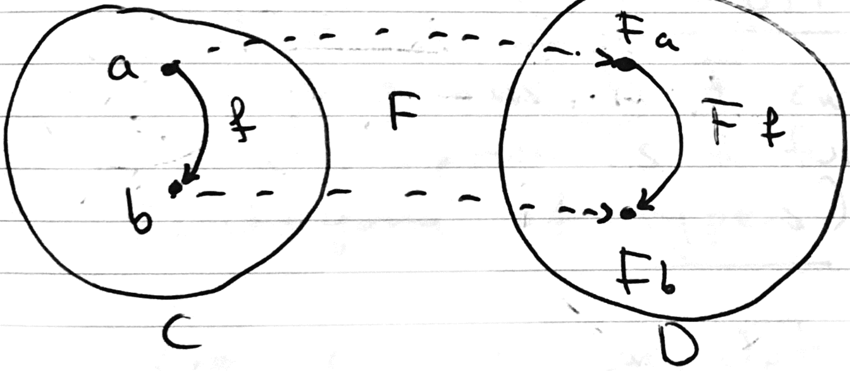
INF-3910 Lecture 7

Functors

- Functors are a simple, yet very powerful abstraction in functional programming
- We have already seen how <u>map</u> abstracts looping for IEnumerables (list, array, seq)
- But functors are much more general than just looping! we can map over a lot of different structures!
- In category theory a category is defined in terms of objects, morphisms and composition of morphisms (think functions and elements):

$$\begin{tikzcd} a \arrow[r, "f"] \arrow[rr, bend right, "g \cdot f"'] & b \arrow[r, "g"] & c \end{tikzcd}$$

- In category theory a functor is a function between two categaries which maps every object in C to an object in D
- Since morphisms (functions) are also objects, all functions are also mapped. This means that all connections betwen objects are preserved!
- Functors also preserve composition and identity morphisms:



note! this is very different from $F(a \to b)$!

$F(a \to Fa)$

$F(f : a \to b) \to (Ff : Fa \to Fb)$ | Functor laws

$F(g \cdot f) \to Fg \cdot Ff$

$F(id_a) \to id_{Fa}$ ← preserve identity

Note! Functors need not be injective! They can collaps structure!

- ok, enough mathematical nonsense! What!?
- In programming, every time we have a <u>generic</u> type, we potentially have a functor!

type $\underbrace{\text{Option}\langle\text{'a}\rangle}$ = | Some 'a | Nothing

this is a type function (constructor), taking a type 'a and returning a new type!

$$\boxed{\text{Option} : \text{'a} \to \text{Option 'a}}$$

Compare this to F on the previous page. Option is a functor (in the category of <u>types</u>)!

- We can create an Option<int> from an int by using one of the <u>data constructors</u>

let x = Some 42 : Option <int>

let y = Nothing : Option<int>

- But what about functions?

Some f : Option <int → int>

is very different from

f': Option <int> → Option <int> !

- How do we transform $f : a \to b$ to $f' : Option<'a> \to Option<'b>$?
- The answer is: we define a function usually called map (or fmap)!

```
let map (f: a -> b) (x: Option<a>) : Option<b> =
    match x with
    | Some v -> Some (f v)
    | Nothing -> Nothing
```

- map is a function which has the following signature:

map : (f : a -> b) -> Option<a> -> Option<b>

which is exactly what we are looking for.

- But what about composition and identity?

fmap id = id (id : 'a -> 'a)

fmap id Nothing = Nothing = id Nothing

fmap id (Some x) = Some (id x)
= Some x
= id (Some x)

composition: $[\text{fmap } g \ll \text{fmap } f = \text{fmap } (g \ll f)]$

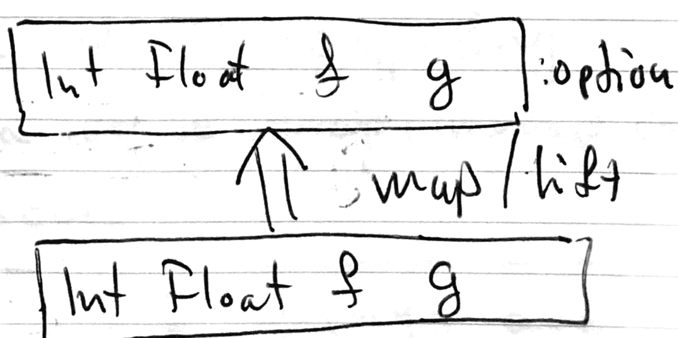
$$
\begin{aligned}
&\text{fmap } (g \ll f) \text{ Nothing} = \text{Nothing} \\
&= \text{fmap } g \text{ Nothing} = \text{fmap } g \text{ (fmap } f \text{ Nothing)} \\
&= (\text{fmap } g \ll \text{fmap } f) \text{ Nothing}
\end{aligned}
$$

$$
\begin{aligned}
&\text{fmap } (g \ll f) \text{ (Some } x) = \text{Some } (g (f\, x)) \\
&= \text{fmap } g \text{ (Some } (f\, x)) \\
&= \text{fmap } g \text{ (fmap } f \text{ (Some } x)) \\
&= (\text{fmap } g \ll \text{fmap } f) \text{ (Some } x)
\end{aligned}
$$

□

Note that for these relations to hold f and g must be pure, stateless functions!

- map/fmap should really be called <u>lift</u> instead, as it lifts functions into an "elevated world":



- lift can be generalized to liftN

- How is all this relevant to programmers?
- There is a very common pattern where we unwrap (pattern match) a type, apply a transformation and wrap it up again. This is handled much more elegantly using map!

Example:

```
let cust : Option<Customer> = getCustomer 123
match cust with
| Some c -> Some <| processCustomer c
| None -> None
```

vs.

```
Option.map processCustomer (getCustomer 123)
```

In particular, when we start composing transformations we get a lot of wrap/unwrap code. Functors to the rescue!

> Keep your eyes open, these things are everywhere!

Examples of functors:

Option<a>

- List<a>: map f x =

```
    match x with
    | [] -> []
    | h :: t -> f h :: map f t
```

Identity<a>: map f = function | Id x -> Id (f x)

Const<a, b>
  Const of a

```
map f =
  | Const x -> Const x
```

Functions! type Fun<a, r> = Fun (a -> r)

map (f: r -> s) (x: a -> r) : a -> s

```
let map f x =
  match x with
  | Fun g -> g >> f   (= f << g)
```

or simply let map = (<<)

map is simply function composition!

- The function functor is also called the Reader functor