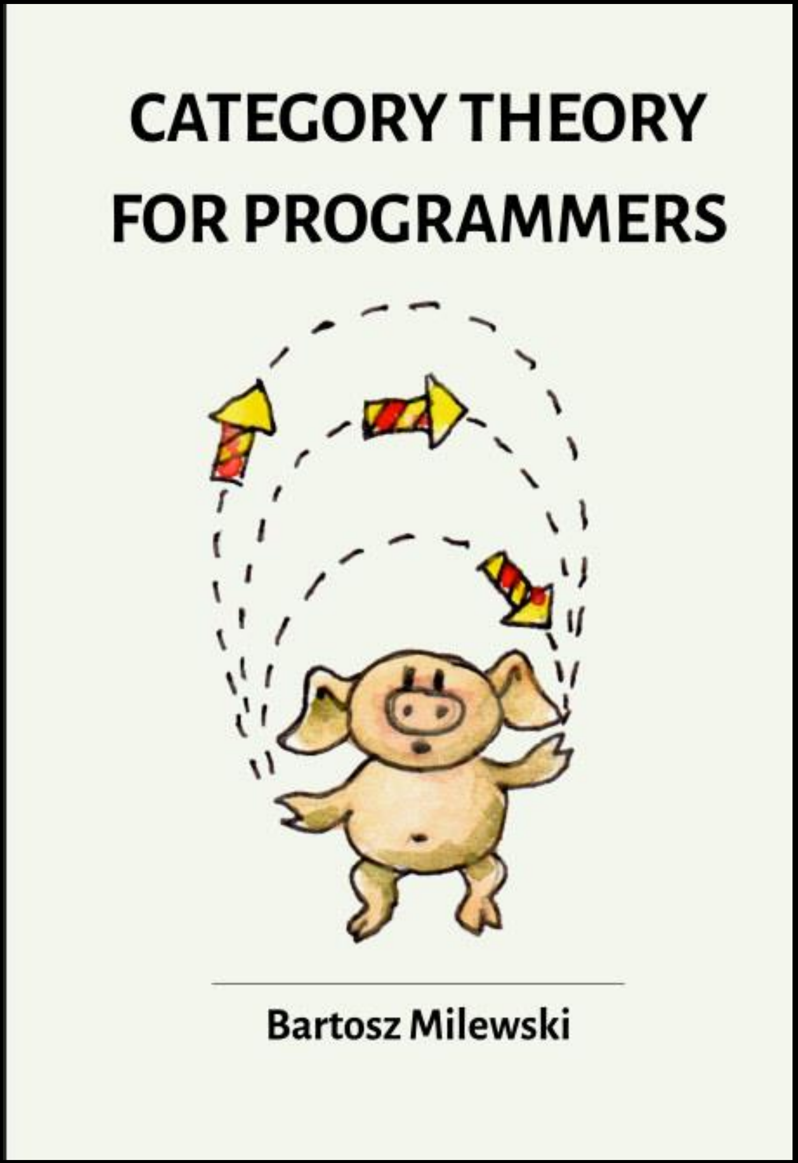
Java

Meetup

Friendly Environment Policy

Berlin Code of Conduct

DISCORD



Category Theory for Programmers

Chapter 8: Functoriality

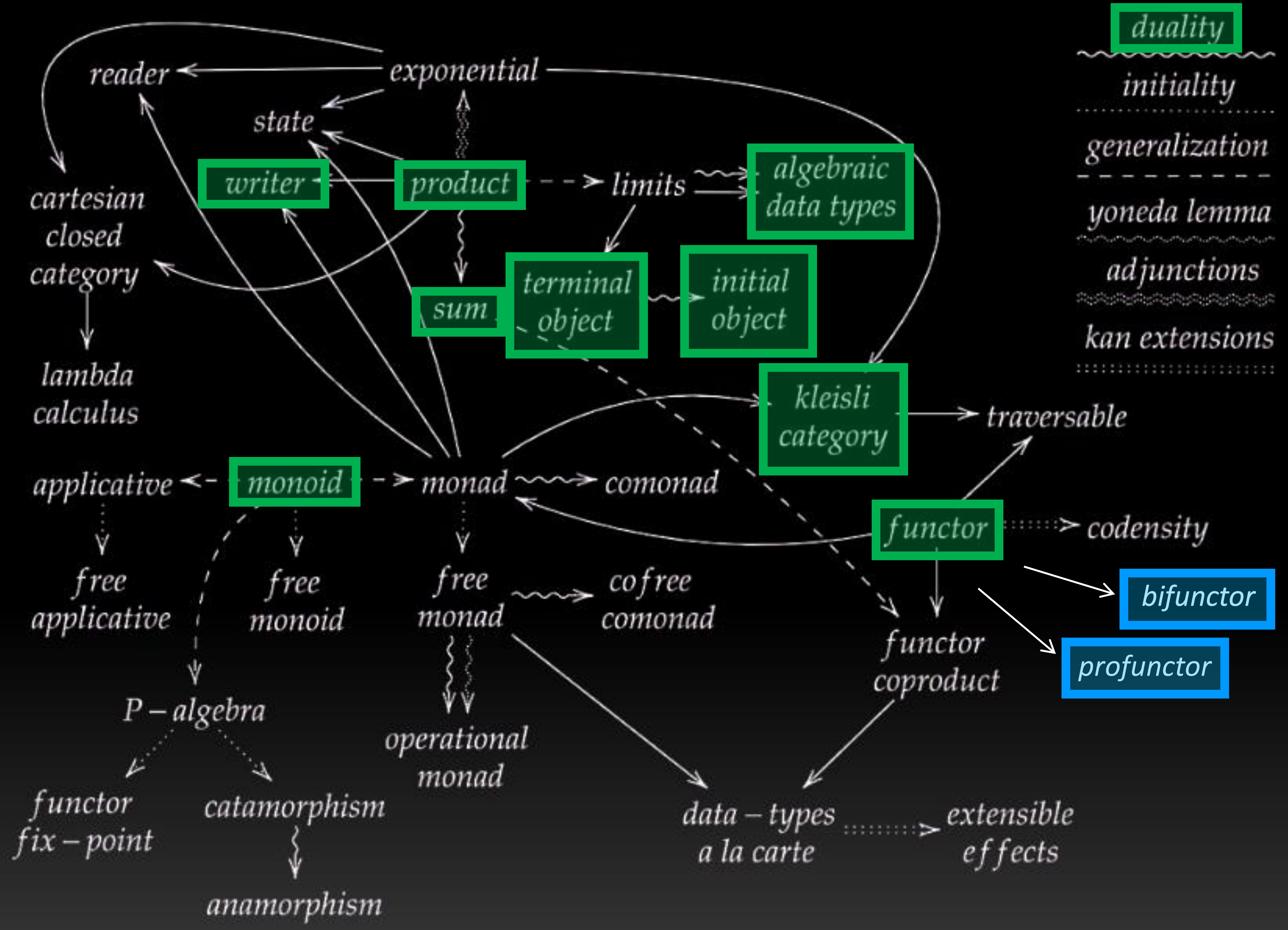
Conor Hoekstra
@code_report

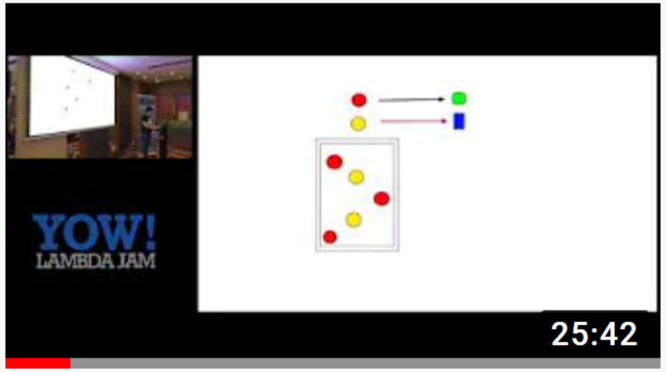
It is 9AM on Saturday morning and I've been studying #CategoryTheory since 7:30 ... how it's going:

😃 functor
🙂 bifunctor
😐 covariant functor
🙁 contravariant functor
☹️ profunctor
😭 hom-functor
😭😭 The diagonal part of the bifunctor is just a functor

8:59 AM · Mar 20, 2021 · Twitter Web App

The Tools for Thought

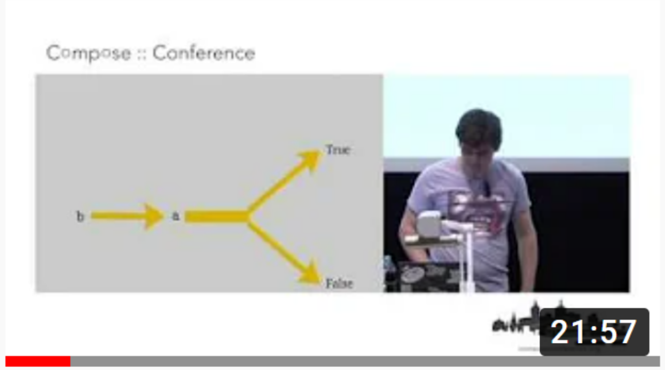




YOW! Lambda Jam 2017 George Wilson - The Extended Functor…

YOW! Conferences • 1K views

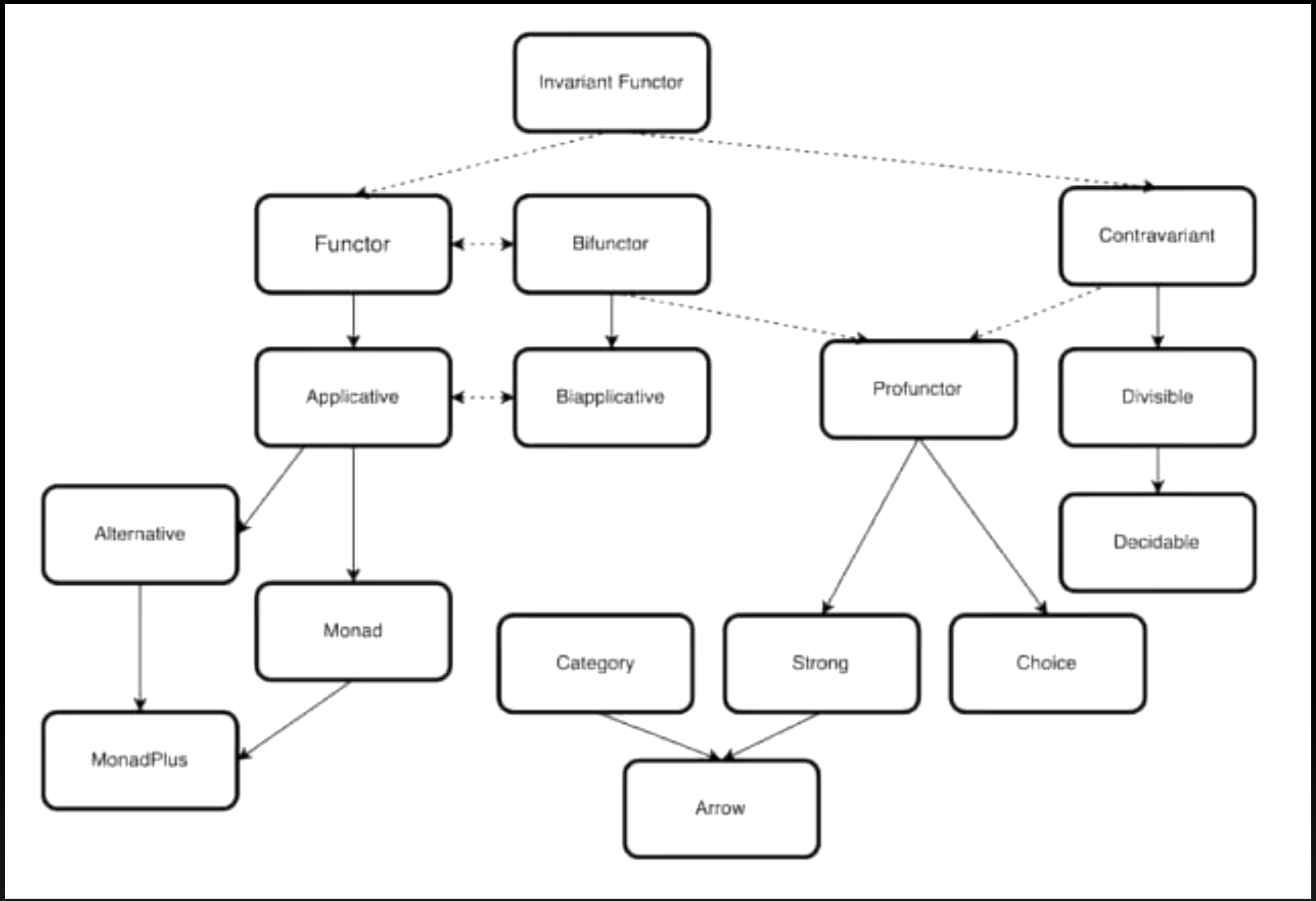
Functors are ubiquitous in modern strongly-typed functional programming. Every Haskell beginner will come across them as

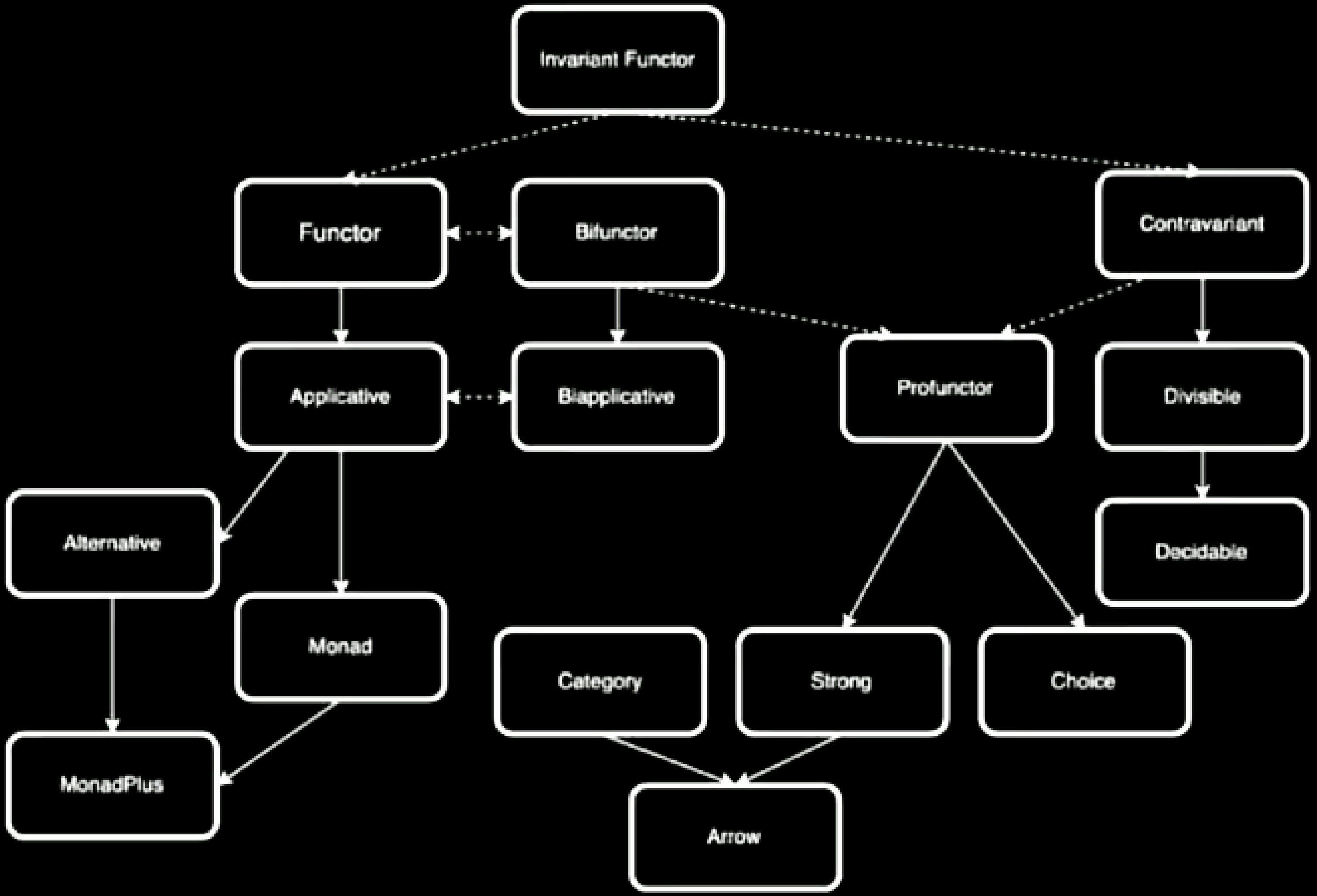


George Wilson - The Extended Functor Family

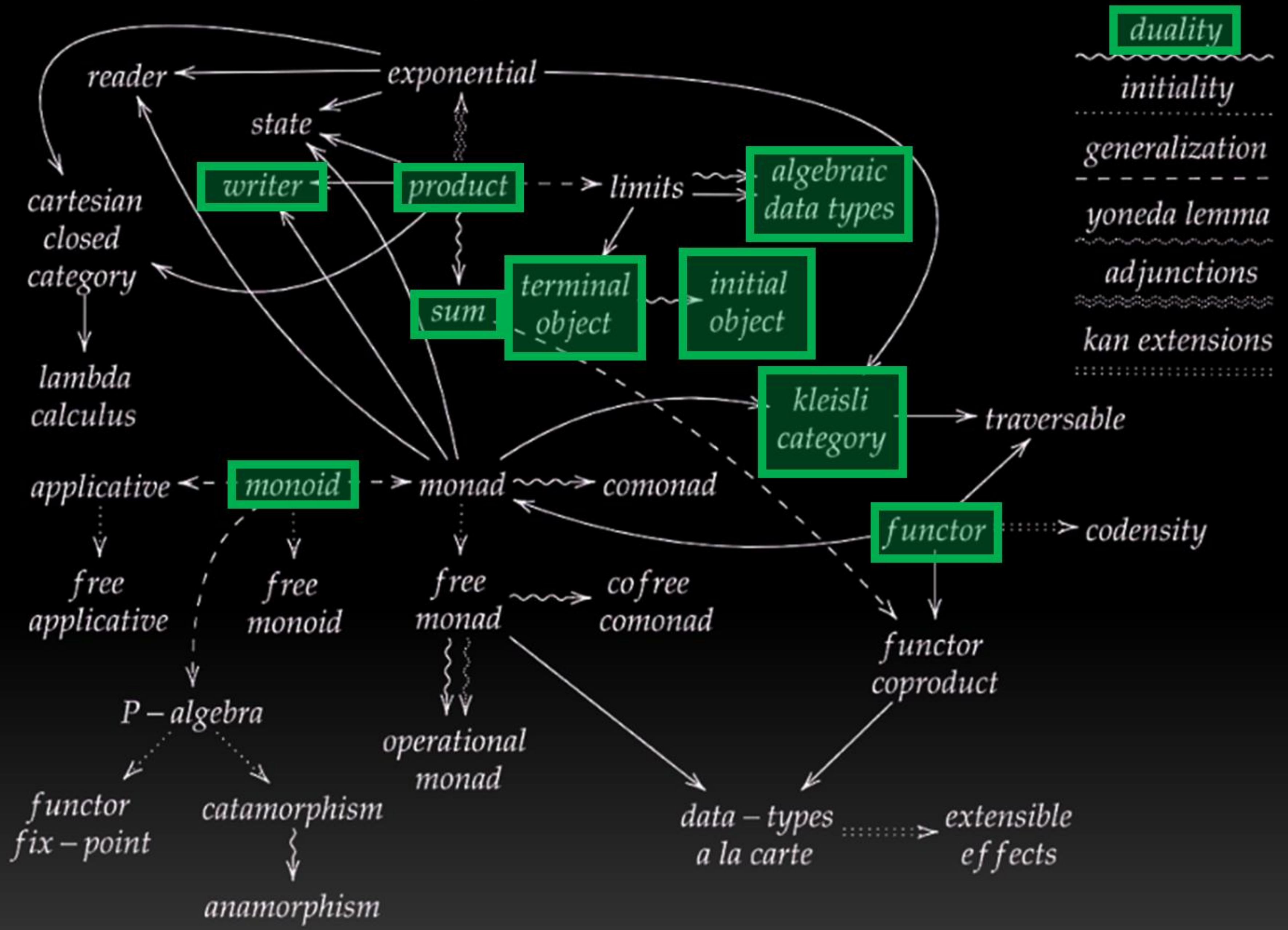
Compose Conference • 11K views

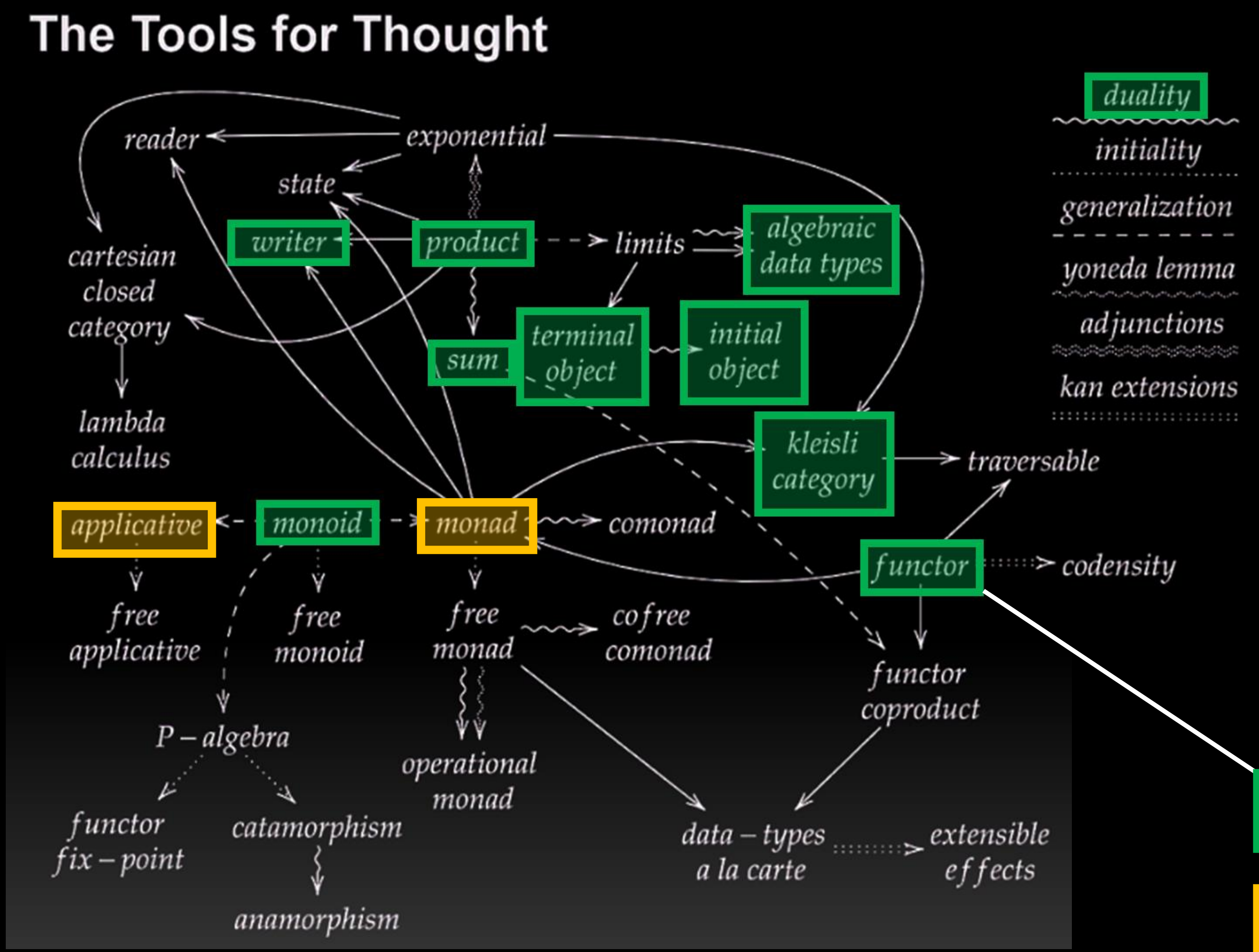
George Wilson's talk at Compose :: Conference in Melbourne, 2016. -- Functors are ubiquitous in modern strongly-typed

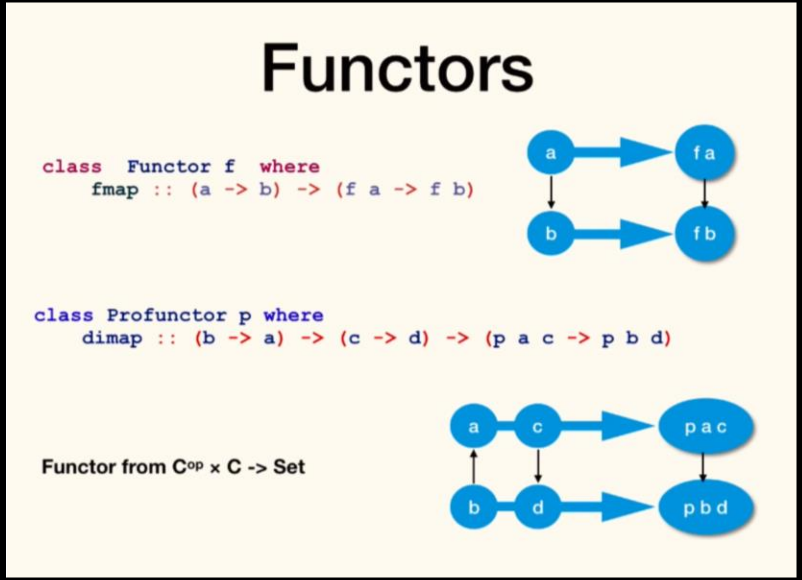

Invariant Functor
Functor
Bifunctor
Contravariant
Applicative
Biapplicative
Profunctor
Divisible
Alternative
Decidable
Monad
Category
Strong
Choice
MonadPlus
Arrow

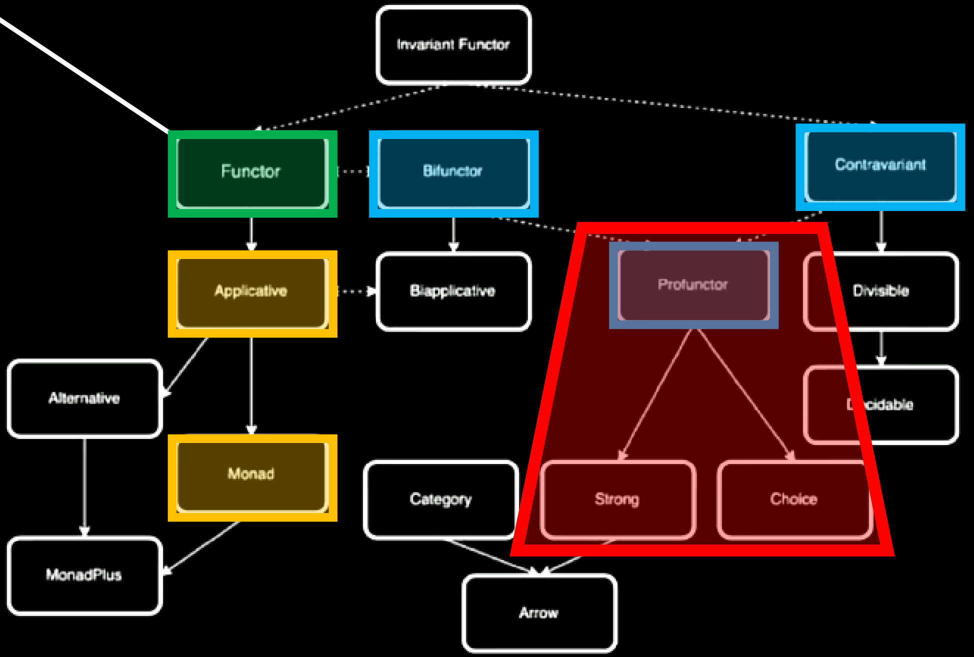

Invariant Functor
Functor
Bifunctor
Contravariant
Applicative
Biapplicative
Profunctor
Divisible
Alternative
Monad
Decidable
Category
Strong
Choice
MonadPlus
Arrow

The Tools for Thought




The Tools for Thought
reader
exponential
state
writer
product
limits
algebraic data types
cartesian closed category
sum
terminal object
initial object
lambda calculus
kleisli category
traversable
applicative
monoid
monad
comonad
functor
codensity
free applicative
free monoid
free monad
cofree comonad
functor coproduct
P – algebra
operational monad
functor fix – point
catamorphism
data – types a la carte
extensible effects
anamorphism
duality
initiality
generalization
yoneda lemma
adjunctions
kan extensions


Functors
class Functor f where
fmap :: (a -> b) -> (f a -> f b)
class Profunctor p where
dimap :: (b -> a) -> (c -> d) -> (p a c -> p b d)
Functor from Cop × C -> Set
a
fa
b
fb
c
d
pac
pbd


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


LAMBDA.WORLD
CÁDIZ 2017
MIDDAY KEYNOTE:
PROFUNCTOR OPTICS: THE CATEGORICAL APPROACH
Bartosz Milewski
Hosted by
45:34
Profunctor Optics: The Categorical Approach - Bartosz...
Lambda World • 10K views
This presentation was the midday keynote of Lambda World 2017 by Bartosz Milewski. Follow: -

8.1 Bifunctors

Since functors are morphisms in **Cat** (the category of categories), a lot of intuitions about morphisms — and functions in particular — apply to functors as well. For instance, just like you can have a function of two arguments, you can have a functor of two arguments, or a *bifunctor*. On objects, a bifunctor maps every pair of objects, one from category **C**, and one from category **D**, to an object in category **E**.

```haskell
class Bifunctor f where
    bimap :: (a -> c) -> (b -> d) -> f a b -> f c d
    bimap g h = first g . second h
    first :: (a -> c) -> f a b -> f c b
    first g = bimap g id
    second :: (b -> d) -> f a b -> f a d
    second = bimap id
```

```haskell
instance Bifunctor Either where
    bimap f _ (Left x) = Left (f x)
    bimap _ g (Right y) = Right (g y)
```

```haskell
-- Tuple below is product type
bimap (+1) (*3) (2, 3)     -- (3, 9)

-- Either below is sum (coproduct) type
bimap (+1) (*3) (Left 3)   -- Left 4
bimap (+1) (*3) (Right 3) -- Right 9
```

8.6 Covariant and Contravariant Functors

A short recap: For every category **C** there is a dual category $\mathbf{C}^{op}$. It's a category with the same objects as **C**, but with all the arrows reversed.

Consider a functor that goes between $\mathbf{C}^{op}$ and some other category **D**:

$$F :: \mathbf{C}^{op} \to \mathbf{D}$$

Such a functor maps a morphism $f^{op} :: a \to b$ in $\mathbf{C}^{op}$ to the morphism $F f^{op} :: F a \to F b$ in **D**. But the morphism f^{op} secretly corresponds to some morphism $f :: b \to a$ in the original category **C**. Notice the inversion.

Now, F is a regular functor, but there is another mapping we can define based on F, which is not a functor — let's call it G. It's a mapping from **C** to **D**. It maps objects the same way F does, but when it comes to mapping morphisms, it reverses them. It takes a morphism $f :: b \to a$ in **C**, maps it first to the opposite morphism $f^{op} :: a \to b$ and then uses the functor F on it, to get $F f^{op} :: F a \to F b$.

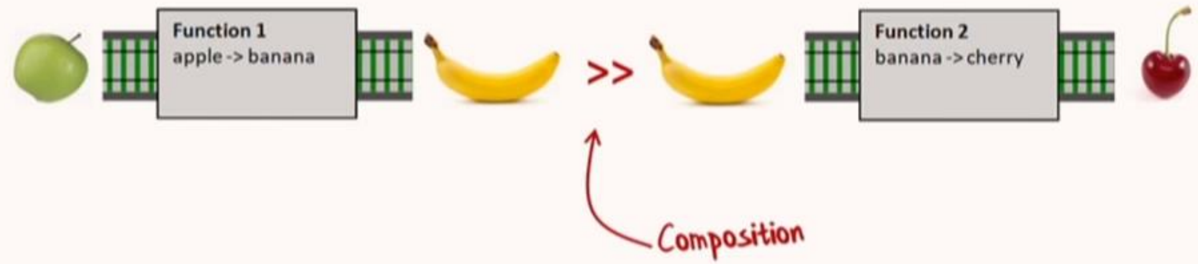
Considering that Fa is the same as Ga and Fb is the same as Gb, the whole trip can be described as: $Gf :: (b \to a) \to (Ga \to Gb)$ It's a "functor with a twist." A mapping of categories that inverts the direction of morphisms in this manner is called a *contravariant functor*. Notice that a contravariant functor is just a regular functor from the opposite category. The regular functors, by the way — the kind we've been studying thus far — are called *covariant* functors.

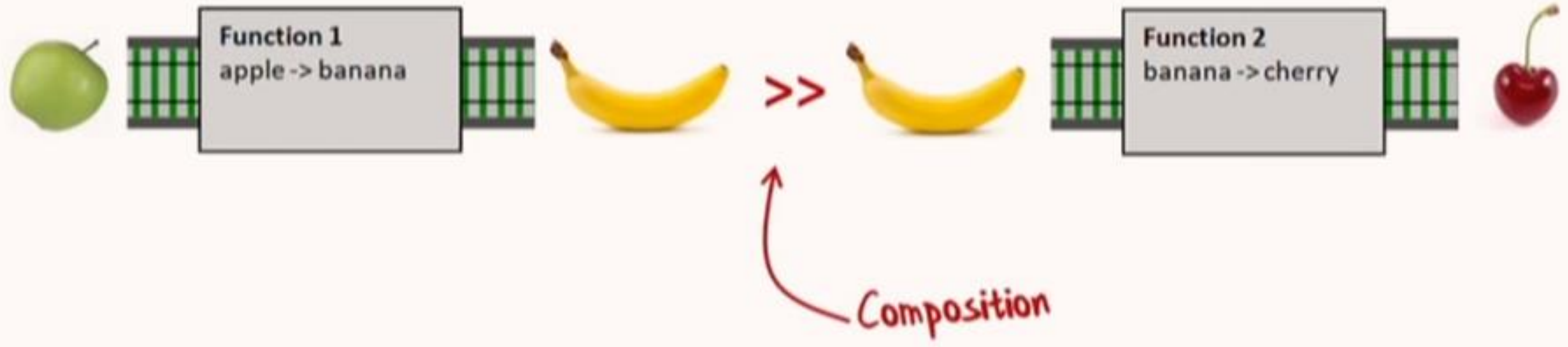
```haskell
class Contravariant f where
    contramap :: (b -> a) -> f a -> f b
```

https://youtu.be/WhEkBCWpDas

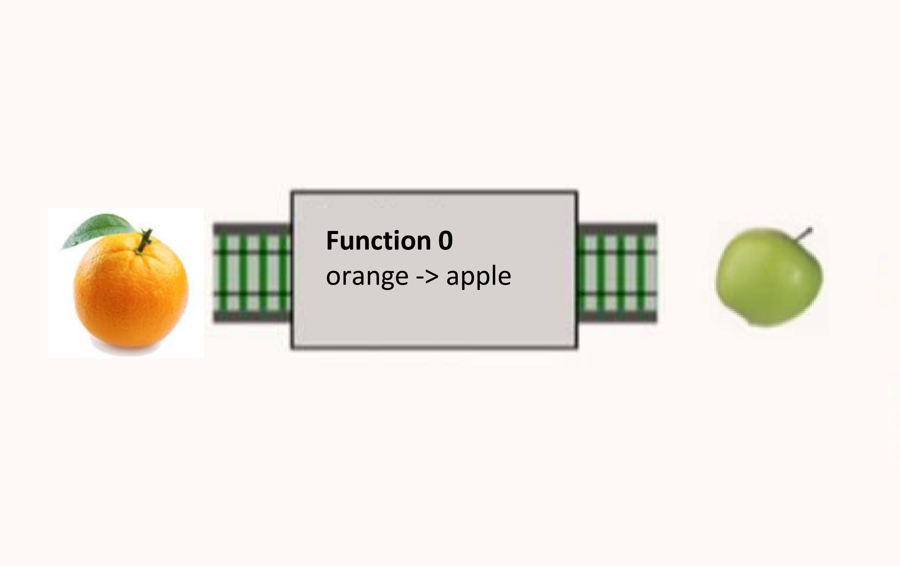
NDC { Oslo }

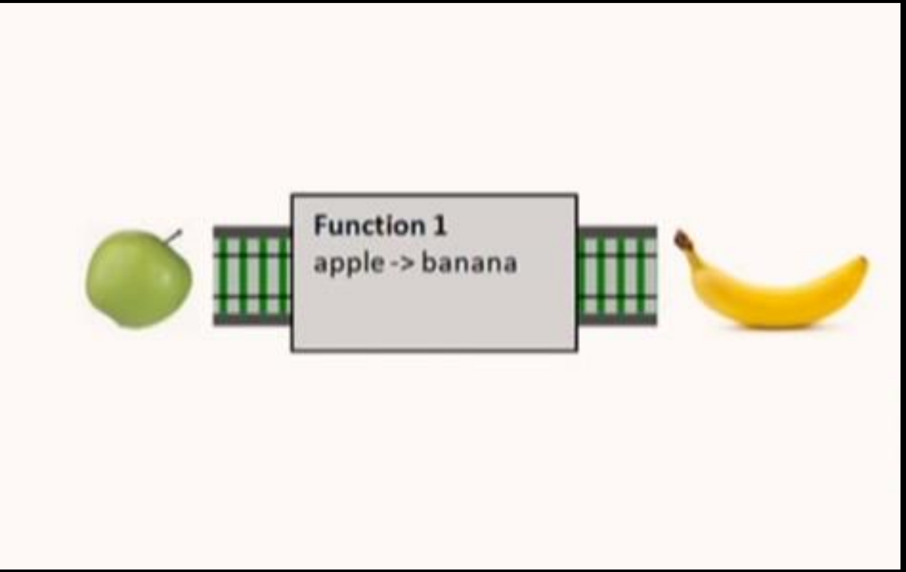
3


Function 1
apple -> banana
>>
Function 2
banana -> cherry
Composition



Function 1
apple -> banana
>>
Function 2
banana -> cherry
Composition

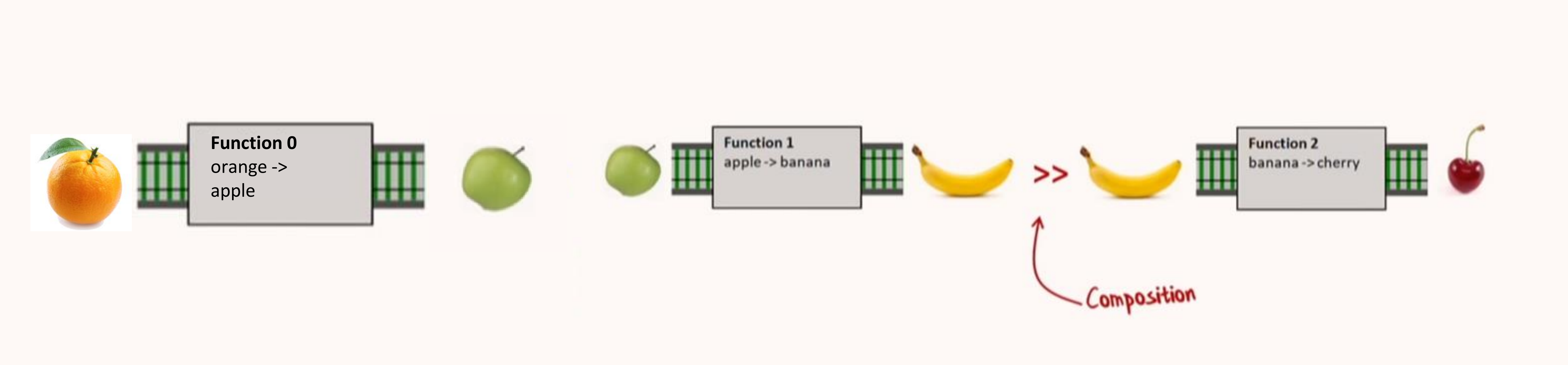

Function 0
orange -> apple


Function 1
apple -> banana

8.7 Profunctors

We've seen that the function-arrow operator is contravariant in its first argument and covariant in the second. Is there a name for such a beast? It turns out that, if the target category is **Set**, such a beast is called a *profunctor*. Because a contravariant functor is equivalent to a covariant functor from the opposite category, a profunctor is defined as:

$$\mathbf{C}^{op} \times \mathbf{D} \to \mathbf{Set}$$


Function 0
orange ->
apple
Function 1
apple -> banana
>>
Function 2
banana -> cherry
Composition

```haskell
class Profunctor p where
    dimap :: (a -> b) -> (c -> d) -> p b c -> p a d
    dimap f g = lmap f . rmap g
    lmap :: (a -> b) -> p b c -> p a c
    lmap f = dimap f id
    rmap :: (b -> c) -> p a b -> p a c
    rmap = dimap id
```

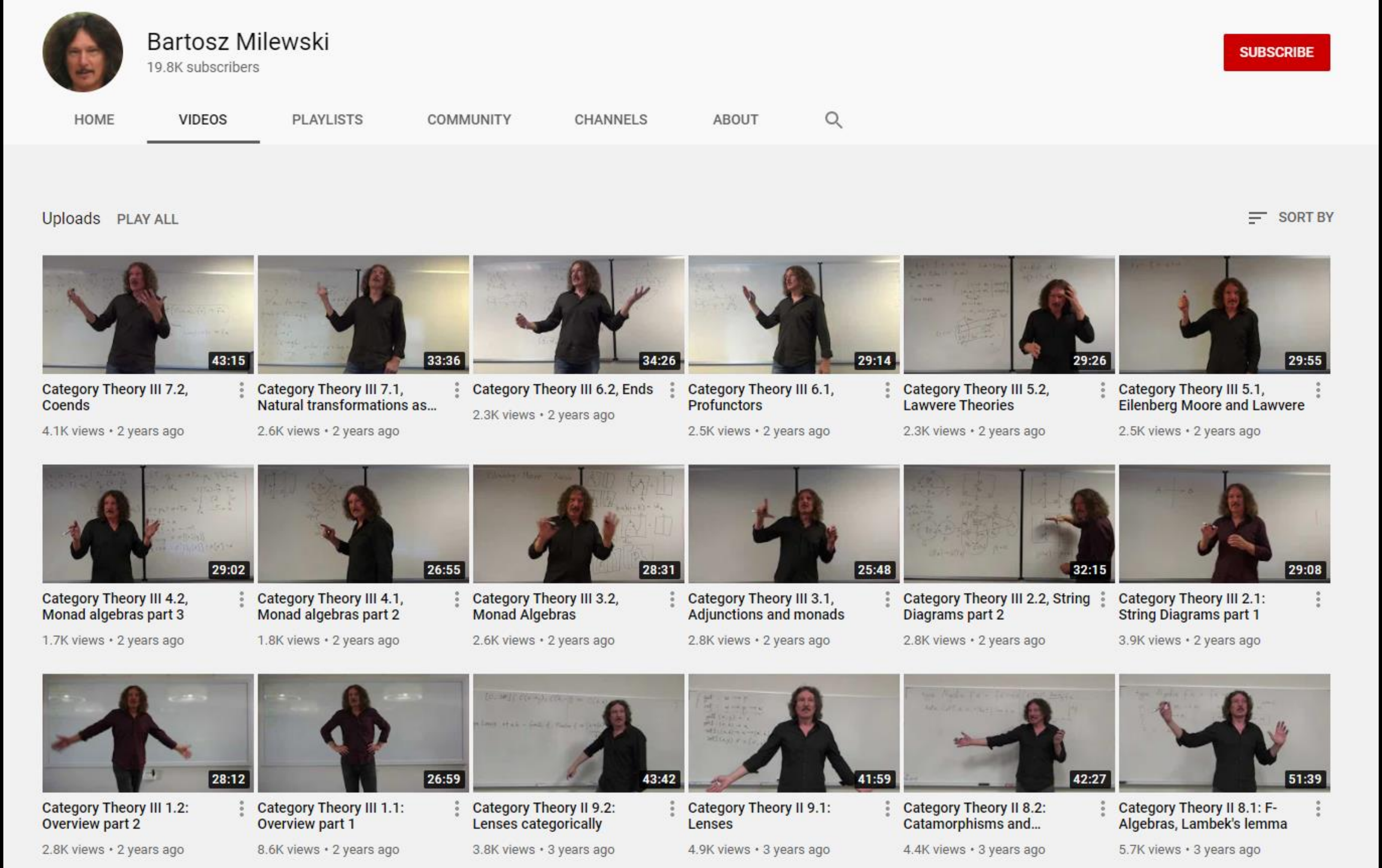
```haskell
class Profunctor p where
    dimap :: (a -> b) -> (c -> d) -> p b c -> p a d
    dimap f g = lmap f . rmap g
    lmap :: (a -> b) -> p b c -> p a c
    lmap f = dimap f id
    rmap :: (b -> c) -> p a b -> p a c
    rmap = dimap id
```

5. Define a bifunctor in a language other than Haskell. Implement `bimap` for a generic pair in that language.

```cpp
using Fn = int(int); // cheating for simplicity

template <typename B>
concept bifunctor = requires(B bf, Fn f, Fn g) {
    { bf.bimap(f, g) } -> std::same_as<B>;
};

struct pair {
    int a, b;
    auto bimap(auto f, auto g) const {
        return pair{f(a), g(b)};
    }
};
```


Bartosz Milewski
19.8K subscribers
SUBSCRIBE
HOME
VIDEOS
PLAYLISTS
COMMUNITY
CHANNELS
ABOUT
Uploads
PLAY ALL
SORT BY
43:15
Category Theory III 7.2, Coends
4.1K views • 2 years ago
33:36
Category Theory III 7.1, Natural transformations as...
2.6K views • 2 years ago
34:26
Category Theory III 6.2, Ends
2.3K views • 2 years ago
29:14
Category Theory III 6.1, Profunctors
2.5K views • 2 years ago
29:26
Category Theory III 5.2, Lawvere Theories
2.3K views • 2 years ago
29:55
Category Theory III 5.1, Eilenberg Moore and Lawvere
2.5K views • 2 years ago
29:02
Category Theory III 4.2, Monad algebras part 3
1.7K views • 2 years ago
26:55
Category Theory III 4.1, Monad algebras part 2
1.8K views • 2 years ago
28:31
Category Theory III 3.2, Monad Algebras
2.6K views • 2 years ago
25:48
Category Theory III 3.1, Adjunctions and monads
2.8K views • 2 years ago
32:15
Category Theory III 2.2, String Diagrams part 2
2.8K views • 2 years ago
29:08
Category Theory III 2.1: String Diagrams part 1
3.9K views • 2 years ago
28:12
Category Theory III 1.2: Overview part 2
2.8K views • 2 years ago
26:59
Category Theory III 1.1: Overview part 1
8.6K views • 2 years ago
43:42
Category Theory II 9.2: Lenses categorically
3.8K views • 3 years ago
41:59
Category Theory II 9.1: Lenses
4.9K views • 3 years ago
42:27
Category Theory II 8.2: Catamorphisms and...
4.4K views • 3 years ago
51:39
Category Theory II 8.1: F-Algebras, Lambek's lemma
5.7K views • 3 years ago

Java

Meetup