

# 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

```
class Monad m where
```

```
class Comonad w where
```

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

```
class Comonad w where
```

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

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

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

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

```
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
```



```
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
```

```
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)
```

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

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

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

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

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

```
extract z
-- 3
```

```
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

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

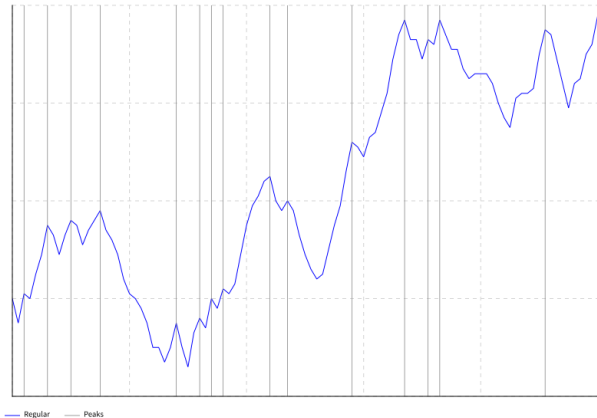
### extend latch z





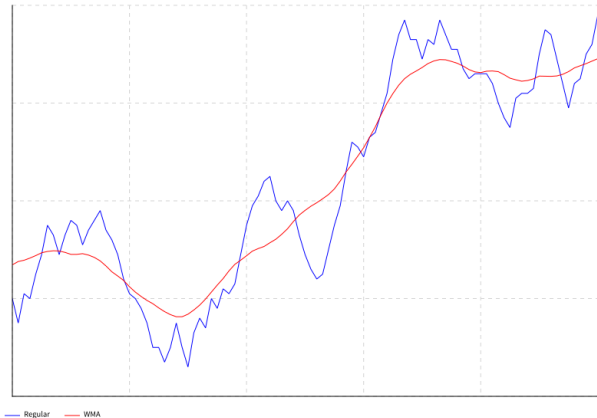
```
peak :: Ord a => Zipper a -> Bool
peak (Zipper l f r) =
  headDef f l < f && f > headDef f r
```

### extend peak z

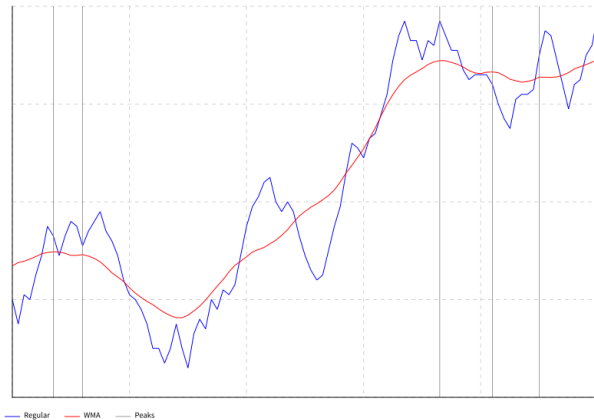


```
wma :: Int -> Zipper Double -> Double
wma n (Zipper l f r) =
  average $ take n l ++ f : take n r
```

**extend (wma 10) z**



## extend peak . extend (wma 10) \$ z



Free

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

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

```
data AdderF k =
```

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

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



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

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

```
instance Functor AdderF where
```

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

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

```
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)
```

```
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)
```

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

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

```
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

```
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
```

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

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

```
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
```

```
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
```

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

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

```
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)
```

```
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

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

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

```
...
```

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

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

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

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

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

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

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

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



```
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)
```

```
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)
```

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

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

```
...
```

```
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

```
type Limit = Int
```

```
type Count = Int
```

```
mkCoAdder :: Limit -> Count -> CoAdder (Limit, Count)
```

```
mkCoAdder limit count = _
```



```
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 = _
```

```
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
```

```
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)
```

```
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)
```

```

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)

```

```
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))
```

```
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))
```



```
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))
```

```
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, _ ) )
```

```
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
```

```
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 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
```

```
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

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

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

```
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)
```

```
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)
```

```
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)
```

```
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 ()
```

```
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
```

```
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))
```

```
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))
```

```
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))
```

```
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))
```

```
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)
```

```
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

```
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
```

```
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
```

```
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
```

```
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
```



```
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
```

```
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
```

```
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
```

```
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
```

```
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

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

```
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
```



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

```
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))
```

```
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
```

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

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

```
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
```

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

```
instance Pairing CoAdderF AdderF where
```



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

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

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

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

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

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

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

```
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
```

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

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

```
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
```

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

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

```
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
```

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

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

```
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
```

```
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"]
```



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

# Coproducts

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

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

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

```
instance Functor AddF where ...
```

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

```
instance Functor CoAddF where ...
```

```
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 ...
```

```
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
```



```
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
```

```
(*:*) :: (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

```
data AddReq = AddReq Int
```

```
data AddRes = AddRes Bool
```

```
data AddReq = AddReq Int
```

```
instance Binary AddReq where ...
```

```
data AddRes = AddRes Bool
```

```
instance Binary AddRes where ...
```

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

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

```
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 ...
```



```
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 ...
```

```
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

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>