The Extended Functor Family

George Wilson

Ephox

george.wilson@ephox.com

May 9, 2017

Google

"fun with functors"

All Videos Maps Images Shopping More Search tools

About 564 results (0.47 seconds)

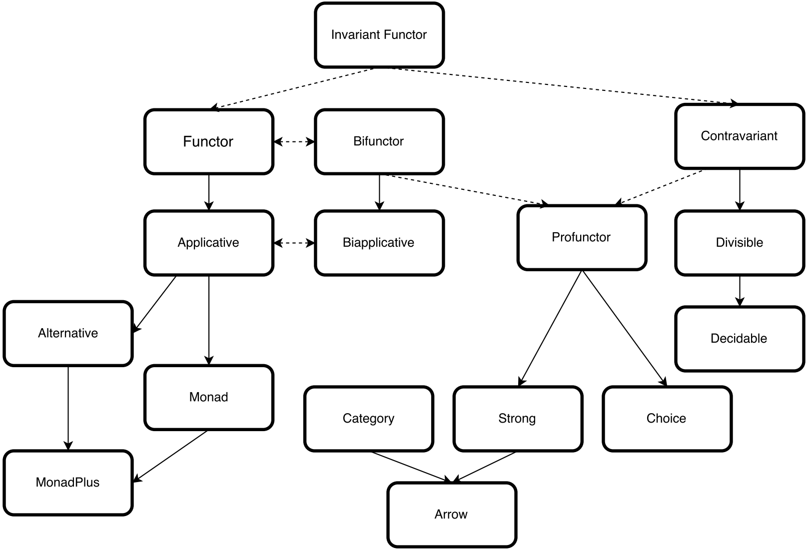
Fun with Functors | Good Math Bad Math
goodmath.scientopia.org/2011/10/25/**fun-with-functors**/
Oct 25, 2011 - So far, we've looked at the minimal basics of categories: what they are, and how to categorize the kinds of arrows that exist in categories in ...

Recitation 8: Functors
www.cs.cornell.edu/courses/cs312/2006fa/recitations/rec08.html
More **fun with functors**. Suppose you have a structure that makes use of not one, but two or more other structures. This seems to call for a functor that can take in ...

cbrad: Fun with functors
bradclow.blogspot.com/2009/02/**fun-with-functors**.html
Feb 15, 2009 - **Fun with functors**. I have been slowly working through some more Haskell with help from Tony. Quickly worked through the previous stuff we did ...

Tiusic: Fun with Functors
tiusic.blogspot.com/2013/06/**fun-with-functors**.html
Jun 4, 2013 - **Fun with Functors**. I just finished implementing the sound system for my game engine. It's a simple, fast, clean wrapper for OpenAL. There's a lot ...

GitHub - PawelPanasewicz/FunctorsAndFriends: Examples of using ...
https://github.com/PawelPanasewicz/FunctorsAndFriends
src/test/scala · using scala check from scala test, 2 months ago .gitattributes · **Fun with Functors**, 3 months ago .gitignore · **Fun with Functors**, 3 months ago .travis.


Invariant Functor
Functor
Bifunctor
Contravariant
Applicative
Biapplicative
Profunctor
Divisible
Alternative
Decidable
Monad
Category
Strong
Choice
MonadPlus
Arrow

Functor

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

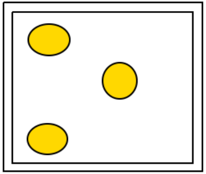
```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

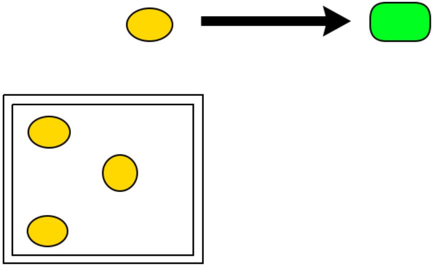
Laws:

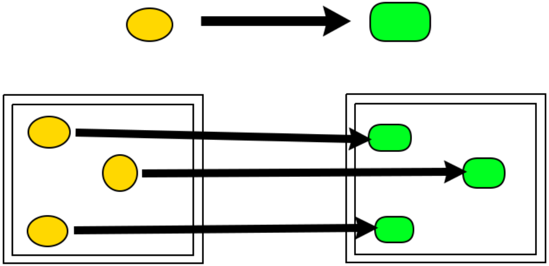
$$fmap\ id = id$$

$$fmap\ f\ .\ fmap\ g\ = fmap\ (f\ .\ g)$$

```haskell
instance Functor [] where
  fmap :: (a -> b) -> [a] -> [b]
  fmap f [] = []
  fmap f (x:xs) = f x : fmap f xs
```

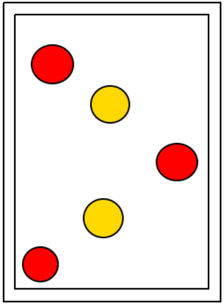


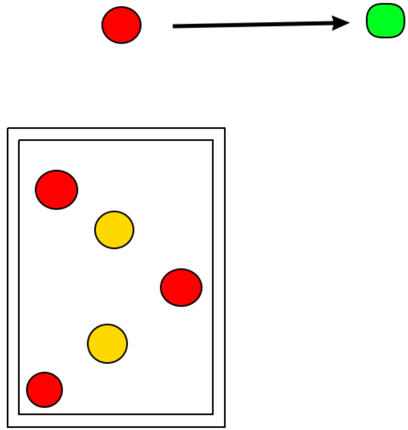


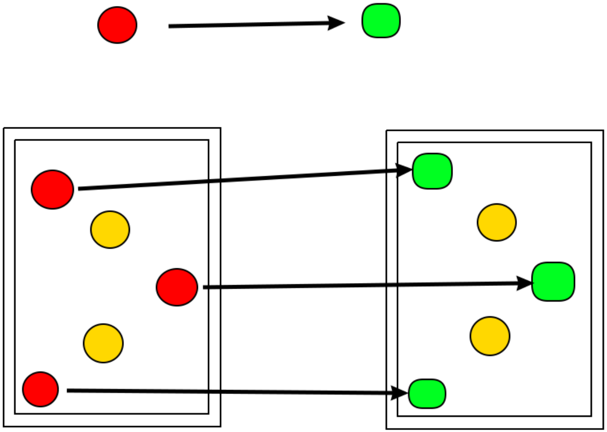


```haskell
instance Functor (x,) where
  fmap :: (a -> b) -> (x, a) -> (x, b)
  fmap f (x, a) = (x, f a)
```

```haskell
instance Functor (x,) where
  fmap :: (a -> b) -> (x, a) -> (x, b)
  fmap f (x, a) = (x, f a)


instance Functor (Either e) where
  fmap :: (a -> b) -> Either e a -> Either e b
  fmap f (Left e) = Left e
  fmap f (Right x) = Right (f x)
```







Bifunctor

```haskell
class Bifunctor p where
  bimap :: (a -> b) -> (x -> y) -> p a x -> p b y
```

```haskell
class Bifunctor p where
  bimap :: (a -> b) -> (x -> y) -> p a x -> p b y

  first :: (a -> b) -> p a x -> p b x
  second :: (x -> y) -> p a x -> p a y
```

```haskell
class Bifunctor p where
  bimap :: (a -> b) -> (x -> y) -> p a x -> p b y

  first :: (a -> b) -> p a x -> p b x
  second :: (x -> y) -> p a x -> p a y
```

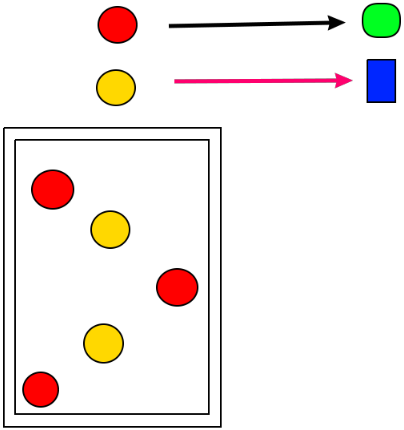
Laws:

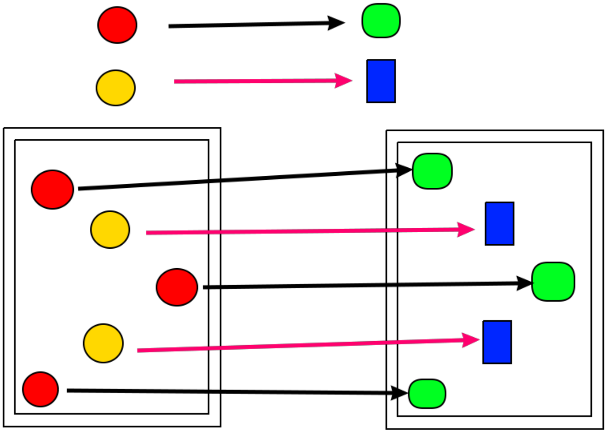
$$bimap\ id\ id\ =\ id$$

$$bimap\ f\ h\ .\ bimap\ g\ i\ =\ bimap\ (f\ .\ g)\ (h\ .\ i)$$

```haskell
instance Bifunctor (,) where
  bimap :: (a -> b) -> (x -> y) -> (a,x) -> (b,y)
  bimap f g (a,x) = (f a, g x)
```

```haskell
instance Bifunctor Either where
  bimap :: (a -> b) -> (x -> y) -> Either a x -> Either b y
  bimap f g (Left a) = Left (f a)
  bimap f g (Right x) = Right (g x)
```





Contravariant Functors

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

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

Is `Predicate` a `Functor`?

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

Is `Predicate` a `Functor`?

Can we write `fmap ::  (a -> b) -> Predicate a -> Predicate b`

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

Is `Predicate` a `Functor`?

Can we write `fmap ::  (a -> b) -> Predicate a -> Predicate b`

No!

Short diversion

Polarity

A type in a type signature can be in *positive* position or in *negative* position

Polarity

A type in a type signature can be in *positive* position or in *negative* position

- A type on its own is in positive position, like
  - `i :: Int`
  - `result :: Maybe String`
  - `snacks :: [Banana]`

Polarity

A type in a type signature can be in *positive* position or in *negative* position

- A type on its own is in positive position, like
  - `i :: Int`
  - `result :: Maybe String`
  - `snacks :: [Banana]`
- Function return types are in positive position, but parameters are in negative position
  - `length :: [a] -> Int`
  - `buildRome :: Romulus -> Remus -> Rome`

Polarity

For `f` to be an instance of `Functor` every `a` in `f a` must be in positive position

We say that `f` is **covariant** in `a`

Polarity

For `f` to be an instance of `Functor` every `a` in `f a` must be in positive position

We say that `f` is **covariant** in `a`

```
data Maybe a = Nothing | Just a
```

Polarity

For `f` to be an instance of `Functor` every `a` in `f a` must be in positive position

We say that `f` is **covariant** in `a`

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
  fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

Polarity

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

Polarity

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

In `Predicate`, we only see `a` in negative position.
We say `Predicate` is **contravariant** in `a`.

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

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

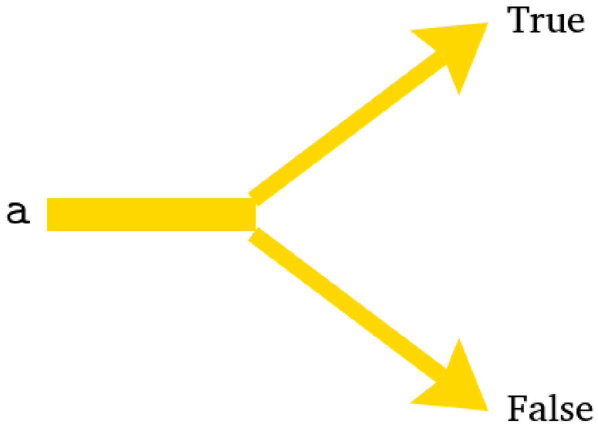
Laws:

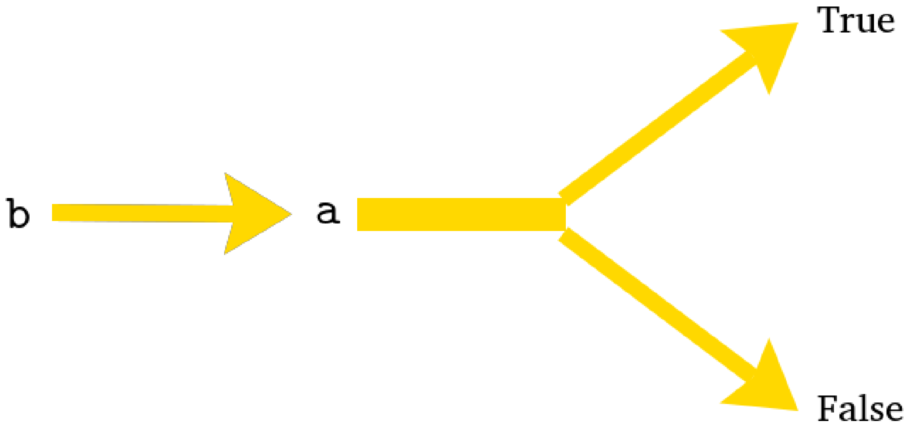
$$contramap\, id \;=\; id$$

$$contramap\ f\ .\ contramap\ g \;=\; contramap\ (g\ .\ f)$$

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

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


instance Contravariant Predicate where
  contramap :: (b -> a) -> Predicate a -> Predicate b
  contramap f (Predicate p) = Predicate (p . f)
```


a
True
False

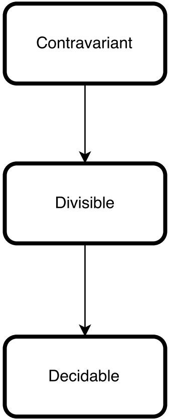

b
a
True
False

We think of a covariant `Functor` as being *full* of `a`'s.
A `Contravariant` functor can be thought of as *consuming* `a`'s.

```haskell
data Ordering = LT | EQ | GT

newtype Comparison a =
  Comparison { runComparison :: a -> a -> Ordering }
```

```haskell
data Ordering = LT | EQ | GT

newtype Comparison a =
  Comparison { runComparison :: a -> a -> Ordering }


instance Contravariant Comparison where
  contramap :: (b -> a) -> Comparison a -> Comparison b
  contramap f (Comparison c) = Comparison (\a b -> c (f a) (f b))
```

Corresponding to the more powerful forms of `Functor`, there are more powerful forms of `Contravariant`:



Why care about Contravariant?

Why care about Contravariant?

Discrimination (Linear-time sorting)

Now that we've talked about Bifunctor and Contravariant, we can finally talk about…

Now that we've talked about Bifunctor and Contravariant, we can finally talk about...

Profunctor

```haskell
class Profunctor p where
  dimap :: (a -> b) -> (c -> d) -> p b c -> p a d
```

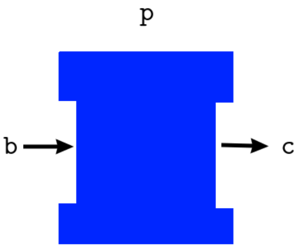
```haskell
class Profunctor p where
  dimap :: (a -> b) -> (c -> d) -> p b c -> p a d

  lmap :: (a -> b) -> p b c -> p a c
  rmap :: (c -> d) -> p b c -> p b d
```

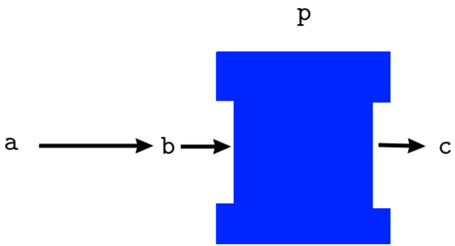
```haskell
class Profunctor p where
  dimap :: (a -> b) -> (c -> d) -> p b c -> p a d

  lmap :: (a -> b) -> p b c -> p a c
  rmap :: (c -> d) -> p b c -> p b d
```

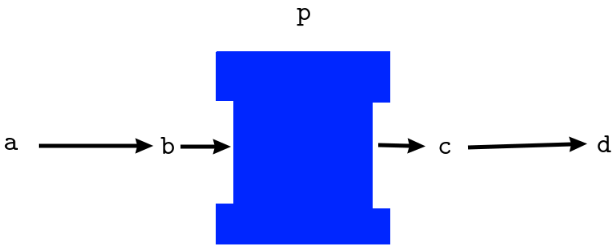
Laws:

$$dimap\ id\ id\ =\ id$$

$$dimap\ f\ g\ .\ dimap\ h\ k\ =\ dimap\ (h\ .\ f)\ (g\ .\ k)$$


p
b
c


p
a
b
c


p
a
b
c
d

```haskell
instance Profunctor (->) where
  dimap :: (a -> b) -> (c -> d) -> (b -> c) -> (a -> d)
  dimap ab cd bc = cd . bc . ab
```

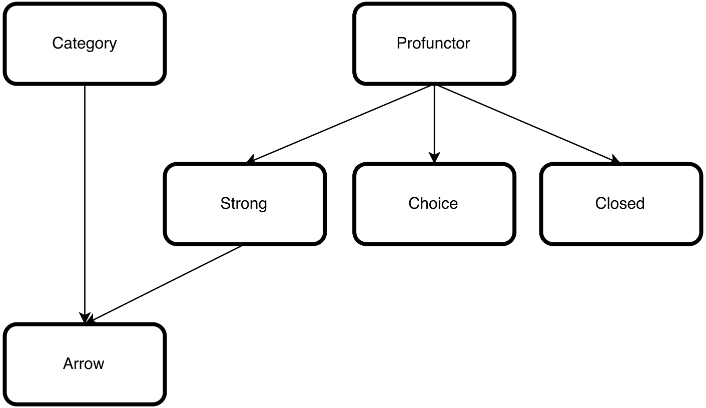
```haskell
-- import Control.Arrow

newtype Kleisli m b c = Kleisli { runKleisli :: b -> m c }
```

```haskell
-- import Control.Arrow

newtype Kleisli m b c = Kleisli { runKleisli :: b -> m c }


instance Monad m => Profunctor (Kleisli m) where
  dimap :: (a -> b) -> (c -> d) -> Kleisli m b c -> Kleisli m a d
  dimap ab cd (Kleisli bmc) =
    Kleisli (liftM cd . bmc . ab)
```

Every `Arrow` is a `Profunctor`!

Every `Arrow` is a `Profunctor`!

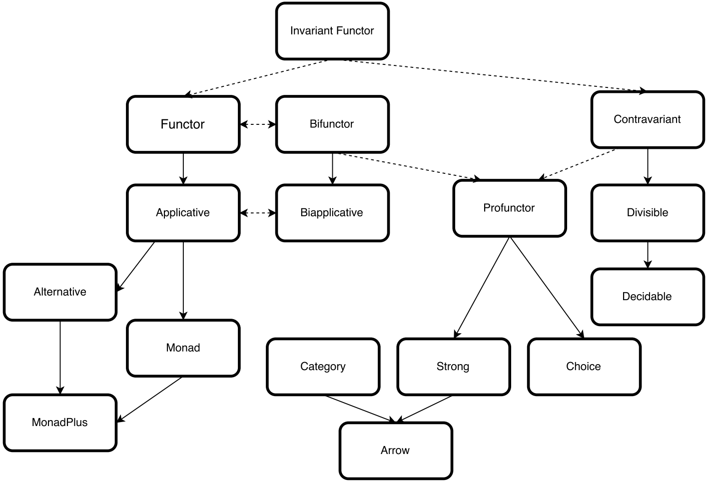
```
newtype WrappedArrow p b c = WrappedArrow { unwrap :: p b c }

instance Arrow p => Profunctor (WrappedArrow p) where
  dimap ab cd (WrappedArrow pbc) =
    WrappedArrow (arr ab . pbc . arr cd)
```


Category
Profunctor
Strong
Choice
Closed
Arrow

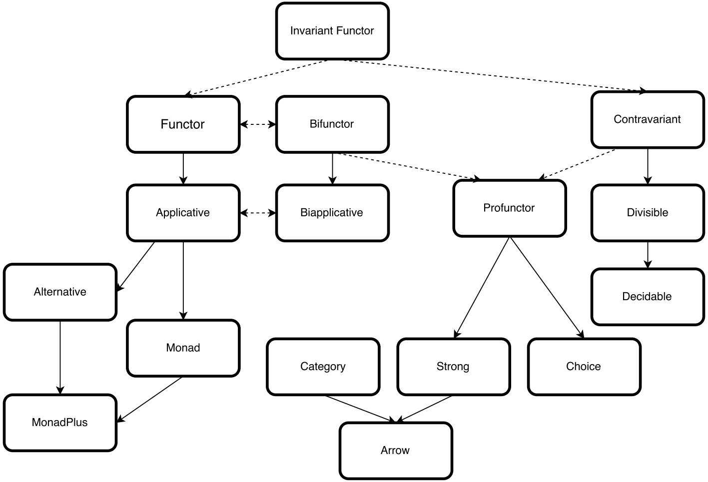
Why care about Profunctors?

Why care about Profunctors?

Lens


Invariant Functor
Functor
Bifunctor
Contravariant
Applicative
Biapplicative
Profunctor
Divisible
Alternative
Decidable
Monad
Category
Strong
Choice
MonadPlus
Arrow

Thanks for listening!



References

- contravariant package
  https://hackage.haskell.org/package/contravariant
- profunctors package
  https://hackage.haskell.org/package/profunctors
- Discrimination is Wrong
  https://yow.eventer.com/yow-lambda-jam-2015-1305/discrimination-is-wrong-improving-productivity-by-edward-kmett-1890
- Fun with Profunctors
  https://www.youtube.com/watch?v=OJtGECfksds