
fstream f g h b as = apo (alt f h) (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.)

5.2. Example of flushing

Consider converting a fraction from base 3 to base 7.

```
> fromBase3 = foldr stepr 0
>   where stepr d x = (d+x)/3

> toBase7 = unfoldr split
>   where split x = Just (floor y, y - floor y) where y=7*x
```

(coercions between numeric types omitted for brevity).

The result will always be infinite, even if the input represents a finite fraction in base 7. (Of course, there is no way of generating a finite output from an infinite input.)

5.3. Invert order of input

The fold is of the wrong kind; refactor to

```
> fromBase3 = extract . foldl stepl (0,1)
>   where stepl (u,v) d = (d+u*3,v/3)
> extract (u,v) = v*u
```

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

5.4. Unfold after a fold

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

Fortunately, the abstraction function fuses with the unfold:

```
> toBase7' = toBase7 . extract
>          = unfoldr split'
>   where split' (u,v) = Just (y, (u-y/(v*7),v*7))
>                      where y = floor (7*u*v)
```

5.5. Streaming condition

The streaming condition does not hold for `stepl` and `split'`.
For example,

```
split' (1, 1/3)           = Just (2, (1/7, 7/3))
split' (stepl (1,1/3) 1) = split' (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.

```
> toBase7' = apo (alt splitS (unfoldr split'))
>   where
>     splitS (u,v)
>       | floor (u*v*7) == floor ((u+1)*v*7) = split' (u,v)
>       | otherwise                          = Nothing
```

The streaming condition holds for `stepl` and `splitS`.

5.6. The complete program

```
> base3toBase7 = fstream splitS stepl (unfoldr split') (0,1)
```

This works for finite or infinite input; it always produces an infinite output.

Output digits are produced whenever possible (that is, whenever completely determined). Input digits are consumed when output is not possible.

Of course, if the input is infinite, it will never be exhausted, and the flushing phase will never be reached.

6. An application: computing π

Here is one of many elegant series for π:

$$\pi \quad = \quad 2 + \tfrac{1}{3}(2 + \tfrac{2}{5}(2 + \tfrac{3}{7}(2 + \tfrac{4}{9}(2 + \cdots))))$$

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)

6.1. Linear fractional transformations

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

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

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

The decimal representation of π is another such composition:

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

(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*.

6.2. 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;
- the identify transformation corresponds to the unit matrix;
- applying the transformation $\begin{pmatrix} q & r \\ s & t \end{pmatrix}$ to a value x yields $(qx + r)/(sx + t)$.

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 $(3, 4)$:

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

As x ranges from 3 to 4, the transformation of x ranges between $\frac{3q+r}{3s+t}$ and $\frac{4q+r}{4s+t}$ (provided that the denominator does not change sign).

6.3. 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 $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)$ 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.

6.4. 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)
```

7. Generic streaming?

- restricted to fold over lists
- that fold must be a `foldl`
- perhaps those constraints are connected: I don't know how to do a generic version of `foldl`
- I have given a generic `scanl` (improved by Alberto Pardo)
- the unfold could be generalized; then a generic invariant property would be involved (like invariants from CMCS paper)
- other applications: Hughes' and Swierstra's online parsers?