Pattern Functors

Fixed points / Free monads / Generics

Hello!

I am Alejandro Serrano

Also known as **@trupill** in Twitter, and **serras** in GitHub

PhD student at Utrecht University in The Netherlands

Daily user of Haskell (sometimes Idris, too)

Author of *Beginning Haskell*

Past contributor to EclipseFP and ghc-mod

First-class functions

Use as arguments and return values

```
map :: (a -> b) -> [a] -> [b]
map f = \lst -> case lst of
                  []   -> []
                  x:xs -> f x : map f xs
```

Combine functions to get new ones

```
(***) :: (a -> b) -> (a' -> b') -> (a,a') -> (b, b')
(&&&) :: (a -> b) -> (a  -> b') -> a      -> (b, b')
```

Speak about functions without giving a name

```
map (\x -> x * x) [0 .. 10]
```

First-class data types?

Types depending on types (parametric polymorphism)

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

Functions at type level (type families)

```
type family   Element container :: *
type instance Element [a]       = a
type instance Element (Tree a)    = a
type instance Element ByteString = Word8
```

What we have

What we are missing

Combining data types

Generating new data types with some structure

Speaking about data types by their shape

0

Fixed points

Setting up the stage

```haskell
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)

fix :: (a -> a) -> a
fix f = let x = f x in x

factorial' :: (Int -> Int) -> Int -> Int
factorial' f 0 = 1
factorial' f n = n * f (n-1)

factorial = fix factorial'
```

Fixed point combinator

```
data ArithExpr = Literal Integer
               | Plus  ArithExpr ArithExpr
               | Times ArithExpr ArithExpr

newtype Fix f = In { out :: f (Fix f) }

data ArithExpr' r = Literal' Integer
                  | Plus'  r r
                  | Times' r r

type ArithExpr = Fix ArithExpr'
```

Fixed point combinator

Pattern functor

Values require In

```
anExpr = Plus (Literal 1)
              (Times (Literal 2) (Literal 3))
```

```
anExpr = In (Plus' (In (Literal' 1))
                   (In (Times' (In (Literal' 2))
                               (In (Literal' 3)))))
```

Pattern synonyms are available since GHC 7.8

```
pattern Literal n = In (Literal' n)
pattern Plus  a b = In (Plus'  a b)
pattern Times a b = In (Times' a b)
```

Why pattern functors?

```
instance Functor ArithExpr' where
  fmap f (Literal' n) = Literal' n
  fmap f (Plus'  x y) = Plus'  (f x) (f y)
  fmap f (Times' x y) = Times' (f x) (f y)
```

This construction always yields a Haskell `Functor`

Decorated data types

```
newtype AnnFix f a = AnnIn (f (AnnFix f a), a)
                       deriving Functor

annExpr :: AnnFix ArithExpr' String
annExpr = AnnIn (Plus' (AnnIn (Literal' 1, "uno"))
   (AnnIn (Times' (AnnIn (Literal' 2, "dos"))
                  (AnnIn (Literal' 3, "tres")),
          "por")), "mas")
```

The pattern functor does not change

Quick recap of folds

```
foldr :: (a -> b -> b)  -- What to do on (:)
      -> b              -- What to do on []
      -> [a]            -- The list to fold over
      -> b              -- The result
foldr _ e []  = e
foldr f e (x:xs) = f x (foldr f e xs)

foldA :: (Integer -> r)  -- What to do on Literal
      -> (r -> r -> r)   -- What to do on Plus
      -> (r -> r -> r)   -- What to do on Times
      -> ArithExpr -> r
foldA l _ _ (Literal n) = l n
foldA l p t (Plus  a b) = p (foldA l p t a) (foldA l p t b)
foldA l p t (Times a b) = t (foldA l p t a) (foldA l p t b)
```

Generic fold

Algebra

```haskell
gfold :: Functor p => (p r -> r) -> Fix p -> r
gfold f (In x) = f (fmap (gfold f) x)

{-# LANGUAGE LambdaCase #-}
eval :: ArithExpr -> Integer
eval = gfold (\case Literal' n -> n
                    Plus'   x y -> x + y
                    Times' x y -> x * y)
```

1

Data types à la carte

How do I smash constructors together?

Our aim: extensibility

```
data ArithLit r = Literal' Integer
data ArithOps r = Plus' r r | Times' r r

type ArithExpr1 = Fix (ArithLit :+: ArithOps)

data ArithInv r = Minus' r r | By' r r

type ArithExpr2 = Fix (ArithLit :+: (ArithOps :+: ArithInv))
```

How should :+: look?

It should tie two functors together

```
data (f :+: g) a = GoL (f a) | GoR (g a)
```

The result should still be a functor

```
instance (Functor f, Functor g)
  => Functor (f :+: g) where
  fmap f (GoL x) = GoL (fmap f x)
  fmap f (GoR y) = GoR (fmap f x)
```

Note the different types involved in fmap

Values require GoL/R

```
anExpr = In (Plus' (In (Literal' 1))
                   (In (Times' (In (Literal' 2))
                               (In (Literal' 3)))))

anExpr = In (GoR (Plus' (In (GoL (Literal' 1)))
   (In (GoR (Times' (In (GoL (Literal' 2)))
                    (In (GoL (Literal' 3))))))))
```

Different GoL/R

Problem

Different combinations of :+: require different GoL/R

Solution

Injections `inj :: f a -> (f :+: g) a`

Details of the construction

- *Data Types à la Carte*, by W. Swierstra
- *Composing and Decomposing Data Types*, by P. Bahr

Extensible functions...

Rough pseudo-code

```
eval :: r -> Integer for r = ArithLit
eval (Literal' n) = n

eval :: r -> Integer for r = ArithOps
eval (Plus'  a b) = eval a + eval b
eval (Times' a b) = eval a * eval b
```

... become true!

```
class Functor p => Evalable p where
  eval_ :: p Integer -> Integer
```

Algebra

```
instance Evalable ArithLit where
  eval_ (Literal' n) = n
instance Evalable ArithLit where
  eval_ (Plus'  a b) = a + b
  eval_ (Times' a b) = a * b

instance (Evalable f, Evalable g) => Evalable (f :+: g) where
  eval_ (InL x) = eval_ x
  eval_ (InR y) = eval_ y
```

Distributing work between branches

```
eval :: Evalable f => Fix f -> Integer
eval (Fix x) = eval_ (fmap eval x)
```

… are really extensible

```
data ArithExp r = Exp' r r

instance Evalable ArithExp where
  eval_ (Exp' a b) = a ^ b

-- No change required to :+: instance
-- No change required to eval
```

The Expression Problem

	Add constructor	Add function
OO	Easy: subclass	**Needs modification** Extension methods
FP	**Needs modification** Data types à la carte	Easy: define it

2

Free Thingies

Creating a Monad out of thin air (and a Functor)

```
class X where
```

A **free** X is a parametric data type *other conditions apply for *the* free X

```
data FreeX a = …
```

for which you can define an instance

```
instance Conditions a => X (FreeX a)
```

with the **less** amount of conditions

If Conditions is again a concept Y we say

Free X over a Y

Free monoid over a type

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m

instance {- empty => -} Monoid (FreeMonoid a)

data [a] = [] | a : [a]
instance {- empty => -} Monoid [a] where
  mempty  = []
  mappend = (++)
```

Free monad, attempt #1

```haskell
instance Conditions f => Monad (FreeMonad f) where
  return :: a -> FreeMonad f a
  (>>=)  :: FreeMonad f a -> (a -> FreeMonad f b)
         -> FreeMonad f b

newtype Fix f = In { out :: f (Fix f) }

gfold :: (f (Fix f) -> Fix f) -> Fix f -> Fix f
gfold f (In x) = f (fmap (gfold f) x)
```

- `gfold` does not change its type like `(>>=)`
- we cannot implement `return`

Let's poke a hole

```haskell
newtype Hole a x = Hole a deriving Functor

type FreeMonad f a = Fix (f :+: Hole a)

instance Functor f => Monad (FreeMonad f) where
  return x = In (GoR (Hole x))
  (In (GoR x)) >>= f = f x
  (In (GoL x)) >>= f = In (GoL (fmap (>>= f) x))
```

- Unfortunately, this is not a valid definition
- Haskell does not allow partially-applied synonyms

Free monad over a functor

```
data FreeMonad f a = Return a
                   | Do (f (FreeMonad f a))

instance Functor f => Monad (FreeMonad f) where
  return x = Return x
  (Return x) >>= f = f x
  (Do     x) >>= f = Do (fmap (>>= f) x)
```

Representing operations

```
data KeyValuePrim k v r = Get k (v -> r)
                        | Put k v r  deriving Functor

type KeyValueM k v a = FreeMonad (KeyValuePrim k v) a

example :: KeyValueM Int String String
example = Do (Put 3 "tres" (Do (Get 3 (\x -> Return x))))

get     :: k      -> KeyValueM k v v
get k    = Do (Get k Return)
put     :: k -> v -> KeyValueM k v ()
put k v = Do (Put k v (Return ()))

example = do put 3 "tres"
             get 3
```

Interpreting operations

Real operations

```haskell
runKV :: KeyValueStoreConnection -> KeyValueM k v a -> IO a
runKV _    (Return x)           = return x
runKV conn (Do (Get k rest))    = do v <- kvStoreGet conn k
                                     runKV (rest v)
runKV conn (Do (Put k v rest)) = do kvStorePut conn (k,v)
                                     runKV rest
```

Mocking operations: great for testing!

```haskell
mockKV :: Eq k => KeyValueM k v a -> [(k,v)] -> (a, [(k,v)])
mockKV (Return x)        s = (x, s)
mockKV (Do (Get k r))    s = let Just v = lookup k s in mockKV (r v) s
mockKV (Do (Put k v r)) s = let new = (k,v) : filter ((/= k) . fst) s
                                 in mockKV rest new
```

Two lessons to be learned

1. Design pattern: represent & interpret
   a. Representations tend to be instances of a concept
   b. Interpretations are folds
2. Haskell gives you a lot of things for free!
   a. Free monoids
   b. Free monads
   c. Free applicatives, alternatives, comonads...
   d. Look at the `free` package from Edward Kmett

3

Generics

The magic behind `deriving Show`

Combining pattern functors

Sum Choice	`data (f :+: g) a = GoL (f a)` `                 | GoR (g a)`
Constant	`newtype Hole x a = Hole x`
Product Both	`data (f :*: g) a = f a :*: g a`
Identity Recursion	`newtype Id a = Id a`
Unit No info	`data Unit a = Unit`

Description of
a constructor

Anonymous data types

```
type ListP a = Unit
          :+: (Hole a :*: Id)
type List  a = Fix (ListP a)

pattern []       = InL Unit
pattern (x:xs) = InR (Hole x :*: Id xs)

type ArithExprP = Hole Integer
            :+: (Id :*: Id)
            :+: (Id :*: Id)
type ArithExpr  = Fix ArithExpr
```

GHC.Generics Fancier, shorter names

<table>
<tr><td>Sum
Choice</td><td><code>data (f :+: g) p = L1 (f p)</code>
<code> | R1 (g p)</code></td></tr>
<tr><td>Constant</td><td><code>newtype K1 c p = K1 c</code></td></tr>
<tr><td>Product
Both</td><td><code>data (f :*: g) p = f p :*: g p</code></td></tr>
<tr><td>Identity
Recursion</td><td>No special type, use <code>K1 p</code></td></tr>
<tr><td>Unit
No information</td><td><code>data U1 p = U1</code></td></tr>
</table>

Generic representations

```
class Generic a where
  type Rep a :: * -> *
  from :: a -> Rep a x
  to   :: Rep a x -> a

instance Generic [a] where
  type Rep [a] = U1 :+: (K1 a :*: K1 [a])
  from []     = L1 U1
  from (x:xs) = R1 (K1 x :*: K1 xs)
  to (L1 U1)                   = []
  to (R1 (K1 x :*: K1 xs)) = x:xs
```

- GHC.Generics **does not use** Fix

deriving Generic

```haskell
{-# LANGUAGE DeriveGeneric #-}
data ArithExpr = Literal n
               | Plus  ArithExpr ArithExpr
               | Times ArithExpr ArithExpr
               deriving Generic
```

Data type-generic (==)

```
class GEq p where geq_ :: p a -> p a -> Bool

geq :: (Generic a, GEq (Rep a)) => a -> a -> Bool
geq x y = geq_ (from x) (from y)

instance GEq U1 where
  geq_ U1 U1 = True
instance GEq c => GEq (K1 c) where
  geq_ (K1 x) (K1 y) = geq_ x y
instance (GEq f, GEq g) => GEq (f :+: g) where
  geq_ (L1 x) (L1 y) = geq_ x y
  geq_ (R1 x) (R1 y) = geq_ x y
  geq_ _      _      = False
instance (GEq f, GEq g) => GEq (f :*: g) where
  geq_ (x1 :*: x2) (y1 :*: y2) = geq_ x1 y1 && geq_ x2 y2
```

Data types à la carte approach

Metadata

How can I implement Show?

```
newtype M1 i c f p = M1 (f p)

> :kind! (Rep Bool)
(Rep Bool) :: * -> *
= M1 D GHC.Generics.D1Bool
       (    M1 C GHC.Generics.C1_0Bool U1
       :+: M1 C GHC.Generics.C1_1Bool U1)
> datatypeName (from True)
"Bool"
> let M1 (R1 unwrapped) = from True in conName unwrapped
"True"
```

More applications

- Eq, Functor, Traversable with generic-deriving
- JSON (de)serialization using generic-aeson
  - And validation using json-schema
- Binary (de)serialization with both binary and cereal
- Deep evaluation using generic-deepseq
- Tree regular expressions with my t-regex package

Summary

1. Data types as first-class objects
2. Data type = pattern functor + fixed point
3. Extensibility of data type and functions
    a. Using data types à la carte approach
    b. Embodied in `GHC.Generics`
4. Free thingies = functor + extra structure
5. Design pattern: represent & interpret

Homework (optional?)

- Read *Data Types à la Carte* and *Composing and Decomposing Data Types*
- Wander around the `free` package
- Browse the docs for `GHC.Generics` and read *A Generic Deriving Mechanism for Haskell*
- Look at `compdata` set of packages
- Download by `t-regex` library

Write less code, more extensibly!

Thanks!

You can find me at:

- The rest of LambdaConf'15
- @trupill
- github.com/serras
- trupill@gmail.com

Credits

Special thanks to all the people who made and released these awesome resources for free:

- Presentation template by SlidesCarnival
- Photographs by Unsplash

Presentation design

This presentation uses the following typographies and colors:

- Titles: Nixie One
- Body copy: Muli

You can download the fonts on this page:

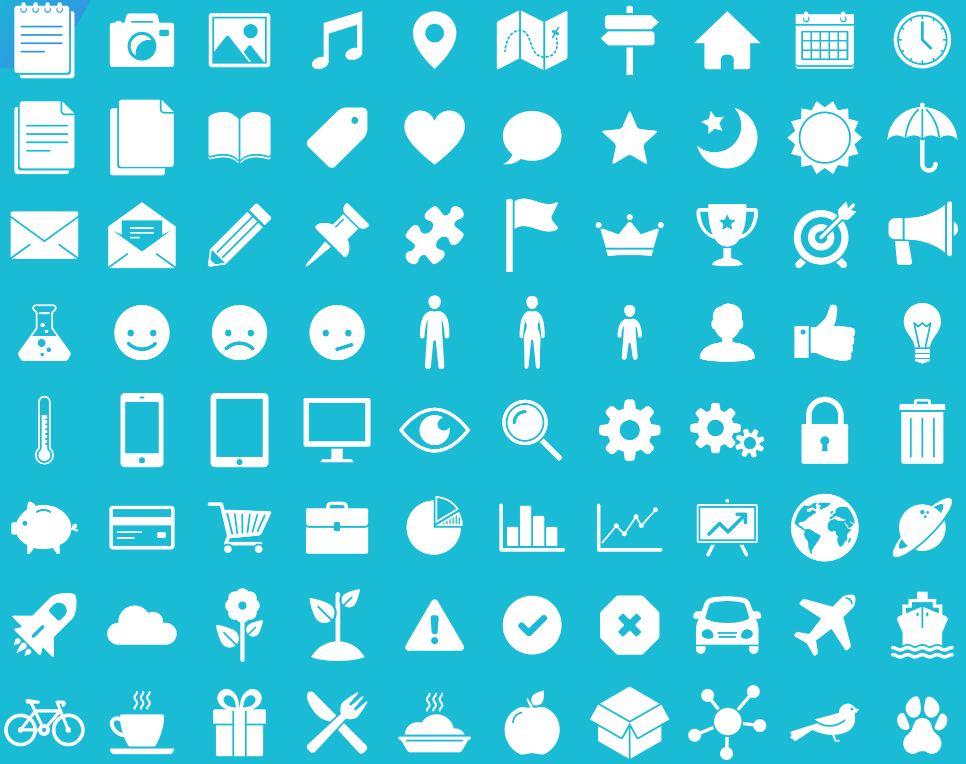
http://www.google.com/fonts#UsePlace:use/Collection:Nixie+One|Muli:300,400,300italic,400italic

Click on the “arrow button” that appears on the top right 

Aquamarina **#00e1c6** / Turquoise **#19bbd5** / Skyblue **#2c9dde** /

Light gray **#c6daec** / Dark blue **#0e293c**

You don’t need to keep this slide in your presentation. It’s only here to serve you as a design guide if you need to create new slides or download the fonts to edit the presentation in PowerPoint®



SlidesCarnival icons are editable shapes.

This means that you can:

- Resize them without losing quality.
- Change fill color and opacity.
- Change line color, width and style.

Isn't that nice? :)

Examples: