

Simplicity in Composition

Adelbert Chang

Scala World 2017



Composition

- A : type (set) of values



Composition

- A : type (set) of values
- $\oplus: A \times A \rightarrow A$



Composition

- A : type (set) of values
- $\oplus: A \times A \rightarrow A$
- 1_A : identity for \oplus



Composition

- A : type (set) of values
- $\oplus: A \times A \rightarrow A$
- 1_A : identity for \oplus

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$



Composition

- A : type (set) of values
- $\oplus: A \times A \rightarrow A$
- 1_A : identity for \oplus

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

$$x = x \oplus 1_A = 1_A \oplus x$$



$$(\mathbb{Z}, +, 0)$$



$([a], ++, [])$



$$(A \rightarrow A, \circ, a \mapsto a)$$



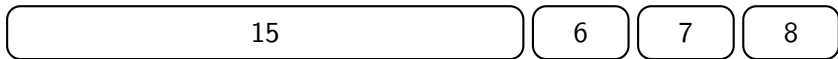
```
trait Monoid[A] {  
  def combine(x: A, y: A): A  
  def empty: A  
}
```











36







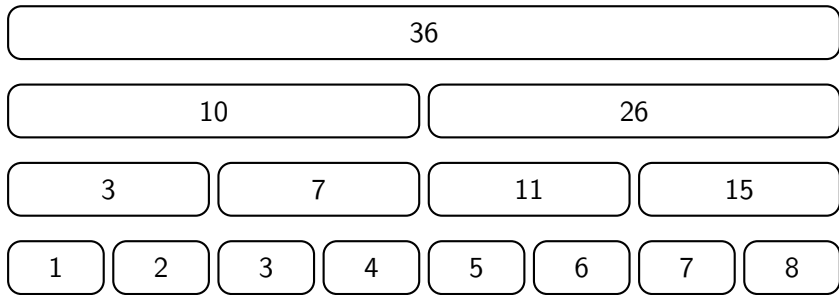
10

26



36





Associative composition allows for **modular**
decomposition and **reasoning**.



Composing programs

A



Composing programs

$F[A]$



Composing programs

$$(F[A], F[A]) \Rightarrow F[A]$$



Composing programs

$$(F[A], F[B]) \Rightarrow F[?]$$



Composing programs

$$(F[A], F[B]) \Rightarrow F[(A, B)]$$



Composing programs

```
def zipOption[A, B]  
  (oa: Option[A], ob: Option[B]): Option[(A, B)] =  
  
  (oa, ob) match {  
    case (Some(a), Some(b)) => Some((a, b))  
    case _                  => None  
  }
```



Composing programs

```
def zipList[A, B]  
  (la: List[A], lb: List[B]): List[(A, B)] =  
  
  la match {  
    case Nil      => Nil  
    case h :: t   => lb.map((h, _)) ++ zipList(t, lb)  
  }
```



Composing programs

```
def zipFunction[A, B, X]  
  (f: X => A, g: X => B): X => (A, B) =  
  
  (x: X) => (f(x), g(x))
```



Composing programs

- $F(-)$: type of program



Composing programs

- $F(-)$: type of program
- $\otimes : F(A) \times F(B) \rightarrow F((A, B))$



Composing programs

- $F(-)$: type of program
- $\otimes : F(A) \times F(B) \rightarrow F((A, B))$
- $\eta : A \rightarrow F(A)$



Composing programs

- $F(-)$: type of program
- $\otimes : F(A) \times F(B) \rightarrow F((A, B))$
- $\eta : A \rightarrow F(A)$

$$fa \otimes (fb \oplus fc) \cong (fa \oplus fb) \oplus fc$$



Composing programs

- $F(-)$: type of program
- $\otimes : F(A) \times F(B) \rightarrow F((A, B))$
- $\eta : A \rightarrow F(A)$

$$fa \otimes (fb \oplus fc) \cong (fa \oplus fb) \oplus fc$$

$$fa \cong fa \otimes \eta_{Unit} \cong \eta_{Unit} \otimes fa$$



```
trait Monoidal[F[_]] {  
  def zip[A, B](fa: F[A], fb: F[B]): F[(A, B)]  
  def pure[A](a: A): F[A]  
  
  /*  
  def map[A, B](fa: F[A])(f: A => B): F[B]  
  */  
}
```



Composing programs

$$(F[A], F[B]) \Rightarrow F[(A, B)]$$



Composing programs

$F[A]$



Composing dependent programs

$$(F[A], A \Rightarrow F[B]) \Rightarrow F[B]$$



Composing dependent programs

```
def flatMapOption[A, B]  
  (oa: Option[A], f: A => Option[B]): Option[B] =  
  
  oa match {  
    case Some(a) => f(a)  
    case None    => None  
  }
```



Composing dependent programs

```
def flatMapList[A, B]  
  (la: List[A], f: A => List[B]): List[B] =  
  
  la match {  
    case Nil => Nil  
    case h :: t => f(h) ++ flatMapList(t, f)  
  }
```



Composing dependent programs

```
def flatMapFunction[A, B, X]  
  (fa: X => A, f: A => (X => B)): X => B =  
  
  (x: X) => f(fa(x))(x)
```



Composing dependent programs

- $F(-)$: type of program



Composing dependent programs

- $F(-)$: type of program
- $\gg=$: $(F(A) \times A \rightarrow F(B)) \rightarrow F(B)$



Composing dependent programs

- $F(-)$: type of program
- $\gg= : (F(A) \times A \rightarrow F(B)) \rightarrow F(B)$
- $\eta : A \rightarrow F(A)$



Composing dependent programs

- $F(-)$: type of program
- $\gg=$: $(F(A) \times A \rightarrow F(B)) \rightarrow F(B)$
- $\eta : A \rightarrow F(A)$

$$(fa \gg= f) \gg= g = fa \gg= (f \gg= g)$$



Composing dependent programs

- $F(_)$: type of program
- $\gg=$: $(F(A) \times A \rightarrow F(B)) \rightarrow F(B)$
- $\eta : A \rightarrow F(A)$

$$\begin{aligned}(fa \gg= f) \gg= g &= fa \gg= (f \gg= g) \\ f(x) &= \eta(x) \gg= f\end{aligned}$$



Composing dependent programs

- $F(-)$: type of program
- $\gg=$: $(F(A) \times A \rightarrow F(B)) \rightarrow F(B)$
- $\eta : A \rightarrow F(A)$

$$(fa \gg= f) \gg= g = fa \gg= (f \gg= g)$$

$$f(x) = \eta(x) \gg= f$$

$$fa = fa \gg= \eta$$



```
trait Monad[F[_]] {  
  def flatMap[A, B](fa: F[A], f: A => F[B]): F[B]  
  def pure[A](a: A): F[A]  
}
```



A nicer monad

$$(fa \gg= f) \gg= g = fa \gg= (f \gg= g)$$

$$f(x) = \eta(x) \gg= f$$

$$fa = fa \gg= \eta$$



A nicer monad

$$f : A \rightarrow F(B)$$

$$g : B \rightarrow F(C)$$

$$h : C \rightarrow F(D)$$



A nicer monad

$$f : A \rightarrow F(B)$$

$$g : B \rightarrow F(C)$$

$$h : C \rightarrow F(D)$$

$$(f \gg g) \gg h = f \gg (g \gg h)$$



A nicer monad

$$f : A \rightarrow F(B)$$

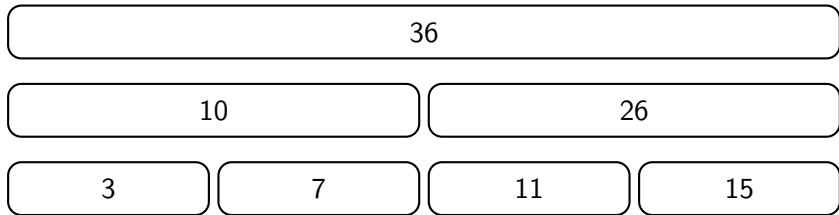
$$g : B \rightarrow F(C)$$

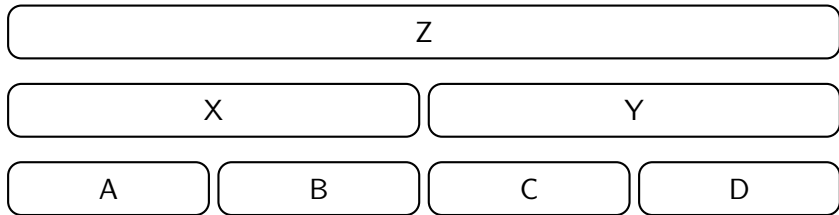
$$h : C \rightarrow F(D)$$

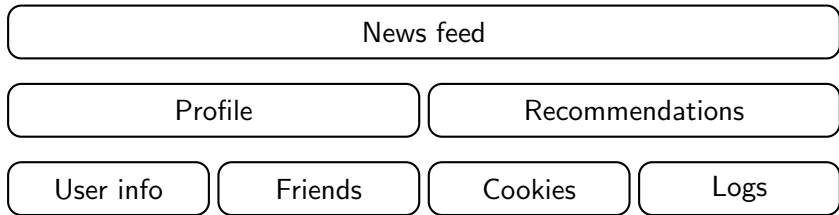
$$(f \rhd g) \rhd h = f \rhd (g \rhd h)$$

$$f = f \rhd \eta = \eta \rhd f$$





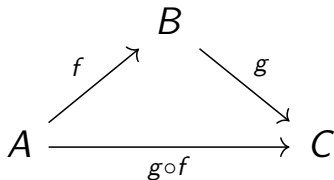




Category theory studies the algebra of composition.



Category theory



Category theory

- objects: $A, B, C \dots$



Category theory

- objects: $A, B, C \dots$
- arrows: $f : A \rightarrow B, g : B \rightarrow C \dots$



Category theory

- objects: $A, B, C \dots$
- arrows: $f : A \rightarrow B, g : B \rightarrow C \dots$
- $1_A : A \rightarrow A$



Category theory

- objects: $A, B, C \dots$
- arrows: $f : A \rightarrow B, g : B \rightarrow C \dots$
- $1_A : A \rightarrow A$

$$(f \circ g) \circ h = f \circ (g \circ h)$$



Category theory

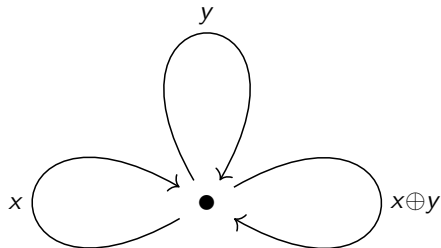
- objects: $A, B, C \dots$
- arrows: $f : A \rightarrow B, g : B \rightarrow C \dots$
- $1_A : A \rightarrow A$

$$(f \circ g) \circ h = f \circ (g \circ h)$$

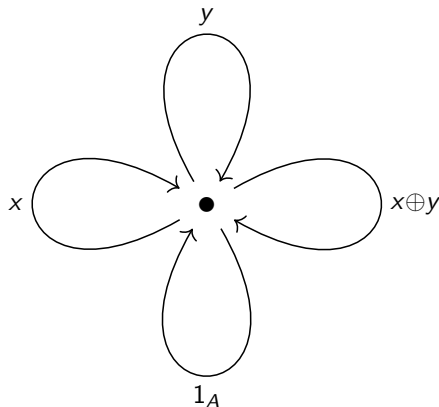
$$f = f \circ 1_A = 1_A \circ f$$



Category theory: monoids



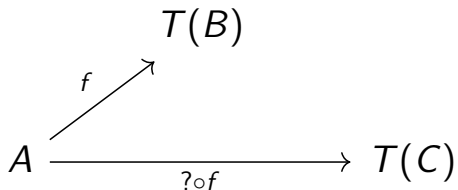
Category theory: monoids



Category theory: monads



Category theory: monads



Category theory: monads

$$\begin{array}{ccc} & & T(B) \\ & \nearrow f & \\ A & \xrightarrow{\quad ? \circ f \quad} & T(C) \end{array}$$

$$\gg =: (F(A) \times A \rightarrow F(B)) \rightarrow F(B)$$



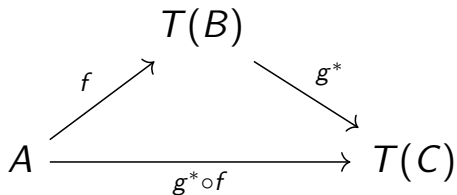
Category theory: monads

$$\begin{array}{ccc} & & T(B) \\ & \nearrow f & \\ A & \xrightarrow{\quad ? \circ f \quad} & T(C) \end{array}$$

$$>> =: (A \rightarrow F(B)) \rightarrow F(A) \rightarrow F(B)$$



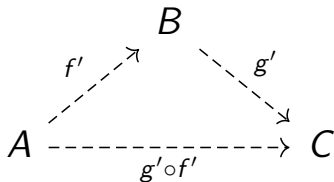
Category theory: monads



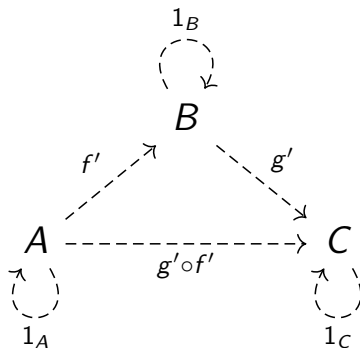
$$\gg =: (A \rightarrow F(B)) \rightarrow F(A) \rightarrow F(B)$$



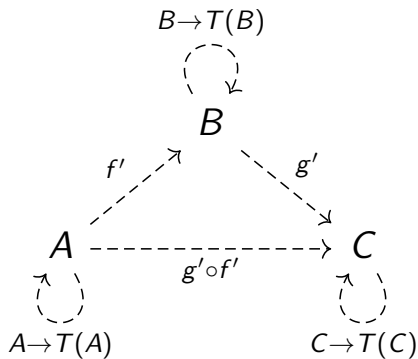
Category theory: monads



Category theory: monads



Category theory: monads



Can we compose the composers?



$$(A, A) \Rightarrow A$$



`(Monoid[A], Monoid[B]) => Monoid[?]`



`(Monoid[A], Monoid[B]) => Monoid[(A, B)]`



$$(F[A], F[B]) \Rightarrow F[(A, B)]$$



`(Monoidal[F], Monoidal[G]) => Monoidal[?]`



```
type L[X] = (F[X], G[X])  
(Monoidal[F], Monoidal[G]) => Monoidal[L]
```



```
type L[X] = (F[X], G[X])  
(Monad[F], Monad[G]) => Monad[L]
```




```
type L[X] = F[G[X]]  
(Monoidal[F], Monoidal[G]) => Monoidal[L]
```



```
type L[X] = F[G[X]]  
(Monad[F], Monad[G]) => Monad[L]
```



~~type L[X] = F[G[X]]
(Monad[F], Monad[G]) => Monad[L]~~

