Applicative Functors

Bernie Pope

Outline

- Functors
- A problem
- A solution
- Compared to Functors and Monads
- Expressiveness
- Application in parser combinators

Functors

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b

instance Functor [] where
   fmap _ [] = []
   fmap g (x:xs) = g x : fmap g xs

instance Functor Maybe where
   fmap _ Nothing = Nothing
   fmap g (Just a) = Just (g a)
```

Functor Laws

```
fmap id = id

fmap (g . h) = fmap g . fmap h
```

A problem

- What if you want to fmap a function or arity higher than one?

- `fmap (+) [1,2,3] :: Num a => [a -> a]`

- For some Functor `f`, if the mapped function has type `(t1 -> t2 ... -> tn)` then we end up with a result of type `f (t2 -> ... -> tn)`

- The problem is, what can we do with that result, now that the function is in the context of `f`?

A solution

```
class Functor f => Applicative f where
   pure :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b

instance Applicative [] where
   pure x = [x]
   gs <*> xs = [ g x | g <- gs, x <- xs ]
```

A solution

- The `<*>` operator gives us a way to use a function embedded in a context.
- The `pure` function gives us a way to put a value into a context.
- These operators can be built from existing monads.

Using monads where you find them

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap =  liftM2 id

liftM2  :: (Monad m) => (a -> b -> c) -> m a -> m b -> m c
liftM2 f m1 m2 = do { x1 <- m1; x2 <- m2; return (f x1 x2) }

instance Applicative Maybe where
   pure = return
   (<*>) = ap

instance Applicative [] where
   pure = return
   (<*>) = ap
```

Example

```
GHCi> (+) `fmap` [1,2,3] <*> [4,5]
[5,6,6,7,7,8]

GHCi> [x + y | x <- [1,2,3], y <- [4,5]]
[5,6,6,7,7,8]

GHCi> do { x <- [1,2,3]; y <- [4,5]; return (x + y) }
[5,6,6,7,7,8]
```

Compared to Monads and Functors

```
($)          ::                          (a -> b) -> a   -> b
fmap         :: Functor t      =>        (a -> b) -> t a -> t b
(<*>)        :: Applicative t => t       (a -> b) -> t a -> t b
flip (>>=)   :: Monad t       =>        (a -> t b) -> t a -> t b
```

Expressiveness

```
miffy :: Monad m => m Bool -> m a -> m a -> m a
miffy mb mt mf = do
   b <- mb
   if b then mt else mf

iffy :: Applicative f => f Bool -> f a -> f a -> f a
iffy fb ft ff = cond `fmap` fb <*> ft <*> ff
   where
   cond b t f = if b then t else f
```

Applications in parser combinators

```
child = do
   e <- element
   cd <- optionMaybe charData
   return (e, cd)
```

```
child = (,) <$> element <*> optionMaybe charData
```

Applications in parser combinators

```
(*>) :: Applicative a => a b -> a c -> a c

(<*) :: Applicative a => a b -> a c -> a b

elementPrefix =
   (,) <$> (string "<" *> name) <*> (spaces *> attributes)
```