Lecture 13: Metamorphisms

We've seen that hylomorphisms, which consist of a fold after an unfold, are a common and useful pattern of computation. In this lecture we study the opposite composition, an unfold after a fold. Some examples:

- regroup n = group n . concat
  to reformat a list of lists to a given width
- heapsort = flattenHeap . buildHeap
  as in Lecture 8
- baseConv (b,c) = toBase b . fromBase c
  to convert from one base to another (more later)
- arithCode = toBits . narrow
  arithmetic encoding.

In all these cases, the fold consumes some representation into an intermediate unconstrained format, and the unfold generates a new representation; the composite effects a change of representation, so we coin the term metamorphism.

Hylomorphisms always fuse.
Metamorphisms only fuse under certain conditions, but when they do they allow infinite representations to be processed.

Recall

$fold :: (\beta \times (\beta \times \alpha \to \beta)) \to List\ \alpha \to \beta$

$unfold_{List} :: (\beta \to 1 + \delta \times \beta) \to \beta \to List\ \delta$

Define

$stream :: ((\beta \to 1 + \delta \times \beta) \times \beta \times (\beta \times \alpha \to \beta)) \to List\ \alpha \to List\ \delta$

```
stream (f,b,g) as = case f b of
    inr (c,b') → cons (c, stream (f,b',g) as)
    inl ()     → case as of
                   cons (a,as') → stream (f, g(b,a), g) as'
                   nil          → nil
```

The streaming condition for f, g is that

$f\ b = inr\ (c, b') \Rightarrow f\ (g\ (b,a)) = inr\ (c, g\ (b', a))\ \forall a$

Under this invariant, for an finite as

$stream\ (f,b,g)\ as = unfold_{List}\ f\ (fold\ (b,g)\ as)$

Eg. str and hold for out_{List} and cat, so the buttering process $unfold_{List}\ out_{List} \cdot fold\ (nil, cat)$ can be streamed.

More generally, the producer might not exhaust the state, so the streaming process will switch into a flushing phase when the input is exhausted.

```
fstream :: ((B → 1 + C × B) × B × (B × A → B) × (B → List C)) →
           List A → List C

fstream (f,b,g,h) as = case f b of
    inr (c,b') → cons (c, fstream (f,b',g,h) as)
    inl ()     → case as of
                   cons (a,as') → fstream (f, g(b,a), g, h) as'
                   nil          → h b
```

If we define (equivalent to apo_{List})

```
apolist :: (B → C × B + List C) → B → List C
apolist f b = case f b of
    inl (c,b') → cons (c, apolist f b')
    inr cs     → cs
```

then, under the streaming condition for f and g,

```
fstream (f,b,g,h) as = apolist (put (f,h)) (fold g b as)
```

for finite as, where

```
put (f,h) b = case f b of  inr (c,b') → inl (c,b')
                           inl ()     → inr (h b)
```

Cautionary untold: more aggressive flushing.

As an extended example, consider converting a fraction from base 3 to base 7:

fromBase3 = foldl (0, step) where step (d,x) = $\frac{d+x}{3}$

toBase7 = unfold$_{\text{fin}}$ split where

split x = let y = 7*x in inr (⌊y⌋, y − ⌊y⌋)

(Always yields infinite output.)

The fold is of the wrong kind, but

fromBase3 = extract · foldl ((0,1), step)

where step ((u,v),d) = (d+u*3, v/3)

extract (u,v) = v*u

Here (u,v) is a defunctionalization of (v*) · (u+).

Now there's an abstraction in the middle, but that fuses with the unfold:

toBase7 · extract = unfold$_{\text{fin}}$ split' where

split' (u,v) = let y = ⌊7*u*v⌋ in

inr (y, (u − y/(v*7), v*7))

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

split' (1, 1/3) = inr (2, (1/7, 7/3))

split' (step ((1, 1/3), 1)) = split' (4, 1/9)

= inr (3, (1/7, 7/9))

That is, $0.1_3 \simeq 0.222_7$, but $0.11_3 \simeq 0.305_7$.

The producer has to be more cautious while input remains. Fortunately

unfold$_{list}$ split' = apo list (alt (split', unfold split'))

where

splitS (u,v) = if ⌊u×v×7⌋ = ⌊(u+1)×v×7⌋
then split' (u,v) else inl ()

(That's true in general: an unfold equals an apo with a guarded version of the same operator.) Now, the streaming condition holds for skipL and splitS. Therefore base conversion is

fstream (splitS, (0,1), skipL, unfold$_{list}$ split')

— but this works even for infinite inputs.

A second example is provided by spigot algorithms for computing the digits of π. It's known that $\pi = 2+\frac{1}{3}(2+\frac{2}{5}(2+\frac{3}{7}(2+\cdots)))$ or $(2+\frac{1}{3}x)(2+\frac{2}{5}x)(2+\frac{3}{7}x)\cdots(2+\frac{i}{2i+1}x)$, an infinite composition of linear fractional transformations $\begin{pmatrix} p & r \\ q & s \end{pmatrix} = x \mapsto (px+r)/(qx+s)$. π itself is another such composition $(3+\frac{1}{10}x)(1+\frac{1}{10}x)(4+\frac{1}{10}x)\cdots$. So computing the digits of π is a matter of converting from one representation to the other — a metamorphism.

Any tail of our expansion of π represents a number between 3 and 4:

$3 = x$ when $x = 2\frac{1}{3}x < 2 + \frac{1}{2+n}\left(2 + \frac{n}{2+n}(\cdots)\right)$.

$< x$ when $x = 2 + \frac{1}{2}x = 4$.

So $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)x$ ranges between $\frac{3q+r}{3s+t}$ and $\frac{4q+r}{4s+t}$ as x varies, provided that $3s+t$ and $4s+t$ have the same sign. The next digit is determined if these two bounds have the same floor, in which case that digit may be output; otherwise another term must be input. So

pi = stream (f, init, nxt) lfts where

init = $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$

lfts = $[\left(\begin{smallmatrix} i & 4i+2 \\ 0 & 2i+1 \end{smallmatrix}\right) \mid i \leftarrow [1..]]$

f $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)$ = inr $\left(n,\ \text{nxt}\left(\begin{smallmatrix} 10 & -10n \\ 0 & 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)\right)$, if $[\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)4] =$

= inl (), otherwise

where $n = \lfloor \left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)3 \rfloor$

nxt $\left(\begin{smallmatrix} q & r \\ s & t \end{smallmatrix}\right)\left(\begin{smallmatrix} u & v \\ w & x \end{smallmatrix}\right) = \left(\begin{smallmatrix} qu+rw & qv+rx \\ su+tw & sv+tx \end{smallmatrix}\right)$

Since the int is infinite, flushing is not needed. This program may be compressed considerably; for instance, s is always 0. The resulting program won a prize in a recent Obfuscated Haskell competition.

(see p106)