Crash Course in Category Theory

Bartosz Milewski (@BartoszMilewski)
(Haskell to Scala translation: Máté Kovács)

Why Categories?

- Categorical semantics
- Composability
- Algebraic data types
- Function types and currying
- The Yoneda lemma

Paradox of Programming

- The paradox of programming: Machine/Human impedance mismatch
    - Local/Global perspective
    - Progress/Goal oriented
    - Detail/Idea
    - Vast memory/Limited memory
    - Pretty reliable/Error prone
    - Machine language/Mathematics
        - Is it easier to think like a machine than to do math?

```
  24011
    312
  -----
  48022
 24011
72033
-------
7491432
```

Semantics

- The meaning of a program
- Operational semantics: local, progress oriented
  - Execute program on an abstract machine in your brain
- Denotational semantics
  - Translate program to math
- *Math*: an ancient language developed for humans

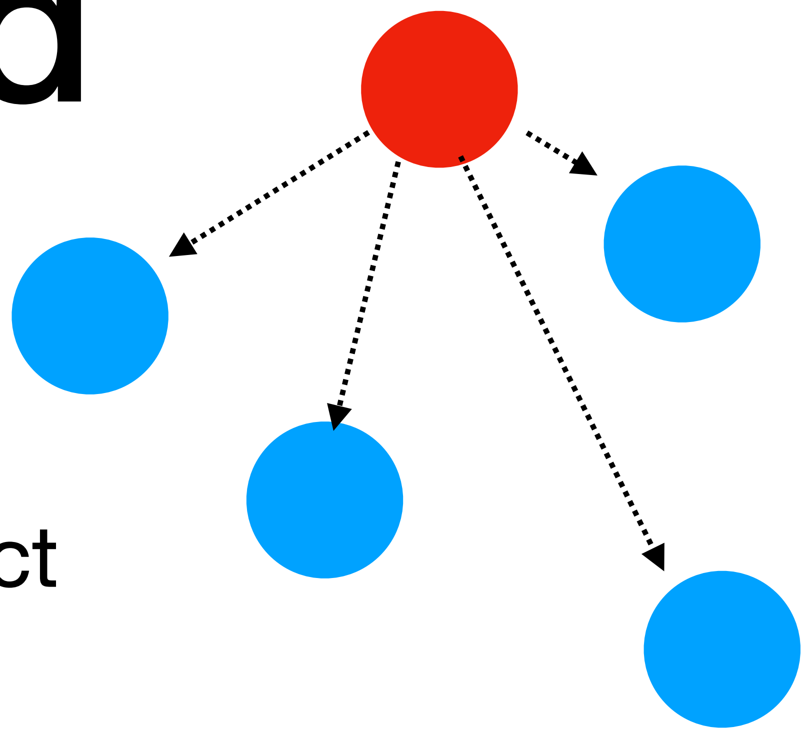
Functional Programming

- Types and functions
  - Types: sets of values
  - (Pure) Functions: mappings between sets
- Categorical view
  - Functions: arrows between objects
  - Types: objects whose properties are defined by arrows

Category

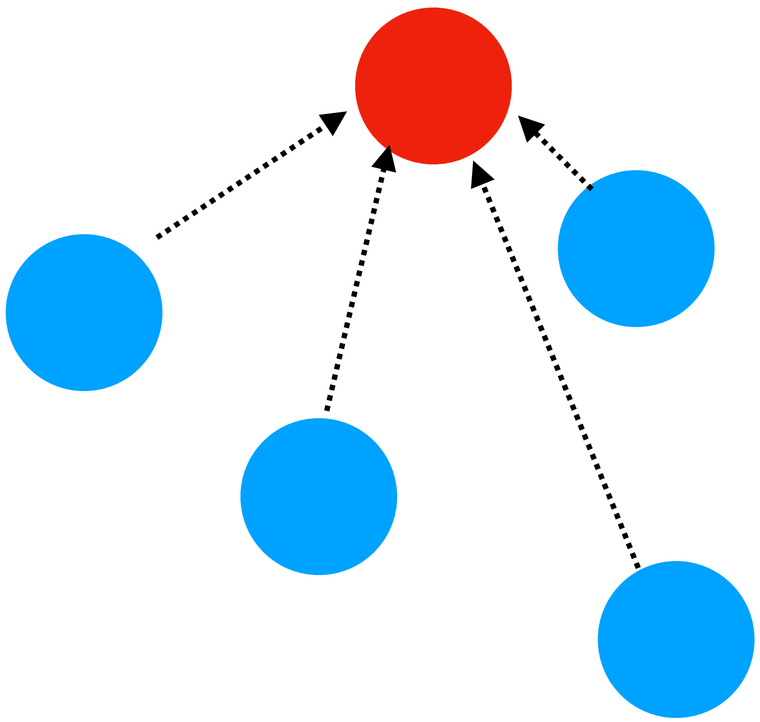
- Objects and arrows (types and functions)
  - composition of arrows (function composition)
  - associativity of composition
    `(f ∘ g) ∘ h = f ∘ (g ∘ h)`
  - identity arrow (identity function)
    `id ∘ f = f ∘ id = f`

Void



- An empty set
- Universal property: Initial object
  - A unique arrow to any object (including self)
- Introduction: No possibility of construction:
  `sealed trait Void`
- Elimination:
  `absurd[A]: Void => A`

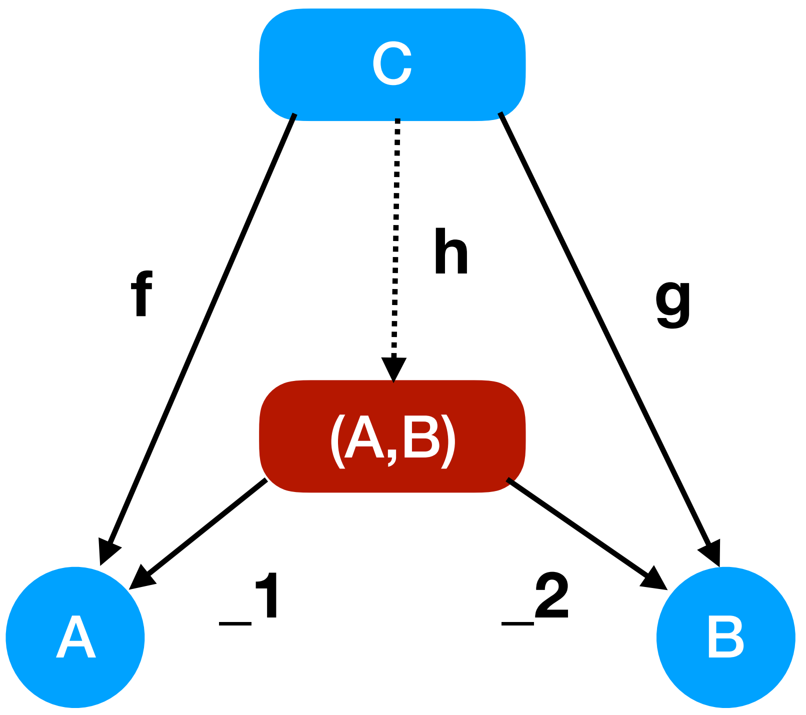
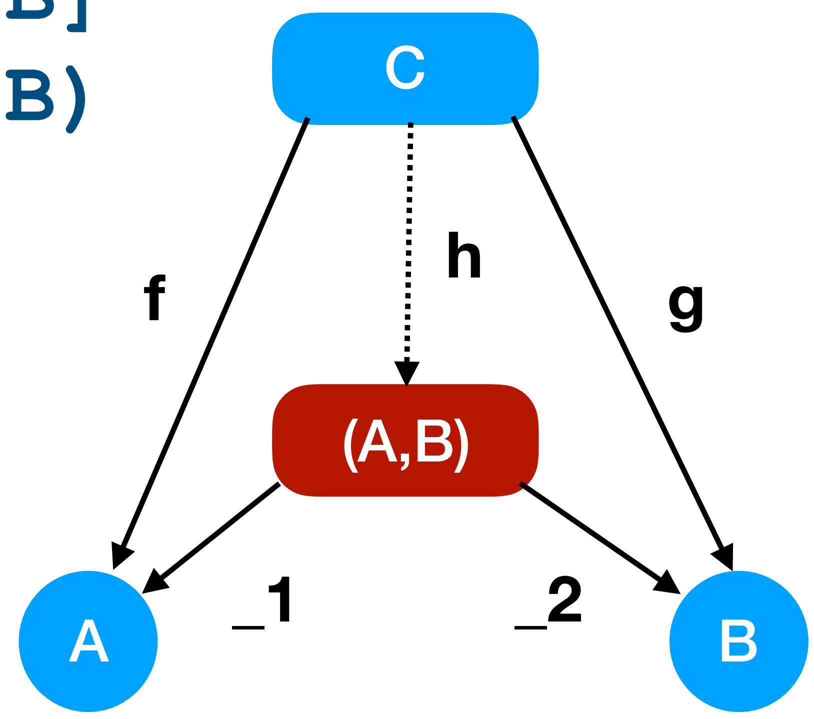
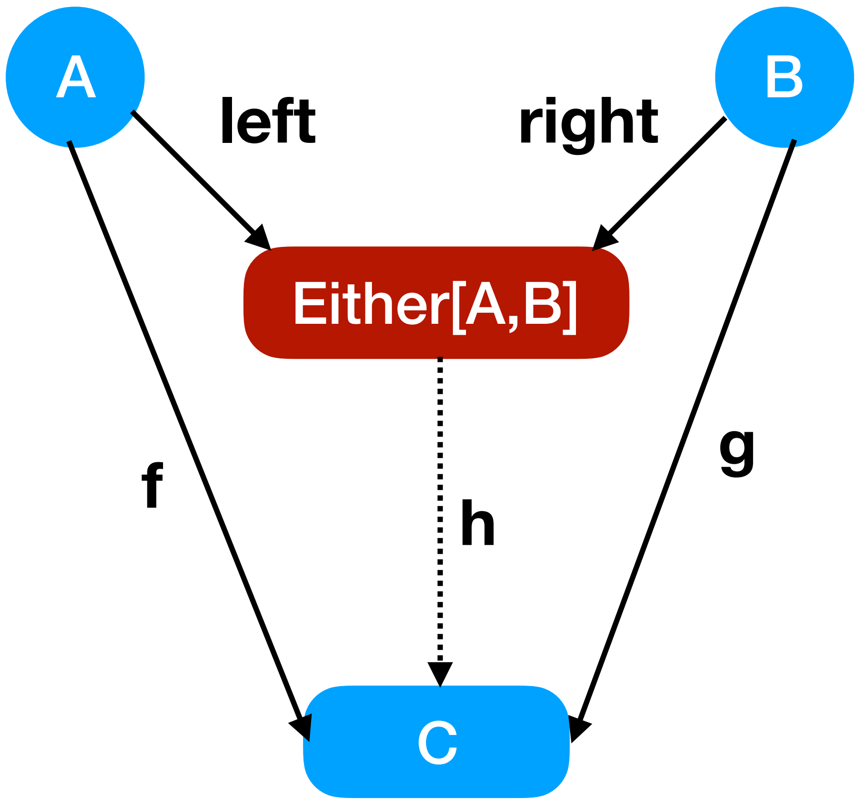
Unit



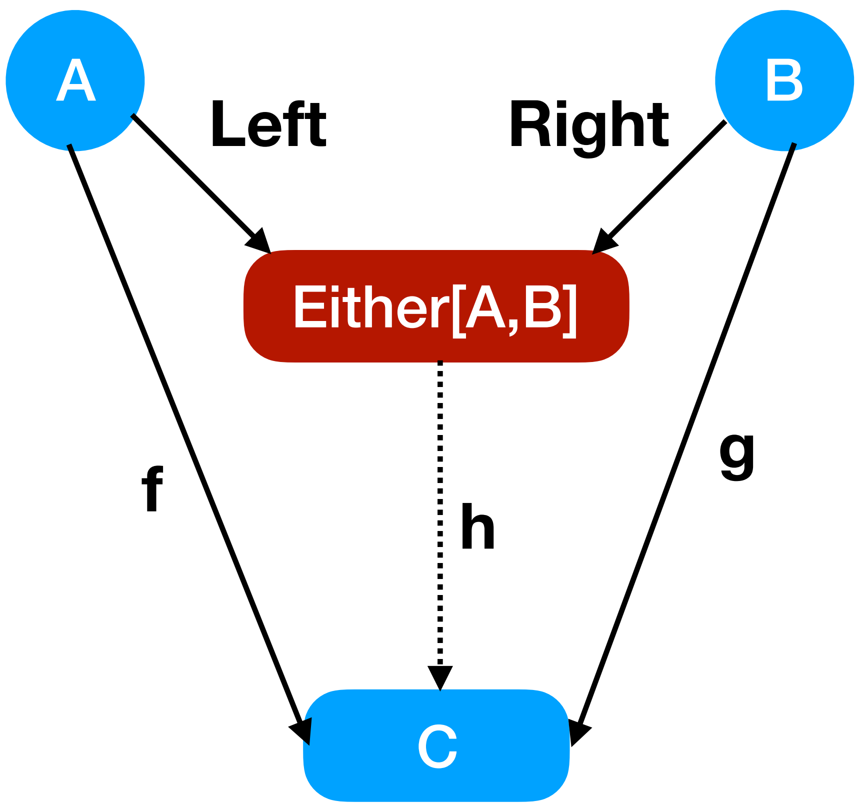
- Singleton set, `Unit` with element `()`
- Universal property: Terminal object
  - A unique arrow from any type
- Introduction:
  `unit[A]: A => Unit`
- Elimination:
  `Unit => A`

Product

- Set of pairs (cartesian product)
- Universal construction
- ```
  f: C => A
  g: C => B
  def h(c: C) = (f(c), g(c))
  ```


  ```

Product Types

- Introduction: `(A, B)` is `Tuple2[A, B]`
  `mkPair[A, B]: A => B => (A, B)`

- Elimination:
  `_1: (A, B) => A`
  `_2: (A, B) => B`

- Pairs, tuples, records



Sum (Coproduct)

- Disjoint union
- Universal construction
- ```
  f: A => C
  g: B => C
  def h(e: Either[A, B]): C = e
  match {
     case Left(a) => f a
     case Right(b) => g b
  }
  ```


  ```

Sum Types



- Introduction

```
Left.apply:  A => Either[A, B]
Right.apply: B => Either[A, B]
```

- Elimination

```
h e = case e of
        Left  a -> f a
        Right b -> g b
```

- Tagged unions

Monoidal Category

- Objects and arrows (types and functions)
- Product of objects (pair type)
  - Associative (up to isomorphism)
    `(A, (B, C)) ≅ ((A, B), C)`
  - Unit:
    `(Unit, A) ≅ A ≅ (A, Unit)`
- Coproduct (sum) of object
  - Associative
    `Either[A, Either[B, C]] ≅ Either[Either[A, B], C]`
  - Unit:
    `Either[Void, A] ≅ A ≅ Either[A, Void]`

Algebra of Types

```
type Bool = Either[Unit, Unit]

            sealed trait Bool
            case class False() extends Bool
            case class True() extends Bool

type Nat = Either[Unit, Either[Unit, Either[Unit, Either[Unit, ...]]]]

            sealed trait Nat
            case class One() extends Nat
            case class Two() extends Nat
            case class Three() extends Nat

type Option[A] = Either[Unit, A]
```

Polymorphic type

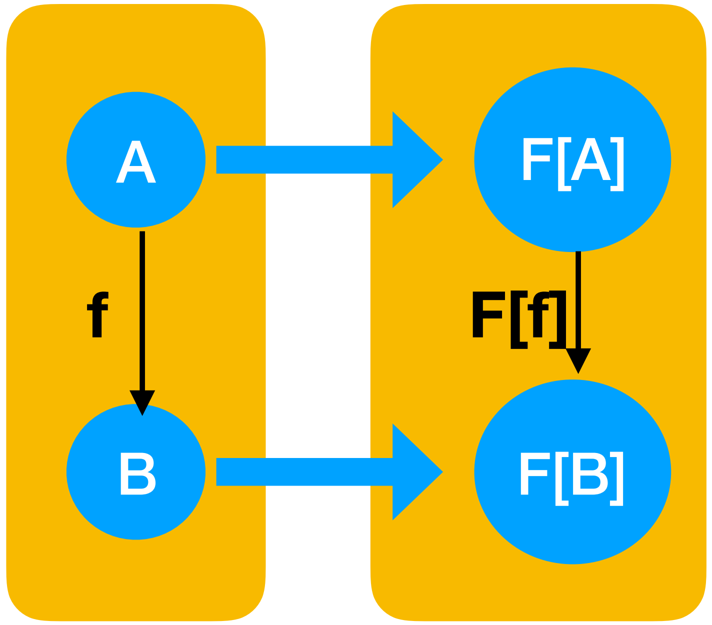
```
            sealed trait Option[A]
            case class None[A]() extends Option[A]
            case class Some[A](a: A) extends Option[A]

type List[A] = Either[Unit, (A, List[A])]
```

Recursive type

```
            sealed trait List[A]
            case class Nil[A]() extends List[A]
            case class Cons[A](head: A, tail: List[A]) extends List[A]
```

Functor

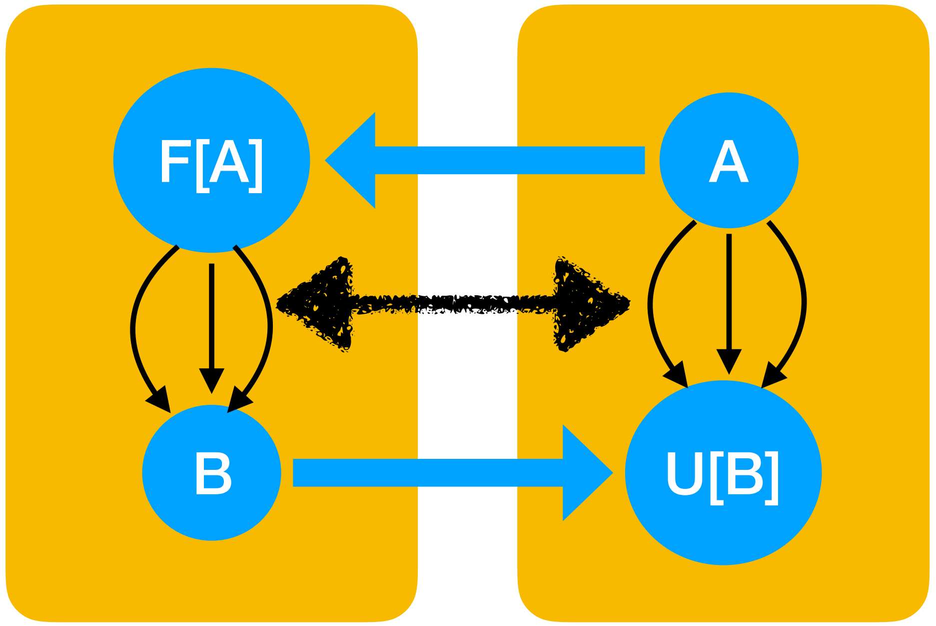
- Mapping between categories preserving structure
    - objects to objects
      (types to types, parametrized types)
    - arrows to arrows
      (functions to functions, `fmap`)




```
trait Functor[F[_]] {
  def fmap[A, B](f: A => B)(v: F[A]): F[B]
}
```

```
implicit object ListFunctor extends Functor[List] {
  def fmap[A, B](f: A => B)(v: List[A]): List[B] = v match {
    case Cons(head, tail) => Cons(f(head), fmap(f)(tail))
    case Nil() => Nil()
  }
}
```

Adjunction



`F[A] => B ≅ A => U[B]`

`Fc A = (A, C)`

`Gc B = C => B`

`(A, C) => B ≅ A => (C => B)`

Exponential: $A^B \cong$ B=>A

Function Types

- A set of functions from `C` to `B`, `C=>B`

- Universal construction

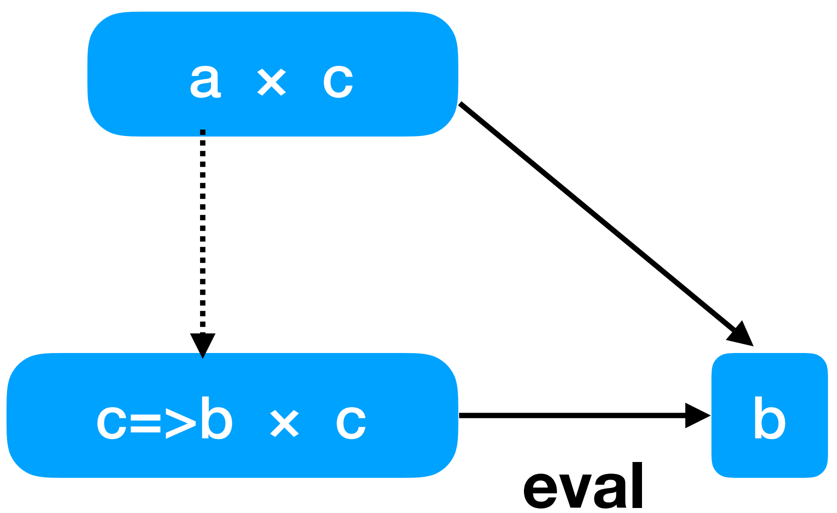
$$
\begin{tikzcd}
A \times C \arrow[d, dotted, "A \Rightarrow (C \Rightarrow B)"'] \arrow[dr, "(A,\ C) \Rightarrow B"] & \\
C \Rightarrow B \times C \arrow[r] & B
\end{tikzcd}
$$

`(A, C) => B ≅ A => (C => B)`

Function Types

- Introduction
  `x : a`
  `e(x) : b`
  `\x. e(x) : a=>b`

- Elimination
  `eval : (c => b, c) => b`

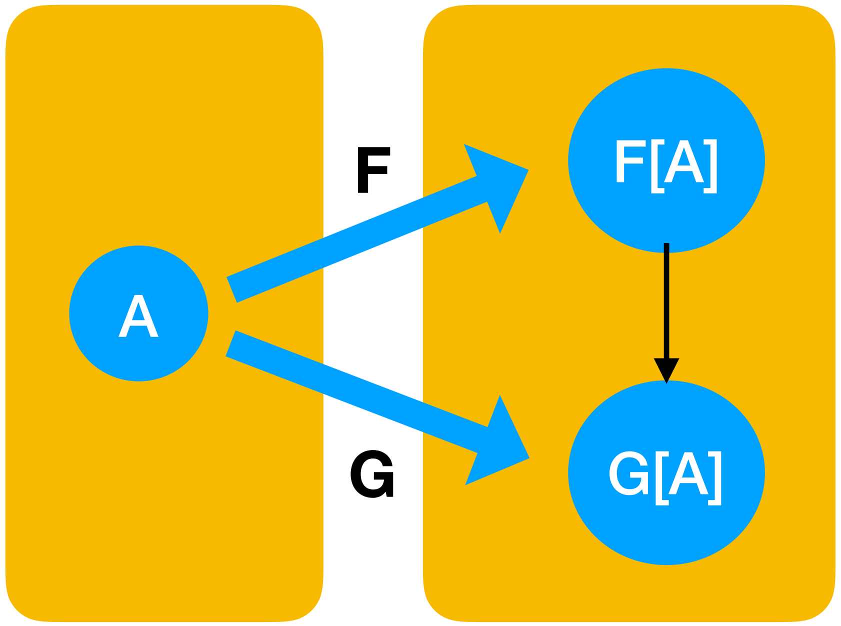


`(a, c) => b ≅ a => (c => b)`

Natural Transformations

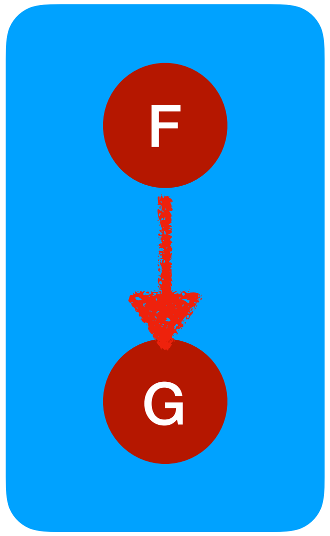
- Two functors F and G
- For every object **`A`**, an arrow
  `F[A] => G[A]`
- Polymorphic function

```
forall A. F[A]=>G[A]
```




```
def safeHead[A](l: List[A]) = l match {
  case Cons(head, _) => Some(head)
  case Nil() => None()
}
```

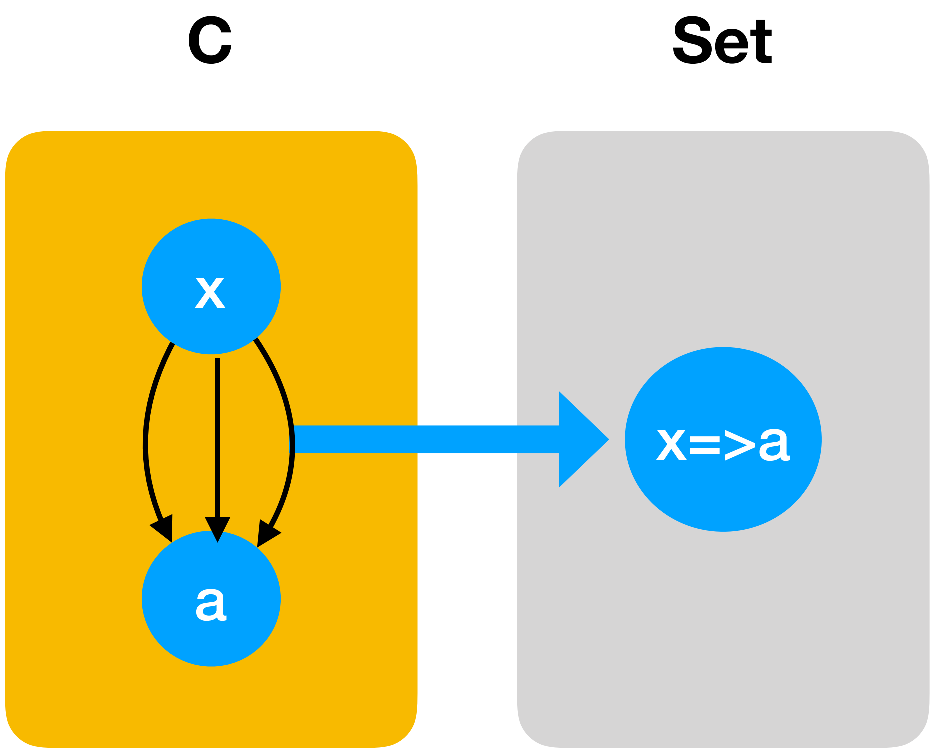
Functor Category



- Pick two categories C and D
- Functors from C to D form category [C, D]
    - objects: functors
    - arrows: natural transformations
- Endofunctors (from C back to C) form a category [C, C]

Object in Context

- Totality of arrows to an object
  - Arrows from `x` to `a` form a set `x=>a`
  - Varying `x` gives us the *totality* of arrows impinging on (fixed) `a`
  - Mapping from `x` to the set `x=>a` is a (contravariant) functor from C to Set



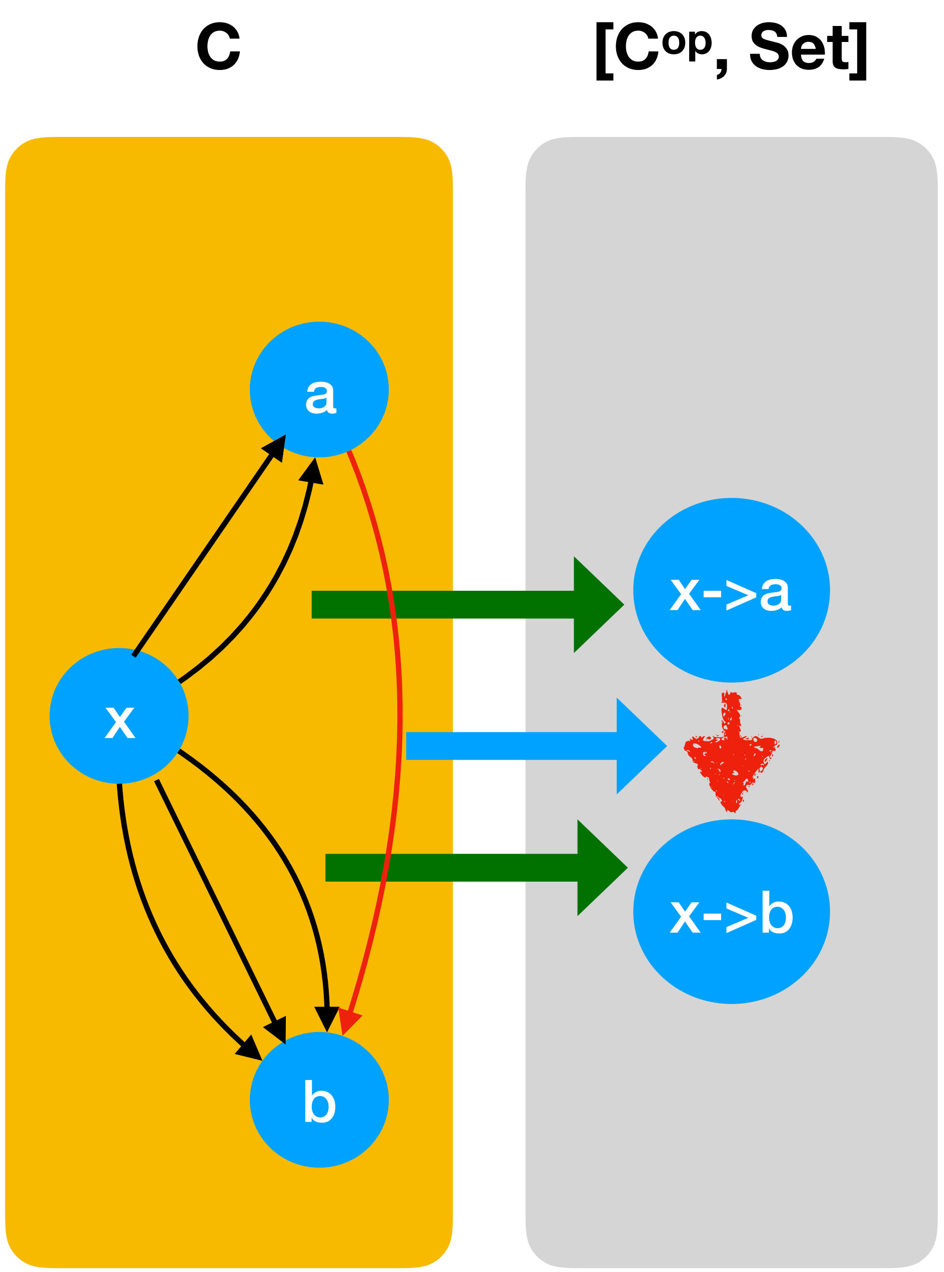
Yoneda Embedding

- Mapping from `x` to the set `x->a` is a (contravariant) functor from C to Set
- Mapping from `a` to this functor *embeds* `a` in functor category
- It's a *fully faithful* embedding: it takes all arrows with it

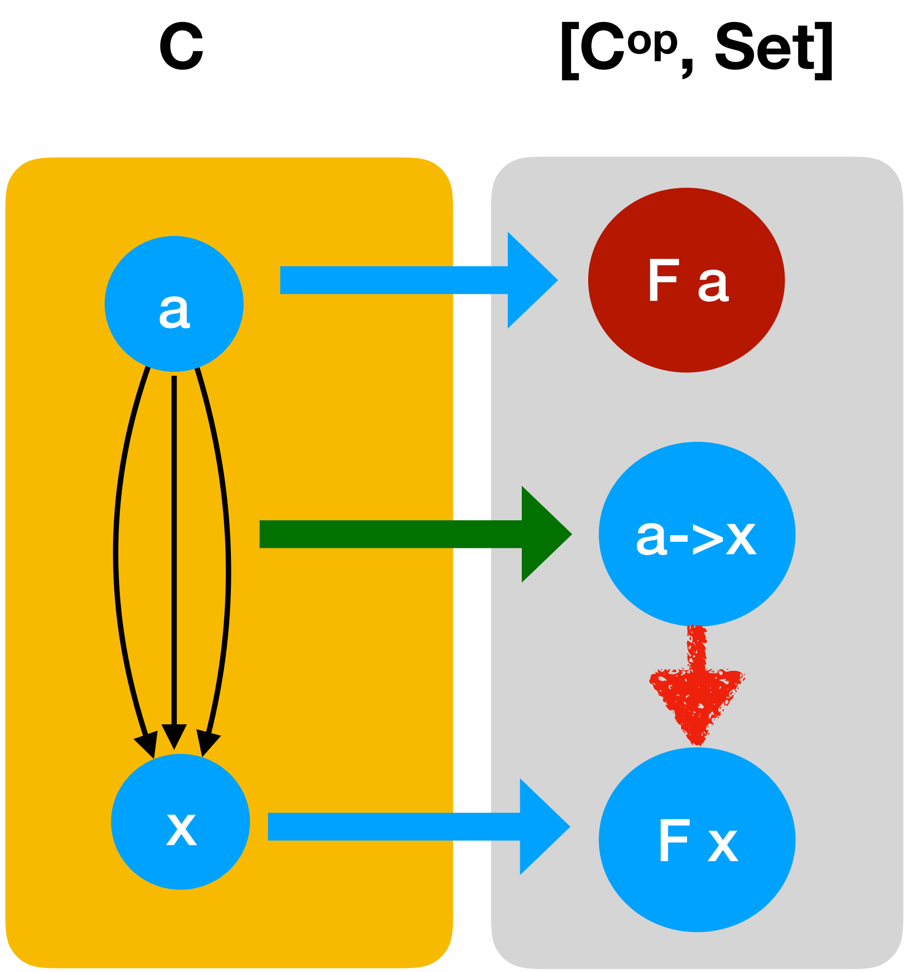
`a->b` ≅ *arrow*

`forall x. (x->a)->(x->b)`

natural transformation



Yoneda Lemma



- Totality of morphisms emanating from `a`
- Any functor `F` from C to Set
- Set of natural transformations between the two
- Isomorphic to the set `F a`
- Representing a parametric data structure `F a` as a polymorphic function

```
forall x . (a -> x) -> F x ≅ F a
```

Continuation Passing

```
forall x . (a -> x) -> F x ≅ F a
```

- Identity functor

```
forall x . (a -> x) -> x ≅ a
```

- Instead of providing `a`, give me a handler for `a`

Lens Library

- Operations on persistent algebraic data structures
- Killer app for Yoneda lemma