Cofun with Cofree Comonads

Dave Laing

YOW! Lambda Jam 2015

Introduction

A cofree functor is right adjoint to a forgetful functor.

~~A cofree functor is right adjoint to a forgetful functor.~~

“Cofree comonads are something I could use if I wanted to”

“I want to use cofree comonads for things”

Fold and unfold for lists are a good starting point

Monadic values are produced in a context

Comonadic values are consumed from a context

Combining the two can have good results

Comonads

```haskell
class Monad m where
```

```haskell
class Comonad w where
```

```haskell
class Monad m where
  return    :: a -> m a
```

```haskell
class Comonad w where
```

```haskell
class Monad m where
  return    :: a -> m a
```

```haskell
class Comonad w where
  extract   :: w a -> a
```

```haskell
class Monad m where
  return    :: a -> m a
  (>>=)     :: m a -> (a -> m b) -> m b


class Comonad w where
  extract   :: w a -> a
```

```haskell
class Monad m where
  return    :: a -> m a
  (>>=)     :: m a -> (a -> m b) -> m b


class Comonad w where
  extract   :: w a -> a
  extend    :: w a -> (w a -> b) -> w b
```

```haskell
class Monad m where
  return    :: a -> m a
  (>>=)     :: m a -> (a -> m b) -> m b
  join      :: m (m a) -> m a

class Comonad w where
  extract   :: w a -> a
  extend    :: w a -> (w a -> b) -> w b
```

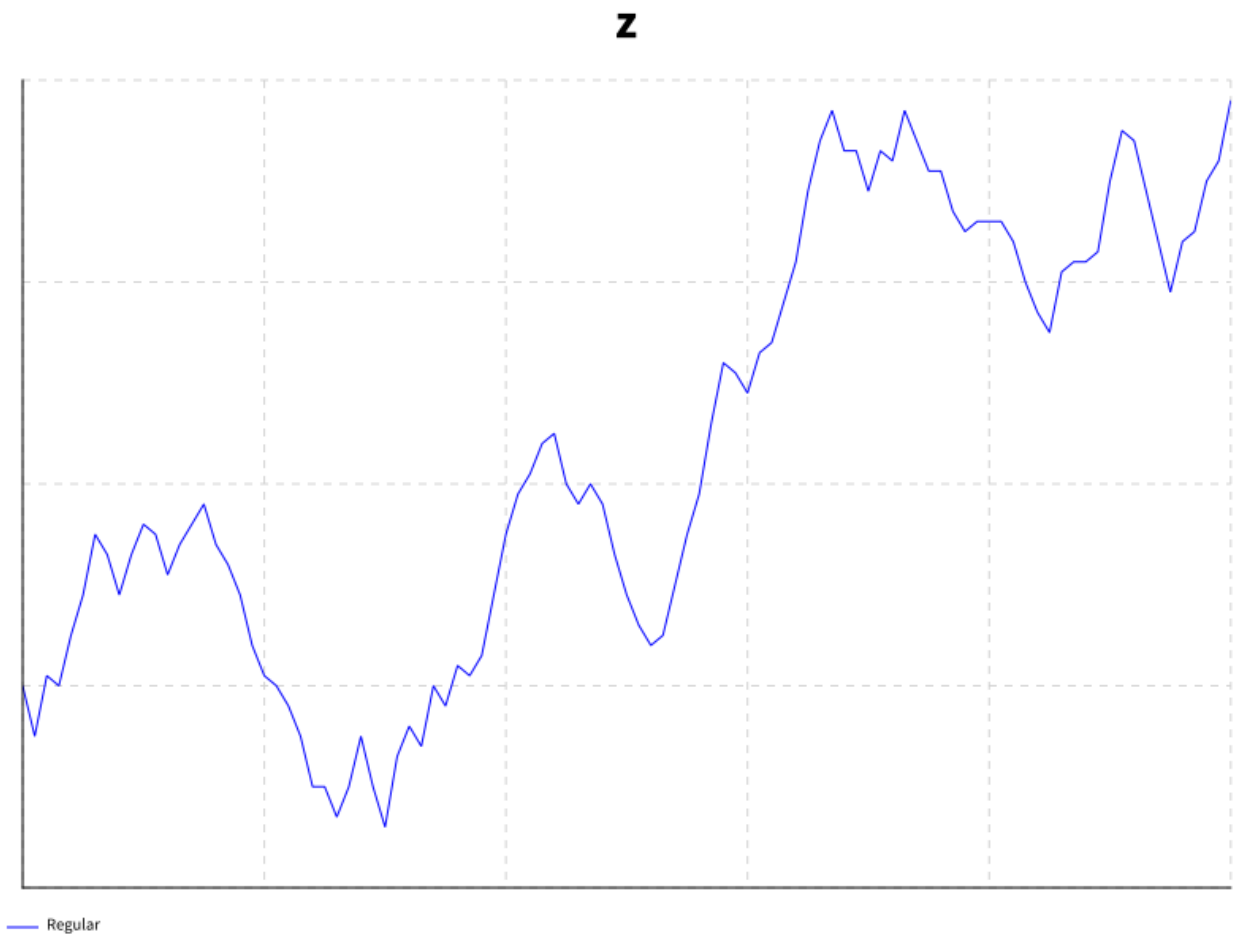
```haskell
class Monad m where
  return    :: a -> m a
  (>>=)     :: m a -> (a -> m b) -> m b
  join      :: m (m a) -> m a

class Comonad w where
  extract   :: w a -> a
  extend    :: w a -> (w a -> b) -> w b
  duplicate :: w a -> w (w a)
```

```haskell
data Zipper a = Zipper [a] a [a]
```

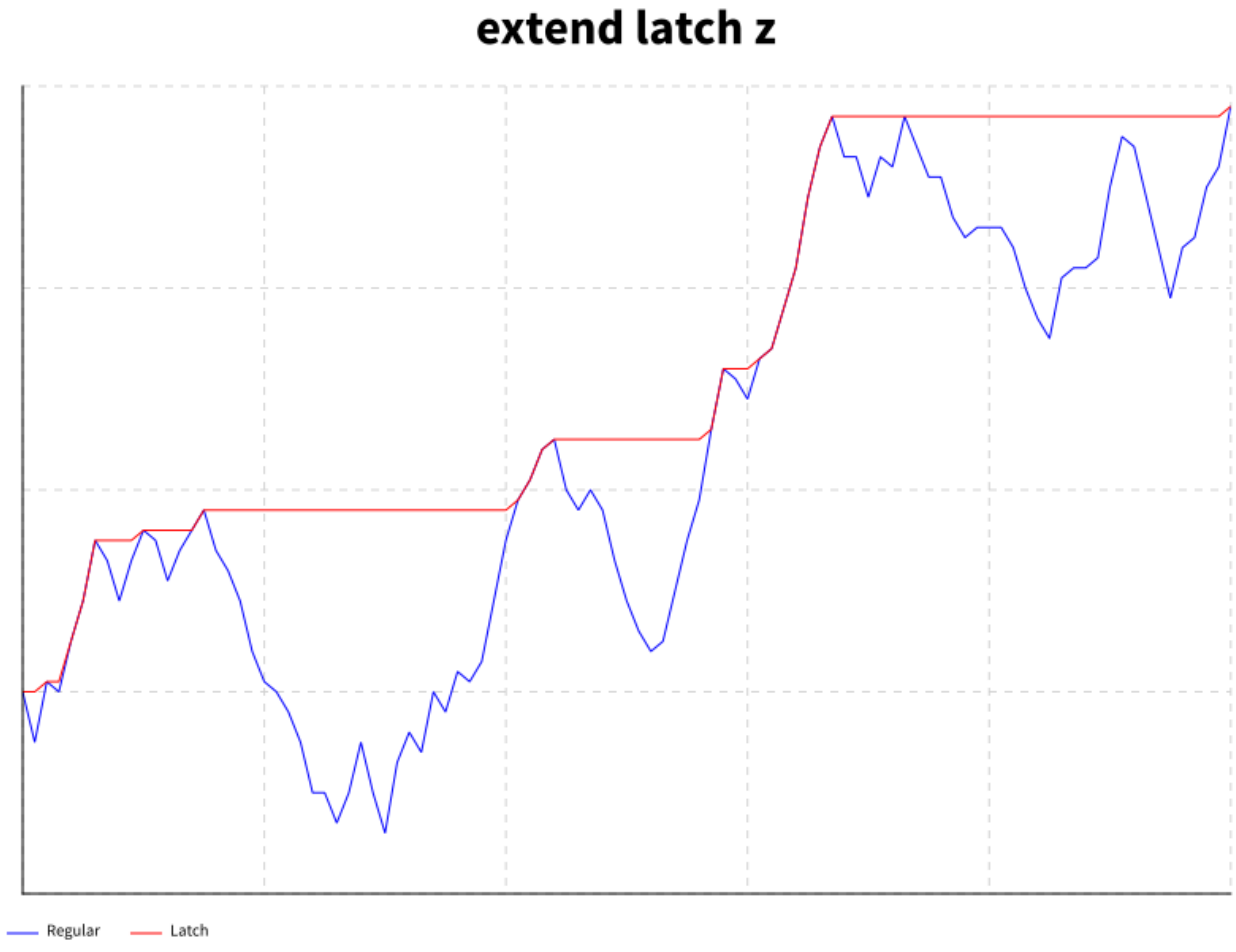
```haskell
z :: Zipper Int
z = Zipper [2, 1] 3 [4]

z
-- | 1 | 2 > 3 < 4 |
```

```haskell
z :: Zipper Int
z = Zipper [2, 1] 3 [4]

z
-- | 1 | 2 > 3 < 4 |

extract z
-- 3
```

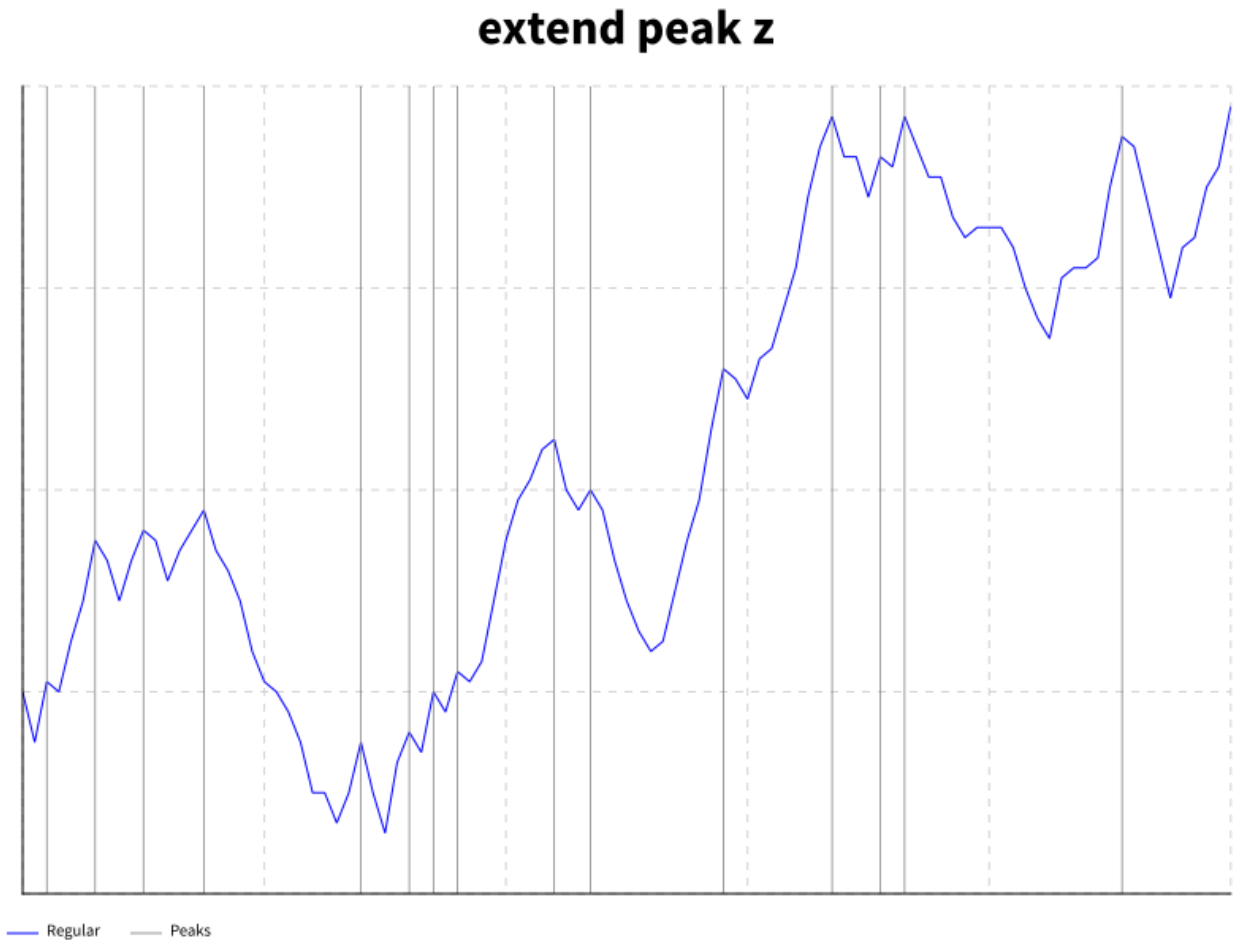
```haskell
z :: Zipper Int
z = Zipper [2, 1] 3 [4]

z
-- | 1 | 2 > 3 < 4 |

extract z
-- 3

duplicate z
-- | >1<2|3|4| |1>2<3|4| > |1|2>3<4| < | |1|2|3>4< |
```


z
Regular

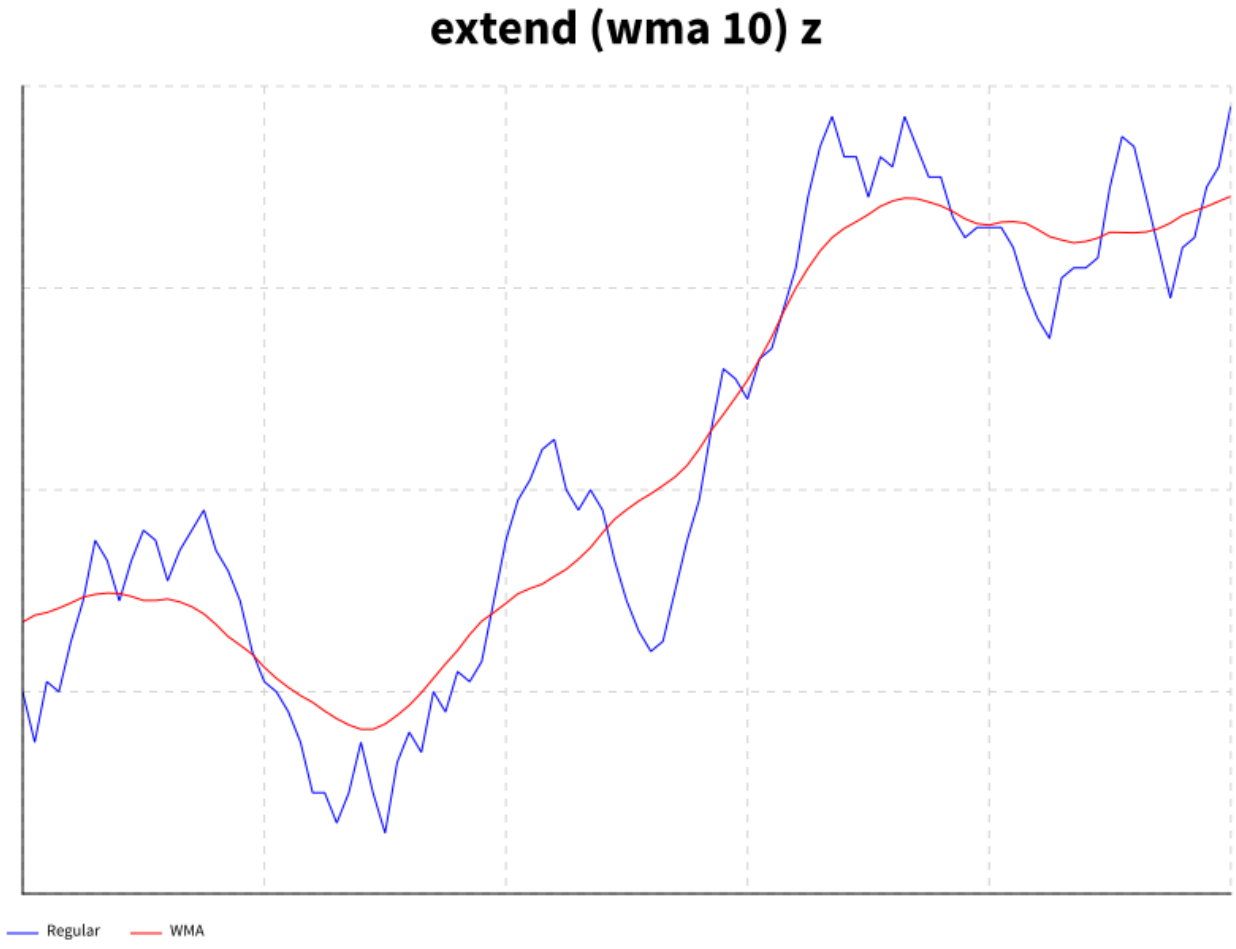
```haskell
latch :: Ord a => Zipper a -> a
latch (Zipper l f _) =
  maximumDef f l
```



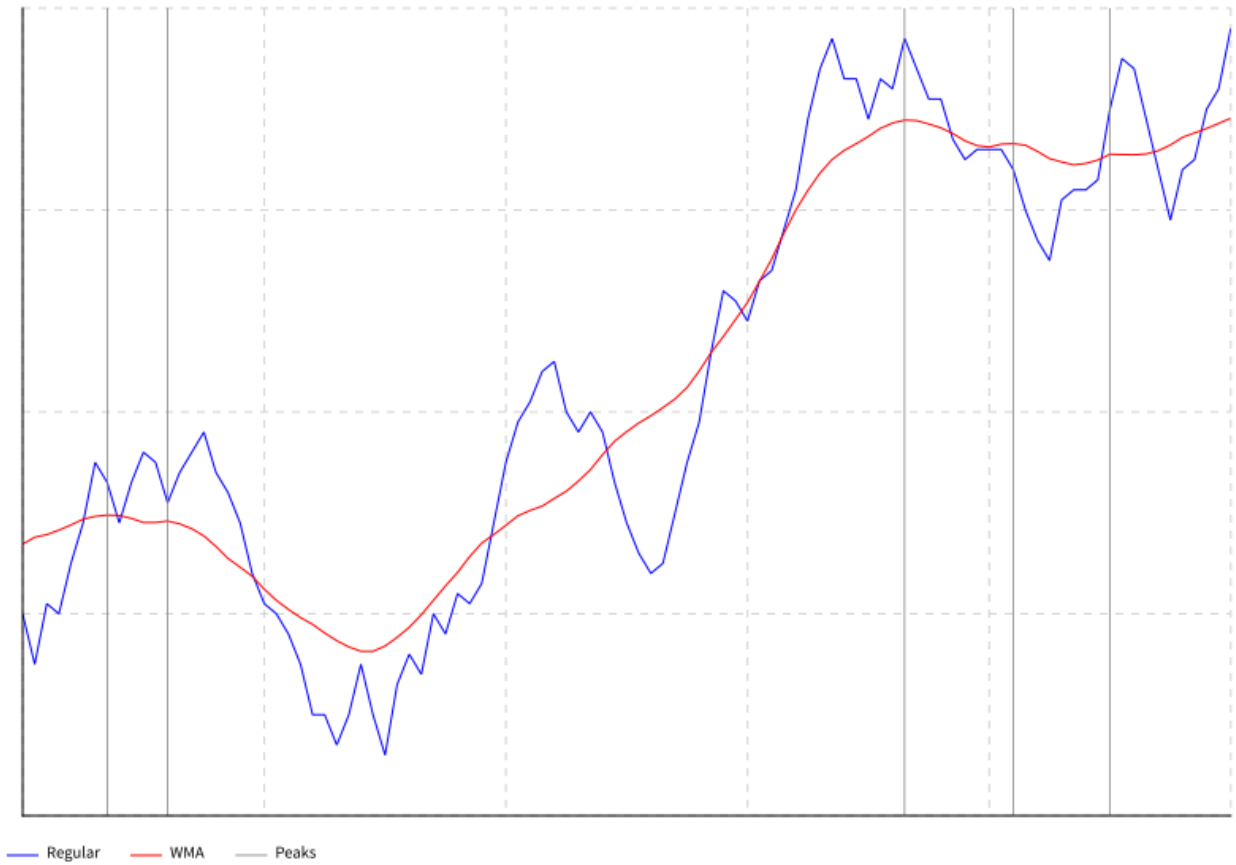
```haskell
peak :: Ord a => Zipper a -> Bool
peak (Zipper l f r) =
  headDef f l < f && f > headDef f r
```



```
wma ::  Int -> Zipper Double -> Double
wma n (Zipper l f r) =
  average $ take n l ++ f : take n r
```



extend peak . extend (wma 10) $ z



Free

```haskell
data Free f a =
    Pure a
  | Free (f (Free f a))

instance Functor f => Monad (Free f) where
  ...
```

```haskell
data AdderF k =
```

```haskell
data AdderF k =
    Add Int (Bool -> k)
```

```haskell
data AdderF k =
    Add Int (Bool -> k)
  | Clear k
```

```haskell
data AdderF k =
    Add Int (Bool -> k)
  | Clear k
  | Total (Int -> k)
```

```haskell
data AdderF k = ...
```

```haskell
instance Functor AdderF where
```

```haskell
data AdderF k =
    Add Int (Bool -> k)
    ...


instance Functor AdderF where
  fmap f (Add x k) = Add x (f . k)
```

```haskell
data AdderF k =
    Add Int (Bool -> k)
  | Clear k
  ...

instance Functor AdderF where
  fmap f (Add x k) = Add x (f . k)
  fmap f (Clear k) = Clear (f k)
```

```haskell
data AdderF k =
    Add Int (Bool -> k)
  | Clear k
  | Total (Int -> k)

instance Functor AdderF where
  fmap f (Add x k) = Add x (f . k)
  fmap f (Clear k) = Clear (f k)
  fmap f (Total k) = Total (f . k)
```

```haskell
data Free f a =
    Pure a
  | Free (f (Free f a))

instance Functor f => Monad (Free f) where
  ...
```

```haskell
type Adder a = Free AdderF a
```

```
type Adder a =
    Pure a
  | Free (AdderF (Adder a))
```

```
type Adder a =
    Pure a
  | Free (Add Int (Bool -> Adder a))
  | Free (Clear (Adder a))
  | Free (Total (Int -> Adder a))
```

Using Free

```haskell
add :: Int -> Adder Bool
add x = liftF $ Add x id

clear :: Adder ()
clear = liftF $ Clear ()

total :: Adder Int
total = liftF $ Total id
```

```
findLimit :: Adder Int
findLimit = do
```

```haskell
findLimit :: Adder Int
findLimit = do
  -- capture the old count
  t <- total
  -- clear the count
  clear
```

```haskell
findLimit :: Adder Int
findLimit = do
  -- capture the old count
  t <- total
  -- clear the count
  clear
  -- seek out the limit
  r <- execStateT findLimit' 0
```

```haskell
findLimit :: Adder Int
findLimit = do
  -- capture the old count
  t <- total
  -- clear the count
  clear
  -- seek out the limit
  r <- execStateT findLimit' 0
  -- restore the old count
  clear
  _ <- add t
```

```haskell
findLimit :: Adder Int
findLimit = do
  -- capture the old count
  t <- total
  -- clear the count
  clear
  -- seek out the limit
  r <- execStateT findLimit' 0
  -- restore the old count
  clear
  _ <- add t
  -- return the result
  return r
```

```
findLimit' :: StateT Int Adder ()
findLimit' = do
```

```haskell
findLimit' :: StateT Int Adder ()
findLimit' = do
  -- add 1 to the total
  r <- lift $ add 1
```

```haskell
findLimit' :: StateT Int Adder ()
findLimit' = do
  -- add 1 to the total
  r <- lift $ add 1
  -- check for overflow
  when r $ do
```

```haskell
findLimit' :: StateT Int Adder ()
findLimit' = do
  -- add 1 to the total
  r <- lift $ add 1
  -- check for overflow
  when r $ do
    -- if no overflow, add to our state counter ...
    modify (+ 1)
```

```haskell
findLimit' :: StateT Int Adder ()
findLimit' = do
  -- add 1 to the total
  r <- lift $ add 1
  -- check for overflow
  when r $ do
    -- if no overflow, add to our state counter ...
    modify (+ 1)
    -- and continue
    findLimit'
```

Separation of the client code from the interpreter code

Reuse code written in terms of the client code

Write different interpreters for different needs

Cofree

```haskell
data Cofree f a = a :< f (Cofree f a)

instance Functor f => Comonad (Cofree f)
  ...
```

```haskell
data CoAdderF k = CoAdderF {
```

```haskell
data CoAdderF k = CoAdderF {
    addH   :: Int -> (Bool, k)
```

```haskell
data CoAdderF k = CoAdderF {
    addH   :: Int -> (Bool, k)
  , clearH :: k
```

```haskell
data CoAdderF k = CoAdderF {
    addH   :: Int -> (Bool, k)
  , clearH :: k
  , totalH :: (Int, k)
  }
```

```haskell
data CoAdderF k = CoAdderF {
    ...
  }


instance Functor CoAdderF where
  fmap f (CoAdderF a c t) = CoAdderF
```

```haskell
data CoAdderF k = CoAdderF {
    addH   :: Int -> (Bool, k)
    ...
  }


instance Functor CoAdderF where
  fmap f (CoAdderF a c t) = CoAdderF
    (fmap (fmap f) a)
```

```haskell
data CoAdderF k = CoAdderF {
    addH   :: Int -> (Bool, k)
  , clearH :: k
    ...
  }

instance Functor CoAdderF where
  fmap f (CoAdderF a c t) = CoAdderF
    (fmap (fmap f) a)
    (f c)
```

```haskell
data CoAdderF k = CoAdderF {
    addH   :: Int -> (Bool, k)
  , clearH :: k
  , totalH :: (Int, k)
  }

instance Functor CoAdderF where
  fmap f (CoAdderF a c t) = CoAdderF
    (fmap (fmap f) a)
    (f c)
    (fmap f t)
```

```haskell
data Cofree f a = a :< f (Cofree f a)

instance Functor f => Comonad (Cofree f)
  ...
```

```haskell
type CoAdder a = Cofree CoAdderF a
```

```
type CoAdder a = a :< CoAdderF (CoAdder a)
```

```
type CoAdder a = a :< CoAdderF {
    Int -> (Bool, CoAdder a)
  , CoAdder a
  , (Int, CoAdder a)
  }
```

Using Cofree

```haskell
type Limit = Int
type Count = Int
```

```haskell
mkCoAdder :: Limit -> Count -> CoAdder (Limit, Count)
mkCoAdder limit count = _
```

```haskell
type Limit = Int
type Count = Int

coiter :: Functor f => (a -> f a) -> a -> Cofree f a

mkCoAdder :: Limit -> Count -> CoAdder (Limit, Count)
mkCoAdder limit count = _
```

```haskell
type Limit = Int
type Count = Int

coiter :: Functor f => (a -> f a) -> a -> Cofree f a

mkCoAdder :: Limit -> Count -> CoAdder (Limit, Count)
mkCoAdder limit count =
          coiter next start
```

```haskell
type Limit = Int
type Count = Int

coiter :: Functor f => (a -> f a) -> a -> Cofree f a

mkCoAdder :: Limit -> Count -> CoAdder (Limit, Count)
mkCoAdder limit count =
          coiter next start

  where
    next w = CoAdderF _ _ _
    start (limit, count)
```

```haskell
type Limit = Int
type Count = Int

coiter :: Functor f => (a -> f a) -> a -> Cofree f a

mkCoAdder :: Limit -> Count -> CoAdder (Limit, Count)
mkCoAdder limit count =
    start
      :< (coiter next <$> next start)
  where
    next w = CoAdderF _ _ _
    start (limit, count)
```

```haskell
type Limit = Int
type Count = Int

coiter :: Functor f => (a -> f a) -> a -> Cofree f a

mkCoAdder :: Limit -> Count -> CoAdder (Limit, Count)
mkCoAdder limit count =
    start :< next start
      :< (coiter next <$> next (next start))
  where
    next w = CoAdderF _ _ _
    start (limit, count)
```

```haskell
type Limit = Int
type Count = Int

coiter :: Functor f => (a -> f a) -> a -> Cofree f a

mkCoAdder :: Limit -> Count -> CoAdder (Limit, Count)
mkCoAdder limit count =
          coiter next start

  where
    next w = CoAdderF (coAdd w) (coClear w) (coTotal w)
    start (limit, count)
```

```
next w = CoAdderF (coAdd w) (coClear w) (coTotal w)

coClear :: (Limit, Count) -> (Limit, Count)


coTotal :: (Limit, Count) -> (Int, (Limit, Count))


coAdd :: (Limit, Count) -> Int -> (Bool, (Limit, Count))
```

```haskell
next w = CoAdderF (coAdd w) (coClear w) (coTotal w)

coClear :: (Limit, Count) -> (Limit, Count)
coClear (limit, count) = (limit, 0)

coTotal :: (Limit, Count) -> (Int, (Limit, Count))


coAdd :: (Limit, Count) -> Int -> (Bool, (Limit, Count))
```

```haskell
next w = CoAdderF (coAdd w) (coClear w) (coTotal w)

coClear :: (Limit, Count) -> (Limit, Count)
coClear (limit, count) = (limit, 0)

coTotal :: (Limit, Count) -> (Int, (Limit, Count))
coTotal (limit, count) = (count, (limit, count))

coAdd :: (Limit, Count) -> Int -> (Bool, (Limit, Count))
```

```haskell
next w = CoAdderF (coAdd w) (coClear w) (coTotal w)

coClear :: (Limit, Count) -> (Limit, Count)
coClear (limit, count) = (limit, 0)

coTotal :: (Limit, Count) -> (Int, (Limit, Count))
coTotal (limit, count) = (count, (limit, count))

coAdd :: (Limit, Count) -> Int -> (Bool, (Limit, Count))
coAdd (limit, count) x = (_    , (limit, _    ))
```

```haskell
next w = CoAdderF (coAdd w) (coClear w) (coTotal w)

coClear :: (Limit, Count) -> (Limit, Count)
coClear (limit, count) = (limit, 0)

coTotal :: (Limit, Count) -> (Int, (Limit, Count))
coTotal (limit, count) = (count, (limit, count))

coAdd :: (Limit, Count) -> Int -> (Bool, (Limit, Count))
coAdd (limit, count) x = (_    , (limit, _    ))
  where
    x' = count + x
    test = x' <= limit
```

```haskell
next w = CoAdderF (coAdd w) (coClear w) (coTotal w)

coClear :: (Limit, Count) -> (Limit, Count)
coClear (limit, count) = (limit, 0)

coTotal :: (Limit, Count) -> (Int, (Limit, Count))
coTotal (limit, count) = (count, (limit, count))

coAdd :: (Limit, Count) -> Int -> (Bool, (Limit, Count))
coAdd (limit, count) x = (test, (limit, _    ))
  where
    x' = count + x
    test = x' <= limit
```

```haskell
next w = CoAdderF (coAdd w) (coClear w) (coTotal w)

coClear :: (Limit, Count) -> (Limit, Count)
coClear (limit, count) = (limit, 0)

coTotal :: (Limit, Count) -> (Int, (Limit, Count))
coTotal (limit, count) = (count, (limit, count))

coAdd :: (Limit, Count) -> Int -> (Bool, (Limit, Count))
coAdd (limit, count) x = (test, (limit, _    ))
  where
    x' = count + x
    test = x' <= limit
    next = if test then x' else count
```

```haskell
next w = CoAdderF (coAdd w) (coClear w) (coTotal w)

coClear :: (Limit, Count) -> (Limit, Count)
coClear (limit, count) = (limit, 0)

coTotal :: (Limit, Count) -> (Int, (Limit, Count))
coTotal (limit, count) = (count, (limit, count))

coAdd :: (Limit, Count) -> Int -> (Bool, (Limit, Count))
coAdd (limit, count) x = (test, (limit, next))
  where
    x' = count + x
    test = x' <= limit
    next = if test then x' else count
```

Separation of the interpreter code from the client code

Reuse code written in terms of the interpreter code

Write different clients for different needs

Comonad transformers

```haskell
class MonadTrans t where
  lift :: Monad m => m a -> t m a

class ComonadTrans t where
  lower :: Comonad w => t w a -> w a
```

```haskell
type Limit = Int
type Count = Int

type CoAdder = Cofree CoAdderF


mkCoAdder :: Limit -> Count -> CoAdder ()
mkCoAdder limit count =
    () <$ coiter  next start
  where
    next w = CoAdderF (coAdd w) (coClear w) (coTotal w)
    start (limit, count)
```

```haskell
type Limit = Int
type Count = Int

type CoAdder = CofreeT CoAdderF Base
type Base = StoreT Count (EnvT Limit Identity)

mkCoAdder :: Limit -> Count -> CoAdder ()
mkCoAdder limit count =
    () <$ coiter  next start
  where
    next w = CoAdderF (coAdd w) (coClear w) (coTotal w)
    start (limit, count)
```

```haskell
type Limit = Int
type Count = Int

type CoAdder = CofreeT CoAdderF Base
type Base = StoreT Count (EnvT Limit Identity)

mkCoAdder :: Limit -> Count -> CoAdder ()
mkCoAdder limit count =
          coiterT next start
  where
    next w = CoAdderF (coAdd w) (coClear w) (coTotal w)
    start (limit, count)
```

```haskell
type Limit = Int
type Count = Int

type CoAdder = CofreeT CoAdderF Base
type Base = StoreT Count (EnvT Limit Identity)

mkCoAdder :: Limit -> Count -> CoAdder ()
mkCoAdder limit count =
          coiterT next start
  where
    next w = CoAdderF (coAdd w) (coClear w) (coTotal w)
    start = flip StoreT count .
            EnvT limit .
            Identity $
            const ()
```

```haskell
coClear :: (Limit, Count) -> (Limit, Count)
coClear (limit, count) = (limit, 0)

coTotal :: (Limit, Count) -> (Int, (Limit, Count))
coTotal (limit, count) = (count, (limit, count))
```

```
get  :: StateT s m s
set  :: s -> StateT s m ()
```

```
pos  :: StoreT s w a -> s
set  :: s -> StateT s m ()
```

```
pos  :: StoreT s w a -> s
seek :: s -> StoreT s w a  -> StoreT s w a
```

```haskell
type Limit, Count = Int

coClear :: (Limit, Count) -> (Limit, Count)
coClear (limit, count) = (limit, 0)

coTotal :: (Limit, Count) -> (Int, (Limit, Count))
coTotal (limit, count) = (count, (limit, count))
```

```haskell
type Base = StoreT Count (EnvT Limit Identity)

coClear :: (Limit, Count) -> (Limit, Count)
coClear (limit, count) = (limit, 0)

coTotal :: (Limit, Count) -> (Int, (Limit, Count))
coTotal (limit, count) = (count, (limit, count))
```

```haskell
type Base = StoreT Count (EnvT Limit Identity)

coClear :: Base a          -> Base a
coClear (limit, count) = (limit, 0)

coTotal :: (Limit, Count) -> (Int, (Limit, Count))
coTotal (limit, count) = (count, (limit, count))
```

```haskell
type Base = StoreT Count (EnvT Limit Identity)

coClear :: Base a          -> Base a
coClear w                  = seek 0 w

coTotal :: (Limit, Count) -> (Int, (Limit, Count))
coTotal (limit, count) = (count, (limit, count))
```

```haskell
type Base = StoreT Count (EnvT Limit Identity)

coClear :: Base a          -> Base a
coClear w                  = seek 0 w

coTotal :: Base a          -> (Int, Base a)
coTotal (limit, count) = (count, (limit, count))
```

```
type Base = StoreT Count (EnvT Limit Identity)

coClear :: Base a          -> Base a
coClear w                  = seek 0 w

coTotal :: Base a          -> (Int, Base a)
coTotal w                  = (pos w, w)
```

```haskell
coAdd :: (Limit, Count) -> Int -> (Bool, (Limit, Count))
coAdd (limit, count) x = (test, (limit, next))
  where
    x' = count + x
    test = x' <= limit
    next = if test then x' else count
```

```
ask  :: ReaderT r m r
```

```
ask  :: EnvT e w a -> e
```

```haskell
type Limit, Count = Int

coAdd :: (Limit, Count) -> Int -> (Bool, (Limit, Count))
coAdd (limit, count) x = (test, (limit, next))
  where


    x' = count + x
    test = x' <= limit
    next = if test then x' else count
```

```haskell
type Base = StoreT Count (EnvT Limit Identity)

coAdd :: (Limit, Count) -> Int -> (Bool, (Limit, Count))
coAdd (limit, count) x = (test, (limit, next))
  where


    x' = count + x
    test = x' <= limit
    next = if test then x' else count
```

```haskell
type Base = StoreT Count (EnvT Limit Identity)

coAdd :: Base a          -> Int -> (Bool, Base a)
coAdd (limit, count) x = (test, (limit, next))
  where


    x' = count + x
    test = x' <= limit
    next = if test then x' else count
```

```haskell
type Base = StoreT Count (EnvT Limit Identity)

coAdd :: Base a          -> Int -> (Bool, Base a)
coAdd w                x = (test, seek next w)
  where
    count = _
    limit = _
    x' = count + x
    test = x' <= limit
    next = if test then x' else count
```

```haskell
type Base = StoreT Count (EnvT Limit Identity)

coAdd :: Base a          -> Int -> (Bool, Base a)
coAdd w                x = (test, seek next w)
  where
    count = pos w
    limit = ask . lower $ w
    x' = count + x
    test = x' <= limit
    next = if test then x' else count
```

```haskell
type Base = StoreT Count (EnvT Limit Identity)

coClear ::
           Base a -> Base a
coClear = seek 0

coTotal ::
           Base a -> (Int, Base a)
coTotal w = (pos w, w)

coAdd ::
         Base a -> Int -> (Bool, Base a)
coAdd w x = (test, seek next w)
  where
    count = pos w
    limit = ask . lower $ w
    x' = count + x
    test = x' <= limit
    next = if test then x' else count
```

```haskell
type Base = StoreT Count (EnvT Limit Identity)

coClear :: ComonadStore Count w
        => Base a -> Base a
coClear = seek 0

coTotal :: ComonadStore Count w
        => Base a -> (Int, Base a)
coTotal w = (pos w, w)

coAdd :: (ComonadEnv Limit w, ComonadStore Count w)
      => Base a -> Int -> (Bool, Base a)
coAdd w x = (test, seek next w)
  where
    count = pos w
    limit = ask . lower $ w
    x' = count + x
    test = x' <= limit
    next = if test then x' else count
```

```haskell
type Base = StoreT Count (EnvT Limit Identity)

coClear :: ComonadStore Count w
        => w    a -> w    a
coClear = seek 0

coTotal :: ComonadStore Count w
        => w    a -> (Int, w    a)
coTotal w = (pos w, w)

coAdd :: (ComonadEnv Limit w, ComonadStore Count w)
      => w    a -> Int -> (Bool, w    a)
coAdd w x = (test, seek next w)
  where
    count = pos w
    limit = ask . lower $ w
    x' = count + x
    test = x' <= limit
    next = if test then x' else count
```

```haskell
type Base = StoreT Count (EnvT Limit Identity)

coClear :: ComonadStore Count w
        => w    a -> w    a
coClear = seek 0

coTotal :: ComonadStore Count w
        => w    a -> (Int, w    a)
coTotal w = (pos w, w)

coAdd :: (ComonadEnv Limit w, ComonadStore Count w)
      => w    a -> Int -> (Bool, w    a)
coAdd w x = (test, seek next w)
  where
    count = pos w
    limit = ask w
    x' = count + x
    test = x' <= limit
    next = if test then x' else count
```

Pairing

```haskell
class (Functor f, Functor g) => Pairing f g where
    pair :: (a -> b -> r) -> f a -> g b -> r
```

```haskell
class (Functor f, Functor g) => Pairing f g where
    pair :: (a -> b -> r) -> f a -> g b -> r

instance Pairing Identity Identity where
  pair f (Identity a) (Identity b) = f a b
```

```haskell
instance Pairing ((->) a) ((,) a) where
  pair p f = uncurry (p . f)
```

```haskell
instance Pairing ((->) a) ((,) a) where
  pair p f = uncurry (p . f)

instance Pairing ((,) a) ((->) a) where
  pair p f g = p (snd f) (g (fst f))
```

```haskell
instance Pairing ((->) a) ((,) a) where
  pair p f = uncurry (p . f)

instance Pairing ((,) a) ((->) a) where
  pair p f g = pair (flip p) g f
```

```haskell
instance Pairing f g => Pairing (Cofree f) (Free g) where
```

```haskell
instance Pairing f g => Pairing (Cofree f) (Free g) where
  pair p (a :< _)  (Pure x)  = p a x
```

```haskell
instance Pairing f g => Pairing (Cofree f) (Free g) where
  pair p (a :< _)  (Pure x)  = p a x
  pair p (_ :< fs) (Free gs) = pair (pair p) fs gs
```

```
data AdderF k =
```

```
data CoAdderF k = CoAdderF {
```

```
instance Pairing CoAdderF AdderF where
```

```haskell
data AdderF k =
    Add Int (Bool -> k)
```

```haskell
data CoAdderF k = CoAdderF {
      addH   :: Int -> (Bool,k)
```

```haskell
instance Pairing CoAdderF AdderF where
```

```haskell
data AdderF k =
    Add Int (Bool -> k)
```

```haskell
data CoAdderF k = CoAdderF {
      addH   :: Int -> (Bool,k)
```

```haskell
instance Pairing CoAdderF AdderF where
  pair f (CoAdderF a _ _) (Add x k) = pair f (a x) k
```

```haskell
data AdderF k =
    Add Int (Bool -> k)
  | Clear k
```

```haskell
data CoAdderF k = CoAdderF {
      addH   :: Int -> (Bool,k)
    , clearH :: k
```

```haskell
instance Pairing CoAdderF AdderF where
  pair f (CoAdderF a _ _) (Add x k) = pair f (a x) k
```

```haskell
data AdderF k =
    Add Int (Bool -> k)
  | Clear k
```

```haskell
data CoAdderF k = CoAdderF {
    addH   :: Int -> (Bool,k)
  , clearH :: k
```

```haskell
instance Pairing CoAdderF AdderF where
  pair f (CoAdderF a _ _) (Add x k) = pair f (a x) k
  pair f (CoAdderF _ c _) (Clear k) = f c k
```

```haskell
data AdderF k =
    Add Int (Bool -> k)
  | Clear k
  | Total (Int -> k)
```

```haskell
data CoAdderF k = CoAdderF {
    addH   :: Int -> (Bool,k)
  , clearH :: k
  , totalH :: (Int,k)
  }
```

```haskell
instance Pairing CoAdderF AdderF where
  pair f (CoAdderF a _ _) (Add x k) = pair f (a x) k
  pair f (CoAdderF _ c _) (Clear k) = f c k
```

```haskell
data AdderF k =
    Add Int (Bool -> k)
  | Clear k
  | Total (Int -> k)
```

```haskell
data CoAdderF k = CoAdderF {
    addH   :: Int -> (Bool,k)
  , clearH :: k
  , totalH :: (Int,k)
  }
```

```haskell
instance Pairing CoAdderF AdderF where
  pair f (CoAdderF a _ _) (Add x k) = pair f (a x) k
  pair f (CoAdderF _ c _) (Clear k) = f c k
  pair f (CoAdderF _ _ t) (Total k) = pair f t k
```

```haskell
runLimit :: CoAdder a -> Int
runLimit w = pair (\_ b -> b) w findLimit

testLimit :: Int -> Bool
testLimit x = runLimit (mkCoAdder x 0) == x
```

```haskell
pairEffect :: (Pairing f g, Comonad w, Monad m)
           => (a -> b -> r)
           -> CofreeT f w a
           -> FreeT g m b -> m r
pairEffect p s c = do
    mb <- runFreeT c
    case mb of
        Pure x -> return $ p (extract s) x
        Free gs -> pair (pairEffect p) (unwrap s) gs
```

```haskell
consoleAdder :: MonadIO m => AdderT m ()
consoleAdder = do
    l <- liftIO getLine
    case words l of
      ["add", x] -> add (read x) >>= \b ->
        output $ "add result: " ++ show b
      ["clear"]  -> clear
      ["total"]  -> total >>= \t ->
        output $ "total result: " ++ show t
      _          -> output prompt
  where
    output = liftIO . putStrLn
    prompt = unlines
      ["Commands:", "  add [int]", "  clear", "  total"]
```

```haskell
testConsole :: IO ()
testConsole = pairEffect (\_ _ -> r)
  (mkCoAdder 10 0)
  (forever consoleAdder)
```

Coproducts

```haskell
data AddF k = Add Int (Bool -> k)
```

```haskell
data CoAddF k = CoAdd (Int -> (Bool, k))
```

```haskell
data AddF k = Add Int (Bool -> k)

instance Functor AddF where ...

data CoAddF k = CoAdd (Int -> (Bool, k))

instance Functor CoAddF where ...
```

```haskell
data AddF k = Add Int (Bool -> k)

instance Functor AddF where ...

data CoAddF k = CoAdd (Int -> (Bool, k))

instance Functor CoAddF where ...

instance Pairing CoAddF AddF where ...
```

```
type AdderF   =  AddF  :+: ClearF   :+: TotalF

type CoAdderF = CoAddF :*: CoClearF :*: CoTotalF
```

```
type AdderF   =  AddF  :+: ClearF   :+: TotalF

type CoAdderF = CoAddF :*: CoClearF :*: CoTotalF

instance (Pairing f1 g1, Pairing f2, g2) =>
         Pairing (f1 :+: f2) (g1 :*: g2) where ...
```

```haskell
type AdderF   =  AddF  :+: ClearF   :+: TotalF

type CoAdderF = CoAddF :*: CoClearF :*: CoTotalF

instance (Pairing f1 g1, Pairing f2, g2) =>
         Pairing (f1 :+: f2) (g1 :*: g2) where ...

-- give us an instance of Pairing for CoAdderF and AdderF
```

```haskell
class (Functor sub, Functor sup) => sub :<: sup where
  inj :: sub a -> sup a

add :: (Functor f, AddF :<: f) => Int -> Free f Bool
add x = liftF . inj $ Add x id
```

```haskell
(*:*) :: (Functor f, Functor g) =>
    (a -> f a) -> (a -> g a) -> a -> (f :*: g) a
(*:*) = liftA2 (:*:)

coAdd :: (Int, Int) -> CoAddF (Int, Int)

mkCoAdder :: Int -> Int -> CoAdder (Int, Int)
mkCoAdder limit count =
  coiter (coAdd *:* coClear *:* coTotal) (limit, count)
```

More cofun

Networking

```haskell
data AddReq = AddReq Int
```

```haskell
data AddRes = AddRes Bool
```

```haskell
data AddReq = AddReq Int

instance Binary AddReq where ...

data AddRes = AddRes Bool

instance Binary AddRes where ...
```

```haskell
data AddClientF m k =
  AddClientF AddReq (m (Either NetError AddRes -> k))
```

```haskell
data CoAddClientF m k =
  CoAddClientF (AddReq -> m (Either NetError AddRes, k))
```

```haskell
data AddClientF m k =
  AddClientF AddReq (m (Either NetError AddRes -> k))

instance Functor m => Functor (AddClientF m) where ...

data CoAddClientF m k =
  CoAddClientF (AddReq -> m (Either NetError AddRes, k))

instance Functor m => Functor (CoAddClientF m) where ...
```

```haskell
data AddClientF m k =
  AddClientF AddReq (m (Either NetError AddRes -> k))

instance Functor m => Functor (AddClientF m) where ...

data CoAddClientF m k =
  CoAddClientF (AddReq -> m (Either NetError AddRes, k))

instance Functor m => Functor (CoAddClientF m) where ...

instance (Functor m, Monad m) =>
  PairingM (CoAddClientF m) (AddClientF m) m where ...
```

```haskell
instance (Functor m, MonadError NetError m) =>
  PairingM (CoAddClientF m) AddF m where ...

instance (Functor m, Monad m) =>
  PairingM CoAddF (AddClientF m) m where ...
```

Testing

Coparsers

MaybeT, EitherT, ComonadError

Tree annotation

Indexed free and cofree

Density

Conclusion

Links

- http://dlaing.org/cofun

Links

- http://dlaing.org/cofun
- http://comonad.com/reader/2008/the-cofree-comonad-and-the-expression-problem/

Links

- http://dlaing.org/cofun
- http://comonad.com/reader/2008/the-cofree-comonad-and-the-expression-problem/
- http://blog.sigfpe.com/2014/05/cofree-meets-free.html