Java

Meetup

Friendly Environment Policy

Berlin Code of Conduct

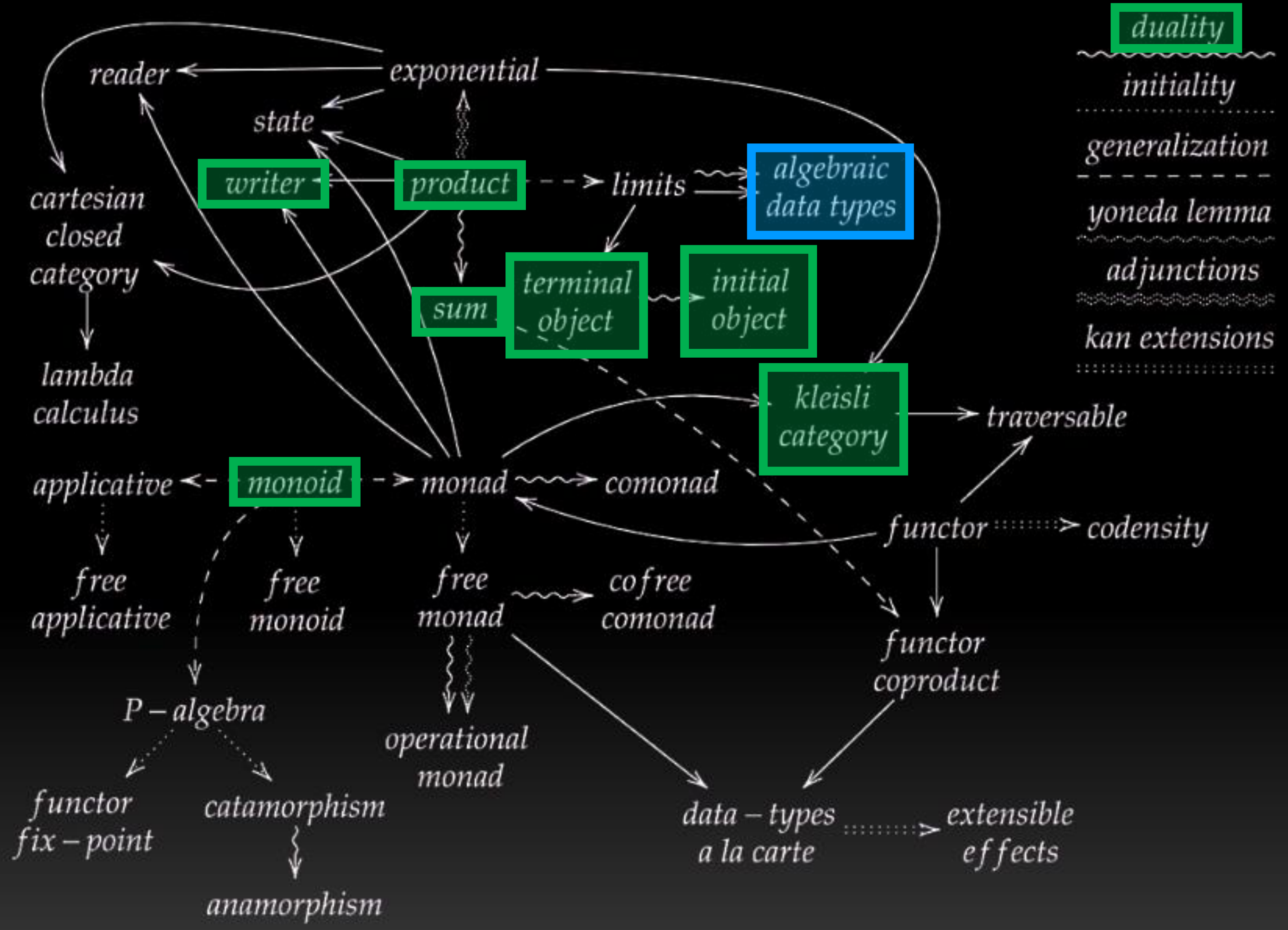
DISCORD

Category Theory for Programmers

Chapter 6:

Simple Algebraic Data Types

The Tools for Thought



6.1 Product Types

The canonical implementation of a product of two types in a programming language is a pair. In Haskell, a pair is a primitive type constructor; in C++ it's a relatively complex template defined in the Standard Library.

These observations can be formalized by saying that **Set** (the category of sets) is a *monoidal category*. It's a category that's also a monoid, in the sense that you can multiply objects (here, take their Cartesian product). I'll talk more about monoidal categories, and give the full definition in the future.

6.2 Records

```haskell
startsWithSymbol :: (String, String, Int) -> Bool
startsWithSymbol (name, symbol, _) = isPrefixOf symbol name
```

This code is error prone, and is hard to read and maintain. It's much better to define a record:

```haskell
data Element = Element { name :: String
                       , symbol :: String
                       , atomicNumber :: Int }
```

The two representations are isomorphic, as witnessed by these two conversion functions

6.3 Sum Types

Just as the product in the category of sets gives rise to product types, the coproduct gives rise to sum types. The canonical implementation of a sum type in Haskell is:

```haskell
data Either a b = Left a | Right b
```

It turns out that **Set** is also a (symmetric) monoidal category with respect to coproduct. The role of the binary operation is played by the disjoint sum, and the role of the unit element is played by the initial object.

Simple sum types that encode the presence or absence of a value are variously implemented in C++ using special tricks and “impossible” values, like empty strings, negative numbers, null pointers, etc. This kind of optionality, if deliberate, is expressed in Haskell using the `Maybe` type:

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

Option	Some	None
Maybe	Just	Nothing
optional	.value()	nullopt
Optional	some	none

6.4 Algebra of Types

Taken separately, product and sum types can be used to define a variety of useful data structures, but the real strength comes from combining the two. Once again we are invoking the power of composition.

Numbers	Types
0	`Void`
1	`()`
$a + b$	`Either a b = Left a | Right b`
$a \times b$	`(a, b)` or `Pair a b = Pair a b`
$2 = 1 + 1$	`data Bool = True | False`
$1 + a$	`data Maybe = Nothing | Just a`

Logic	Types
false	`Void`
true	`()`
$a \mathbin{\|\|} b$	`Either a b = Left a | Right b`
$a \mathbin{\&\&} b$	`(a, b)`

Numbers	Types
0	`Void`
1	`()`
$a+b$	`Either a b = Left a | Right b`
$a \times b$	`(a, b)` or `Pair a b = Pair a b`
$2=1+1$	`data Bool = True | False`
$1+a$	`data Maybe = Nothing | Just a`

Logic	Types
false	`Void`
true	`()`
$a \mid\mid b$	`Either a b = Left a | Right b`
$a \mathbin{\&\&} b$	`(a, b)`

This analogy goes deeper, and is the basis of the Curry-Howard isomorphism between logic and type theory. We'll come back to it when we talk about function types.

1. Show the isomorphism between `Maybe a` and `Either () a`.

```haskell
maybeToEither :: Maybe a -> Either () a
maybeToEither (Just x) = Right x
maybeToEither Nothing  = Left ()

eitherToMaybe :: Either () a -> Maybe a
eitherToMaybe (Left ()) = Nothing
eitherToMaybe (Right x) = Just x

main :: IO ()
main = do
    print $ maybeToEither (Just 42)    -- Right 42
    print $ eitherToMaybe (Right 1729) -- Just 1729
```

Implement `Shape` in C++ or Java as an interface and create two classes: `Circle` and `Rect`. Implement `area` as a virtual function.

Concepts vs Typeclasses vs Traits vs Protocols vs Type Constraints

Conor Hoekstra

code_report

NVIDIA. RAPIDS

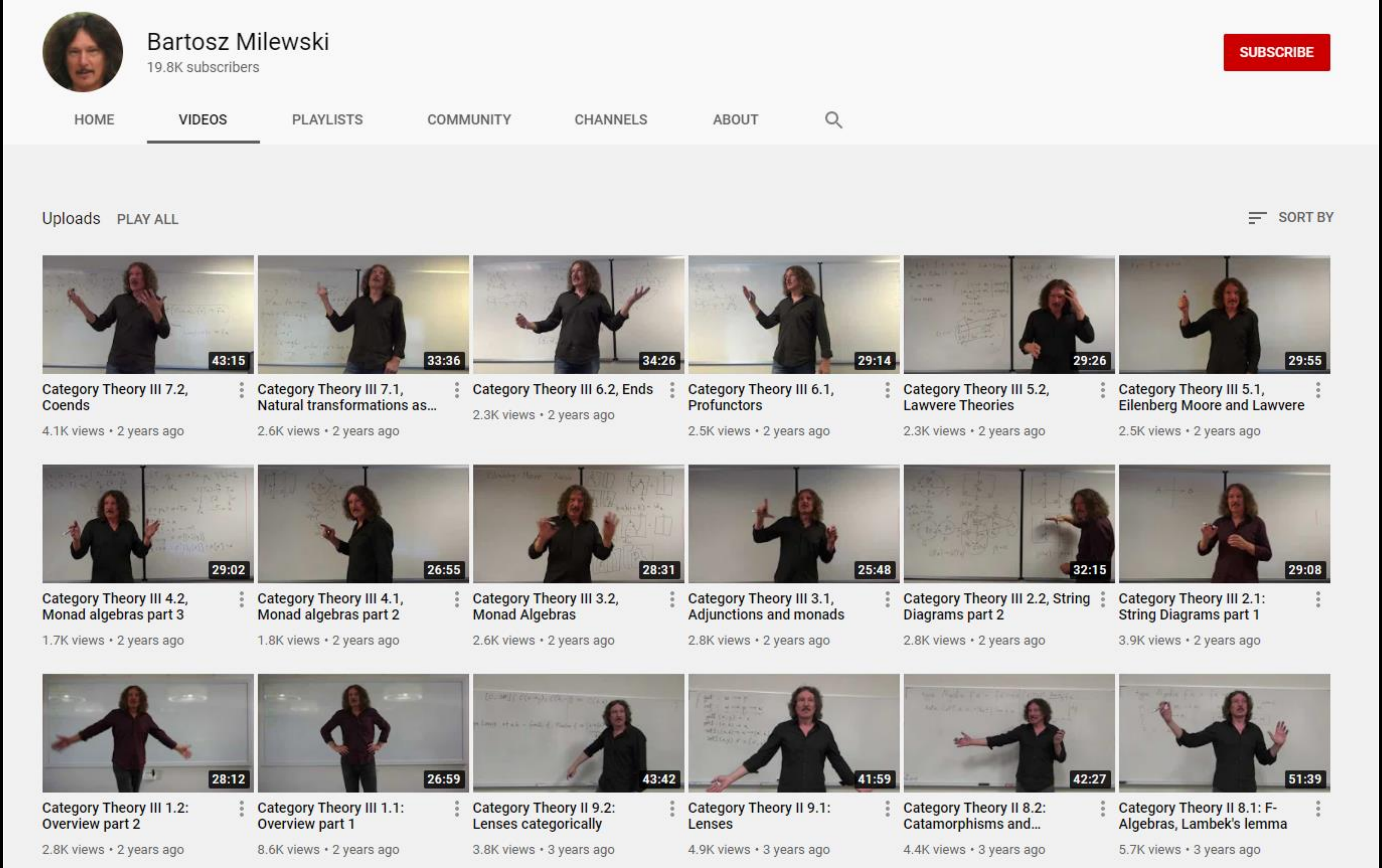
#include

```swift
protocol Shape {
    func name()      -> String
    func area()      -> Float
    func perimeter() -> Float
}

class Rectangle : Shape {
    let w, h: Float
    init(w: Float, h: Float) { self.w = w; self.h = h }
    func name()      -> String { "Rectangle" }
    func area()      -> Float  { w * h }
    func perimeter() -> Float  { 2 * w + 2 * h }
}

class Circle : Shape {
    let r: Float
    init(r: Float) { self.r = r }
    func name()      -> String { "Circle" }
    func area()      -> Float  { Float.pi * r * r }
    func perimeter() -> Float  { 2 * Float.pi * r }
}
```

```swift
class Square : Shape {
    let w: Float
    init(w: Float) { self.w = w }
    func name()      -> String { "Square" }
    func area()      -> Float  { w * w }
    func perimeter() -> Float  { 4 * w }
}
```


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