

All About Applicative

Adelbert Chang (@adelbertchang)

April 9, 2017

```
{-# 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 <*>
```

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

```
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]
```

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

infixl 4 <*>
```

```
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
```

```
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
```

```
-- 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:

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

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

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

```
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
```

$(fa \text{ <*> } fb) \text{ <*> } fc \quad \sim \quad fa \text{ <*> } (fb \text{ <*> } fc)$

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

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

$(fa \lt ** \gt fb) \lt ** \gt fc \quad \sim \quad fa \lt ** \gt (fb \lt ** \gt fc)$

$(\text{Const } \$ \text{ getConst } fa \text{ `mappend` getConst } fb) \lt ** \gt fc$

$(a \text{ `mappend` } b) \lt ** \gt fc$

$(fa \text{ <*> } fb) \text{ <*> } fc \quad \sim \quad fa \text{ <*> } (fb \text{ <*> } fc)$

$(\text{Const } \$ \text{ getConst } fa \text{ `mappend` getConst } fb) \text{ <*> } fc$

$(a \text{ `mappend` } b) \text{ <*> } fc$

$(a \text{ `mappend` } b) \text{ `mappend` } c$

$(fa \text{ <*> } fb) \text{ <*> } fc \quad \sim \quad fa \text{ <*> } (fb \text{ <*> } fc)$

$(\text{Const } \$ \text{ getConst } fa \text{ `mappend` getConst } fb) \text{ <*> } fc$

$(a \text{ `mappend` } b) \text{ <*> } fc$

$(a \text{ `mappend` } b) \text{ `mappend` } c$

$a \text{ `mappend` } (b \text{ `mappend` } c)$

`(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?

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

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

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

```
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]
```

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

```
-- f = IO
```

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

```
-- 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]
```

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

```

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]

```

```
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
```

```
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
```

```
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
```



```
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
```

```
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?

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

```
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
```

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

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

```
-- 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)
```

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

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



```
-- 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)
```

```
-- 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
```

```
-- 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?

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

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

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

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

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

```
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 (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