

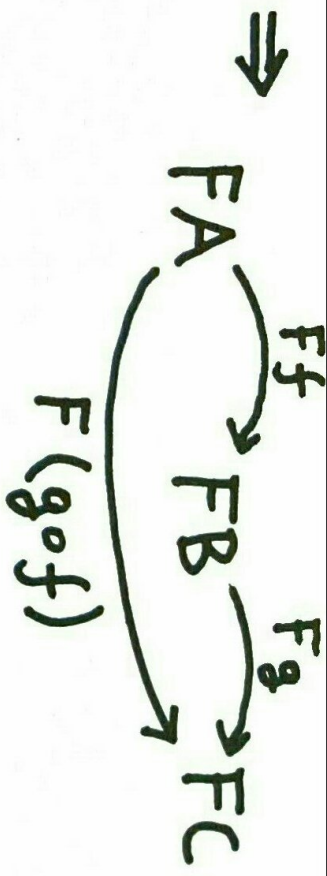
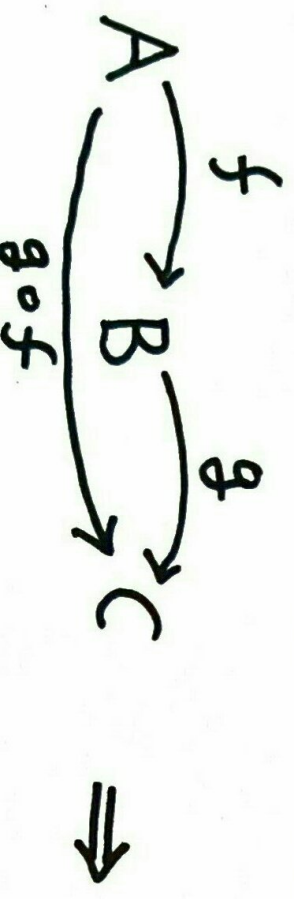
A functor is an operator that maps types to types  
 $X \text{ type} \Rightarrow FX \text{ type}$   
 and functions to functions

$$f: A \rightarrow B \Rightarrow Ff: FA \rightarrow FB$$

It must preserve identities and compositions

$$A \xrightarrow{id_A} A \Rightarrow FA \xrightarrow{Fid_A} FA$$

$$Fid_A = id_{FA}$$



$$F(g \circ f) = Fg \circ Ff$$

Strictly Positive Functor

$FX$  is defined by an expression where  $X$  occurs only on the right of arrows

Examples:

$$FX = \mathbb{1} + X$$

$$FX = \mathbb{1} + A \times X \text{ for fixed } A$$

$$FX = A + X \times X$$

$$FX = (X \rightarrow A) \rightarrow A$$

← not strictly positive

Every strictly positive functor specifies the structure of

a recursive type  $\mu F$

•  $FX = 1 + X$

one constant initial element  $\Downarrow$  0  
one unary constructor  $\Uparrow$  succ

$\mu F$  corresponds to Nat

•  $FX = 1 + X \times X$

one constant  $\Downarrow$  nil  
unary constructor with parameter A  $\Uparrow$  (::)

$\mu F$  correspond to List<sub>A</sub>

•  $FX = A + X \times X$

a constant for every element of A  $\Downarrow$  leaf  
binary constructor  $\Uparrow$  node

$\mu F$  corresponds to Tree<sub>A</sub>

# Rules for the Inductive Type

$\mu F$

## Introduction

$$\frac{t : F \mu F}{\text{in}_F t : \mu F}$$

• Example: if  $FX = \mathbb{1} + X$

then  $\mu F \cong \text{Nat}$

The rule says:

if  $t : \mathbb{1} + \text{Nat}$

then  $\text{in } t : \text{Nat}$

## Elements of $F \mu F = \mathbb{1} + \text{Nat}$

$$\mathbb{1} + \text{Nat}$$

$w$

$w$

$\text{inl } *$

$\text{inr } n$

↑  
only element  
of  $\mathbb{1}$

↑  
any element  
of  $\text{Nat}$

$\parallel$

$\parallel$

zero

succ n

• Example: if  $FX = \mathbb{1} + A \times X$   
then  $\mu F \cong \text{List}_A$

Elements of

$$F \mu F = \mathbb{1} + A \times \text{List}_A$$

$\text{inl } *$

$\text{inr } \langle a, \ell \rangle$

$\parallel$

$\parallel$

$\text{nil}$

$a :: \ell$

# Elimination

For every type  $X$

$$f: FX \rightarrow X$$

$$\text{cata } f: \mu F \rightarrow X$$

Reduction

$$\text{cata } f \text{ (in } t)$$

$$\rightsquigarrow f(F(\text{cata } f) t)$$

Explanation:

$f$  tells us how to compute on the constructors and the recursive calls  
cata  $f: \mu F \rightarrow X$  is called the catamorphism of  $f$

Since the functor  $F$  can be applied to functions

$$\text{cata } f: \mu F \rightarrow X \Rightarrow F(\text{cata } f):$$

$$F \mu F \rightarrow FX$$

So we can apply it to  $t: F \mu F$

$$F(\text{cata } f) t: FX$$

If we now apply  $f$  to it:

$$f(F(\text{cata } f) t): X$$

The elimination principle corresponds to iteration:  
we iterate  $f$  down the structure.

• Example:

$$\text{For } FX = \mathbb{1} + X$$

the elimination rule says

$$\frac{f : \mathbb{1} + X \longrightarrow X}{\text{cata } f : \text{Nat} \longrightarrow X}$$

equivalent to

$$\frac{x_0 : X \quad g : X \longrightarrow X}{\text{iterate } x_0 \ g : \text{Nat} \longrightarrow X}$$

$$\text{iterate } x_0 \ g \ n \rightsquigarrow^* \underbrace{g(g \dots (g x_0))}_{n \text{ times}}$$

$$x_0 = f(\text{inl } *)$$

$$g = \lambda x. f(\text{inr } x)$$

• Example

$$\text{For } FX = \mathbb{1} + A \times X$$

the elimination rule says

$$\frac{f : \mathbb{1} + A \times X \longrightarrow X}{\text{cata } f : \text{List}_A \longrightarrow X}$$

equivalent to

$$\frac{x_0 : X \quad g : A \longrightarrow X \longrightarrow X}{\text{iterate } x_0 \ g : \text{List}_A \longrightarrow X}$$

$$\text{(called foldr in Haskell)}$$