

Contravariant: The Other Side of the Coin

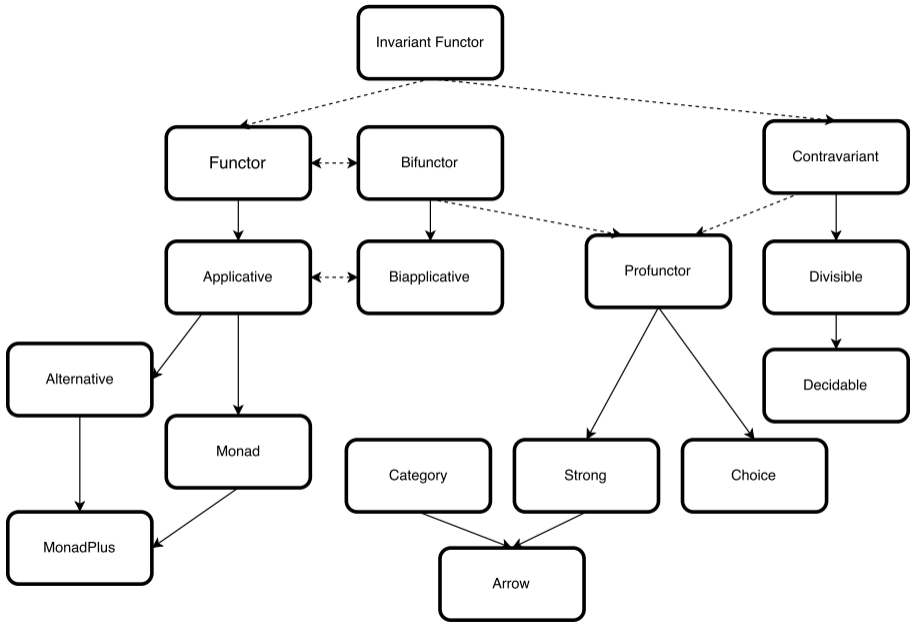
George Wilson

Data61/CSIRO

george.wilson@data61.csiro.au

22nd May 2018





Contravariant

```
newtype Predicate a =  
  Predicate { runPredicate :: a -> Bool }
```

```
newtype Predicate a =  
  Predicate { runPredicate :: a -> Bool}
```

```
evenP :: Predicate Int
```

```
evenP = Predicate (\i -> i `mod` 2 == 0)
```

```
newtype Predicate a =  
  Predicate { runPredicate :: a -> Bool}
```

```
evenP :: Predicate Int
```

```
evenP = Predicate (\i -> i `mod` 2 == 0)
```

```
x :: Bool
```

```
x = runPredicate evenP 7
```

```
gt5 :: Predicate Int
gt5  = Predicate (\i ->
                  i > 5)
```

```
gt5 :: Predicate Int
gt5  = Predicate (\i ->
                  i > 5)
```

```
lenGT5 :: Predicate String
lenGT5 = Predicate (\str ->
                    let i = length str
                    in  i > 5)
```



```
gt5 :: Predicate Int
gt5  = Predicate (\i ->
                  i > 5)
```

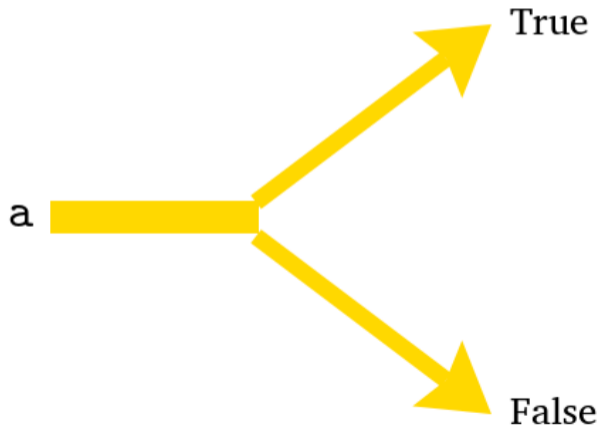
```
lenGT5 :: Predicate String
lenGT5 = Predicate (\str ->
                    let i = length str
                    in  i > 5)
```

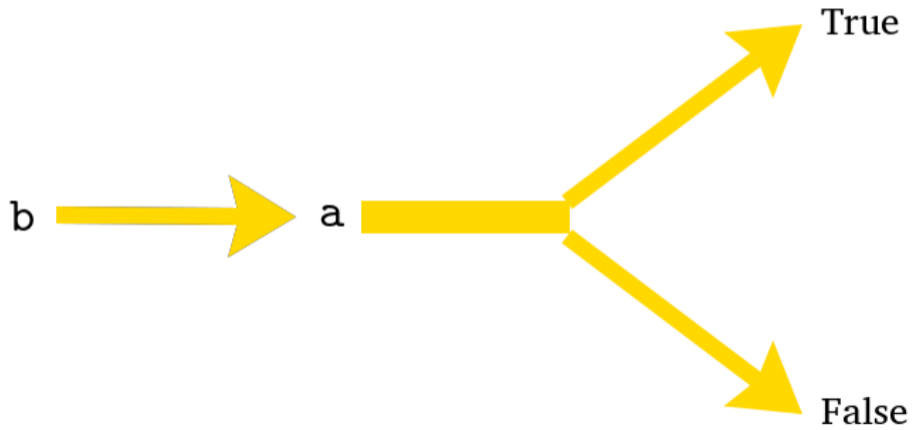
```
mapP :: (b -> a) -> Predicate a -> Predicate b
mapP ba (Predicate abool) =
  Predicate (\b ->
            let a = ba b
            in  abool a)
```

```
gt5 :: Predicate Int
gt5  = Predicate (\i ->
                  i > 5)
```

```
lenGT5' :: Predicate String
lenGT5' = mapP length gt5
```

```
mapP :: (b -> a) -> Predicate a -> Predicate b
mapP ba (Predicate abool) =
  Predicate (\b ->
             let a = ba b
             in  abool a)
```





Is Predicate a Functor?

Is Predicate a Functor?

`mapP :: (b -> a) -> Predicate a -> Predicate b`

`fmap :: (a -> b) -> Predicate a -> Predicate b`

I contain `Ints`!
You can access them if you'd like

```
[13, 74, 63, 12] :: List Int
```

I contain Ints!
You can access them if you'd like

```
[13, 74, 63, 12] :: List Int
```

I am in need of an Int!

Int

even?

Bool

:: Predicate Int


```
class Contravariant f where  
  contramap :: (b -> a) -> f a -> f b
```

```
class Contravariant f where  
  contramap :: (b -> a) -> f a -> f b
```

Laws:

```
contramap id = id
```

```
contramap f . contramap g = contramap (g . f)
```

```
class Contravariant f where  
  contramap :: (b -> a) -> f a -> f b
```

```
instance Contravariant Predicate where  
  contramap :: (b -> a) -> Predicate a -> Predicate b  
  contramap ba (Predicate abool) = Predicate (abool . ba)
```

We need more power!

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

```
class Functor f => ApplicativeL f where  
  liftA2 :: ((a, b) -> c) -> f a -> f b -> f c  
  pure   :: a -> f a
```

```
class Functor f => ApplicativeL f where
```

```
  liftA2 :: ((a, b) -> c) -> f a -> f b -> f c
```

```
  pure   :: a -> f a
```

```
(<*>) :: ApplicativeL f => f (a -> b) -> f a -> f b
```

```
(<*>) fab fa = liftA2 (\(ab,a) -> ab a) fab fa
```

```
class Contravariant f => Divisible f where  
  divide  :: (c -> (a, b)) -> f a -> f b -> f c  
  conquer :: f a
```



```
class Contravariant f => Divisible f where  
  divide  :: (c -> (a, b)) -> f a -> f b -> f c  
  conquer :: f a
```

Laws:

```
divide f m conquer = contramap (fst . f) m
```

```
divide f conquer m = contramap (snd . f) m
```

```
divide f (divide g m n) o = divide f' m (divide id n o)
```

where

```
f' a = case f a of (bc,d) -> case g bc of (b,c) -> (a, (b,c))
```

```
class Contravariant f => Divisible f where  
  divide :: (c -> (a, b)) -> f a -> f b -> f c  
  conquer :: f a
```

```
instance Divisible Predicate where  
  divide cab (Predicate pa) (Predicate pb) =  
    Predicate $ \c ->  
      case cab c of  
        (a,b) -> pa a && pb b  
  
  conquer = Predicate (\_ -> True)
```



```
ingredients :: (Banana, IceCream)
```

```
ingredients :: (Banana, IceCream)
```

```
ripe :: Predicate Banana
```

```
frozen :: Predicate IceCream
```

```
ingredients :: (Banana, IceCream)
```

```
ripe :: Predicate Banana
```

```
frozen :: Predicate IceCream
```

```
divide      :: Divisible f => (c -> (a,b)) -> f a -> f b -> f c
```

```
divide id   :: Divisible f => f a -> f b -> f (a,b)
```

```
ingredients :: (Banana, IceCream)
```

```
ripe :: Predicate Banana
```

```
frozen :: Predicate IceCream
```

```
divide      :: Divisible f => (c -> (a,b)) -> f a -> f b -> f c
```

```
divide id   :: Divisible f => f a -> f b -> f (a,b)
```

```
divide id ripe frozen :: Predicate (Banana, IceCream)
```

```
ingredients :: (Banana, IceCream)
```

```
ripe :: Predicate Banana
```

```
frozen :: Predicate IceCream
```

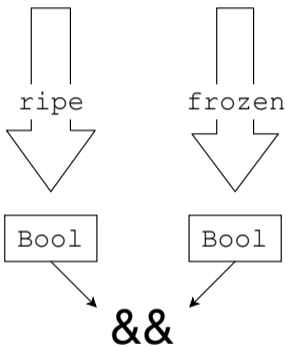
```
divide :: Divisible f => (c -> (a,b)) -> f a -> f b -> f c
```

```
divide id :: Divisible f => f a -> f b -> f (a,b)
```

```
divide id ripe frozen :: Predicate (Banana, IceCream)
```

```
runPredicate (divide id ripe frozen) ingredients :: Bool
```


(Banana, IceCream)



```
data Kitchen = Kitchen Rice Curry Banana Apple IceCream
```

```
data Kitchen = Kitchen Rice Curry Banana Apple IceCream
```

```
ripe :: Predicate Banana
```

```
frozen :: Predicate IceCream
```

```
data Kitchen = Kitchen Rice Curry Banana Apple IceCream
```

```
ripe :: Predicate Banana
```

```
frozen :: Predicate IceCream
```

```
getIngredients :: Kitchen -> (Banana, IceCream)
```

```
getIngredients (Kitchen _ _ b _ i) = (b,i)
```

```
data Kitchen = Kitchen Rice Curry Banana Apple IceCream
```

```
ripe :: Predicate Banana
```

```
frozen :: Predicate IceCream
```

```
getIngredients :: Kitchen -> (Banana, IceCream)
```

```
getIngredients (Kitchen _ _ b _ i) = (b,i)
```

```
divide :: Divisible f => (c -> (a,b)) -> f a -> f b -> f c
```

```
data Kitchen = Kitchen Rice Curry Banana Apple IceCream
```

```
ripe :: Predicate Banana
```

```
frozen :: Predicate IceCream
```

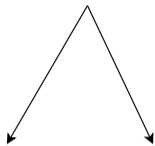
```
getIngredients :: Kitchen -> (Banana, IceCream)
```

```
getIngredients (Kitchen _ _ b _ i) = (b,i)
```

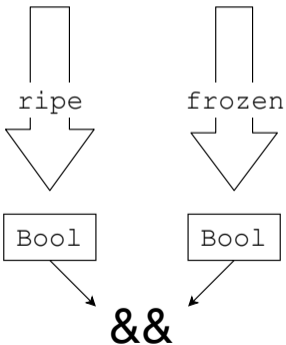
```
divide :: Divisible f => (c -> (a,b)) -> f a -> f b -> f c
```

```
divide getIngredients ripe frozen :: Predicate Kitchen
```

Kitchen



(Banana, IceCream)





What about Alternative?

```
class Contravariant f => Divisible f where  
  divide  :: (c -> (a, b)) -> f a -> f b -> f c  
  conquer :: f a
```

```
class Contravariant f => Divisible f where  
  divide  :: (c -> (a, b)) -> f a -> f b -> f c  
  conquer :: f a
```

```
class Divisible f => Decidable f where  
  choose :: (c -> Either a b) -> f a -> f b -> f c  
  lose   :: (a -> Void) -> f a
```

```
data Void
```

```
absurd :: Void -> a
```

```
absurd v = case v of {}
```

```
data Void
```

```
absurd :: Void -> a
```

```
absurd v = case v of {}
```

```
left :: Either a Void -> a
```

```
left = either id absurd
```

```
right :: Either Void b -> b
```

```
right = either absurd id
```

```
class Divisible f => Decidable f where  
  choose :: (c -> Either a b) -> f a -> f b -> f c  
  lose  :: (a -> Void) -> f a
```

```
class Divisible f => Decidable f where  
  choose :: (c -> Either a b) -> f a -> f b -> f c  
  lose  :: (a -> Void) -> f a
```

Laws:

```
choose Left m (lose f) = m
```

```
choose Right (lose f) m = m
```

```
choose f (choose g m n) o = choose f' m (choose id n o) where  
  f' bcd = either (either id (Right . Left) . g) (Right . Right) . f
```

```
class Divisible f => Decidable f where  
  choose :: (c -> Either a b) -> f a -> f b -> f c  
  lose  :: (a -> Void) -> f a
```

```
instance Decidable Predicate where  
  choose cab (Predicate pa) (Predicate pb) =  
    Predicate $ \c ->  
      case cab c of  
        Left a -> pa a  
        Right b -> pb b  
  
  lose av = Predicate (\a -> absurd (av a))
```


Predicates are boring

```
newtype Printer a = Printer {  
  runPrinter :: a -> String  
}
```

```
string :: Printer String
```

```
string = Printer id
```

```
konst :: String -> Printer a
```

```
konst s = Printer (const s)
```

```
showP :: Show a => Printer a
```

```
showP = Printer show
```

```
int :: Printer Int
```

```
int = showP
```

```
newline :: Printer ()
```

```
newline = konst "\n"
```

```
instance Contravariant Printer where
  contramap ba (Printer as) = Printer (as . ba)
```

```
instance Contravariant Printer where
```

```
  contramap ba (Printer as) = Printer (as . ba)
```

```
instance Divisible Printer where
```

```
  divide cab (Printer as) (Printer bs) = Printer $ \c ->
```

```
    case cab c of
```

```
      (a,b) -> as a <> bs b
```

```
  conquer = Printer (const "")
```

```
instance Decidable Printer where
```

```
  choose cab (Printer as) (Printer bs) = Printer $ \c ->
```

```
    case cab c of
```

```
      Left a -> as a
```

```
      Right b -> bs b
```

```
lose av = Printer (\a -> absurd (av a))
```

(>\$<) :: **Contravariant** f => (b -> a) -> f a -> f b

(>\$<) = contramap

(>\$<) :: **Contravariant** f => (b -> a) -> f a -> f b

(>\$<) = contramap

(>*<) :: **Divisible** f => f a -> f b -> f (a,b)

(>*<) = divide id

(>\$<) :: **Contravariant** f => (b -> a) -> f a -> f b

(>\$<) = contramap

(>*<) :: **Divisible** f => f a -> f b -> f (a,b)

(>*<) = divide id

(>|<) :: **Decidable** f => f a -> f b -> f (**Either** a b)

(>|<) = choose id

(>\$<) :: **Contravariant** f => (b -> a) -> f a -> f b

(>\$<) = contramap

(>*<) :: **Divisible** f => f a -> f b -> f (a,b)

(>*<) = divide id

(>|<) :: **Decidable** f => f a -> f b -> f (**Either** a b)

(>|<) = choose id

(>*) :: **Divisible** f => f a -> f () -> f a

(>*) = divide (\a -> (a, ()))

(*<) :: **Divisible** f => f () -> f a -> f a

(*<) = divide (\a -> ((), a))

infixr 3 >\$<

infixr 4 >*<

infixr 3 >|<

infixr 4 >*

infixr 4 *<

```
data Car = Car
  { make    :: String
  , model   :: String
  , engine  :: Engine
  }
```

```
data Engine = Pistons Int | Rocket
```

```
data Car = Car
  { make    :: String
  , model   :: String
  , engine  :: Engine
  }
```

```
data Engine = Pistons Int | Rocket
```

```
car :: Car
```

```
car = Car "Toyota" "Corolla" (Pistons 4)
```

```
engineToEither :: Engine -> Either Int ()
engineToEither e = case e of
  Pistons i -> Left i
  Rocket    -> Right ()
```

```
enginePrint :: Printer Engine
enginePrint =
  engineToEither
    >$< konst "Pistons: " *< int
    >|< konst "Rocket "
```

```
carToTuple :: Car -> (String, (String, Engine))
carToTuple (Car ma mo e) = (ma, (mo, e))
```

```
carPrint :: Printer Car
```

```
carPrint =
```

```
  carToTuple
```

```
    >$< (konst "Make:  " *< string >* newline)
```

```
    >*< (konst "Model: " *< string >* newline)
```

```
    >*< enginePrint
```

```
putStrLn $ runPrinter carPrint car
```

```
Make: Toyota  
Model: Corolla  
Pistons: 4
```


- `Applicative` and `Alternative` let us talk about how to *combine multiple results*
- `Divisible` and `Decidable` let us talk about how to *consume multiple inputs*

Thanks for listening!

Questions?

- <https://hackage.haskell.org/package/contravariant>
- <https://github.com/qfpl/invariant-extras>
- <https://hackage.haskell.org/package/generics-eot>
- **Discrimination is Wrong**
<https://www.youtube.com/watch?v=eXDJ5Jcbgk8>

“Is there a contravariant Monad?” No

```
class Applicative f => MonadJ f where  
  join :: f (f a) -> f a
```

but

```
newtype Compose f g a = Compose (f (g a))
```

```
instance (Contravariant f, Contravariant g) =>  
  Functor (Compose f g) where  
  fmap z (Compose fga) = Compose $ contramap (contramap z) fga
```

“Is anything both contravariant and covariant?”

```
data Const c a = Const c
```

```
instance Functor (Const c) where  
  fmap f (Const c) = Const c
```

```
instance Contravariant (Const c) where  
  contramap f (Const c) = Const c
```

Every a is in positive position

Every a is in negative position