Evolution Of A Haskell Programmer

Aditya Siram

May 5, 2016

Outline

What

Freshman

Freshman Haskell programmer

```haskell
fac n = if n == 0
           then 1
           else n * fac (n-1)
```

(3 * (2 * (1 * 1)))

Foldr

Senior Haskell programmer

(voted for ~~Nixon~~ ~~Buchanan~~ Bush — "leans right")

```
fac n = foldr (*) 1 [1..n]
```

```
(1 * (2 * (3 * 1)))
```

Foldl

Another senior Haskell programmer

(voted for ~~McGovern~~ ~~Biafra~~ Nader — "leans left")

```
fac n = foldl (*) 1 [1..n]
```

```
(((1 * 1) * 2) * 3)
```

Foldr As Foldl

Yet another senior Haskell programmer

(leaned so far right he came back left again!)

```
-- using foldr to simulate foldl

fac n = foldr (\x g n -> g (x*n)) id [1..n] 1
```

```
fac n = foldr (\x g -> (\n -> g (x * n))) id [1 .. n] 1
(\n -> (\n -> (\n -> id (3 * n)) (2 * n)) (1 * n)) 1
(\n -> (\n -> id (3 * n)) (2 * n)) (1 * 1)
(\n -> id (3 * n)) ((1 * 1) * 2)
id (((1 * 1) * 2) *3)
(((1 * 1) * 2) *3)
```

Memoizing Haskell programmer

(takes Ginkgo Biloba daily)

```
facs = scanl (*) 1 [1..]

fac n = facs !! n
```

```
facs = scanl (*) 1 [1 ..]
facs = [1,1 * 1, <-- * 2, <-- * 3, ...]
                                ^
                                |
facs !! 3 = --------------------+
```

Accum

Accumulating Haskell programmer

(building up to a quick climax)

```haskell
facAcc a 0 = a
facAcc a n = facAcc (n*a) (n-1)

fac = facAcc 1
```

```
fac 3
facAnn (3 * 1)         2
facAnn (3 * 1 * 2)     1
facAnn (3 * 1 * 2 * 1) 0
= (3 * 1 * 2 * 1)
```

Continuation

Continuation-passing Haskell programmer

(raised RABBITS in early years, then moved to New Jersey)

```
facCps k 0 = k 1
facCps k n = facCps (k . (n *)) (n-1)

fac = facCps id
```

```
fac 3
facCps (id . (3 *)) 2
facCps ((id . (3 *)) . (2 *)) 1
facCps (((id . (3 *)) . (2 *)) (1 *))
(((id . (3 *)) . (2 *)) . (1 *)) 1
id (((1 * 1) * 2) * 3)
```

Knot

Boy Scout Haskell programmer

(likes tying knots; always "reverent," he belongs to the Church of the Least Fixed-Point [8])

```
y f = f (y f)

fac = y (\f n -> if (n==0) then 1 else n * f (n-1))
```

Knot

```
y f = f (y f)
    = f (f (y f))
    = f (f (f ...

fac = y (\f n -> if (n == 0) then 1 else n * f (n - 1)))

fac 3
 = y (\f n -> ... 3 * f 2)
 = 3 * (y (\f n -> ... 2 * f 1))
 = 3 * 2 * (y (\f n -> ... 1 * f 0))
 = 3 * 2 * 1 * (y (\f n -> if (n == 0) then 1 ...))
 = 3 * 2 * 1 * 1
```

Interpretive - Grammar

Interpretive Haskell programm

(never "met a language" he didn't lik

```haskell
-- a dynamically-typed term language

data Term = Occ Var
          | Use Prim
          | Lit Integer
          | App Term Term
          | Abs Var  Term
          | Rec Var  Term

type Var  = String
type Prim = String


-- a domain of values, including functions

data Value = Num  Integer
           | Bool Bool
           | Fun (Value -> Value)
```

Interpretive - Projection

```
prjFun (Fun f) = f
prjFun  _      = error "bad function value"

prjNum (Num n) = n
prjNum  _      = error "bad numeric value"

prjBool (Bool b) = b
prjBool  _       = error "bad boolean value"
```

Interpretive - Binary Op 1

```
binOp inj f = Fun (\i -> (Fun (\j -> inj (f (prjNum i) (prjNum j)))))
```

Interpretive - Binary Op 2

```
-- a (fixed) "environment" of language primitives

times = binOp Num  (*)
minus = binOp Num  (-)
equal = binOp Bool (==)
cond  = Fun (\b -> Fun (\x -> Fun (\y -> if (prjBool b) then x else y)))

prims = [ ("*", times), ("-", minus), ("==", equal), ("if", cond) ]
```

Interpretive - Environment

```haskell
-- environments mapping variables to values

type Env = [(Var, Value)]

getval x env =  case lookup x env of
                  Just v  -> v
                  Nothing -> error ("no value for " ++ x)
```

Interpretive - Eval

```
-- an environment-based evaluation function

eval env (Occ x) = getval x env
eval env (Use c) = getval c prims
eval env (Lit k) = Num k
eval env (App m n) = prjFun (eval env m) (eval env n)
eval env (Abs x m) = Fun  (\v -> eval ((x,v) : env) m)
eval env (Rec x m) = f where f = eval ((x,f) : env) m
```

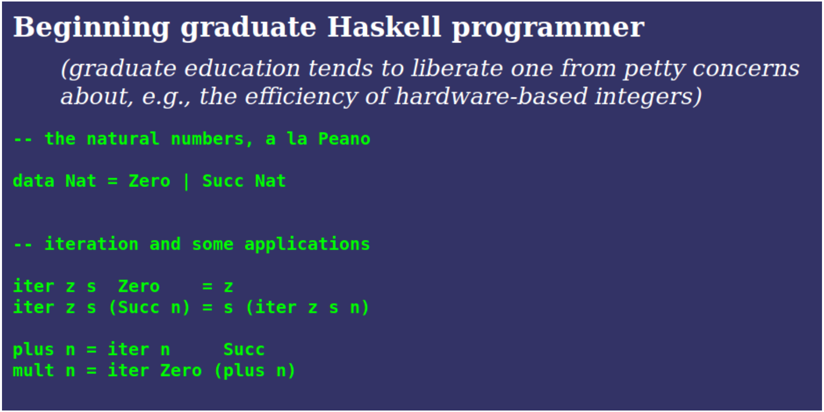
Interpretive - Runner

```haskell
-- a term representing factorial and a "wrapper" for evaluation

facTerm = Rec "f" (Abs "n"
              (App (App (App (Use "if")
                   (App (App (Use "==") (Occ "n")) (Lit 0))) (Lit 1))
                   (App (App (Use "*")  (Occ "n"))
                        (App (Occ "f")
                             (App (App (Use "-") (Occ "n")) (Lit 1))))))

fac n = prjNum (eval [] (App facTerm (Lit n)))
```

Peano




```haskell
Zero                -- 0
(Succ Zero)         -- 1
(Succ (Succ Zero))  -- 2
```

Peano plus

```
plus (Succ Zero) (Succ (Succ Zero)) =
  Succ (iter (Succ Zero) (Succ Zero))
       Succ (iter (Succ Zero) Zero)
             Succ Zero
```

Peano Fac

```
-- primitive recursion

primrec z s  Zero     = z
primrec z s (Succ n) = s n (primrec z s n)


-- two versions of factorial

fac  = snd . iter (one, one) (\(a,b) -> (Succ a, mult a b))
fac' = primrec one (mult . Succ)
```

Peano Fac

```
fac  = snd . iter (one, one) (\(a,b) -> (Succ a, mult a b))
=> (\(a,b) -> (Succ a, mult a b)
    (\(a,b) -> (Succ a, mult a b)
     (\(a,b) -> (Succ a, mult a b)
      (Succ Zero, Succ Zero)
=> (\(a,b) -> (Succ a, mult a b)
    (\(a,b) -> (Succ a, mult a b)
     (Succ (Succ Zero)), (Succ Zero)
=> (\(a,b) -> (Succ a, mult a b)
    (Succ (Succ Zero), Succ Zero)
=> (_ , (Succ ... Zero)) -- first is unevaluated!
=> (Succ (Succ (Succ (Succ (Succ (Succ Zero))))))
```

Peano Fac

```
fac' = primrec (Succ Zero) (mult . Succ)
=> (mult (Succ (Succ (Succ Zer0)))
    (mult (Succ (Succ Zero)))
      (mult (Succ Zero))
         (Succ Zero)
```

Origamist Fold

Origamist Haskell programmer

(always starts out with the "basic Bird fold")

```
-- (curried, list) fold and an application

fold c n []     = n
fold c n (x:xs) = c x (fold c n xs)

prod = fold (*) 1
```

```
prod = fold (*) 1 [3,2,1]
=> 3 * (fold (*) 1 [2,1])
=> 3 * 2 * (fold (*) 1 [1])
=> 3 * 2 * 1 * 1
```

Origamist Unfold

```
-- (curried, boolean-based, list) unfold and an application

unfold p f g x =
  if p x
     then []
     else f x : unfold p f g (g x)

downfrom = unfold (==0) id pred


-- hylomorphisms, as-is or "unfolded" (ouch! sorry ...)

refold  c n p f g   = fold c n . unfold p f g

refold' c n p f g x =
  if p x
     then n
     else c (f x) (refold' c n p f g (g x))
```

Origamist Unfold

```
downFrom 3 = unfold (== 0) id pred
=> (id 3) : (unfold (== 0) id 2)
=> (id 3) : (id 2) : (unfold (== 0) id 1)
=> (id 3) : (id 2) : (id 1) : []
=> [3,2,1]
```

Origamist Fac

```
-- several versions of factorial, all (extensionally) equivalent

fac   = prod . downfrom
fac'  = refold  (*) 1 (==0) id pred
fac'' = refold' (*) 1 (==0) id pred
```

Cartesian Cata

Cartesianally-inclined Haskell programmer

(prefers Greek food, avoids the spicy Indian stuff; inspired by Lex Augusteijn's "Sorting Morphisms" [3])

```
-- (product-based, list) catamorphisms and an application

cata (n,c) []     = n
cata (n,c) (x:xs) = c (x, cata (n,c) xs)

mult = uncurry (*)
prod = cata (1, mult)
```

Cartesian Ana

```
uncurry f (a,b)= f a b
prod = cata (1, uncurry (*)) [3,2,1]
=> (uncurry (*) (3,
    (uncurry (*) (2,
     (uncurry (*) (1, 1)))
```

Cartesian Ana

```haskell
-- (co-product-based, list) anamorphisms and an application

ana f = either (const []) (cons . pair (id, ana f)) . f

cons = uncurry (:)

downfrom = ana uncount

uncount 0 = Left  ()
uncount n = Right (n, n-1)
```

```haskell
pair (f,g) (x,y) = (f x, g y)
```

Cartesian Ana

```
pair (f,g) (x,y) = (f x, g y)
ana f = either (const []) (cons . pair (id, ana f)) . f
cons = uncurry (:)
ana uncount 3
=> (uncurry (:) (3,
     uncurry (:) (2,
      uncurry (:) (1, (const [])))))
=> [3,2,1]
```

Cartesian Hylo

```
hylo  f  g      = cata g . ana f
```

```
hylo = fold . unfold
```

Cartesian Fac

```
fac    = prod . downfrom
fac'   = hylo  uncount (1, mult)
```

Phd Mu

Ph.D. Haskell programmer

(ate so many bananas that his eyes bugged out, now he needs new lenses!)

```
-- explicit type recursion based on functors

newtype Mu f = Mu (f (Mu f))  deriving Show

in      x  = Mu x
out (Mu x) = x
```

```
-- notice the similarity
newtype Mu f = Mu (f (Mu f))
y f = y (f y)
```

Phd Cata

```
-- cata- and ana-morphisms, now for *arbitrary* (regular) base functors

cata phi = phi . fmap (cata phi) . out
ana  psi = in  . fmap (ana  psi) . psi
```

```
-- injection/projection
out (Mu x) = x
in  x      = Mu x

-- morphisms
cata phi = phi . fmap (cata phi) . out
ana  psi = in .  fmap (ana  psi) . psi
```

Phd Nats

```haskell
-- base functor and data type for natural numbers,
-- using a curried elimination operator

data N b = Zero | Succ b  deriving Show

instance Functor N where
  fmap f = nelim Zero (Succ . f)

nelim z s  Zero     = z
nelim z s (Succ n) = s n

type Nat = Mu N
```

```haskell
Mu (Succ (Mu Succ (Mu Succ (Mu Zero)))) -- 3
```

Phd Lists

```
-- base functor and data type for lists

data L a b = Nil | Cons a b  deriving Show

instance Functor (L a) where
  fmap f = lelim Nil (\a b -> Cons a (f b))

lelim n c  Nil        = n
lelim n c (Cons a b) = c a b

type List a = Mu (L a)
```

```
Mu (Cons 3 (Mu (Cons 2 (Mu (Cons 1 (Mu Nil)))))) -- [3,2,1]
```

Phd Nats

```haskell
zero = in   Zero
suck = in . Succ       -- pardon my "French" (Prelude conflict)

plus n = cata (nelim n     suck   )
mult n = cata (nelim zero (plus n))
```

```
plus (Mu (Succ (Mu (Succ (Mu Zero))))) (Mu (Succ (Mu Zero)))
=>   (in . Succ . in . Succ)            (Mu (Succ (Mu Zero)))
=>   (Mu (Succ (Mu (Succ (Mu (Succ (Mu Zero)))))))
```

Phd Fac

```
prod = cata (lelim (suck zero) mult)

upto = ana (nelim Nil (diag (Cons . suck)) . out)

diag f x = f x x

fac = prod . upto
```

```
fac (Mu (Succ (Mu (Succ (Mu (Succ (Mu Zero))))))) = 6
```

Tenured professor

(teaching Haskell to freshmen)

```
fac n = product [1..n]
```

Resources

http://www.willamette.edu/~fruehr/haskell/evolution.html?