# Functors

September 20, 2024

# Functions

- Consider the successor function:

# Functions

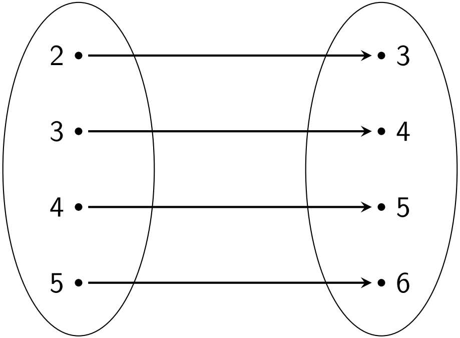
- Consider the successor function:
  - `succ 0 = 1`

# Functions

- Consider the successor function:
  - `succ 0 = 1`
  - `succ 1 = 2`

# Functions

- Consider the successor function:
  - `succ 0 = 1`
  - `succ 1 = 2`

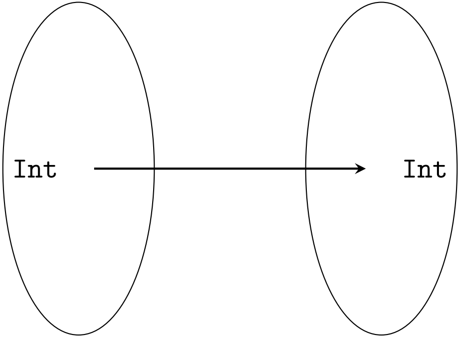


# Functions

- More generally, consider types as our objects of interest.

# Functions

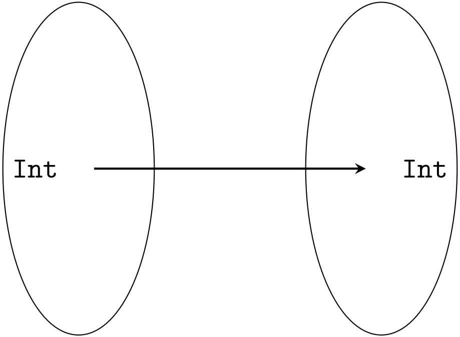
- More generally, consider types as our objects of interest.




- `succ :: Int -> Int`

# Functions

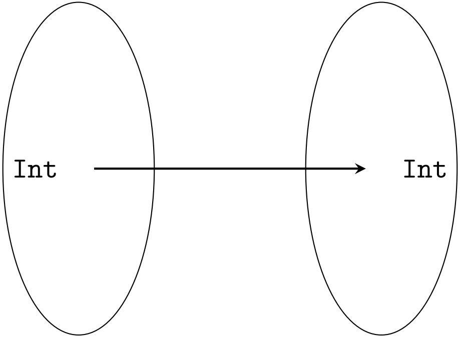
- More generally, consider types as our objects of interest.




- `succ :: Enum a => a -> a`

# Functions

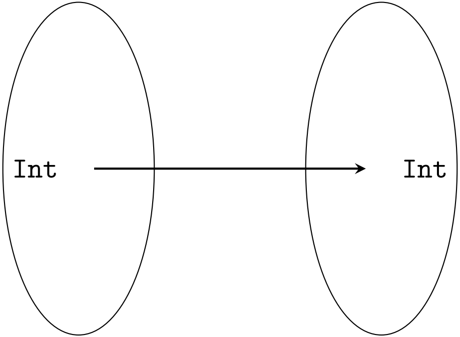
- More generally, consider types as our objects of interest.




- `succ :: Int -> Int`

# Functions

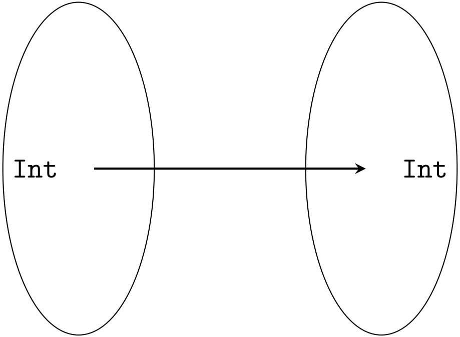
- More generally, consider types as our objects of interest.




- `succ :: Int -> Int`
- `succ` is a function from `Int`s to `Int`s.

# Functions

- More generally, consider types as our objects of interest.




- `succ :: Int -> Int`
- `succ` is a morphism from `Int`s to `Int`s.

# Maps

- Let us look at maps:

# Maps

- Let us look at maps:
  - `map succ [] = []`

# Maps

- Let us look at maps:
  - `map succ [] = []`
  - `map succ [0] = [1]`

# Maps

- Let us look at maps:
  - `map succ [] = []`
  - `map succ [0] = [1]`
  - `map succ [1,2] = [2,3]`

# Maps

- Let us look at maps:
  - `map succ [] = []`
  - `map succ [0] = [1]`
  - `map succ [1,2] = [2,3]`
  - `map succ [3,4,5] = [4,5,6]`

# Maps

- Let us look at maps:
  - `map succ [] = []`
  - `map succ [0] = [1]`
  - `map succ [1,2] = [2,3]`
  - `map succ [3,4,5] = [4,5,6]`
- van Eijck and Unger: The function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.

# Maps

- Let us look at maps:
  - `map succ [] = []`
  - `map succ [0] = [1]`
  - `map succ [1,2] = [2,3]`
  - `map succ [3,4,5] = [4,5,6]`
- van Eijck and Unger: The function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.
- `map :: (a -> b) -> [a] -> [b]`

# Maps

- Now let us curry `map`:

# Maps

- Now let us curry `map`:

# Maps

- Now let us curry `map`:

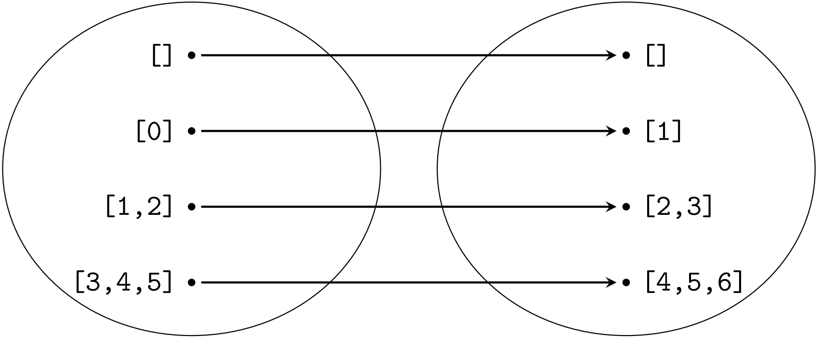
- `map :: (a -> b) -> ([a] -> [b])`

# Maps

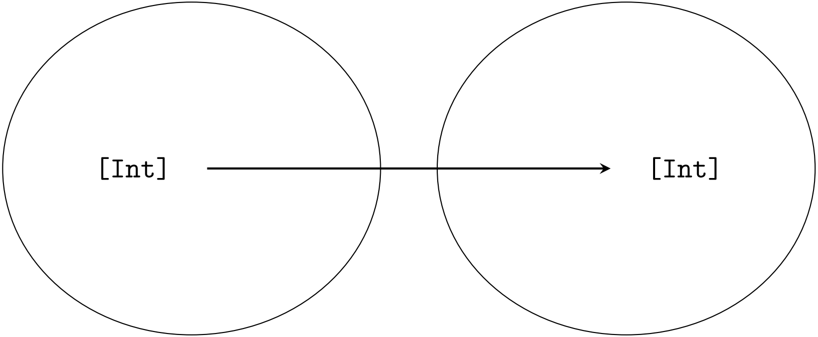
- Now let us curry `map`:

- `map :: (a -> b) -> ([a] -> [b])`
- The function `map` takes a function from `a` to `b` and returns a function from `[a]` to `[b]`.

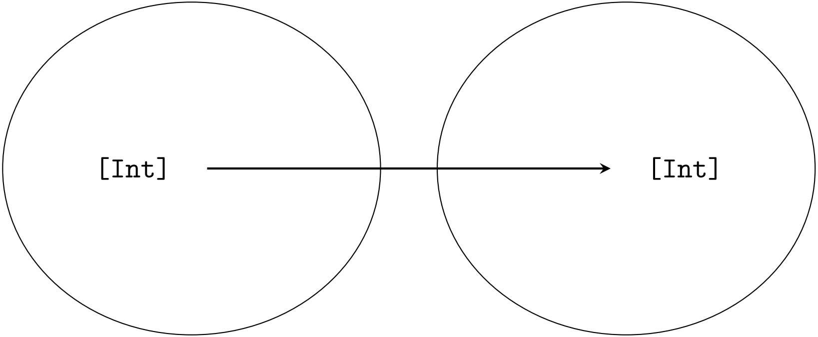
# Maps



# Maps



# Maps




- `map succ :: [Int] -> [Int]`

# Functors

- Wikipedia: Let $C$ and $D$ be categories. A functor $F$ from $C$ to $D$ is a mapping that
  - associates to each object $X$ in $C$ an object $F(X)$ in $D$,
  - associates to each morphism $f\colon X \to Y$ in $C$ a morphism $F(f)\colon F(X) \to F(Y)$ in $D$ such that the following two conditions hold:
    - $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ for every object $X$ in $C$,
    - $F(g \circ f) = F(g) \circ F(f)$ for all morphisms $f\colon X \to Y$ and $g\colon Y \to Z$ in $C$.

  That is, functors must preserve identity morphisms and composition of morphisms.

# Functors

- Let $C = D = \texttt{Hask}$.
  - $\texttt{Hask}$ = the category of Haskell types
    - category = objects + morphisms
      - objects = types
      - morphisms = functions

A functor $\texttt{F}$ from $C$ to $D$ is a mapping that

- associates to each object $X$ in $C$ an object $F(X)$ in $D$,
- associates to each morphism $f\colon X \to Y$ in $C$ a morphism $F(f)\colon F(X) \to F(Y)$ in $D$ such that the following two conditions hold:
  - $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ for every object $X$ in $C$,
  - $F(g \circ f) = F(g) \circ F(f)$ for all morphisms $f\colon X \to Y$ and $g\colon Y \to Z$ in $C$.

That is, functors must preserve identity morphisms and composition of morphisms.

# Functors

- Let $C = D =$ `Hask`.
  - `Hask` = the category of Haskell types
    - category = objects + morphisms
      - objects = types
      - morphisms = functions

A functor `F` from $C$ to $D$ is a mapping that

- associates to each type `a` a type `F a`,
- associates to each morphism $f\colon X \to Y$ in $C$ a morphism $F(f)\colon F(X) \to F(Y)$ in $D$ such that the following two conditions hold:
  - $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ for every object $X$ in $C$,
  - $F(g \circ f) = F(g) \circ F(f)$ for all morphisms $f\colon X \to Y$ and $g\colon Y \to Z$ in $C$.

That is, functors must preserve identity morphisms and composition of morphisms.

# Functors

- Let $C = D =$ `Hask`.
  - `Hask` = the category of Haskell types
    - category = objects + morphisms
      - objects = types
      - morphisms = functions

A functor `F` from $C$ to $D$ is a mapping that

- associates to each type `a` a type `F a`,
- associates to each function `f :: a -> b` a function `fmap f :: F a -> F b` such that the following two conditions hold:
  - $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ for every object $X$ in $C$,
  - $F(g \circ f) = F(g) \circ F(f)$ for all morphisms $f\colon X \to Y$ and $g\colon Y \to Z$ in $C$.

That is, functors must preserve identity morphisms and composition of morphisms.

# Functors

- Let $C = D =$ `Hask`.
  - `Hask` = the category of Haskell types
    - category = objects + morphisms
      - objects = types
      - morphisms = functions

A functor `F` from $C$ to $D$ is a mapping that

- associates to each type `a` a type `F a`,
- associates to each function `f :: a -> b` a function `fmap f :: F a -> F b` such that the following two conditions hold:
  - `fmap id = id`,
  - $F(g \circ f) = F(g) \circ F(f)$ for all morphisms $f : X \to Y$ and $g : Y \to Z$ in $C$.

That is, functors must preserve identity morphisms and composition of morphisms.

# Functors

- Let $C = D =$ `Hask`.
  - `Hask` = the category of Haskell types
    - category = objects + morphisms
      - objects = types
      - morphisms = functions

A functor `F` from $C$ to $D$ is a mapping that

- associates to each type `a` a type `F a`,
- associates to each function `f :: a -> b` a function `fmap f :: F a -> F b` such that the following two conditions hold:
  - `fmap id = id`,
  - `fmap (g . f) = fmap g . fmap f`.

That is, functors must preserve identity morphisms and composition of morphisms.

# Functors

- Let $C = D =$ `Hask`.
  - `Hask` = the category of Haskell types
    - category = objects + morphisms
      - objects = types
      - morphisms = functions

A functor `F` from $C$ to $D$ is a mapping that

- associates to each type `a` a type `F a`,
- associates to each function `f :: a -> b` a function `fmap f :: F a -> F b` such that the following two conditions hold:
  - `fmap id = id`,
  - `fmap (g . f) = fmap g . fmap f`.

That is, functors must preserve identity morphisms and composition of morphisms. (Haskell will not do this for you—you have to do it yourself)

# Lists are Functors

- ```
  instance Functor [] where
      fmap = map
  ```
  ```

# Lists are Functors

- ```
  instance Functor [] where
      fmap = map
  ```
- For each type a there is a type [a]
  ```

# Lists are Functors

- `instance Functor [] where`
  `    fmap = map`
- For each type `a` there is a type `[a]`
- For each function `f :: a -> b` there is a function `map f :: [a] -> [b]` such that the following two conditions hold:

# Lists are Functors

- Identity: `map id = id`

# Lists are Functors

- Identity: `map (\y -> y) xs = (\y -> y) xs = xs`

# Lists are Functors

- Identity: `map (\y -> y) xs = (\y -> y) xs = xs`

```
map (\y -> y) [] = []
```

# Lists are Functors

- Identity: `map (\y -> y) xs = (\y -> y) xs = xs`

```
map (\y -> y) [] = []

map (\y -> y) (x:xs) = (\y -> y) x :  map (\y -> y) xs
```

# Lists are Functors

- Identity: `map (\y -> y) xs = (\y -> y) xs = xs`

```
map (\y -> y) [] = []

map (\y -> y) (x:xs) = (\y -> y) x :  map (\y -> y) xs
                     = (\y -> y) x :  xs
```

# Lists are Functors

- Identity: `map (\y -> y) xs = (\y -> y) xs = xs`

```
map (\y -> y) [] = []

map (\y -> y) (x:xs) = (\y -> y) x :  map (\y -> y) xs
                     = (\y -> y) x :  xs
                     = (x:xs)
```

# Lists are Functors

- Composition: `(map g . map f) xs = map (g . f) xs`

# Lists are Functors

- Composition: `(map g . map f) xs = map (g . f) xs`

```
(map g . map f) [] = map g (map f [])
```

# Lists are Functors

- Composition: `(map g . map f) xs = map (g . f) xs`

```
(map g . map f) [] = map g (map f [])
                   = map g []
```

# Lists are Functors

- Composition: `(map g . map f) xs = map (g . f) xs`

```
(map g . map f) [] = map g (map f [])
                   = map g []
                   = []
```

# Lists are Functors

- Composition: `(map g . map f) xs = map (g . f) xs`

```
(map g . map f) [] = map g (map f [])
                   = map g []
                   = []

(map g . map f) (x:xs) = map g (map f (x:xs))
```

# Lists are Functors

- Composition: `(map g . map f) xs = map (g . f) xs`

```
(map g . map f) [] = map g (map f [])
                   = map g []
                   = []

(map g . map f) (x:xs) = map g (map f (x:xs))
                       = map g (f x : map f xs)
```

# Lists are Functors

- Composition: `(map g . map f) xs = map (g . f) xs`

```
(map g . map f) [] = map g (map f [])
                   = map g []
                   = []

(map g . map f) (x:xs) = map g (map f (x:xs))
                       = map g (f x : map f xs)
                       = g (f x) : map g (map f xs)
```

# Lists are Functors

- Composition: `(map g . map f) xs = map (g . f) xs`

```
(map g . map f) [] = map g (map f [])
                   = map g []
                   = []

(map g . map f) (x:xs) = map g (map f (x:xs))
                       = map g (f x : map f xs)
                       = g (f x) : map g (map f xs)
                       = (g . f) x : (map g . map f) xs
```

# Lists are Functors

- Composition: `(map g . map f) xs = map (g . f) xs`

```
(map g . map f) [] = map g (map f [])
                   = map g []
                   = []

(map g . map f) (x:xs) = map g (map f (x:xs))
                       = map g (f x : map f xs)
                       = g (f x) : map g (map f xs)
                       = (g . f) x : (map g . map f) xs
                       = (g . f) x : map (g . f) xs
```

# Lists are Functors

- Composition: `(map g . map f) xs = map (g . f) xs`

```
(map g . map f) [] = map g (map f [])
                   = map g []
                   = []

(map g . map f) (x:xs) = map g (map f (x:xs))
                       = map g (f x : map f xs)
                       = g (f x) : map g (map f xs)
                       = (g . f) x : (map g . map f) xs
                       = (g . f) x : map (g . f) xs
                       = map (g . f) (x:xs)
```

# Functors

- Other examples of functors:

# Functors

- Other examples of functors:
  - `Maybe`

# Functors

- Other examples of functors:
  - `Maybe`
  - `IO`

# Functors

- Other examples of functors:
  - `Maybe`
  - `IO`
  - Functions (`((->) r`)

# Functors

- Functors are boxes

# Functors

- Functors are boxes
    - That implement maps that lift normal functions (of type `a -> b`) to functions over boxes (of type `F a -> F b`)

# Functors

- Functors represent context
  - That implement maps that lift normal functions (of type `a -> b`) to functions over context (of type `F a -> F b`)

# Functors

- Functors represent context
  - That implement maps that lift normal functions (of type `a -> b`) to functions over context (of type `F a -> F b`)
    - `IO`: input/output
    - `Maybe`: possible failure
    - `[]`: nondeterminism