All About Applicative

Adelbert Chang (@adelbertchang)

April 9, 2017

```haskell
{-# LANGUAGE NoImplicitPrelude #-}
import Prelude hiding ((<*>), Applicative, pure)

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

infixl 4 <*>
```

```haskell
instance Applicative (Either e) where
  Right f <*> Right a = Right $ f a
  Left e  <*> _       = Left e
  _       <*> Left e  = Left e
  pure                = Right
```

```haskell
instance Applicative (Either e) where
  Right f <*> Right a = Right $ f a
  Left e  <*> _       = Left e
  _       <*> Left e  = Left e
  pure                = Right

instance Applicative [] where
  (fh : ft) <*> fs = fmap fh fs ++ (ft <*> fs)
  _         <*> _  = []

  pure a           = [a]
```

```haskell
class Functor f => Monoidal f where
  (<**>) :: f a -> f b -> f (a, b)
  unit   :: a   -> f a

infixl 4 <**>
```

```haskell
class Functor f => Monoidal f where
  (<**>) :: f a -> f b -> f (a, b)
  unit   :: a   -> f a

infixl 4 <**>

fmap :: (Functor f) => (a -> b) -> f a -> f b
```

```haskell
class Functor f => Monoidal f where
  (<**>) :: f a -> f b -> f (a, b)
  unit   :: a   -> f a

infixl 4 <**>

fmap :: (Functor f) => (a -> b) -> f a -> f b

(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

```haskell
-- Applicative ~> Monoidal
ff <*> fa = fmap (uncurry ($)) (ff <**> fa)
pure      = unit

-- Monoidal ~> Applicative
fa <**> fb = (,) <$> fa <*> fb
unit       = pure
```

Associativity:

$$(fa \otimes fb) \otimes fc \sim fa \otimes (fb \otimes fc)$$

Associativity:

$$(fa \otimes fb) \otimes fc \sim fa \otimes (fb \otimes fc)$$

Identity:

$$fa \otimes unit()$$

$$\sim$$

$$unit() \otimes fa$$

$$\sim$$

$$fa$$

Associativity:

$$(a \times b) \times c \equiv a \times (b \times c)$$

Identity:

$$a \times 1 \equiv 1 \times a \equiv a$$

```haskell
import Control.Applicative (Const(..))

-- newtype Const a b = Const { getConst :: a }
```

```haskell
import Control.Applicative (Const(..))

-- newtype Const a b = Const { getConst :: a }

instance Monoid a => Monoidal (Const a) where
  fa <**> fb  = Const $ getConst fa `mappend` getConst fb
  unit        = const $ Const mempty
```

```
(fa <**> fb) <**> fc    ~    fa <**> (fb <**> fc)
```

```
(fa <**> fb) <**> fc    ~    fa <**> (fb <**> fc)

(Const $ getConst fa `mappend` getConst fb) <**> fc
```

```
(fa <**> fb) <**> fc    ~    fa <**> (fb <**> fc)

(Const $ getConst fa `mappend` getConst fb) <**> fc

(a  `mappend`  b)  <**> fc
```

```
(fa <**> fb) <**> fc    ~    fa <**> (fb <**> fc)

(Const $ getConst fa `mappend` getConst fb) <**> fc

(a  `mappend`  b)  <**> fc

(a  `mappend`  b)  `mappend`  c
```

```
(fa <**> fb) <**> fc    ~    fa <**> (fb <**> fc)

(Const $ getConst fa `mappend` getConst fb) <**> fc

(a  `mappend`  b)  <**> fc

(a  `mappend`  b)  `mappend`  c

 a  `mappend` (b   `mappend`  c)
```

```
(fa <**> fb) <**> fc    ~    fa <**> (fb <**> fc)

(Const $ getConst fa `mappend` getConst fb) <**> fc

(a  `mappend`  b)  <**> fc

(a  `mappend`  b)  `mappend`  c

 a  `mappend` (b   `mappend`  c)

fa <**> (fb <**> fc)
```

Questions?

```haskell
traverseL :: (Monoidal f) => (a -> f b) -> [a] -> f [b]
```

```haskell
traverseL :: (Monoidal f) => (a -> f b) -> [a] -> f [b]

traverseL f = foldr acc (unit [])
```

```haskell
traverseL :: (Monoidal f) => (a -> f b) -> [a] -> f [b]

traverseL f = foldr acc (unit [])

  where acc a bs = fmap (uncurry (:)) (f a <**> bs)
```

```
-- traverseL :: (Monoidal f) => (a -> f b) -> [a] -> f [b]
```

```haskell
-- traverseL :: (Monoidal f) => (a -> f b) -> [a] -> f [b]

-- f = IO
traverseLIO :: (a -> IO b) -> [a] -> IO [b]
```

```haskell
-- traverseL :: (Monoidal f) => (a -> f b) -> [a] -> f [b]

-- f = IO
traverseLIO :: (a -> IO b) -> [a] -> IO [b]

-- f = Either e
traverseLEither :: (a -> Either e b) -> [a] -> Either e [b]
```

```haskell
instance Monoidal (Either e) where
  Right a <**> Right b = Right (a, b)
  Left  e <**> _       = Left e
  _       <**> Left  e = Left e
  unit                 = Right
```

```haskell
instance Monoidal (Either e) where
  Right a <**> Right b = Right (a, b)
  Left  e <**> _       = Left e
  _       <**> Left  e = Left e
  unit                 = Right

ex0 = traverseL validate [2, 4, 6]
  where validate i = if i `mod` 2 == 0 then Left [i]
                                       else Right i

-- Left [2]
```

```haskell
data Validated e a = Failure e | Success a

instance Functor (Validated e) where
  fmap _ (Failure e) = Failure e
  fmap f (Success a) = Success $ f a
```

```haskell
import Data.Semigroup ((<>), Semigroup)

instance Semigroup e => Monoidal (Validated e) where
  Success a <**> Success b = Success (a, b)
  Failure e <**> Success _ = Failure e
  Success _ <**> Failure e = Failure e
```

```haskell
import Data.Semigroup ((<>), Semigroup)

instance Semigroup e => Monoidal (Validated e) where
  Success a <**> Success b = Success (a, b)
  Failure e <**> Success _ = Failure e
  Success _ <**> Failure e = Failure e

  Failure e <**> Failure f = Failure $ e <> f
```

```haskell
import Data.Semigroup ((<>), Semigroup)

instance Semigroup e => Monoidal (Validated e) where
  Success a <**> Success b = Success (a, b)
  Failure e <**> Success _ = Failure e
  Success _ <**> Failure e = Failure e

  Failure e <**> Failure f = Failure $ e <> f

  unit                     =  Success
```

```haskell
ex1 = traverseL validate [2, 4, 6]
  where validate i = if i `mod` 2 == 0 then Failure [i]
                                       else Success i

-- Failure [2,4,6]
```

Questions?

```haskell
class (Functor t, Foldable t) => Traversable t
  where traverse :: (Applicative f) =>
                    (a -> f b) -> t a -> f (t b)
```

```haskell
class (Functor t, Foldable t) => Traversable t
  where traverse :: (Applicative f) =>
                    (a -> f b) -> t a -> f (t b)

-- Functor
fmap :: (a - b) -> t a -> t b

-- Foldable
foldMap :: (Monoid m) => (a -> m) -> t a -> m
```

```haskell
-- traverse :: (Traversable t, Applicative f) =>
--             (a -> f b) -> t a -> f (t b)

fmapT :: (Traversable t) => (a -> b) -> t a -> t b
```

```haskell
-- traverse :: (Traversable t, Applicative f) =>
--             (a -> f b) -> t a -> f (t b)

fmapT :: (Traversable t) => (a -> b) -> t a -> t b

import Data.Functor.Identity (Identity(..))

fmapT f = runIdentity . traverse (Identity . f)
```

```haskell
-- traverse :: (Traversable t, Applicative f) =>
--             (a -> f b) -> t a -> f (t b)

foldMapT :: (Traversable t, Monoid m) =>
                    (a -> m) -> t a -> m
```

```haskell
-- traverse :: (Traversable t, Applicative f) =>
--             (a -> f b) -> t a -> f (t b)

foldMapT :: (Traversable t, Monoid m) =>
                    (a -> m) -> t a -> m

-- (a -> Const m b) -> t a -> Const m (t b)
```

```haskell
-- traverse :: (Traversable t, Applicative f) =>
--             (a -> f b) -> t a -> f (t b)

foldMapT :: (Traversable t, Monoid m) =>
                    (a -> m) -> t a -> m

-- (a -> Const m b) -> t a -> Const m (t b)

-- (a ->       m  ) -> t a ->       m
```

```haskell
-- traverse :: (Traversable t, Applicative f) =>
--             (a -> f b) -> t a -> f (t b)

foldMapT :: (Traversable t, Monoid m) =>
                    (a -> m) -> t a -> m

-- (a -> Const m b) -> t a -> Const m (t b)

-- (a ->       m  ) -> t a ->       m

foldMapT f = getConst . traverse (Const . f)
```

Questions?

```haskell
-- data Compose f g a = Compose (f (g a))

instance (Applicative f, Applicative g) =>
          Applicative (Compose f g)
```

```haskell
import Data.Functor.Product (Product(..))

-- data Product f g a = Pair (f a) (g a)

-- instance (Applicative f, Applicative g) =>
--          Applicative (Product f g)
```

```haskell
instance (Monoidal f, Monoidal g) =>
         Monoidal (Product f g) where
  -- (f a, g a) -> (f b, g b) -> (f (a, b), g(a, b))
  Pair fa ga <**> Pair fa' ga' =
    Pair (fa <**> fa') (ga <**> ga')

  unit a = Pair (unit a) (unit a)
```

```haskell
instance (Monoidal f, Monoidal g) =>
         Monoidal (Product f g) where
  -- (f a, g a) -> (f b, g b) -> (f (a, b), g(a, b))
  Pair fa ga <**> Pair fa' ga' =
    Pair (fa <**> fa') (ga <**> ga')

  unit a = Pair (unit a) (unit a)

instance (Monoid a, Monoid b) =>
         Monoid (a, b) where
  -- (a, b) -> (a, b) -> (a, b)
  (a, b) `mappend` (a', b') =
    (a `mappend` a', b `mappend` b')

  mempty = (mempty, mempty)
```

Questions?

Case study: Free Applicative algebras

EOF