The Evolution of a Haskell Programmer

Fritz Ruehr, Willamette University

See Iavor Diatchki’s page “The Evolution of a Programmer” for the “original” (though he is not the author), and also below for the story behind this version.

(This page has been translated into the Serbo-Croatian language by Anja Skrba from Webhostinggeeks.com. *Thanks, Anja, for all your hard work!*)

Freshman Haskell programmer

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

Sophomore Haskell programmer, at MIT

(studied Scheme as a freshman)

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

Junior Haskell programmer

(beginning Peano player)

```
fac  0    =  1
fac (n+1) = (n+1) * fac n
```

Another junior Haskell programmer

(read that n+k patterns are “a disgusting part of Haskell” [1]
and joined the “Ban n+k patterns”-movement [2])

```
fac 0 = 1
fac n = n * fac (n-1)
```

Senior Haskell programmer

(voted for ~~Nixon~~ ~~Buchanan~~ Bush — “leans right”)

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

Another senior Haskell programmer

(voted for ~~McGovern~~ ~~Biafra~~ Nader — “leans left”)

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

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
```

Memoizing Haskell programmer

(takes Ginkgo Biloba daily)

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

fac n = facs !! n
```

~~Pointless~~ *(ahem) “Points-free”* Haskell programmer

(studied at Oxford)

```
fac = foldr (*) 1 . enumFromTo 1
```

Iterative Haskell programmer

(former Pascal programmer)

```
fac n = result (for init next done)
        where init = (0,1)
              next   (i,m) = (i+1, m * (i+1))
              done   (i,_) = i==n
              result (_,m) = m

for i n d = until d n i
```

Iterative one-liner Haskell programmer

(former APL and C programmer)

```
fac n = snd (until ((>n) . fst) (\(i,m) -> (i+1, i*m)) (1,1))
```

Accumulating Haskell programmer

(building up to a quick climax)

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

fac = facAcc 1
```

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
```

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))
```

Combinatory Haskell programmer

(eschews variables, if not obfuscation;
all this currying's just a phase, though it seldom hinders)

```
s f g x = f x (g x)

k x y   = x

b f g x = f (g x)

c f g x = f x g

y f     = f (y f)

cond p f g x = if p x then f x else g x

fac  = y (b (cond ((==) 0) (k 1)) (b (s (*)) (c b pred)))
```

List-encoding Haskell programmer

(prefers to count in unary)

```
arb = ()    -- "undefined" is also a good RHS, as is "arb" :)

listenc n = replicate n arb
listprj f = length . f . listenc

listprod xs ys = [ i (x,y) | x<-xs, y<-ys ]
                 where i _ = arb

facl []         = listenc  1
facl n@(_:pred) = listprod n (facl pred)

fac = listprj facl
```

Interpretive Haskell programmer

(never "met a language" he didn't like)

```
-- 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)

instance Show Value where
  show (Num  n) = show n
  show (Bool b) = show b
  show (Fun  _) = ""

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"

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


-- 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)


-- 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


-- 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) ]


-- 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)))
```

Static Haskell programmer

(he does it with class, he's got that fundep Jones!
After Thomas Hallgren's "Fun with Functional Dependencies" [7])

```
-- static Peano constructors and numerals
```

```
data Zero
data Succ n

type One   = Succ Zero
type Two   = Succ One
type Three = Succ Two
type Four  = Succ Three


-- dynamic representatives for static Peanos

zero  = undefined :: Zero
one   = undefined :: One
two   = undefined :: Two
three = undefined :: Three
four  = undefined :: Four


-- addition, a la Prolog

class Add a b c | a b -> c where
  add :: a -> b -> c

instance              Add  Zero    b  b
instance Add a b c => Add (Succ a) b (Succ c)


-- multiplication, a la Prolog

class Mul a b c | a b -> c where
  mul :: a -> b -> c

instance                         Mul  Zero    b Zero
instance (Mul a b c, Add b c d) => Mul (Succ a) b d


-- factorial, a la Prolog

class Fac a b | a -> b where
  fac :: a -> b

instance                                Fac  Zero    One
instance (Fac n k, Mul (Succ n) k m) => Fac (Succ n) m

-- try, for "instance" (sorry):
--
--     :t fac four
```

Beginning graduate Haskell programmer

(graduate education tends to liberate one from petty concerns about, e.g., the efficiency of hardware-based integers)

```
-- the natural numbers, a la Peano

data Nat = Zero | Succ Nat


-- iteration and some applications

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

plus n = iter n     Succ
mult n = iter Zero (plus n)


-- 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)


-- for convenience and testing (try e.g. "fac five")

int = iter 0 (1+)

instance Show Nat where
  show = show . int

(zero : one : two : three : four : five : _) = iterate Succ Zero
```

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


-- (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))


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

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

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)


-- (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)


-- two variations on list hylomorphisms

hylo  f  g    = cata g . ana f

hylo' f (n,c) = either (const n) (c . pair (id, hylo' f (c,n))) . f

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


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

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

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


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

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


-- 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


-- conversion to internal numbers, conveniences and applications

int = cata (nelim 0 (1+))

instance Show Nat where
  show = show . int

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

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


-- 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)


-- conversion to internal lists, conveniences and applications

list = cata (lelim [] (:))

instance Show a => Show (List a) where
  show = show . list

prod = cata (lelim (suck zero) mult)

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

diag f x = f x x

fac = prod . upto
```

Post-doc Haskell programmer

(from Uustalu, Vene and Pardo’s “Recursion Schemes from Comonads” [4])

```
-- explicit type recursion with functors and catamorphisms

newtype Mu f = In (f (Mu f))

unIn (In x) = x

cata phi = phi . fmap (cata phi) . unIn


-- base functor and data type for natural numbers,
-- using locally-defined "eliminators"

data N c = Z | S c

instance Functor N where
  fmap g  Z    = Z
  fmap g (S x) = S (g x)

type Nat = Mu N
```

```
zero   = In  Z
suck n = In (S n)

add m = cata phi where
  phi  Z    = m
  phi (S f) = suck f

mult m = cata phi where
  phi  Z    = zero
  phi (S f) = add m f


-- explicit products and their functorial action

data Prod e c = Pair c e

outl (Pair x y) = x
outr (Pair x y) = y

fork f g x = Pair (f x) (g x)

instance Functor (Prod e) where
  fmap g = fork (g . outl) outr


-- comonads, the categorical "opposite" of monads

class Functor n => Comonad n where
  extr :: n a -> a
  dupl :: n a -> n (n a)

instance Comonad (Prod e) where
  extr = outl
  dupl = fork id outr


-- generalized catamorphisms, zygomorphisms and paramorphisms

gcata :: (Functor f, Comonad n) =>
           (forall a. f (n a) -> n (f a))
             -> (f (n c) -> c) -> Mu f -> c

gcata dist phi = extr . cata (fmap phi . dist . fmap dupl)

zygo chi = gcata (fork (fmap outl) (chi . fmap outr))

para :: Functor f => (f (Prod (Mu f) c) -> c) -> Mu f -> c
para = zygo In


-- factorial, the *hard* way!

fac = para phi where
  phi  Z             = suck zero
  phi (S (Pair f n)) = mult f (suck n)


-- for convenience and testing

int = cata phi where
  phi  Z    = 0
  phi (S f) = 1 + f

instance Show (Mu N) where
  show = show . int
```

Tenured professor

(teaching Haskell to freshmen)

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

Background

On 19 June 2001, at the OGI PacSoft Tuesday Morning Seminar Series , Iavor Diatchki presented the paper “Recursion Schemes from Comonads” by Uustalu, Vene and Pardo [4]. I attended Iavor’s excellent presentation and remarked that I found the end of the paper rather anti-climactic: after much categorical effort and the definition of several generalized recursion combinators, the main examples were the factorial and Fibonacci functions. (Of course, I offered no better examples myself, so this was rather unfair carping.)

Some time later, I came across Iavor’s "jokes" page, including a funny bit called “The Evolution of a Programmer” in which the traditional imperative "Hello, world" program is developed through several variations, from simple beginnings to a ridiculously complex extreme. A moment’s thought turned up the factorial function as the best functional counterpart of "Hello, world". Suddenly the Muse struck and I knew I must write out these examples, culminating (well, almost) in the heavily generalized categorical version of factorial provided by Uustalu, Vene and Pardo.

I suppose this is what you’d have to call “small-audience” humour.

PS: I’ve put all the code into a better-formatted text file for those who might like to experiment with the different variations (you could also just cut and paste a section from your browser).

PPS: As noted above, Iavor is not the original author of “The Evolution of a Programmer.” A quick web search suggests that there are thousands of copies floating around and it appears (unattributed) in humor newsgroups as far back as 1995. But I suspect some version of it goes back much further than that. Of course, if anyone does know who wrote the original, please let me know so that I may credit them here.

But seriously, folks, ...

On a more serious note, I think that the basic idea of the joke (successive variations on a theme, building in complexity) can serve a good pedagogical purpose as well as a humorous one. To that end, and for those who may not be familiar with all of the ideas represented above, I offer the following comments on the variations:

The first version (straight recursion with conditionals) is probably familiar to programmers of all stripes; fans of LISP and Scheme will find the sophomore version especially readable, except for the funny spelling of “lambda” and the absence of “define” (or “defun”). The use of patterns may seem only a slight shift in perspective, but in addition to mirroring mathematical notation, patterns encourage the view of data types as initial algebras (or as inductively defined).

The use of more “structural” recursion combinators (such as foldr and foldl) is square in the spirit of functional programming: these higher-order functions abstract away from the common details of different instances of recursive definitions, recovering the specifics through function arguments. The “points-free” style (defining functions without explicit reference to their formal parameters) can be compelling, but it can also be over-done; here the intent is to foreshadow similar usage in some of the later, more stridently algebraic variations.

The accumulating-parameter version illustrates a traditional technique for speeding up functional code. It is the second fastest implementation here, at least as measured in terms of number of reductions reported by Hugs, with the iterative versions coming in third. Although the latter run somewhat against the spirit of functional programming, they do give the flavor of the functional simulation of state as used in denotational semantics or, for that matter, in monads. (Monads are woefully un-represented here; I would be grateful if someone could contribute a few (progressive) examples in the spirit of the development above.) The continuation-passing version recalls a denotational account of control (the references are to Steele’s RABBIT compiler for Scheme and the SML/NJ compiler).

The fixed-point version demonstrates that we can isolate recursion in a general Y combinator. Thecombinatory version provides an extreme take on the points-free style inspired by Combinatory Logic, isolating dependence on variable names to the definitions of a few combinators. Of course we could go further, defining the Naturals and Booleans in combinatory terms, but note that the predecessor function will be a bit hard to accomodate (this is one good justification for algebraic types). Also note that we cannot define the Y combinator in terms of the others without running into typing problems (due essentially to issues of self-application). Interestingly, this is the fastest of all of the implementations, perhaps reflecting the underlying graph reduction mechanisms used in the implementation.

The list-encoded version exploits the simple observation that we can count in unary by using lists of arbitrary elements, so that the length of a list encodes a natural number. In some sense this idea foreshadows later versions based on recursive type definitions for Peano’s naturals, since lists of units are isomorphic to naturals. The only interesting thing here is that multiplication (numeric product) is seen to arise naturally out of combination (Cartesian product) by way of cardinality. Typing issues make it hard to express this correspondence as directly as we’d like: the following definition of listprod would break the definition of the facl function due to an occurs-check/infinite type:

```
listprod xs ys = [ (x,y) | x<-xs, y<-ys ]
```

Of course we could also simplify as follows, but only at the expense of obscuring the relationship between the two kinds of products:

```
listprod xs ys = [ arb | x<-xs, y<-ys ]
```

The interpretive version implements a small object language rich enough to express factorial, and then implements an interpreter for it based on a simple environment model. Exercises along these lines run all through the latter half of the Friedman, Wand and Haynes text ([6]), albeit expressed there in Scheme. We used to get flack from students atOberlin when we made them implement twelve interpreters in a single week-long lab, successively exposing more of the implementation by moving the real work from the meta-language to the interpreter. This implementation leaves a whole lot on the shoulders of the meta-language, corresponding to about Tuesday or Wednesday in their week. Industrious readers are invited to implement a compiler for a Squiggol-like language of polytypic folds and unfolds, targeting (and simulating) a suitable categorical abstract machine (see [9]), and then to implement factorial in that setting (but don't blame me if it makes you late for lunch ...).

The statically-computed version uses type classes and *functional dependencies* to facilitate computation at compile time (the latter are recent extensions to the Haskell 98 standard by Mark Jones, and are available in Hugs and GHC). The same kinds of techniques can also be used to encode behaviors more often associated with dependent types and polytypic programming, and are thus a topic of much recent interest in the Haskell community. The code shown here is based on an account by Thomas

Hallgren (see [7]), extended to include factorial. Prolog fans will find the definitions particularly easy to read, if a bit backwards.

The first of the “graduate” versions gets more serious about recursion, defining natural numbers as a recursive algebraic datatype and highlighting the difference between iteration and primitive recursion. The “origamist” and “cartesian” variations take a small step backwards in this regard, as they return to the use of internal integer and list types. They serve, however, to introduce anamorphic and hylomorphic notions in a more familiar context.

The “Ph.D” example employs the categorical style of BMF/Squiggol in a serious way (we could actually go a bit further, by using co-products more directly, and thus eliminate some of the overt dependence on the “internal sums” of the data type definition mechanism).

By the time we arrive at the “pièce de résistance”, the comonadic version of Uustalu, Vene and Pardo, we have covered most of the underlying ideas and can (hopefully) concentrate better on their specific contributions. The final version, using the Prelude-defined product function and ellipsis notation, is how I think the function is most clearly expressed, presuming some knowledge of the language and Prelude definitions. (This definition also dates back at least to David Turner’s KRC* language [5].)

It is comforting to know that the Prelude ultimately uses a recursion combinator (foldl', the strict version of foldl) to define product. I guess we can all hope to see the day when the Prelude will define gcatamorphic, zygomorphic and paramorphic combinators for us, so that factorial can be defined both conveniently *and* with greater dignity :) .

* *KRC may or may not be a trademark of Research Software, Ltd.,*
but you can bet your sweet bippy that Miranda ™ *is!*

Revision history

- *Sometime in the 2010’s*: corrected the attribution for the *Church of the LFP* creed, which is due to Calvin Ostrum (and added a link to an illuminated version; see above). Also added a link to a Serbo-Croatian version of this page, although I was later told that this kindly-provided service is some sort of click-bait scheme. I don’t know what to make of all that: if the translation itself is somehow horribly wrong, I suppose I could cut the link back out. But perhaps the whole incident merely suggests that I am deteriorating into a clueless old fart who can‘t make sense of all these new-fangled Internet scams, which is probably becoming more true all the time. (Extra points to anyone who combines Dynamic Logic with Denotational Semantics and Fuzzy Logic to develop a system in which to explore the notion of propositions “becoming true” slowly, over time. I can also see potential applications to sorities paradoxes involving greying hair ... .)

- *20 August 01*: added the interpretive version, based on an environment model of a small object language (no, not in *that* sense of object ...). I’m thinking about re-arranging the order of the examples, so that longer ones that are not part of the main line of development don't intrude so much. I also advertised the page on the Haskell Café mailing list and requested that a link be added to the Haskell humor page. Finally, I have an interesting new example in the works that may actually have some original research value; more on this soon.

- *14 August 01 (afternoon)*: added the combinatory version, now the fastest of the bunch, as measured in number of reductions reported by Hugs.

- *14 August 01 (morning)*: adjusted the sophomore/Scheme version to use an explicit "lambda" (though we spell it differently in Haskell land) and added the fixed-point version.

- *10 August 01*: added the list-encoding and static computation versions (the latter uses type classes and functional dependencies to compute factorial during type-checking; it is an extended version of code from Thomas Hallgren’s “Fun with Functional Dependencies” [7]).

- *1 August 01*: added accumulating-parameter and continuation-passing versions (the latter is a revised transliteration from Friedman, Wand and Haynes’ “Essentials of Programming Languages” [5]).

- *11 July 01*: date of the original posting.

References

1. *Highlights from nhc - a Space-efficient Haskell Compiler,* Niklas Röjemo. In the **FPCA ‘95 proceedings**. ACM Press, 1995 (see also CiteSeer or Chalmers ftp archive)

2. *n+k patterns,* Lennart Augustsson. Message to the haskell mailing list, Mon, 17 May 93 (see the mailing list archive)

3. *Sorting Morphisms,* Lex Augusteijn. In **Advanced Functional Programming**, (LNCS 1608). Springer-Verlag, 1999 (see also CiteSeer).

4. *Recursion Schemes from Comonads,* T. Uustalu, V. Vene and A. Pardo. **Nordic Journal of Computing**, to appear (see also Tarmo Uustalu’s papers page).

5. *Recursion Equations as a Programming Language,* D. A. Turner. In **Functional Programming and its Applications**. Cambridge University Press, 1982.

6. *Essentials of Programming Languages,* D. Friedman, M. Wand and C. Haynes. MIT PRess and McGraw-Hill, 1994.

7. *Fun with Functional Dependencies,* Thomas Hallgren. **Joint Winter Meeting of the Departments of Science and Computer Engineering**, Chalmers University of Technology and Göteborg University, Varberg, Sweden, 2001 (available at the author’s web archive).

8. *The Church of the Least Fixed-Point,* ~~authour unknown~~ by Calvin Ostrum, Feburary 1983. A bit of lambda calculus humor which circulated in the mid-1980’s (at least that’s when I saw it), probably from the comp.lang.functional newsgroup or somesuch. (Thanks to Ron, and perhaps another few kind souls, for bringing the correct attribution to my attention. I apparently created the following “illuminated manuscript” version of the *Church of the LFP* creed somewhere along the way.)

9. *Categorical Combinators, Sequential Algorithms and Functional Programming,* Pierre-Louis Curien. Springer Verlag (2nd edition), 1993.