



February 2020
BOB Conference, Berlin

The Essence of Programming

INOQ



Ludvig Sundström

Consultant
INNOQ Deutschland GmbH

ludvig.sundstroem@innoq.com

My Agenda

- Structure in problem solving

My Agenda

- Structure in problem solving
- Using function composition

My Agenda

- Structure in problem solving
- Using function composition
- Learning about categories

My Agenda

- Structure in problem solving
- Using function composition
- Learning about categories



Stockholm, Sweden



- Haskell

- Haskell
- Clojure

- Haskell
- Clojure
- Erlang

- Haskell
- Clojure
- Erlang
- Scala

- Haskell
- Clojure
- Erlang
- Scala
- etc.

λ



Uppsala, Schweden









[O]



Avoid!

- Quick fixes

Avoid!

- Quick fixes
- Unnecessary layers of technology

Avoid!

- Quick fixes
- Unnecessary layers of technology
- Misunderstanding the problem itself

The Problem of Solving the Problem

"How to change and modify parts of a system without making the system as a whole more complicated"

Today

I'll tackle the problem of the problem by explaining why ...

Why Functional Programming Matters

John Hughes
The University, Glasgow

FAQ

- If functional programming matters ...

FAQ

- If functional programming matters ...
- ...Why is the functional fan-club so small? [2]

FAQ: Answers

- We are technical people

FAQ: Answers

- We are technical people
- We have technical discussions

FAQ: Answers

- We are technical people
- We have technical discussions
- We learn to say "it depends"

FAQ: Answers

- We are technical people
- We have technical discussions
- We learn to say "it depends"
- And that software engineering is about trade-offs

FAQ: Answers

- We are technical people
- We have technical discussions
- We learn to say "it depends"
- And that software engineering is about trade-offs
- But most of all to have faith in what is already working

"Faith triumphs over science [in programming]"
- Philip Waldler

Instead of ...

- Having tunnel vision

Instead of ...

- Having tunnel vision
- Justifying what we already know

Instead of ...

- Having tunnel vision
- Justifying what we already know
- Getting lost in technical discussions

We should ...

- Have an open mind

We should ...

- Have an open mind
- Justify science, logic

We should ...

- Have an open mind
- Justify science, logic
- Start with the fundamentals of problem solving

Only then can we lift ourselves over the everyday programming grind!

Why FP?

Why FP?

- Functional programming does not only matter

Why FP?

- Functional programming does not only matter
- It is universal! (direct correspondence with logic) [2]

Why FP?

- Functional programming does not only matter
- It is universal! (direct correspondence with logic) [2]
- \implies Lets us talk about the structure of problem solving

Why FP?

- Functional programming does not only matter
- It is universal! (direct correspondence with logic) [2]
- \implies Lets us talk about the structure of problem solving
- \implies Lets us focus on the essence of programming

Why FP?

- Functional programming does not only matter
- It is universal! (direct correspondence with logic) [2]
- \implies Lets us talk about the structure of problem solving
- \implies Lets us focus on the essence of programming

The Fundamentals of Problem Solving



Nick

@Zorchenhimer



Found this in production today. I need a drink.

```
public static bool CompareBooleans(bool orig, bool val)
{
    return AreBooleansEqual(orig, val);
}

internal static bool AreBooleansEqual(bool orig, bool val)
{
    if(orig == val)
        return false;
    return true;
}
```

12:54 AM · 31 May 19 · [Twitter Web Client](#)

2,519 Retweets **7,054** Likes



We do our best to create modular, loosely coupled, composable abstractions



- We program in order to solve problems (Only? 🤖)

- We program in order to solve problems (Only? 🤖)
- Then how do we solve problems?

CS: 101

CS: 101

Divide and Conquer

- Elegant code = Code that is easy to understand

- Elegant code = Code that is easy to understand
- Elegant code = Code broken up into just big enough pieces

- Elegant code = Code that is easy to understand
- Elegant code = Code broken up into just big enough pieces (by divide and conquer)

CS: 101

Divide and Conquer

A computer program is ...

- A solution to a problem

A computer program is ...

- A solution to a problem
- A solution to many smaller problems

A computer program is ...

- A solution to a problem
- A solution to many smaller problems
- Complexity, split up into pieces

A computer program is ...

- A solution to a problem
- A solution to many smaller problems
- Complexity, split up into pieces
- Information flowing in a structure

A computer program is ...

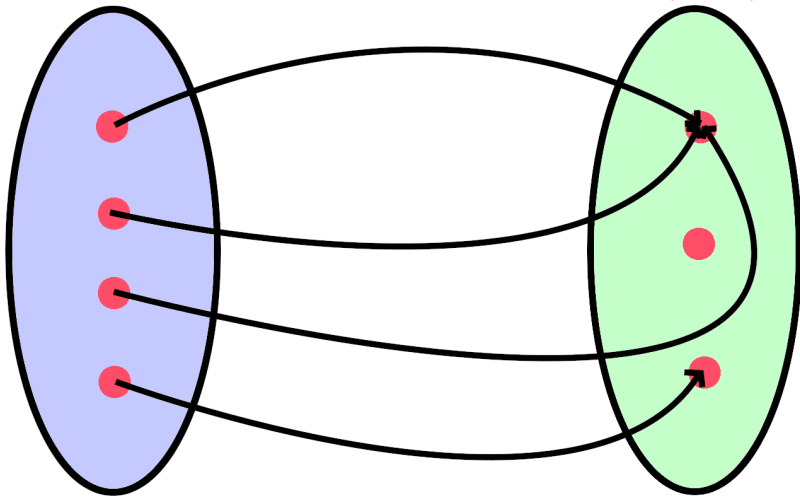
- A solution to a problem
- A solution to many smaller problems
- Complexity, split up into pieces
- Information flowing in a structure (by divide and conquer)

How do we build information flow?

Enter the function

X

$f(X)$



- We'd like to use the mathematical model of functions

- We'd like to use the mathematical model of functions
- But in programming, we cannot have mathematical functions

- We'd like to use the mathematical model of functions
- But in programming, we cannot have mathematical functions
- However, we can get close enough 😊

- We'd like to use the mathematical model of functions
- But in programming, we cannot have mathematical functions
- However, we can get close enough 😊
- So let's think about our programs as a collection of pure functions ...

- We'd like to use the mathematical model of functions
- But in programming, we cannot have mathematical functions
- However, we can get close enough 😊
- So let's think about our programs as a collection of pure functions ...
- ...composed together in a certain structure

- We'd like to use the mathematical model of functions
- But in programming, we cannot have mathematical functions
- However, we can get close enough 😊
- So let's think about our programs as a collection of pure functions ...
- ...composed together in a certain structure (by divide and conquer)

Divide and Conquer \implies Structure

Divide and Conquer \implies Structure = Function Composition

Divide and Conquer \implies Structure =
Function Composition = The Essence of Programming!

Now show me how to study the essence of programming!

Category Theory **(without most of the theory)**



[3]

Category Theory

- Is the science of patterns

Category Theory

- Is the science of patterns
- Is the study of composition

Category Theory

- Is the science of patterns
- Is the study of composition
- Is a language that abstracts structure across different fields

Category Theory

- Is the science of patterns
- Is the study of composition
- Is a language that abstracts structure across different fields
- Applies well to programming ...


```

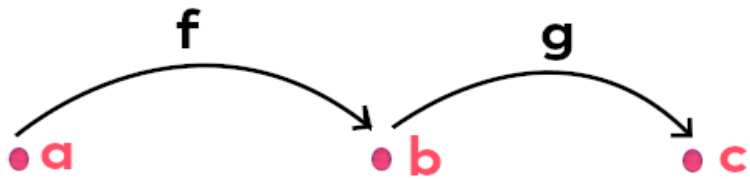
    }
    public static int isEven(int a) {
        if (a == 0) return 1;
        if (a == 2) return 1;
        if (a == 4) return 1;
        if (a == 6) return 1;
        if (a == 8) return 1;
        if (a == 10) return 1;
        if (a == 12) return 1;
        if (a == 14) return 1;
        // TODO: Add more checks.
        return 0;
    }
}

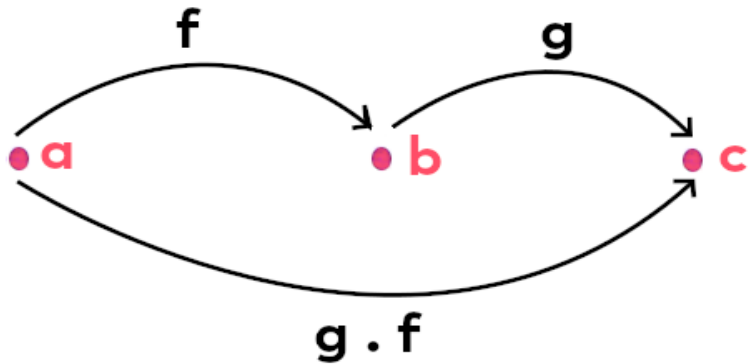
```

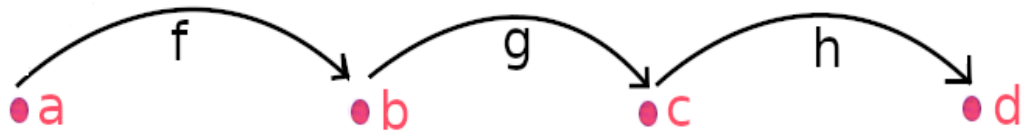
Category Theory

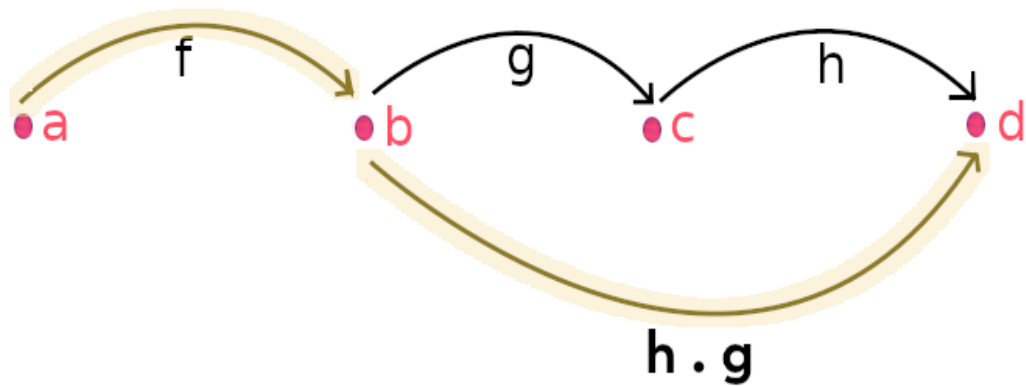
- Is the study of composition
- Is the science of patterns
- Is a language that abstracts structure across different fields
- Applies well to programming ...
- ...because programming is all about structure

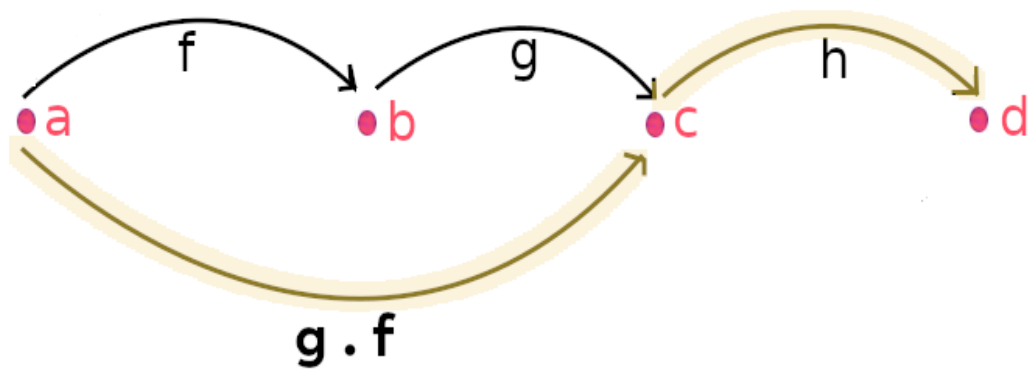
Ok, show me what a category is.



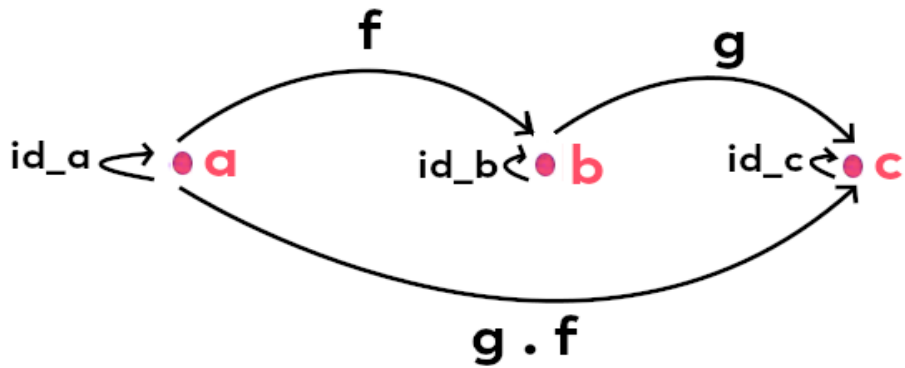


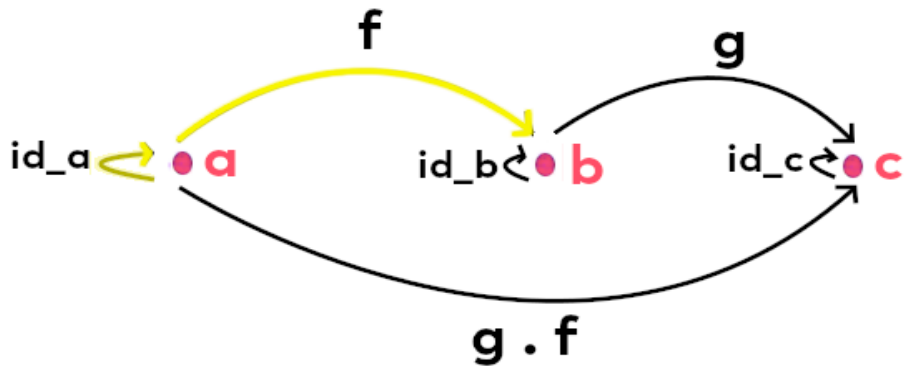


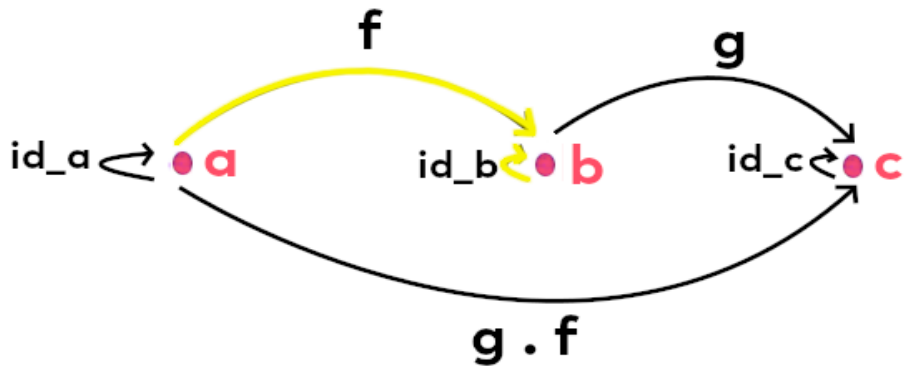












- That's it.

- That's it.
- CT leaves it to us to discover the meaning behind this simple structure

Then show me how to define a category with some meaning!

How to define a category

1. Say what the objects are

How to define a category

1. Say what the objects are
2. Say what the arrows are

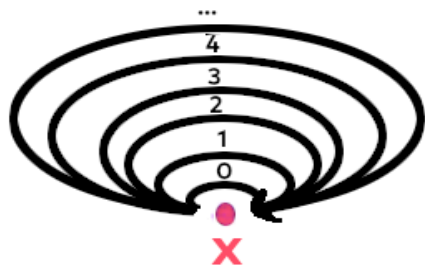
How to define a category

1. Say what the objects are
2. Say what the arrows are
3. Say what the identities are

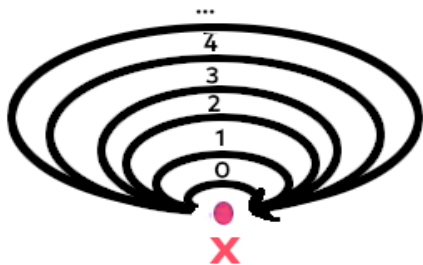
How to define a category

1. Say what the objects are
2. Say what the arrows are
3. Say what the identities are
4. Say how the arrows compose

Category M

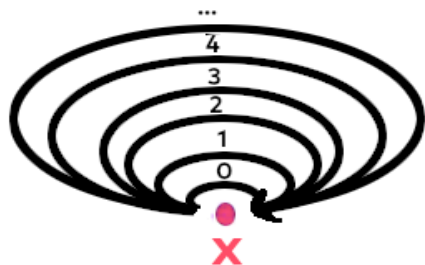


Category M



$$\text{Obj}(M) = \{x\}$$

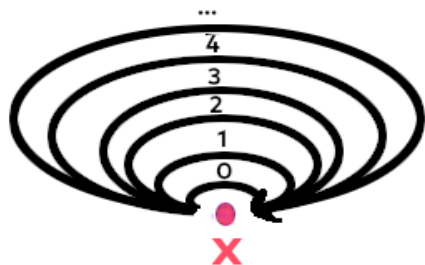
Category M



$$\text{Obj}(\mathbf{M}) = \{x\}$$

$$\text{Hom}(\mathbf{M}) = \mathbb{N}$$

Category M

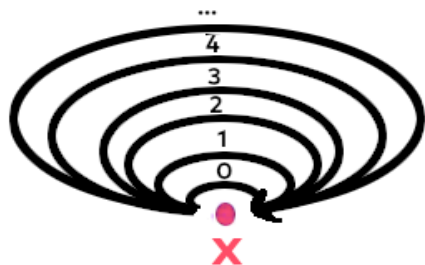


$$\text{Obj}(M) = \{x\}$$

$$\text{id}_x = 0$$

$$\text{Hom}(M) = \mathbb{N}$$

Category M

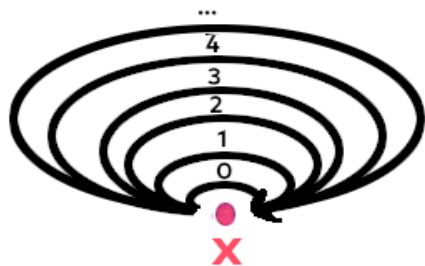


$$\text{Obj}(M) = \{x\}$$

$$\text{id}_x = 0$$

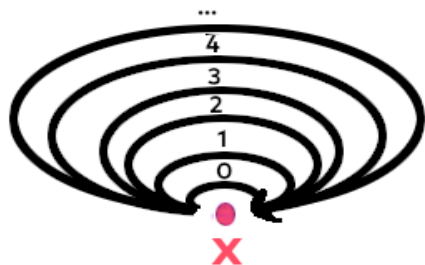
$$\text{Hom}(M) = \mathbb{N} \quad \text{composition} = (+)$$

Category M



Composition: For any two arrows n and m , there exists a composite arrow $(n + m)$

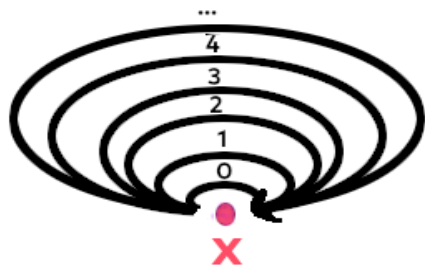
Category M



Composition: For any two arrows n and m , there exists a composite arrow $(n + m)$

Identity: Any arrow can be composed with identity $(n + 0)$

Category M

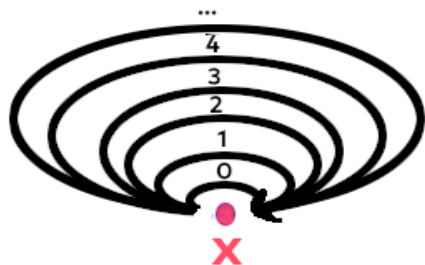


Composition: For any two arrows n and m , there exists a composite arrow $(n + m)$

Identity: Any arrow can be composed with identity $(n + 0)$

Associativity: Composing arrows $(i + j) + k$ is the same as composing $i + (j + k)$

Category M



Composition: For any two arrows n and m , there exists a composite arrow $(n + m)$

Identity: Any arrow can be composed with identity $(n + 0)$

Associativity: Composing arrows $(i + j) + k$ is the same as composing $i + (j + k)$

All logic is encoded in the composition

Programmers Category

- Programmers talk in data ...

- Programmers talk in data ...
- ...and give the data **types**

- Programmers talk in data ...
- ...and give the data **types**
- They spend their days transforming it with **functions** ...

- Programmers talk in data ...
- ...and give the data **types**
- They spend their days transforming it with **functions** ...
- ...and **compose** those functions in order to D.R.Y

The Category of Types and Functions

The Category of Types and Functions

1. Objects \rightarrow Types

The Category of Types and Functions

1. Objects \rightarrow Types
2. Arrows \rightarrow Functions

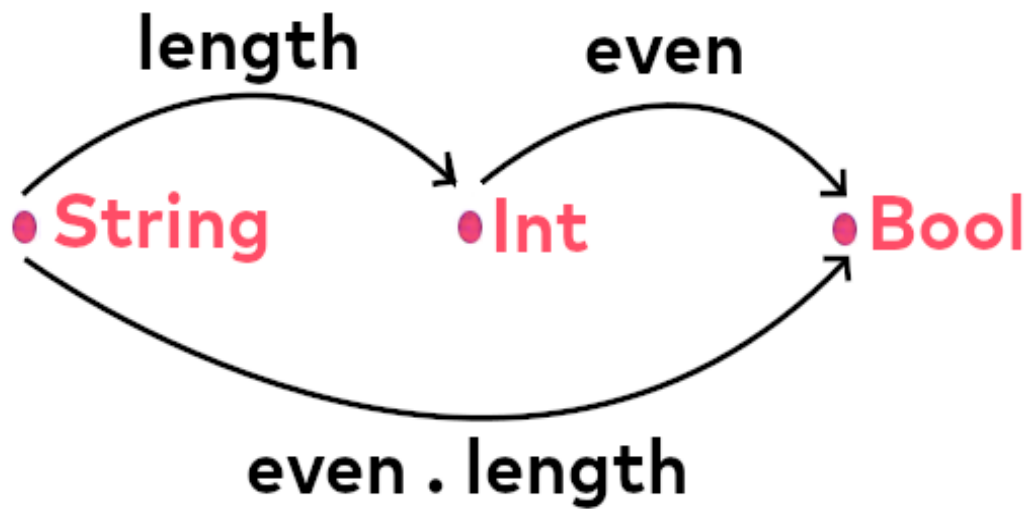
The Category of Types and Functions

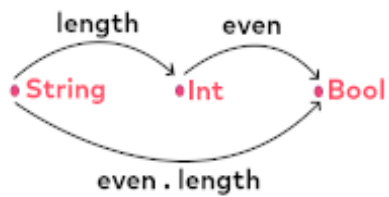
1. Objects \rightarrow Types
2. Arrows \rightarrow Functions
3. Composition \rightarrow Function composition

The Category of Types and Functions

1. Objects \rightarrow Types
2. Arrows \rightarrow Functions
3. Composition \rightarrow Function composition

A tool to study **essence of programming!**



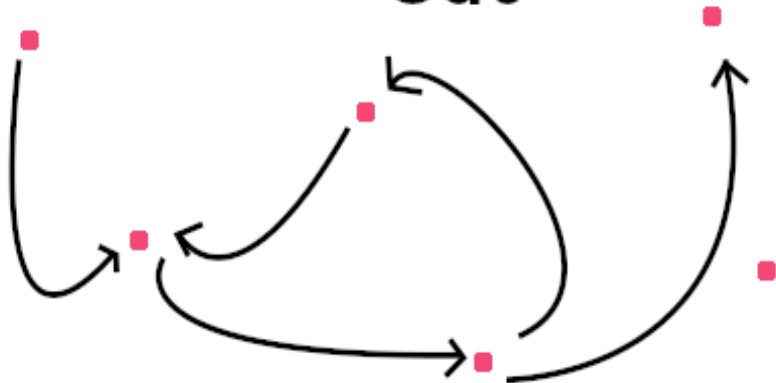








Cat



$\text{Ob}(\mathbf{Cat}) = \text{categories}$

$\text{Hom}(\mathbf{Cat}) = \text{functors}$



The Functor

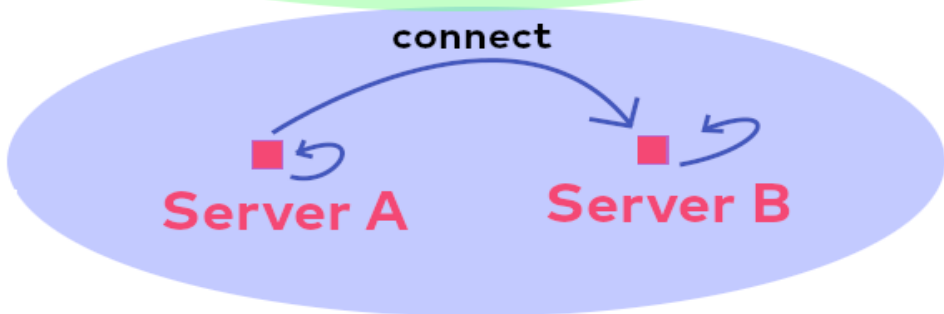
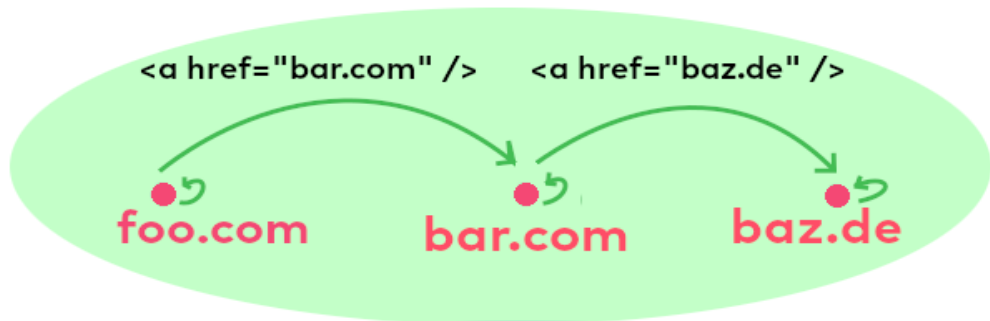
- Is a mapping between categories

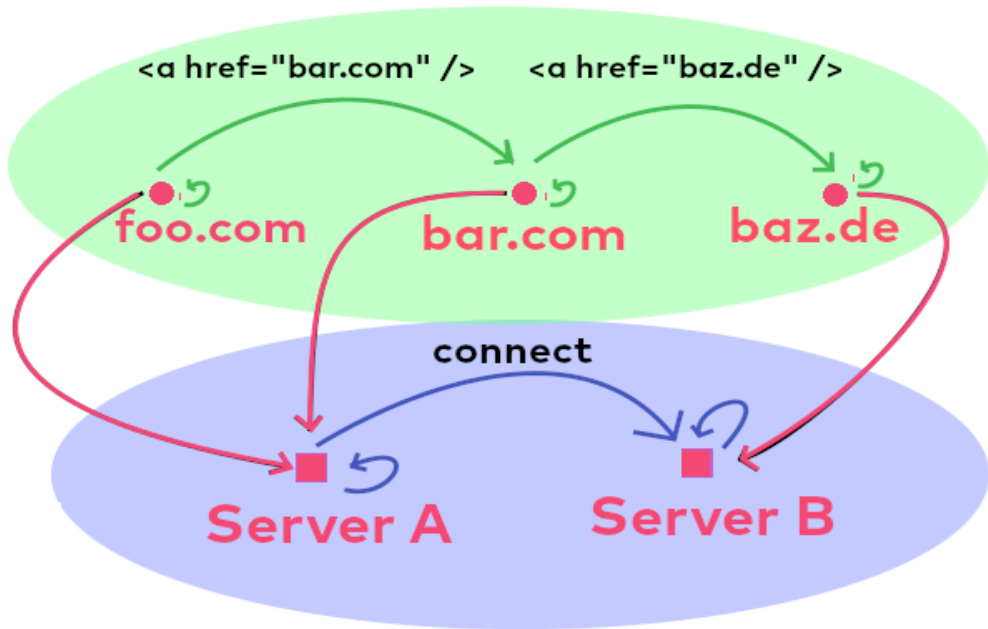
The Functor

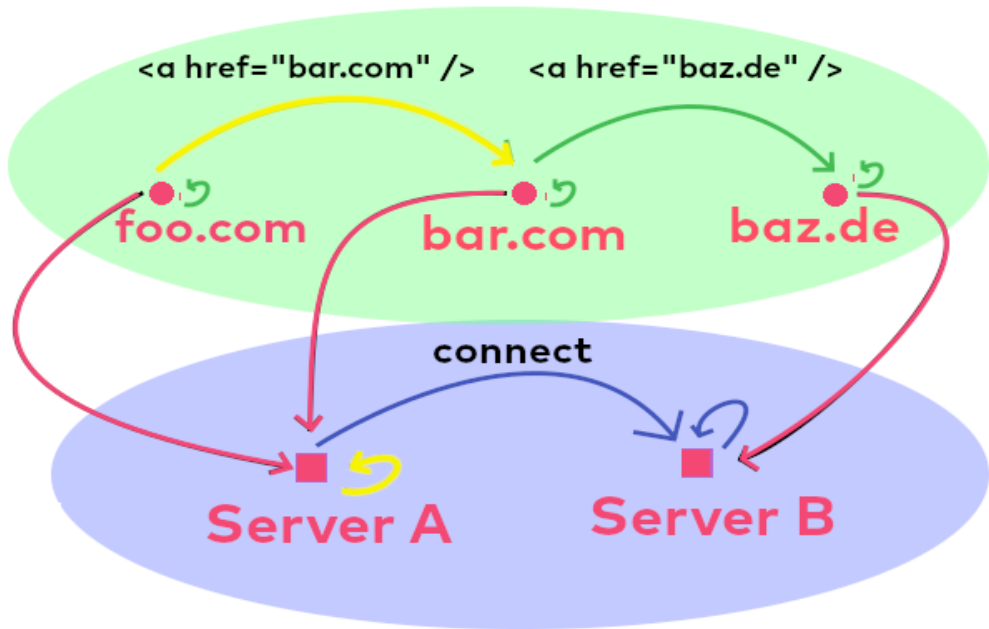
- Is a mapping between categories
- Maps objects into objects and arrows into arrows ...

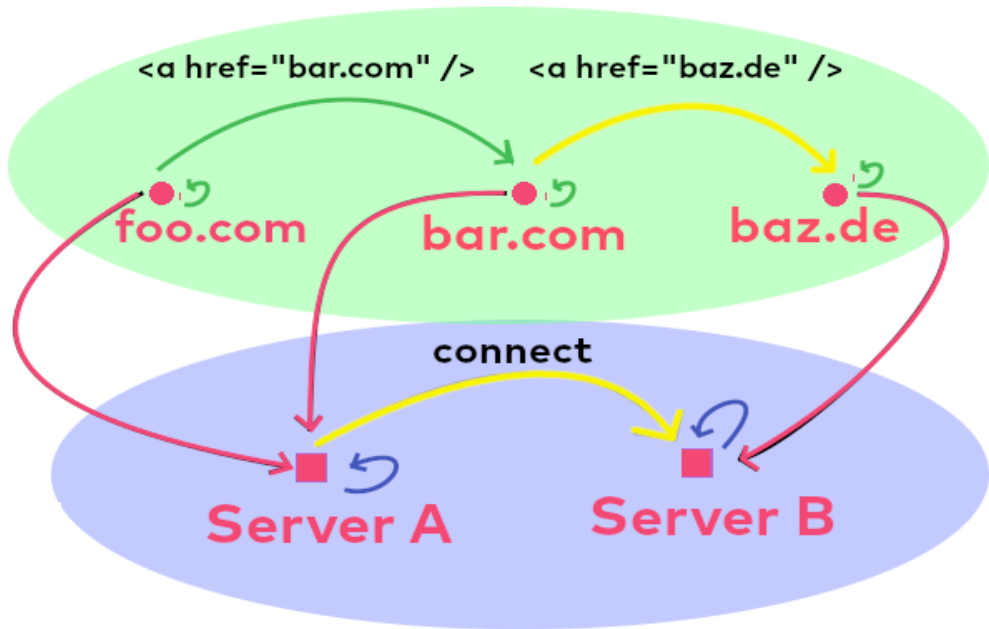
The Functor

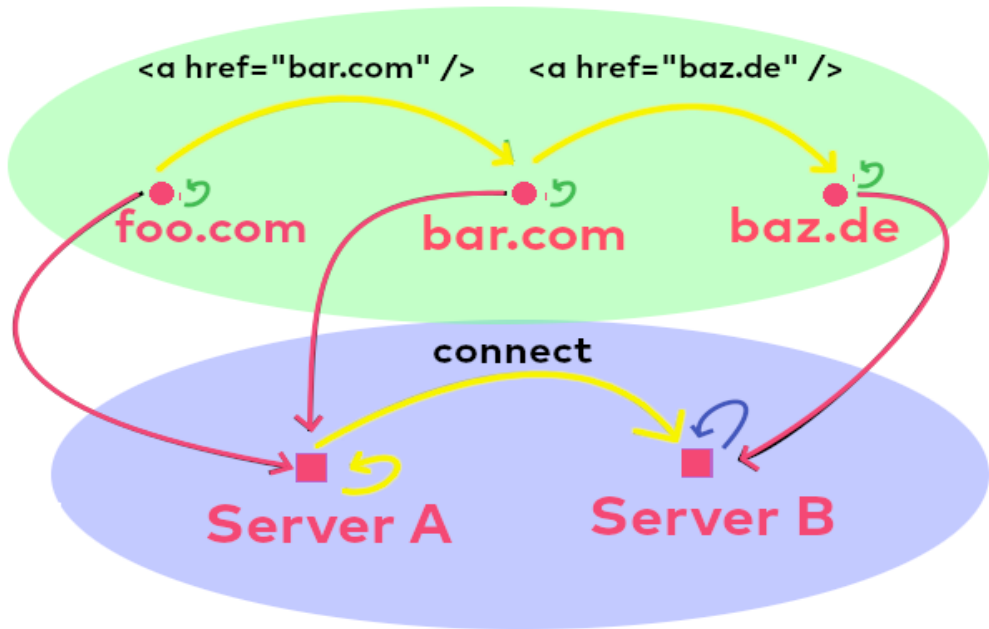
- Is a mapping between categories
- Maps objects into objects and arrows into arrows ...
- ...Preserving structure! (or meaning)

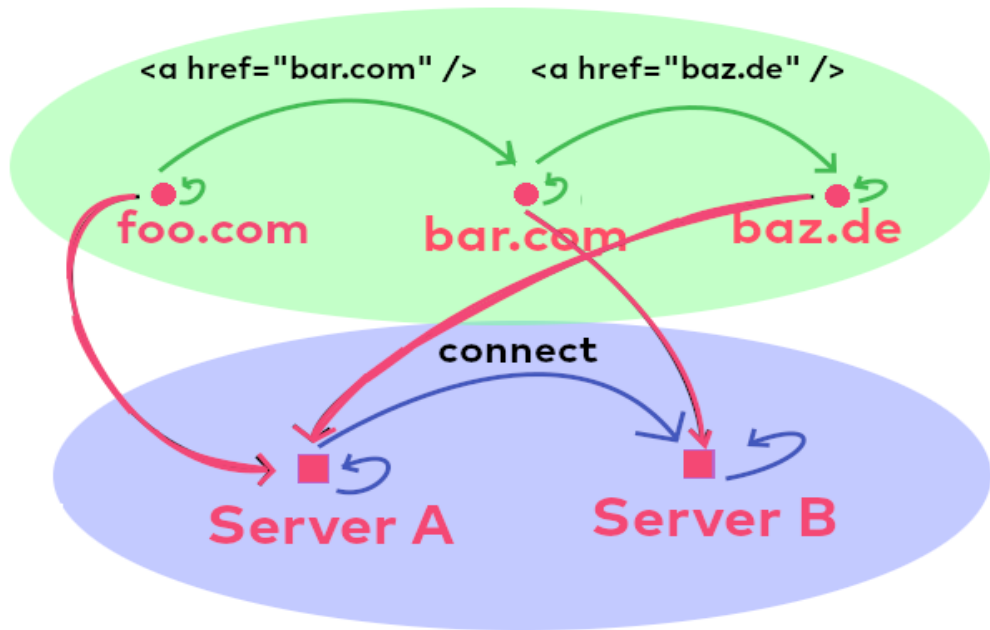




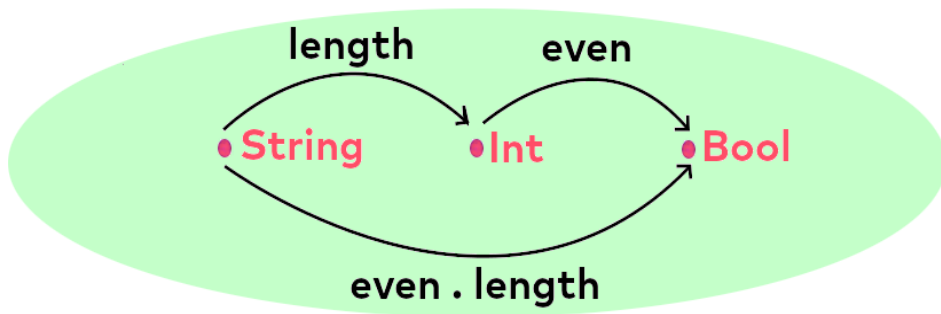
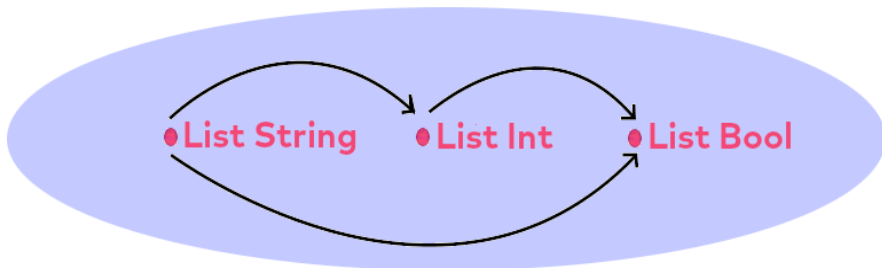


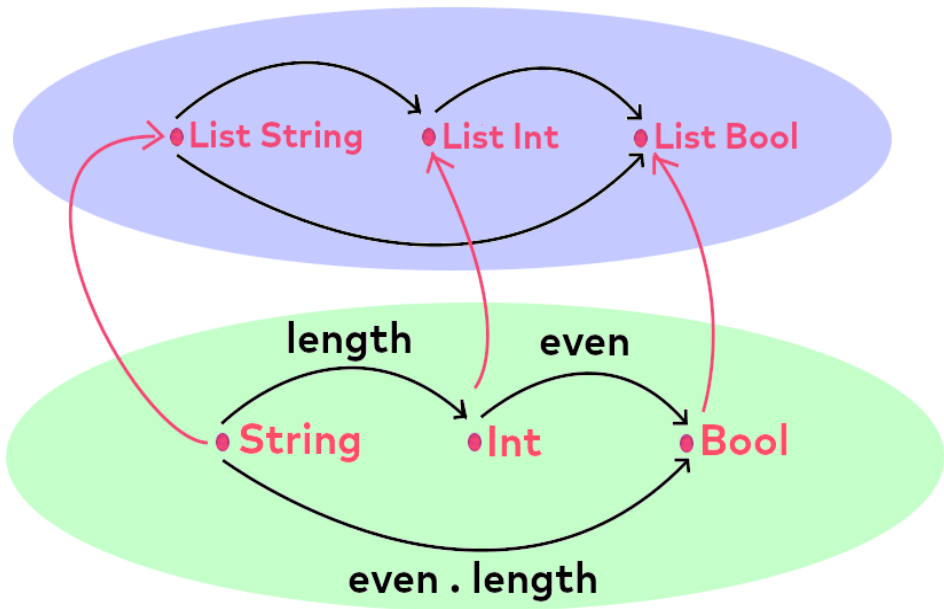


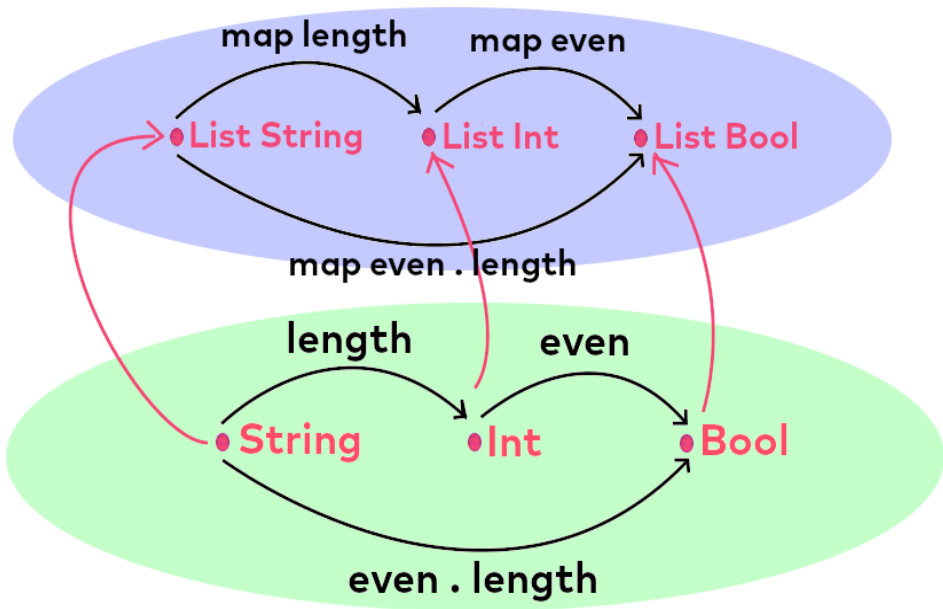




Functors in Programming







A Functor in ...

- Category theory: Mapping between categories

A Functor in ...

- Category theory: Mapping between categories
- Programming: Way to construct a richer type from a simpler type (e.g. `Int -> List Int`)

A Functor in ...

- Category theory: Mapping between categories
- Programming: Way to construct a richer type from a simpler type (e.g. `Int -> List Int`)

How do we do this in practice?

Enter: *fmap*

- The programmatic way of of **mapping** between types and functions.

Enter: *fmap*

- The programmatic way of of **mapping** between types and functions.
- **Lifting** simpler types into richer types

Enter: *fmap*

- The programmatic way of **mapping** between types and functions.
- **Lifting** simpler types into richer types
- Represented by the Functor class (by implementing `fmap`)


```
-- Functor interface
```

```
--
```

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

```
--                               Input 1: Function  
--                               ^^^^^^^^
```

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

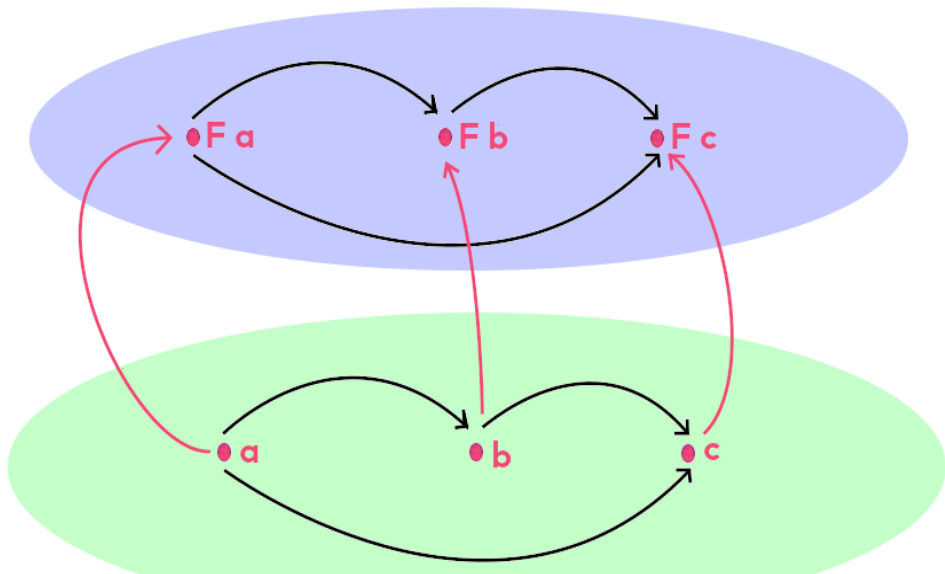
```
--                                     Input 2: Enriched type  
--                                     ^^^  
fmap :: Functor f => (a -> b) -> f a -> f b
```

```
--                                     Output: Enriched type  
--                                     ^^^  
fmap :: Functor f => (a -> b) -> f a -> f b
```

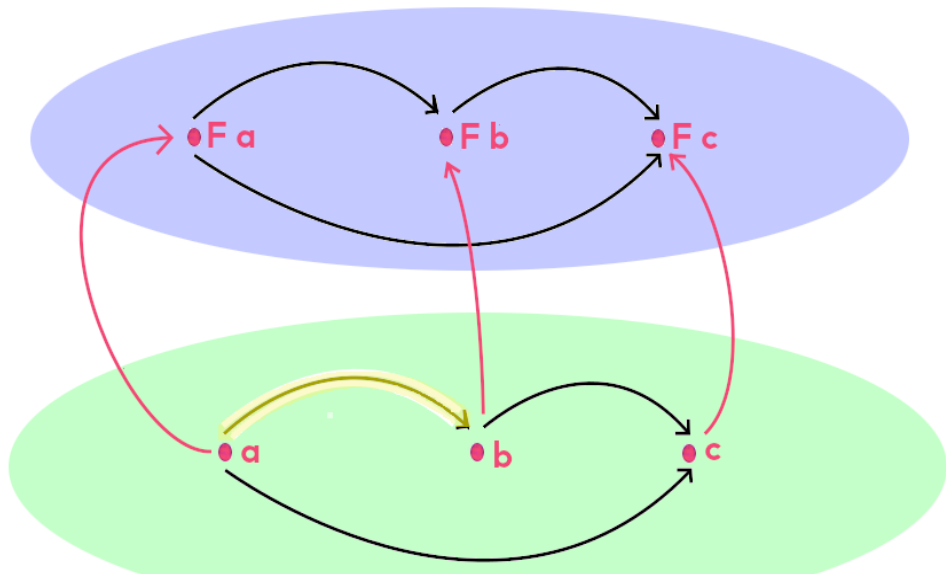
```
--                               Input 1: Function  
--                               ^^^^^^^^
```

```
fmap :: Functor f => (a -> b) -> (f a -> f b)
```

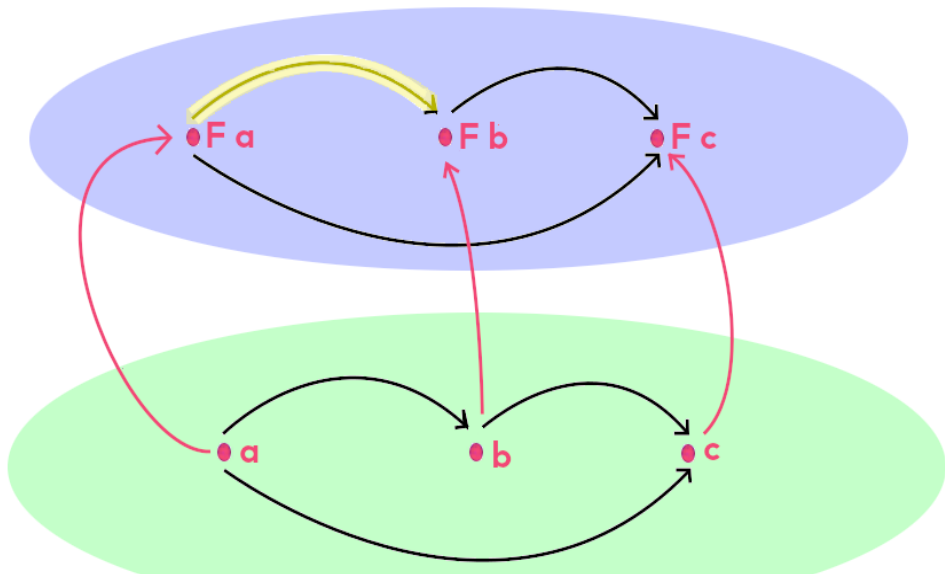
```
--                                Output: Enriched function
--                                ^^^^^^^^^^^^^^^
fmap :: Functor f => (a -> b) -> (f a -> f b)
```



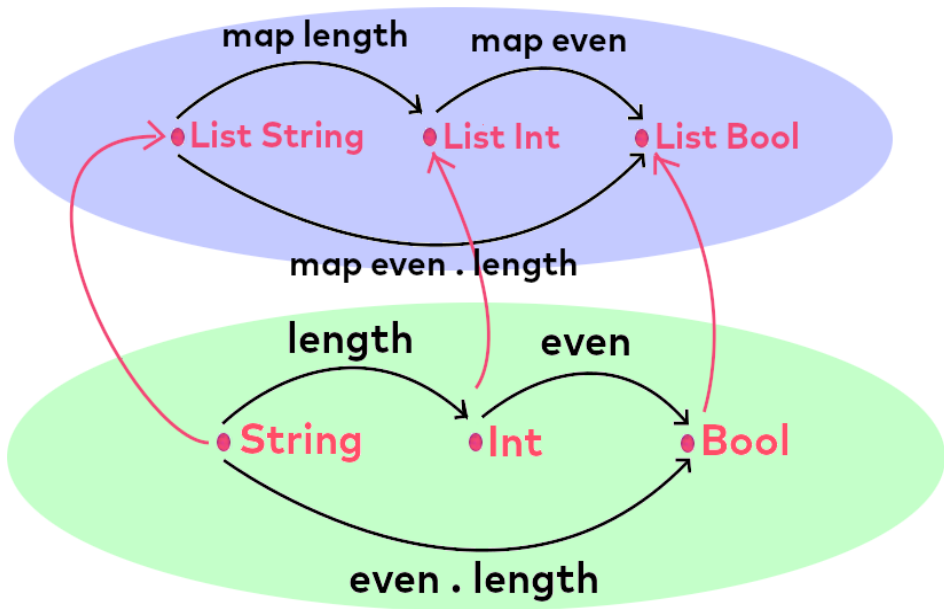
`fmap :: Functor F => (a -> b) -> (F a -> F b)`



`fmap :: Functor F => (a -> b) -> (F a -> F b)`



`fmap :: Functor F => (a -> b) -> (F a -> F b)`



The Functor in ...

- Category Theory: Represents new parts of categories

The Functor in ...

- Category Theory: Represents new parts of categories
- Programming: Represents new computational contexts

The Functor in ...

- Category Theory: Represents **new parts** of categories
 - ▶ Retaining structure!
- Programming: Represents **new computational contexts**
 - ▶ Retaining structure!

The Functor in ...

- Category Theory: Represents **new parts** of categories
 - ▶ Retaining structure!
- Programming: Represents **new computational contexts**
 - ▶ Retaining structure!

⇒ Lets us **focus on original program structure** in a new context

Example Contexts

Example Contexts

- List: Where computations may have multiple return values

Example Contexts

- List: Where computations may have multiple return values
- Maybe (Optional): Where failures might occur

Example Contexts

- List: Where computations may have **multiple return values**
- Maybe (Optional): Where **failures** might occur
- IO: Where **side effects** can happen

Example Contexts

- List: Where computations may have **multiple return values**
- Maybe (Optional): Where **failures** might occur
- IO: Where **side effects** can happen

→ Use the functor to abstract over the context!

Now show me how to make a type a functor!

```
--  
-- How to make List a functor  
--  
instance Functor [] where  
    fmap f xs = map f xs
```

```
--  
-- How to make Maybe a functor  
--  
instance Functor Maybe where  
    fmap f (Just x) = Just (f x)  
    fmap f Nothing = Nothing
```

```
--  
-- How to make IO a functor  
--  
instance Functor IO where  
    fmap f action = do  
        x <- action  
        return (f x)
```

List Implements fmap!

```
prompt> fmap length ["Y0", "Y00", "Y000"]  
[2,3,4]
```

```
prompt> fmap even [1..10]  
[False,True,False,True,False,True,False,True,False,True]
```

```
prompt> fmap (even . length) ["ah", "aha", "ehhhhh"]  
[True,False,True]
```


Maybe Implements fmap!

```
ghci> fmap even Nothing
```

```
Nothing
```

```
ghci> fmap length (Just "Y000")
```

```
Just 4
```

```
ghci> fmap (even . Length) (Just "Y000")
```

```
(Just True)
```

IO implements fmap!

IO implements fmap!

Get a string from the command line...

```
prompt> fmap length getLine
```

```
HELLOWORLD
```

```
10
```

... and an integer

```
prompt> fmap even getInt
```

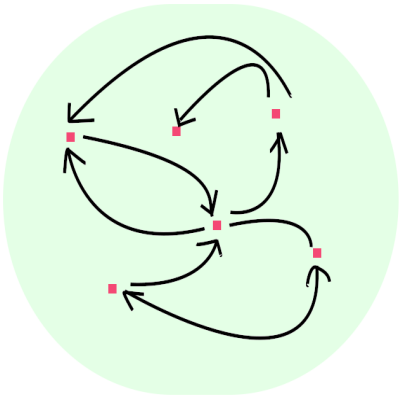
```
33
```

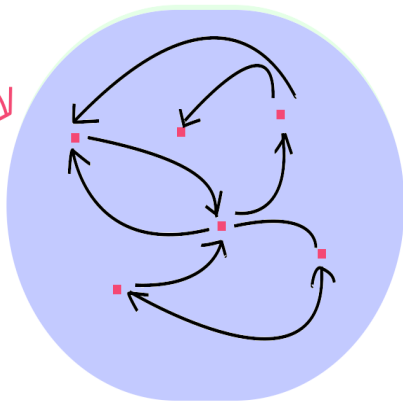
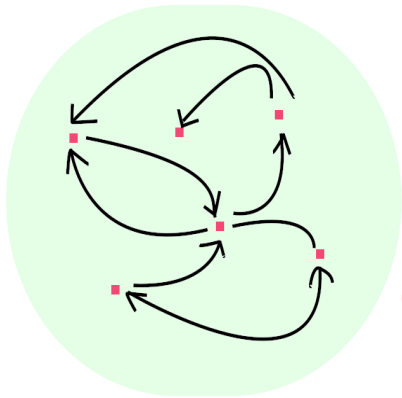
```
False
```

```
prompt> fmap (even . length) getLine
```

```
HELLO
```
























```
False
```





Instances

| | | |
|--|----------|-----------------|
|  Functor [] | # Source | Since: 2.1 |
|  Functor Maybe | # Source | Since: 2.1 |
|  Functor IO | # Source | Since: 2.1 |
|  Functor Par1 | # Source | Since: 4.9.0.0 |
|  Functor NonEmpty | # Source | Since: 4.9.0.0 |
|  Functor ReadP | # Source | Since: 2.1 |
|  Functor ReadPrec | # Source | Since: 2.1 |
|  Functor Down | # Source | Since: 4.11.0.0 |
|  Functor Product | # Source | Since: 4.8.0.0 |
|  Functor Sum | # Source | Since: 4.8.0.0 |
|  Functor Dual | # Source | Since: 4.8.0.0 |
|  Functor Last | # Source | Since: 4.8.0.0 |
|  Functor First | # Source | Since: 4.8.0.0 |
|  Functor STM | # Source | Since: 4.3.0.0 |
|  Functor Handler | # Source | Since: 4.6.0.0 |
|  Functor Identity | # Source | Since: 4.8.0.0 |
|  Functor ZipList | # Source | Since: 2.1 |
|  Functor ArgDescr | # Source | Since: 4.6.0.0 |
|  Functor OptDescr | # Source | Since: 4.6.0.0 |
|  Functor ArgOrder | # Source | Since: 4.6.0.0 |
|  Functor Option | # Source | Since: 4.9.0.0 |
|  Functor Last | # Source | Since: 4.9.0.0 |
| Functor First | # Source | Since: 4.9.0.0 |

| | | |
|---|----------|-----------------|
|  Functor Max | # Source | Since: 4.9.0.0 |
|  Functor Min | # Source | Since: 4.9.0.0 |
|  Functor Complex | # Source | Since: 4.9.0.0 |
|  Functor (Either a) | # Source | Since: 3.0 |
|  Functor (V1 :: Type -> Type) | # Source | Since: 4.9.0.0 |
|  Functor (U1 :: Type -> Type) | # Source | Since: 4.9.0.0 |
|  Functor ((,) a) | # Source | Since: 2.1 |
|  Functor (ST s) | # Source | Since: 2.1 |
|  Functor (Proxy :: Type -> Type) | # Source | Since: 4.7.0.0 |
|  Arrow a => Functor (ArrowMonad a) | # Source | Since: 4.6.0.0 |
|  Monad m => Functor (WrappedMonad m) | # Source | Since: 2.1 |
|  Functor (ST s) | # Source | Since: 2.1 |
|  Functor (Arg a) | # Source | Since: 4.9.0.0 |
|  Functor f => Functor (Rec1 f) | # Source | Since: 4.9.0.0 |
|  Functor (URec Char :: Type -> Type) | # Source | Since: 4.9.0.0 |
|  Functor (URec Double :: Type -> Type) | # Source | Since: 4.9.0.0 |
|  Functor (URec Float :: Type -> Type) | # Source | Since: 4.9.0.0 |
|  Functor (URec Int :: Type -> Type) | # Source | Since: 4.9.0.0 |
|  Functor (URec Word :: Type -> Type) | # Source | Since: 4.9.0.0 |
|  Functor (URec (Ptr ()) :: Type -> Type) | # Source | Since: 4.9.0.0 |
|  Functor f => Functor (Alt f) | # Source | Since: 4.8.0.0 |
|  Functor f => Functor (Ap f) | # Source | Since: 4.12.0.0 |
|  Functor (Const m :: Type -> Type) | # Source | Since: 2.1 |
| Arrow a => Functor (WrappedArrow a b) | # Source | Since: 2.1 |

Thinking categorically gives us

- Structure for free

Thinking categorically gives us

- Structure for free
- Instant context switching

Thinking categorically gives us

- Structure for free
- Instant context switching
- In other words

Thinking categorically gives us

- Structure for free
- Instant context switching
- In other words
 - ▶ Flexibility

Thinking categorically gives us

- Structure for free
- Instant context switching
- In other words
 - ▶ Flexibility
 - ▶ Code reuse

Thinking categorically gives us

- Structure for free
- Instant context switching
- In other words
 - ▶ Flexibility
 - ▶ Code reuse
 - ▶ Separation of concerns

Thinking categorically gives us

- Structure for free
- Instant context switching
- In other words
 - ▶ Flexibility
 - ▶ Code reuse
 - ▶ Separation of concerns
 - ▶ Modularity





[0]

- Thinking categorically will might not make webpack compile

- Thinking categorically will might not make webpack compile
- But It will:

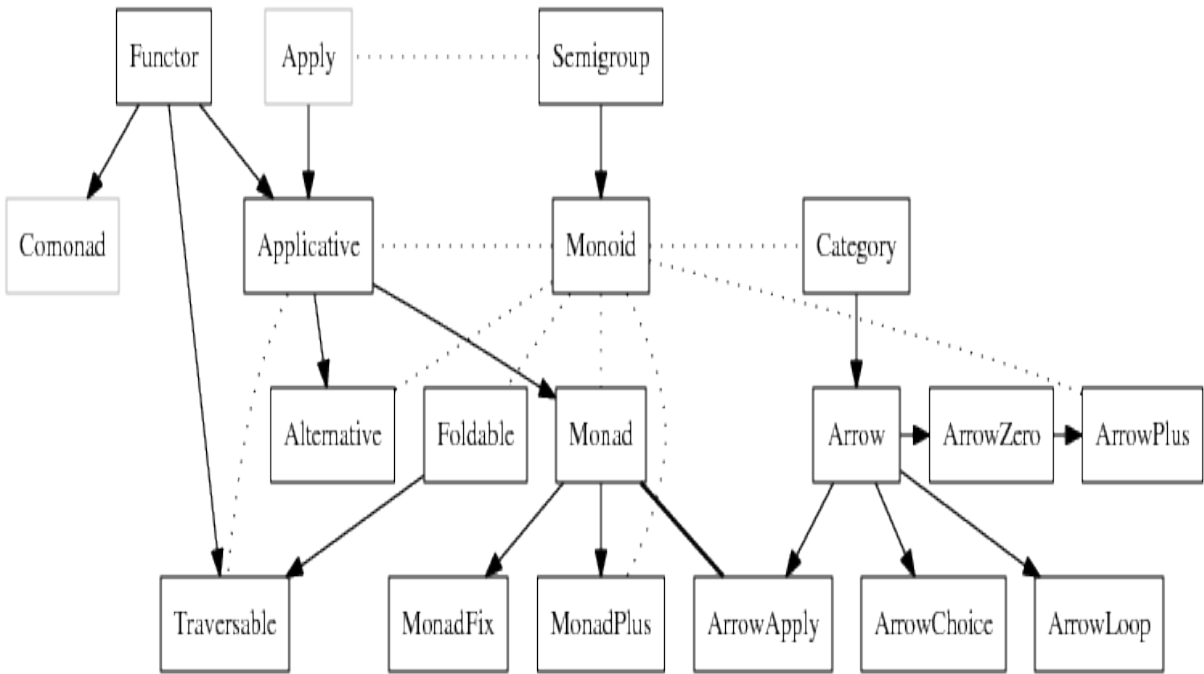
- Thinking categorically will might not make webpack compile
- But It will:
 - ▶ Help us expose structure

- Thinking categorically will might not make webpack compile
- But It will:
 - ▶ Help us expose structure
 - ▶ Give us a different view on context

- Thinking categorically will might not make webpack compile
- But It will:
 - ▶ Help us expose structure
 - ▶ Give us a different view on context
 - ▶ Help us change and extend locally without complecting globally

- Thinking categorically will might not make webpack compile
- But It will:
 - ▶ Help us expose structure
 - ▶ Give us a different view on context
 - ▶ Help us change and extend locally without complecting globally

The functor is just the beginning ...



A functor as a Type Class

- Extends the behavior of data

A functor as a Type Class

- Extends the behavior of data
 - ▶ Not the data itself

A functor as a Type Class

- Extends the behavior of data
 - ▶ Not the data itself
- Is part of a relation to other behaviors

A functor as a Type Class

- Extends the behavior of data
 - ▶ Not the data itself
- Is part of a relation to other behaviors
 - ▶ Not a part of a rigid type hierarchy

A functor as a Type Class

- Extends the behavior of data
 - ▶ Not the data itself
- Is part of a relation to other behaviors
 - ▶ Not a part of a rigid type hierarchy
- Its implementation is open

A functor as a Type Class

- Extends the behavior of data
 - ▶ Not the data itself
- Is part of a relation to other behaviors
 - ▶ Not a part of a rigid type hierarchy
- Its implementation is open
 - ▶ Not bound to the class that implements the interface

A functor as a Type Class

- Extends the behavior of data
 - ▶ Not the data itself
- Is part of a relation to other behaviors
 - ▶ Not a part of a rigid type hierarchy
- Its implementation is open
 - ▶ Not bound to the class that implements the interface
- Gives us information about a function without looking at its implementation

A functor as a Type Class

- Extends the behavior of data
 - ▶ Not the data itself
- Is part of a relation to other behaviors
 - ▶ Not a part of a rigid type hierarchy
- Its implementation is open
 - ▶ Not bound to the class that implements the interface
- Gives us information about a function without looking at its implementation
 - ▶ Does not require us learning each context independently

A functor as a Type Class

- Extends the behavior of data
 - ▶ Not the data itself
- Is part of a relation to other behaviors
 - ▶ Not a part of a rigid type hierarchy
- Its implementation is open
 - ▶ Not bound to the class that implements the interface
- Gives us information about a function without looking at its implementation
 - ▶ Does not require us learning each context independently
- \implies Blends out the details, focus on the interactions

Summary

- Fundamentally, we solve all problems the same way: splitting, solving, recursing, composing

Summary

- Fundamentally, we solve all problems the same way: splitting, solving, recursing, composing
- Structure emerges through composition

Summary

- Fundamentally, we solve all problems the same way: splitting, solving, recursing, composing
- Structure emerges through composition
- Pure, math-inspired functions are the most natural tool to model problem solving in computer programming

Summary

- Fundamentally, we solve all problems the same way: splitting, solving, recursing, composing
- Structure emerges through composition
- Pure, math-inspired functions are the most natural tool to model problem solving in computer programming
- Category theory lets us study composition ...

Summary

- Fundamentally, we solve all problems the same way: splitting, solving, recursing, composing
- Structure emerges through composition
- Pure, math-inspired functions are the most natural tool to model problem solving in computer programming
- Category theory lets us study composition ...
- ...and provides tools such as the functor that encourages us to focus on interactions between things, not things themselves

Structurize, don't optimize 😊

Thank you! Questions?



Ludvig Sundström
ludvig.sundstroem@innoq.com



+49 1516 1181270



@l5und

innoQ Deutschland GmbH

Krischerstr. 100
40789 Monheim a. Rh.
Germany
+49 2173 3366-0

Ohlauer Str. 43
10999 Berlin
Germany

Ludwigstr. 180E
63067 Offenbach
Germany

Kreuzstr. 16
80331 München
Germany

c/o WeWork
Hermannstrasse 13
20095 Hamburg
Germany

innoQ Schweiz GmbH

Gewerbestr. 11
CH-6330 Cham
Switzerland
+41 41 743 01 11

Albulastr. 55
8048 Zürich
Switzerland

- 0 : <http://cdn.makeuseof.com/wp-content/uploads/2014/09/stress-free-programming-frustration.jpg?x92042>
- 1 : <https://insights.stackoverflow.com/survey/2019#technology>
- 2 : https://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspondence
- 3 : https://en.wikipedia.org/wiki/Design_Patterns
- 4 : https://golem.ph.utexas.edu/category/2012/01/vorsicht_funktor.html

Laws

- Associativity in a category: $h \cdot g \cdot f = (h \cdot g) \cdot f = h \cdot (g \cdot f)$
- Identity in a category (for $f :: a \rightarrow b$): $f \cdot \text{id}_a = f$, $\text{id}_b \cdot f = f$
- Functor retains structure under composition:
if $h = g \cdot f$, then $F h = F g \cdot F f$
- Functor retains structure under identity: $F \text{id}_a = \text{id}_{\{F a\}}$

Curry-Howard Isomorphism

- $\text{Void} \iff \text{False}$
- $() \iff \text{True}$
- $\text{Product Types} \iff \text{OR}$
- $\text{Sum Types} \iff \text{AND}$
- $A \rightarrow B \iff \text{If } A \text{ then } B$

Notes on functor as a typeclass

Interfaces methods are always associated with an object instance. In other words, there is always an implied 'this' parameter that is the object on which the method is called. All inputs to a type class function are explicit.