Software components in Haskell via monadic Mealy machines (I)

J.N. Oliveira & L.S. Barbosa

MFES - 2012-13

Abstract: Draft of a component-ware library in Haskell based on the *components as coalgebras* approach proposed in [1, 2].

1 Latex

In `vim` add commands

```
map <C-G> :w!<CR>:!ghci -XNPlusKPatterns -XMultiParamTypeClasses
-XFlexibleInstances -XFlexibleContexts
map <C-U> :w!<CR>:!lhs2TeX -o mmm08.tex mmm08.lhs; latex
mmm08<CR>i
```

At any time: type `<C-U>` to redo DVI. At any time: type `<C-G>` to run Haskell.

2 Imports

The following modules are imported:

- Basic library of pointfree combinators:

  import Cp

- State monad library:

  import St

- State monad transformer library:

  import SMT

- Maybe monad transformer library (for partial behaviour):

  import MMT

- Distribution and list monads:

  import $Probability$
  import $Data.List$
  import $Qais$

Auxiliary functions can be found in the appendix.

3 Main construction steps based on Mealy machines as coalgebras

Mealy machines (monadic and deterministic) of type $a \to b$ are monadic functions of type $(Monad\ m) \Rightarrow (s, a) \to m\ (s, b)$, where s is the state of the machine. Data type

$$\mathbf{data}\ Mealy\ m\ s\ a\ b = Mealy\ \{mealy :: (s, a) \to m\ (s, b)\}$$

could be introduced later as the basis for the library:

1. Lift function to machine Functions become machines once paralleled with identity on states:

$$\begin{aligned}&f2m :: (Monad\ m) \Rightarrow (a \to b) \to (s, a) \to m\ (s, b)\\&f2m\ f = \eta \cdot (id \times f)\end{aligned}$$

2. Copy This is the "do nothing" machine:

$$\begin{aligned}&copy :: (Monad\ m) \Rightarrow (s, c) \to m\ (s, c)\\&\quad \text{-- identity}\\&copy = f2m\ id\end{aligned}$$

3. Sequential composition of machines Pipeline of two machines:

$$\begin{aligned}&seqc :: (Strong\ \mathsf{F}, Monad\ \mathsf{F}) \Rightarrow\\&\quad \text{-- input machines}\\&\quad ((s, i) \to \mathsf{F}\ (s, o)) \to\\&\quad ((r, o) \to \mathsf{F}\ (r, k)) \to\\&\quad \text{-- output machine}\\&\quad ((s, r), i) \to \mathsf{F}\ ((s, r), k)\\&seqc\ p\ q = ((\mathsf{F}\ \mathsf{a}^{\circ}) \cdot \tau_l \cdot (id \times q) \cdot \mathsf{xl}) \bullet \tau_r \cdot (p \times id) \cdot \mathsf{xr}\end{aligned}$$

In the algebra original description in [2] this is written as

$$seqc\ p\ q = \mu \cdot (\mathsf{F}\ (\mathsf{F}\ \mathsf{a}^{\circ})) \cdot (\mathsf{F}\ \tau_l) \cdot (\mathsf{F}\ (id \times q)) \cdot (\mathsf{F}\ \mathsf{xl}) \cdot \tau_r \cdot (p \times id) \cdot \mathsf{xr}.$$

The version above is shorter thanks to Kleisli composition (precedence according to $(f \bullet g) \cdot h = f \bullet (g \cdot h)$).

4. Parallel composition of machines $times :: (Monad\ \mathsf{F}, Strong\ \mathsf{F}) \Rightarrow$

$$\begin{aligned}&\quad \text{-- input machines}\\&\quad ((s, i) \to \mathsf{F}\ (s, o)) \to ((t, j) \to \mathsf{F}\ (t, r)) \to\\&\quad \text{-- output machine}\\&\quad ((s, t), (i, j)) \to \mathsf{F}\ ((s, t), (o, r))\\&times\ p\ q = (\mathsf{F}\ m) \cdot deltal \cdot (p \times q) \cdot m\\&\quad \mathbf{where}\ m\ ((p, q), (i, j)) = ((p, i), (q, j))\end{aligned}$$

Laws in the algebra can also be run, e.g.

$$lemma36\ f\ g = [times\ (f2m\ f)\ (f2m\ g), f2m\ (f \times g)]$$

etc

5. External choice of machines $\cdot \boxplus \cdot :: (Strong\ \mathsf{F}) \Rightarrow$
-- input machines
$((s, i) \to \mathsf{F}\ (s, o)) \to ((t, j) \to \mathsf{F}\ (t, r)) \to$
-- output machine
$((s, t), i + j) \to \mathsf{F}\ ((s, t), o + r)$
$p \boxplus q = [(\mathsf{F}\ (id \times i1))\ , (\mathsf{F}\ (id \times i2))] \cdot$
$(\mathsf{F}\ \mathsf{xr} + (\mathsf{F}\ \mathsf{a}^\circ)) \cdot (\tau_r \cdot (p \times id) + \tau_l \cdot (id \times q)) \cdot$
$(\mathsf{xr} + \mathsf{a}) \cdot \mathsf{dr}$

6. Feedback This runs a machine twice:

$feedb :: (Monad\ \mathsf{F}) \Rightarrow (a \to \mathsf{F}\ a) \to a \to \mathsf{F}\ a$
$feedb\ p = p \bullet p$

This works for every endo-machine of type $a \to a$.

7. Partial feedback Operator precedence in the following is according to law $(f \bullet g) \cdot h = f \bullet (g \cdot h)$):

$pfeedb :: (Functor\ \mathsf{F}, Monad\ \mathsf{F}) \Rightarrow$
-- input machine
$((s, i + z) \to \mathsf{F}\ (s, o + z)) \to$
-- output machine
$(s, i + z) \to \mathsf{F}\ (s, o + z)$
$pfeedb\ p = (\nabla \cdot (\eta + p) \cdot (id \times i1 + id \times i2) \cdot \mathsf{dr}) \bullet p$

8. Interface wrapping $wrap :: (Functor\ \mathsf{F}) \Rightarrow ((b, e) \to \mathsf{F}\ (a, c)) \to (i \to e) \to (c \to d) \to (b, i) \to \mathsf{F}\ (a, d)$
$wrap\ p\ f\ g = fmap\ (id \times g) \cdot p \cdot (id \times f)$

Now summing up Mealy machines as in [3]; the main difference is the fact that in the paper the (relational) sum happens before transposition (introduction of the powerset or maybe monad):

1. Binary sum:

   $sum2 :: (Functor\ \mathsf{F}) \Rightarrow ((s, a) \to \mathsf{F}\ (r, b)) \to ((s, i) \to \mathsf{F}\ (r, c)) \to (s, a + i) \to \mathsf{F}\ (r, b + c)$
   $sum2\ m1\ m2 = (fmap\ undr) \cdot codelta \cdot (m1 + m2) \cdot \mathsf{dr}$

2. Ternary sum:

   $sum3 :: (Functor\ \mathsf{F}) \Rightarrow$
   $((s, e) \to \mathsf{F}\ (a, b)) \to ((s, b1) \to \mathsf{F}\ (a, c)) \to ((s, b2) \to \mathsf{F}\ (a, d)) \to$
   $(s, (e + b1) + b2) \to \mathsf{F}\ (a, (b + c) + d)$
   $sum3\ m1\ m2\ m3 = after \cdot ((m1 + m2) + m3) \cdot ldr2$

   where

   $after :: Functor\ \mathsf{F} \Rightarrow ((\mathsf{F}\ (s, a)) + (\mathsf{F}\ (s, b))) + (\mathsf{F}\ (s, c)) \to \mathsf{F}\ (s, (a + b) + c)$
   $after = (\mathsf{F}\ lundr2) \cdot lcodelta2$

4 Example of stacks and folders, in several steps and layers

(a) First layer: a partial stack at functional level:

$$push = flip\ (:)$$
$$pop = tail$$
$$top = head$$
$$empty\ a = length\ a \equiv 0$$

(b) Individual Mealy (Maybe) machines, one per totalised (partial) function

$$push' :: ([b], b) \to \mathsf{M}\ ([b], ())$$
$$push' = \eta \cdot \langle (uncurry\ push), (!) \rangle \quad \text{-- total writer}$$
$$pop' :: ([a], b) \to \mathsf{M}\ ([a], a)$$
$$pop' = tot\ \langle pop, top \rangle\ (\neg \cdot empty) \cdot p1 \quad \text{-- partial writer}$$
$$top' :: ([a], b) \to \mathsf{M}\ ([a], a)$$
$$top' = tot\ \langle id, top \rangle\ (\neg \cdot empty) \cdot p1 \quad \text{-- partial reader}$$
$$empty' :: ([a], b) \to \mathsf{M}\ ([a], \mathbb{B})$$
$$empty' = \eta \cdot \langle id, empty \rangle \cdot p1 \quad \text{-- total reader}$$

where tot totalizes a partial function by fusion with its precondition:

$$tot :: (a \to b) \to (a \to \mathbb{B}) \to a \to \mathsf{M}\ b$$
$$tot\ f\ p = Cp.cond\ p\ (Just \cdot f)\ \underline{Nothing}$$

(c) summing up all these (NB: leaving out empty', which is not used in the folder example):

$$stack :: ([p], ((\,) + (\,)) + p) \to \mathsf{M}\ ([p], (p + p) + (\,))$$
$$stack = sum3\ pop'\ top'\ push'$$

(d) start building a folder by wrapping two stacks:

$$rightS :: ([b], ((\,) + (\,)) + (b + b)) \to \mathsf{M}\ ([b], (b + b) + (\,))$$
$$rightS = wrap\ stack\ (id + \nabla)\ id$$
$$leftS :: ([c], (\,) + c) \to \mathsf{M}\ ([c], c + (\,))$$
$$leftS = wrap\ stack\ (i1 + id)\ ((id + (!)) \cdot coassocr)$$

(e) now we put everything together...

$$almostFolder = pfeedb\ (wrap\ (leftS \boxplus rightS)\ wi\ wo)$$

where

$$wi = [[[[i11\ , i211]\ , i212]\ , i222]\ , [i221\ , i12]]$$
$$wo = [[i21\ , i111]\ , [[i22\ , i112]\ , i12]]$$

That is:

$$wi = [[[(i1 + i11)\ , i212]\ , i222]\ , (coswap \cdot (i21 + i2))]$$
$$wo = [(coswap \cdot (i1 + i11))\ , [(coswap \cdot (i2 + i12))\ , i12]]$$

(f) instead of doing the final wrapping by *fi,fo* as in the original calculus I've tried to introduced some linguistic stuff in, via explicit APIs. Because stack and folder are separable components both exhibit the same input and output interfaces:

```
data StackAPI a b c = Pop a | Top b | Push c deriving Show
data FolderAPI a b c d = Tr a | Tl b | Rd c | In d deriving (Eq, Show, Ord)
```

Then we may write:

```
apiFolder :: (([d], [d]), FolderAPI () () () d) → M (([d], [d]), FolderAPI () () d d1)
apiFolder = wrap almostFolder wi wo
  where wi (Tr a) = i1111 a
        wi (Tl a) = i1112 a
        wi (Rd a) = i112 a
        wi (In a) = i12 a
        wo (i1 (i2 a)) = Tr a
        wo (i1 (i1 (i1 a))) = Tl a
        wo (i1 (i1 (i2 a))) = Rd a
```

and

```
apiStack :: ([c], StackAPI () () c) → M ([c], StackAPI c c ())
apiStack = wrap stack wi wo
  where wi (Pop ()) = i11 ()
        wi (Top ()) = i12 ()
        wi (Push a) = i2 a
        wo (i1 (i1 o)) = Pop o
        wo (i1 (i2 o)) = Top o
        wo (i2 o) = Push o
```

(g) need to prepare *apiStack* for the given *wi, wo* wires — (much of this should go away via fusion laws):

```
rightApiS :: ([b], (() + ()) + (b + b)) → M ([b], (b + b) + ())
rightApiS = wrap apiStack wi wo
            where wi = [[Pop , Top] , [Push , Push]]
                  wo (Pop p) = i11 p
                  wo (Top p) = i12 p
                  wo (Push ()) = i2 ()
leftApiS :: ([b], () + b) → M ([b], b + ())
leftApiS = wrap apiStack [Pop , Push] wo
            where wo (Pop p) = i1 p
                  wo (Push ()) = i2 ()
```

(h) putting both of these together

```
almostFolderApi = pfeedb (wrap (leftApiS ⊞ rightApiS) wi wo)
```

(i) finishing up:

```
folderApi :: (([d], [d]), FolderAPI () () () d) → M (([d], [d]), FolderAPI () () d d1)
folderApi = wrap almostFolderApi wi wo
```

```
where wi (Tr a) = i1111 ()
  wi (Tl a) = i1 (i112 ())
  wi (Rd a) = i112 ()
  wi (In a) = i12 a
  wo (i1 (i2 a)) = Tr a
  wo (i1 (i1 (i1 a))) = Tl a
  wo (i1 (i1 (i2 a))) = Rd a
```

4.1 The coalgebraic twist

A "curried Mealy machine" is a coalgebra, eg.

```
folderCoalg = curry folderApi
```

over which we can alreadty do single step testing: check behaviour of *folderCoalg* over given state

```
state = (["L1","L2"],["R1"])
testTr = folderCoalg state (Tr ())   -- testTr should be Just ((["L2"],["L1","R1"]),Tr ())
testTl = folderCoalg state (Tl ())   -- testTl should be Just ((["R1","L1","L2"],[]),Tl ())
testRd = folderCoalg state (Rd ())   -- testRd should be Just ((["L1","L2"],["R1"]),Rd "R1")
testIn = folderCoalg state (In "A")   -- testIn should be Just ((["L1","L2"],["A","R1"]),Tr ())
```

Note the strange "Tr()" in *testIn*, due to the fact that in actual wirings, *Tr* and *In* null output are joined. Thus the "d1" in the type of *folderApi*.

4.2 Flipping Mealy coalgebras to obtain state monadic components

In the example, *folderApi*'s behaviour animation is obtained by flipping the coalgebra we obtain a monadic state transformer

```
folderApiSt = flip folderCoalg
```

Now use the library library for the state monad transformer SMT.

```
folderApiSt' i s = (fmap swap) (folderApiSt i s)
```

where *swap* is required because SMT state and ouput are swapped in *SMT*. Finally, we get a Maybe state monadic transformer

```
folderAcceptMachine :: FolderAPI () () () d → SMT ([d],[d]) M (FolderAPI () () d d1)
folderAcceptMachine i = SMT (folderApiSt' i)
```

"Stream" animation emulates the final coalgebra by mapping input streams in batch mode into output streams, monadically.

```
batch :: (Monad (SMT s m), Monad m, Functor m) ⇒ (b → SMT s m a) → [b] → s → m [a]
batch mst inp seed = runSMT (sequence (map mst inp)) seed   -- result is output stream
stream1 = [Rd (), Tl (), Rd (), In "C"]
seed1 = (["A"],["B"])
out1 = batch folderAcceptMachine stream1 seed1   -- out1 is Nothing: folder dies upon second Rd
stream2 = [In "A", Tl (), In "B", Tr (), Rd ()]
seed2 = ([],[])
out2 = batch folderAcceptMachine stream2 seed2   -- ok, but note the strange Tr()
    -- outputs already mentioned.
```

4.3 Bringing IO in via MMT (Maybe monad transformer)

So as to have interactive machines:

```
folderApiSMT :: FolderAPI () () () String →
  SMT ([String], [String]) (MMT IO) (FolderAPI () () String String)
folderApiSMT = m2smt folderApiSt'


script :: SMT ([String], [String]) (MMT IO) ()
script = do {i ← dlift (
    do {putStr "\nDo what? "; getLine}
  );
  o ← folderApiSMT (recognize i);
  dlift (
    do putStr ("\nInput was: " ++ (show (recognize i)));
      putStr ("\nOutput is: " ++ (show o) ++ "\n");
    );
    s ← iSt (get);
    dlift (writeFile "_folder.txt" (show s))
  } where
    recognize s = let l = words s
      in case l of
        "In" : (s: _) → In s
        "Tr": _ → Tr ()
        "Tr()": _ → Tr ()
        "Tl": _ → Tl ()
        "Tl()": _ → Tl ()
        otherwise → Rd ()
```

Finally, an interactive shell:

```
main = unlift (runSMT (loop script) seed2)
```

which dies (silently) as soon as an invalid operation takes place...

A Auxiliary functions

The following functor instances,

```
D :: (a → b) → D a → D b
D = fmap


M :: (a → b) → M a → M b
M = fmap
```

and

```
F :: Functor F ⇒ (a → b) → F a → F b
F = fmap
```

as well as shortcut

```
DM = D · M
```

are intended for pretty printing and tighter type checking. We also define

$\mu_{\mathsf{DM}} :: \mathsf{D}\ (\mathsf{M}\ (\mathsf{D}\ (\mathsf{M}\ b))) \to \mathsf{D}\ (\mathsf{M}\ b)$
$\mu_{\mathsf{DM}} = \mu \cdot \mathsf{D}\ (\mathsf{D}\ (\mu) \cdot cl)$

$rstrD :: (Strong\ f, Monad\ m) \Rightarrow (m\ (f\ a), b) \to m\ (f\ (a, b))$
$rstrD\ (d, b) = \mathbf{do}\ \{m \leftarrow d; \eta\ (\tau_r\ (m, b))\}$
$lstrD :: (Strong\ f, Monad\ m) \Rightarrow (b, m\ (f\ a)) \to m\ (f\ (b, a))$
$lstrD\ (a, d) = \mathbf{do}\ \{m \leftarrow d; \eta\ (\tau_l\ (a, m))\}$

$deltar :: (Monad\ m, Strong\ m) \Rightarrow (m\ a, m\ b) \to m\ (a, b)$
$deltar = \tau_r \bullet \tau_l$
$deltal :: (Monad\ m, Strong\ m) \Rightarrow (m\ b, m\ a) \to m\ (b, a)$
$deltal = \tau_l \bullet \tau_r$
$\nabla :: b + b \to b$
$\nabla = [id\ , id]$
$\mathsf{xr} :: ((a, b), c) \to ((a, c), b)$
$\mathsf{xr}\ ((u, u'), i) = ((u, i), u')$
$\mathsf{xl} :: ((a, b), c) \to (a, (c, b))$
$\mathsf{xl}\ ((u, u'), i) = (u, (i, u'))$
$\mathsf{dr} :: (a, c + b) \to (a, c) + (a, b)$
$\mathsf{dr}\ (a, i_1\ b) = i_1\ (a, b)$
$\mathsf{dr}\ (a, i_2\ c) = i_2\ (a, c)$
$dr2 :: (c, d + (a + b)) \to (c, d) + ((c, a) + (c, b))$
$dr2 = (id + \mathsf{dr}) \cdot \mathsf{dr}$
$dr3 = (id + dr2) \cdot \mathsf{dr}$
$ldr2 = (\mathsf{dr} + id) \cdot \mathsf{dr}$
$undr :: (a, b) + (a, c) \to (a, b + c)$
$undr = [(id \times i1)\ , (id \times i2)]$
$undr2 :: (a, b) + ((a, b1) + (a, c)) \to (a, b + (b1 + c))$
$undr2 = undr \cdot (id + undr)$
$undr3 = undr \cdot (id + undr2)$
$lundr2 = undr \cdot (undr + id)$
$codelta :: (Functor\ m) \Rightarrow (m\ a) + (m\ b) \to m\ (a + b)$
$codelta = [(fmap\ i_1)\ , (fmap\ i_2)]$
$merge :: Functor\ f \Rightarrow (a \to f\ a1) \to (b \to f\ b1) \to a + b \to f\ (a1 + b1)$
$merge\ f\ g = [((fmap\ i1) \cdot f)\ , ((fmap\ i2) \cdot g)]$
$codelta2 :: (Functor\ m) \Rightarrow (m\ a) + ((m\ c) + (m\ b)) \to m\ (a + (c + b))$
$codelta2 = codelta \cdot (id + codelta)$
$codelta3 :: (Functor\ m) \Rightarrow (m\ a) + ((m\ a1) + ((m\ c) + (m\ b))) \to m\ (a + (a1 + (c + b)))$
$codelta3 = codelta \cdot (id + codelta2)$
$lcodelta2 :: (Functor\ m) \Rightarrow ((m\ a) + (m\ c)) + (m\ b) \to m\ ((a + c) + b)$
$lcodelta2 = codelta \cdot (codelta + id)$
$i11\ \ = i1 \cdot i1$
$i111 = i1 \cdot i1 \cdot i1$
$i1111 = i1 \cdot i1 \cdot i1 \cdot i1$
$i1112 = i1 \cdot i1 \cdot i1 \cdot i2$
$i112 = i1 \cdot i1 \cdot i2$
$i12\ \ = i1 \cdot i2$
$i121 = i1 \cdot i2 \cdot i1$
$i122 = i1 \cdot i2 \cdot i2$

$$
\begin{aligned}
i21 &= i2 \cdot i1 \\
i211 &= i2 \cdot i1 \cdot i1 \\
i212 &= i2 \cdot i1 \cdot i2 \\
i22 &= i2 \cdot i2 \\
i221 &= i2 \cdot i2 \cdot i1 \\
i222 &= i2 \cdot i2 \cdot i2
\end{aligned}
$$

References

[1] L.S. Barbosa. *Towards a calculus of state-based software components*. *Journal of Universal Computer Science*, 9 (8):891–909, 2003.

[2] L.S. Barbosa and J.N. Oliveira. *State-based components made generic*. *Elect. Notes in Theor. Comp. Sci. (CMCS'03 - Workshop on Coalgebraic Methods in Computer Science)*, 82.1, 2003.

[3] L.S. Barbosa and J.N. Oliveira. *Transposing partial components — an exercise on coalgebraic refinement*. *Theoretical Computer Science*, 365(1):2–22, 2006.

[4] J.N. Oliveira. Towards a linear algebra of programming. *Formal Asp. Comput.*, 24(4-6):433–458, 2012.