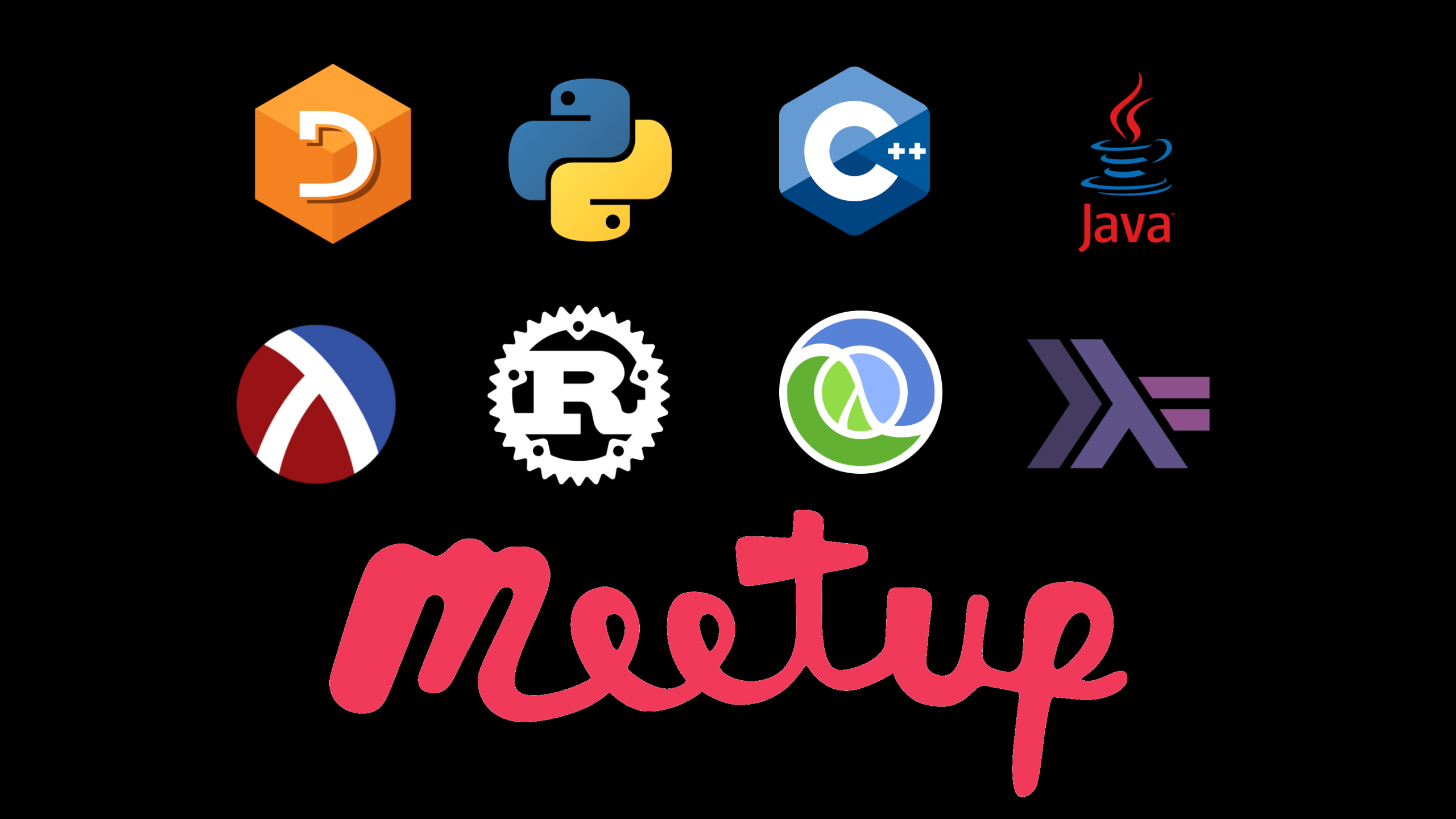
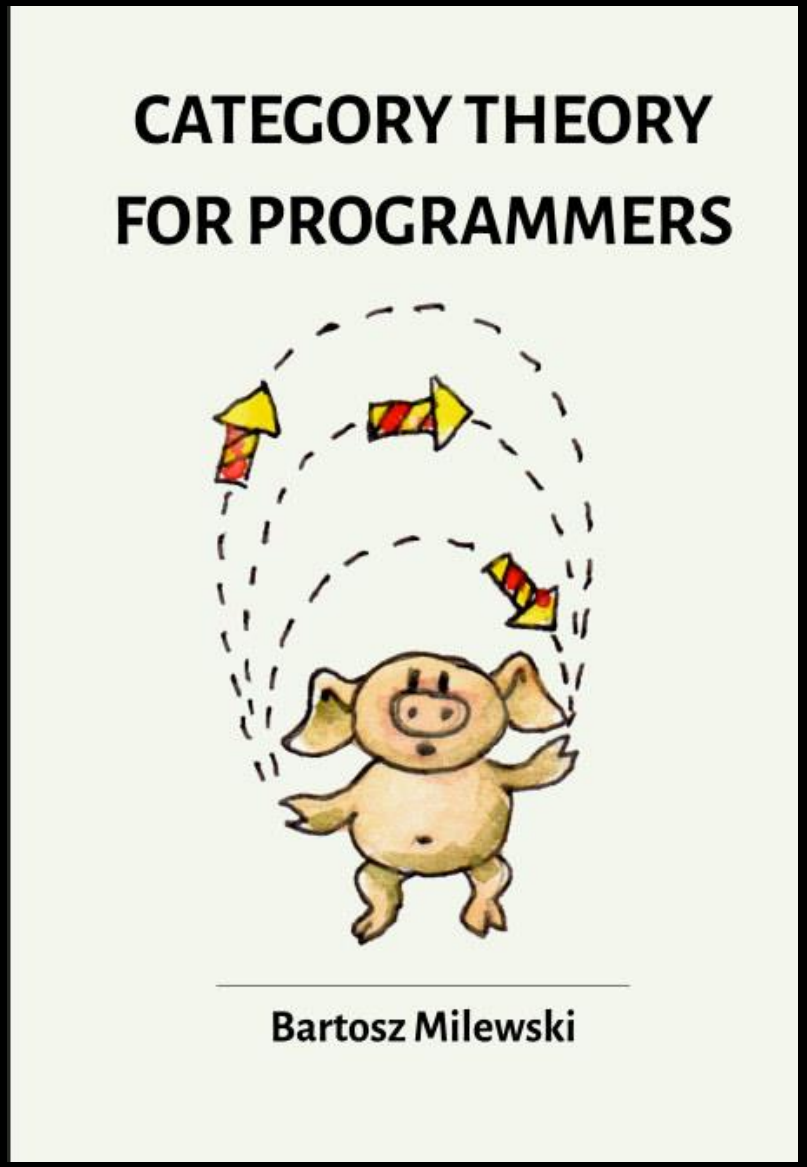

Java
meetup

Friendly Environment Policy

Berlin Code of Conduct



Category Theory for Programmers

Chapter 3:

Categories Great & Small

3.1 No Objects

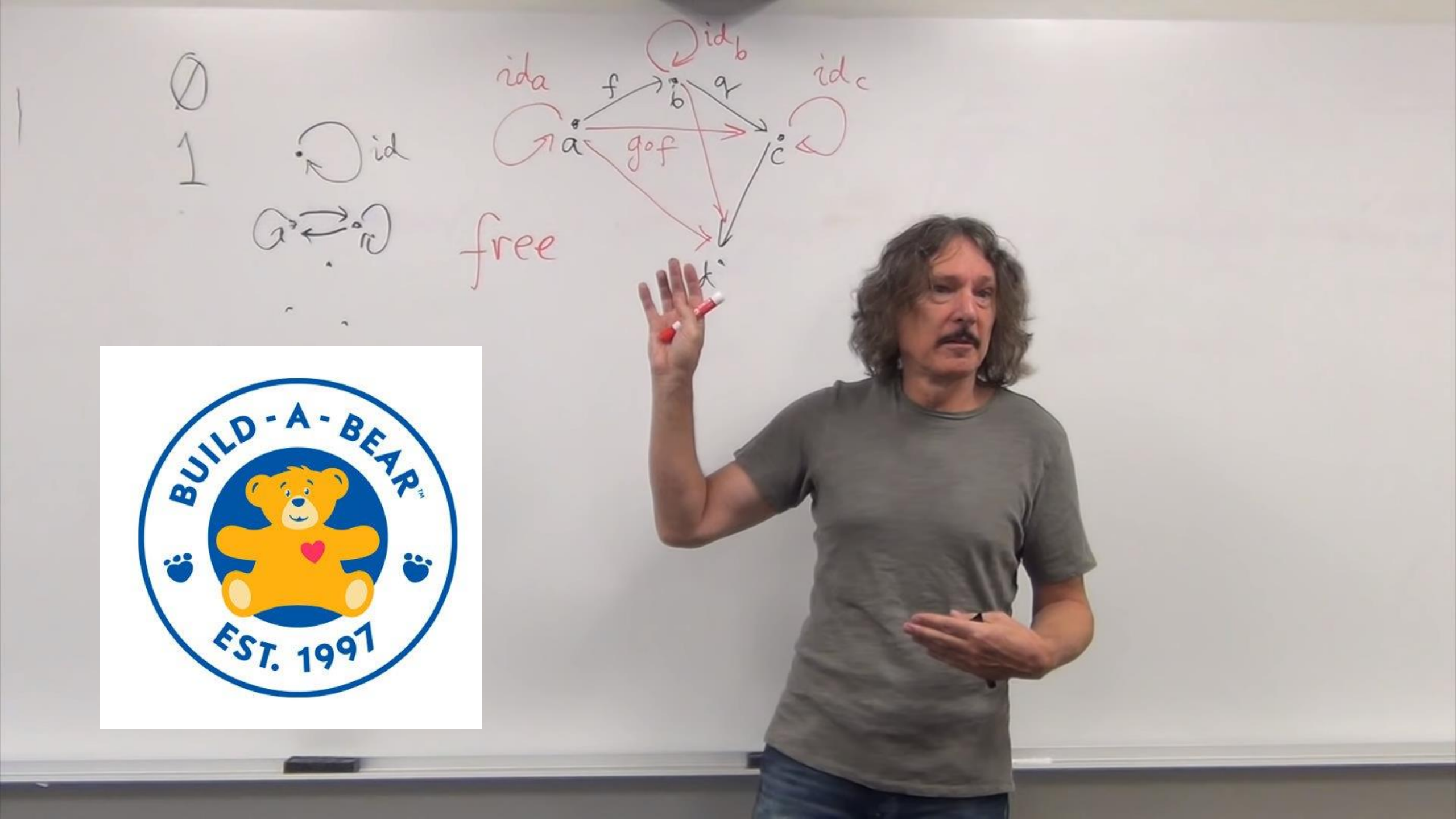
The most trivial category is one with zero objects and, consequently, zero morphisms. It's a very sad category by itself, but it may be important in the context of other categories, for instance, in the category of all categories (yes, there is one). If you think that an empty set makes sense, then why not an empty category?

3.2 Simple Graphs

You can build categories just by connecting objects with arrows. You can imagine starting with any directed graph and making it into a category by simply adding more arrows. First, add an identity arrow at each node. Then, for any two arrows such that the end of one coincides with the beginning of the other (in other words, any two *composable* arrows), add a new arrow to serve as their composition. Every time you add a new arrow, you have to also consider its composition with any other arrow (except for the identity arrows) and itself. You usually end up with infinitely many arrows, but that's okay.

Another way of looking at this process is that you're creating a category, which has an object for every node in the graph, and all possible *chains* of composable graph edges as morphisms. (You may even consider identity morphisms as special cases of chains of length zero.)

Such a category is called a *free category* generated by a given graph. It's an example of a free construction, a process of completing a given structure by extending it with a minimum number of items to satisfy its laws (here, the laws of a category). We'll see more examples of it in the future.


∅
1
id
free
BUILD-A-BEAR™
EST. 1997

3.4 Monoid as Set

Monoid is an embarrassingly simple but amazingly powerful concept.

Traditionally, a monoid is defined as a set with a binary operation. All that's required from this operation is that it's associative, and that there is one special element that behaves like a unit with respect to it.

For instance, natural numbers with zero form a monoid under addition. Associativity means that:

```
(a + b) + c = a + (b + c)
```

(In other words, we can skip parentheses when adding numbers.)

The neutral element is zero, because:

```
0 + a = a
```

and

```
a + 0 = a
```

C++ now 2019 MAY 6-10 cppnow.org

Ben Deane

Identifying Monoids: Exploiting Compositional Structure in Code

WHAT IS A MONOID?

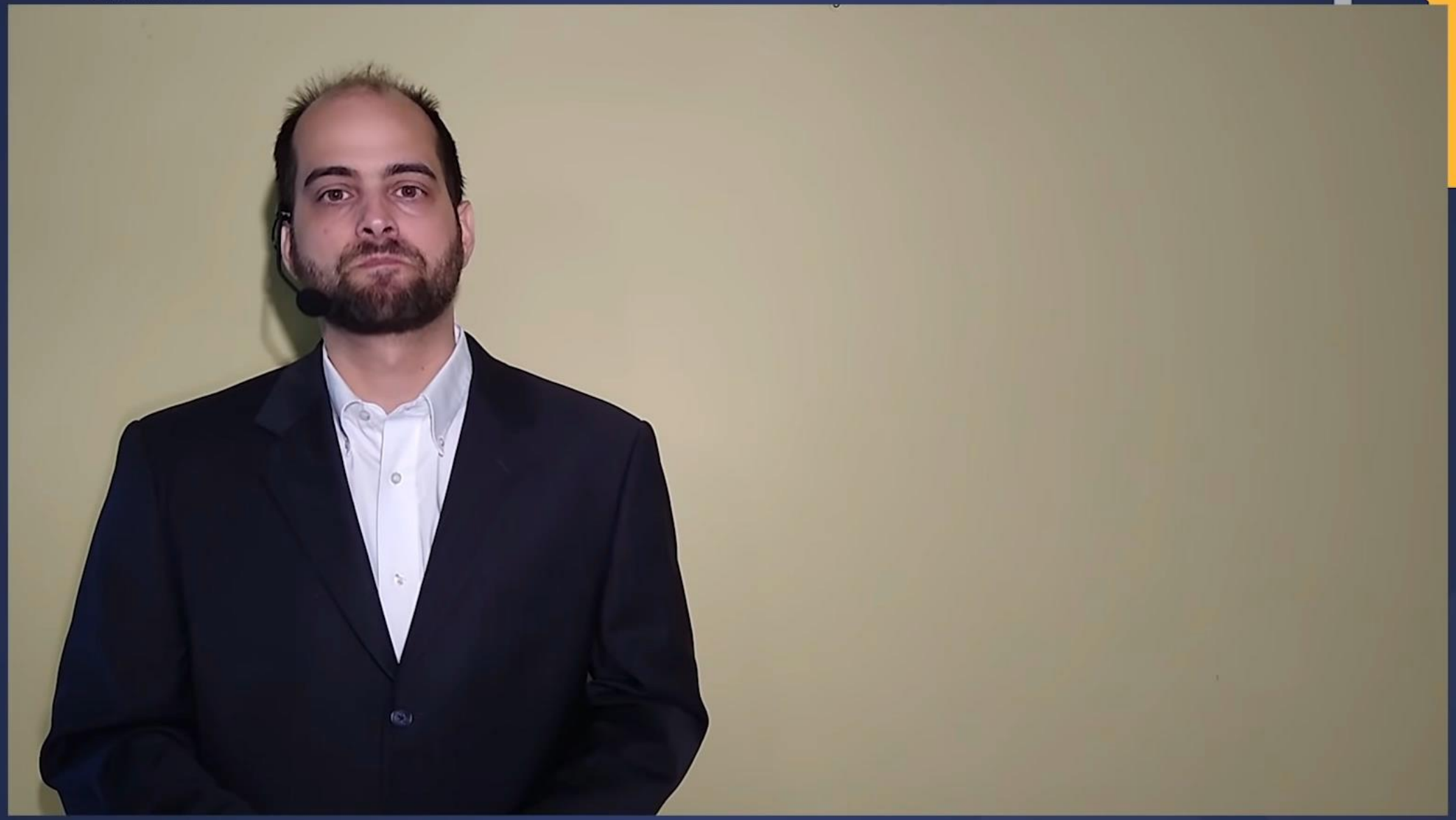
"Monoidi sunt omnes divisi in partes tres."

-- Julius Caesar, De Bello Monoido

1. A set of values.
   - finite or infinite
2. A binary operation.
   - closed
   - associative
3. One special value in the set.
   - the identity

Cppcon
The C++ Conference
September 13-18

David Sankel



Monoids, Monads, and Applicative Functors: Repeated Software Patterns

ansatz

C++ now 2019 MAY 6-10 cppnow.org

Ben Deane

Identifying Monoids: Exploiting Compositional Structure in Code

WHAT IS A MONOID?

"Monoidi sunt omnes divisi in partes tres."

-- Julius Caesar, De Bello Monoido

1. A set of values.
   - finite or infinite
2. A binary operation.
   - closed
   - associative
3. One special value in the set.
   - the identity

Ben Deane

Identifying Monoids: Exploiting Compositional Structure in Code

THE OBVIOUS MONOIDS

There's a reason why the default operation of `accumulate` is addition.

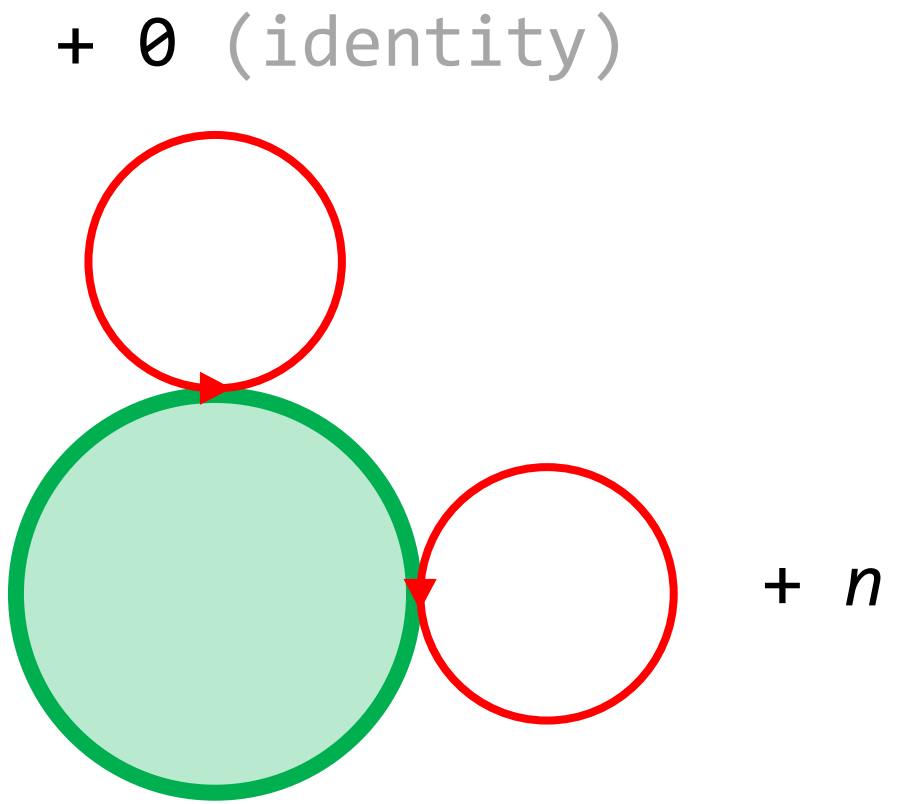
- $\{\mathbb{R}, +, 0\}$
- $\{\mathbb{R}, \times, 1\}$

For $\mathbb{R}$, read also $\mathbb{Z}$ or $\mathbb{N}$. (And also $\mathbb{C}$).

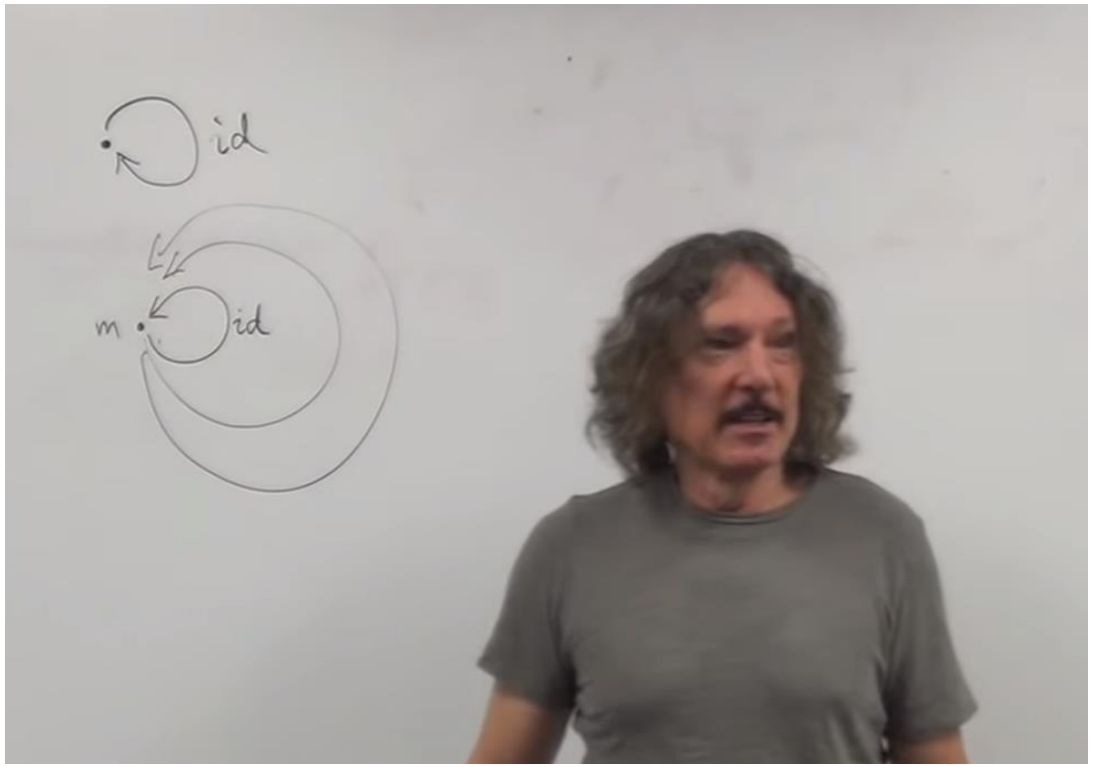
David Sankel

Monoids, Monads, and Applicative Functors: Repeated Software Patterns

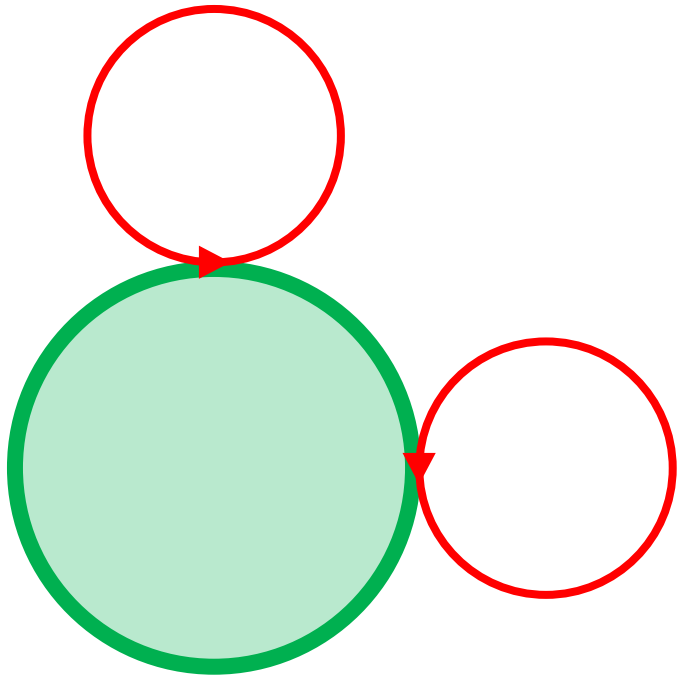
1. **Type:** Integers
2. **Binary operation:** +
3. **Identity value:** 0



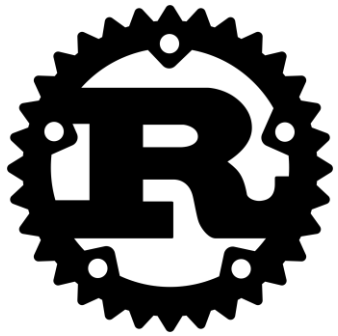
1. **Type:** Integers
2. **Binary operation:** +
3. **Identity value:** 0



+ 0 (identity)



+ *n*



```rust
fn sum(n: i32) -> i32 {
    (1..=n).fold(0, |a, b| a + b )
}

fn product(n: i32) -> i32 {
    (1..=n).fold(1, |a, b| a * b )
}
```

```
using namespace std::ranges;

auto fact(int n) -> int {
    return views::iota(1, n) | fold(1, std::multiplies{});
}
```

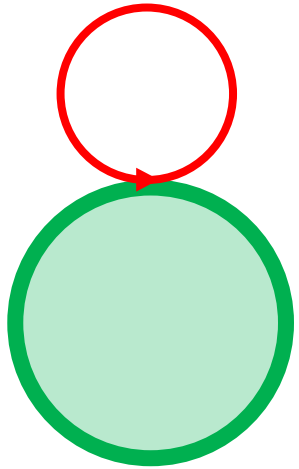
In Haskell we can define a type class for monoids — a type for which there is a neutral element called `mempty` and a binary operation called `mappend`:

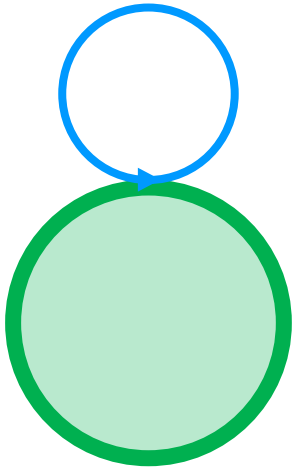
```
class Monoid m where
    mempty  :: m
    mappend :: m -> m -> m
```

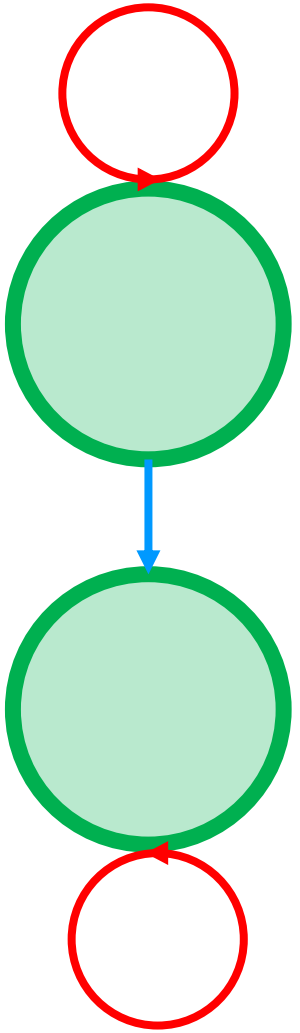
Since the values of arguments are sometimes called *points* (as in: the value of f at point x), this is called point-wise equality. Function equality without specifying the arguments is described as *point-free*. (Incidentally, point-free equations often involve composition of functions, which is symbolized by a point, so this might be a little confusing to the beginner.)

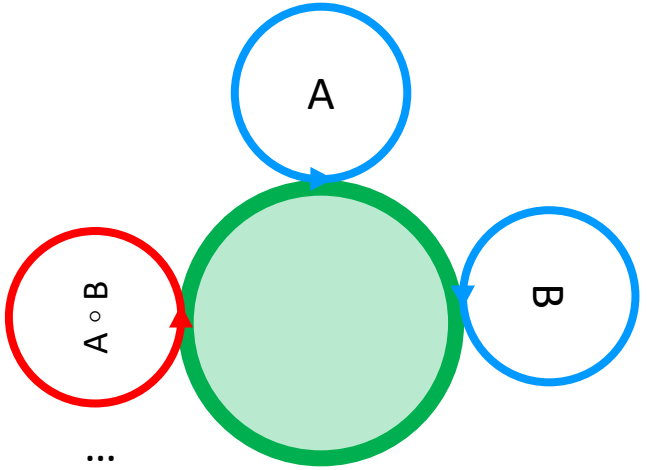
1. Generate a free category from:

   (a) A graph with one node and no edges
   (b) A graph with one node and one (directed) edge (hint: this edge can be composed with itself)
   (c) A graph with two nodes and a single arrow between them
   (d) A graph with a single node and 26 arrows marked with the letters of the alphabet: a, b, c … z.









1. Generate a free category from:

   (a) A graph with one node and no edges
   (b) A graph with one node and one (directed) edge (hint: this edge can be composed with itself)
   (c) A graph with two nodes and a single arrow between them
   (d) A graph with a single node and 26 arrows marked with the letters of the alphabet: a, b, c ... z.

3. Considering that Bool is a set of two values True and False, show that it forms two (set-theoretical) monoids with respect to, respectively, operator && (AND) and || (OR).

3. Considering that Bool is a set of two values True and False, show that it forms two (set-theoretical) monoids with respect to, respectively, operator && (AND) and || (OR).

```
a AND (∧)
1∧1 a 1
1∧0 a 0
0∧1 a 0
0∧0 a 0
∧/θ a 1 (identity)
```

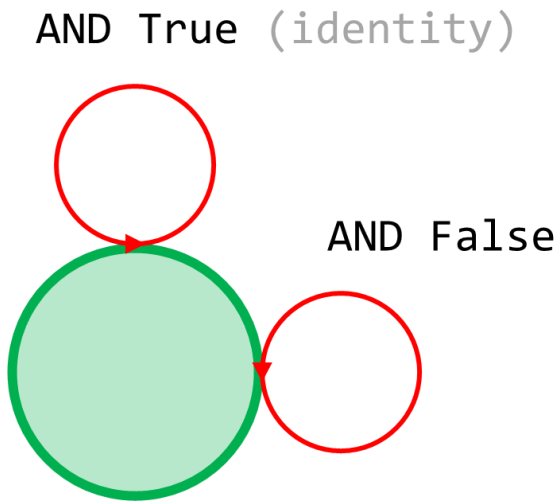
```
a OR (∨)
1∨1 a 1
1∨0 a 1
0∨1 a 1
0∨0 a 0
∨/θ a 0 (identity)
```

4. Represent the Bool monoid with the AND operator as a category: List the morphisms and their rules of composition.

4. Represent the Bool monoid with the AND operator as a category: List the morphisms and their rules of composition.

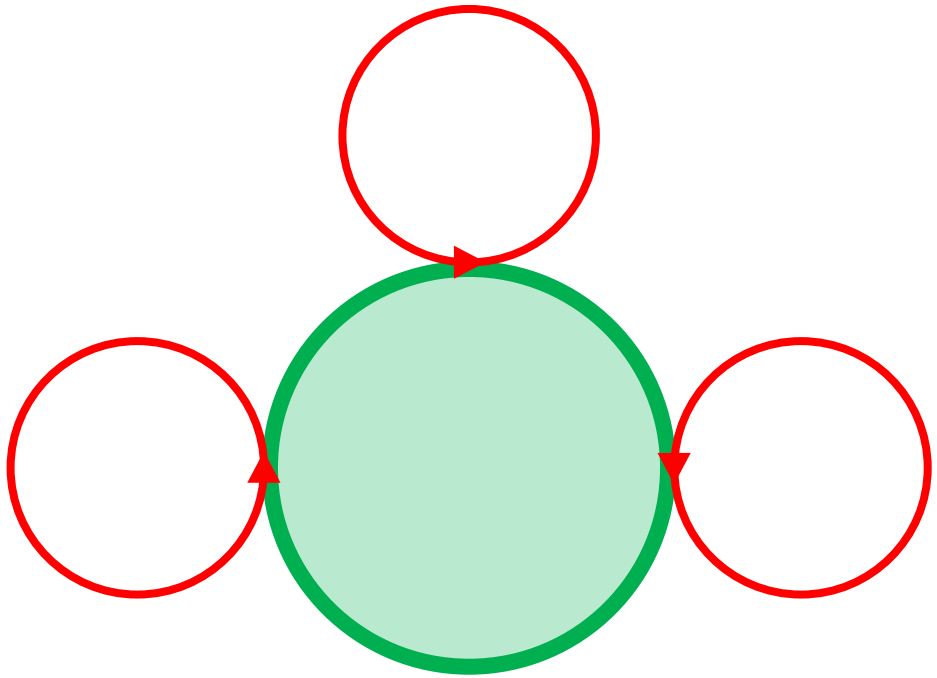
Single object is the `Bool` Type. There are two morphisms, `AND False` and `AND True`. When composed, you get:

A	B	A ∘ B
`AND True`	`AND True`	`AND True`
`AND True`	`AND False`	`AND False`
`AND False`	`AND True`	`AND False`
`AND False`	`AND False`	`AND False`



5. Represent addition modulo 3 as a monoid category.

+ 0 mod 3 (identity)



+ 2 mod 3

+ 1 mod 3

Bartosz Milewski

19.8K subscribers

SUBSCRIBE

HOME | VIDEOS | PLAYLISTS | COMMUNITY | CHANNELS | ABOUT

Uploads PLAY ALL

SORT BY

Category Theory III 7.2, Coends

4.1K views • 2 years ago

Category Theory III 7.1, Natural transformations as...

2.6K views • 2 years ago

Category Theory III 6.2, Ends

2.3K views • 2 years ago

Category Theory III 6.1, Profunctors

2.5K views • 2 years ago

Category Theory III 5.2, Lawvere Theories

2.3K views • 2 years ago

Category Theory III 5.1, Eilenberg Moore and Lawvere

2.5K views • 2 years ago

Category Theory III 4.2, Monad algebras part 3

1.7K views • 2 years ago

Category Theory III 4.1, Monad algebras part 2

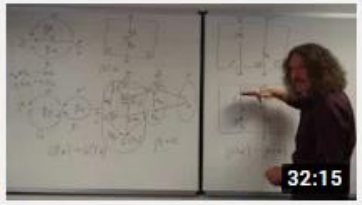
1.8K views • 2 years ago

Category Theory III 3.2, Monad Algebras

2.6K views • 2 years ago

Category Theory III 3.1, Adjunctions and monads

2.8K views • 2 years ago



Category Theory III 2.2, String Diagrams part 2

2.8K views • 2 years ago

Category Theory III 2.1: String Diagrams part 1

3.9K views • 2 years ago

Category Theory III 1.2: Overview part 2

2.8K views • 2 years ago

Category Theory III 1.1: Overview part 1

8.6K views • 2 years ago

Category Theory II 9.2: Lenses categorically

3.8K views • 3 years ago

Category Theory II 9.1: Lenses

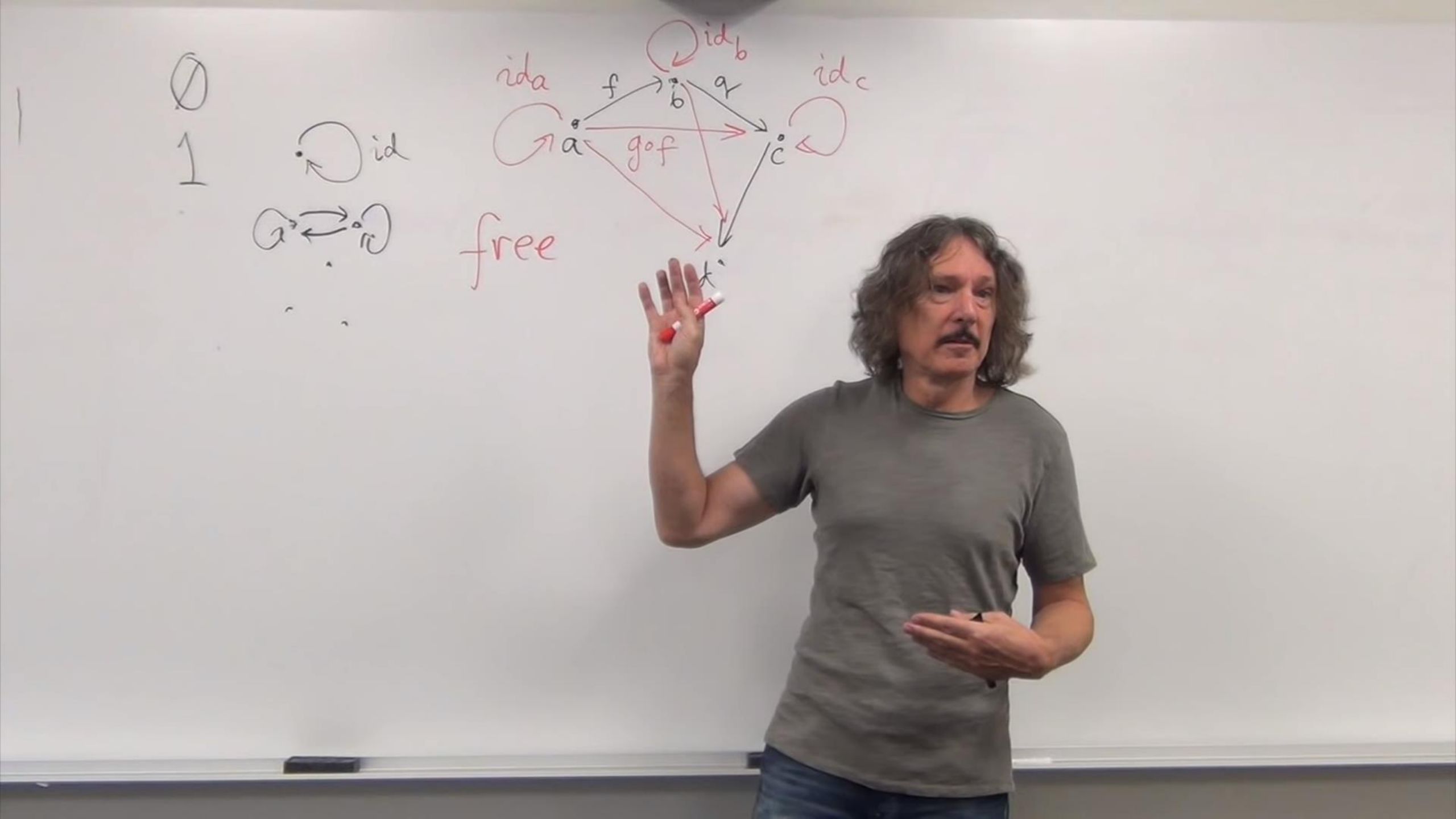
4.9K views • 3 years ago

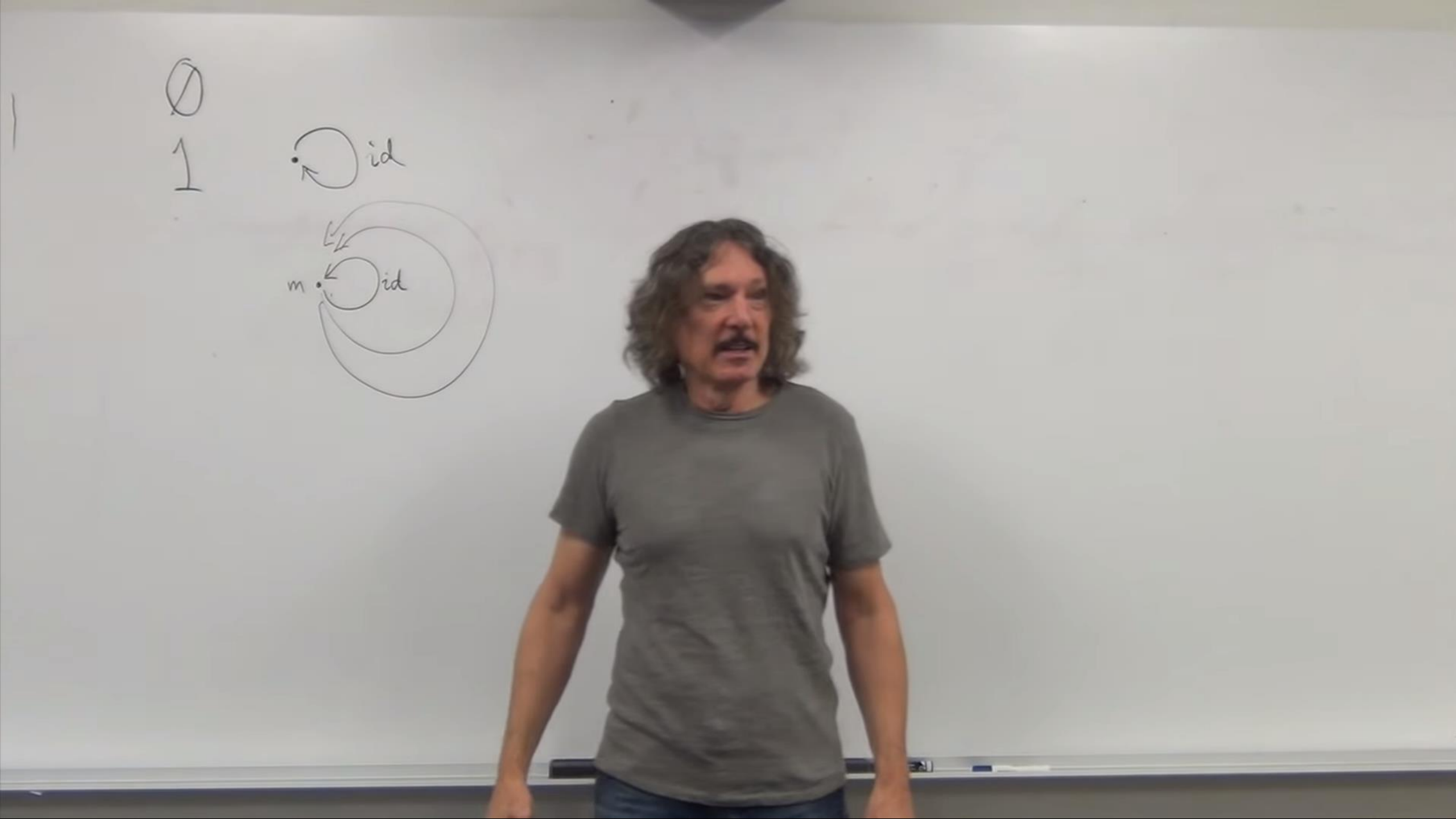
Category Theory II 8.2: Catamorphisms and...

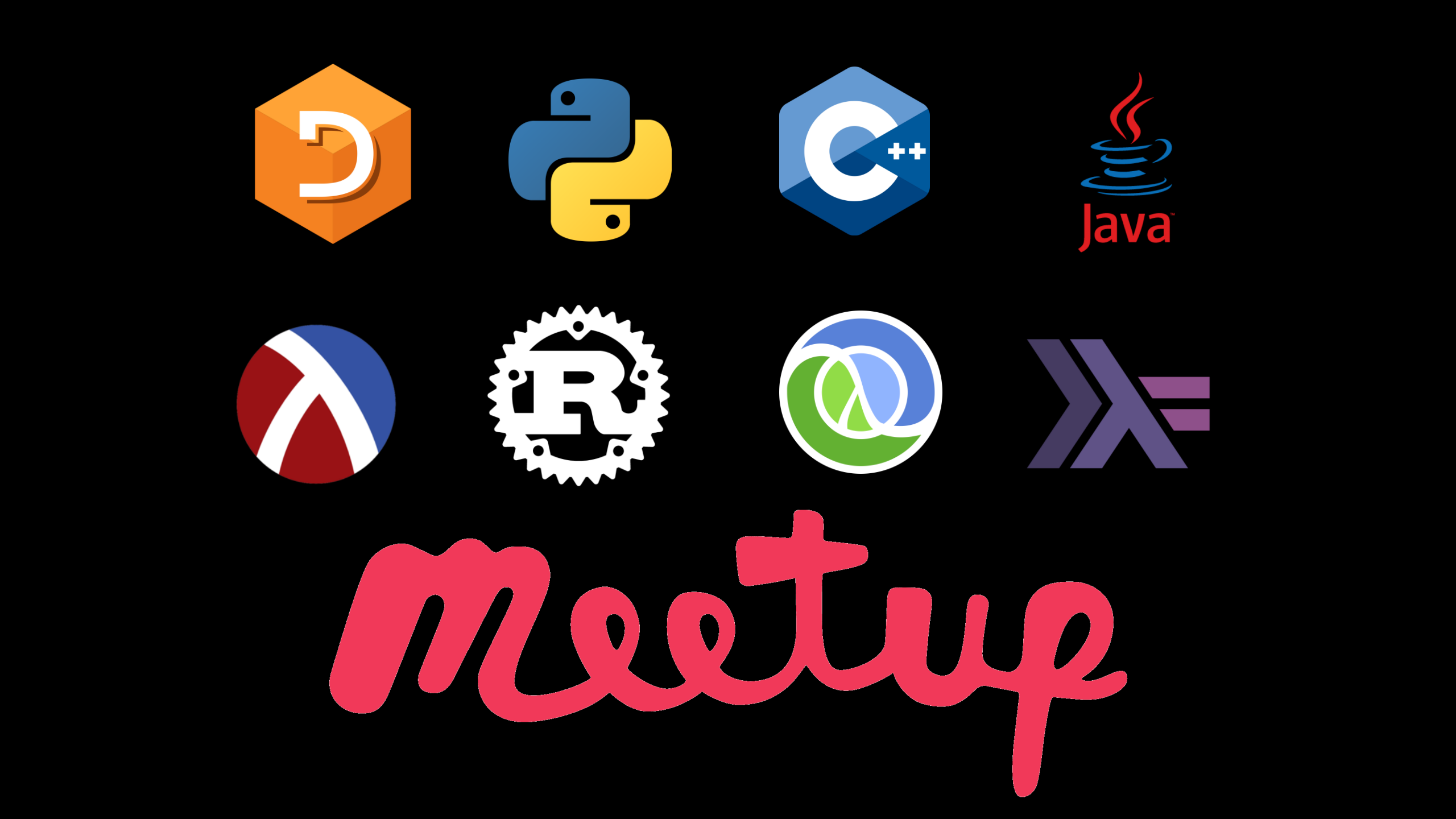
4.4K views • 3 years ago

Category Theory II 8.1: F-Algebras, Lambek's lemma

5.7K views • 3 years ago


∅
1
id
free
id_a
id_b
id_c
f
g
a
b
c
g∘f


∅
1
id
m
id


Java
meetup