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
- $\oplus$: $A \times A \to A$
- 1_A: identity for $\oplus$

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

Composition

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

$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$
$$x = x \oplus 1_A = 1_A \oplus x$$

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

$$(\texttt{[a]}, +\!\!+, \texttt{[]})$$

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

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

1	2	3	4	5	6	7	8

3	3	4	5	6	7	8


6
4
5
6
7
8

15	6	7	8

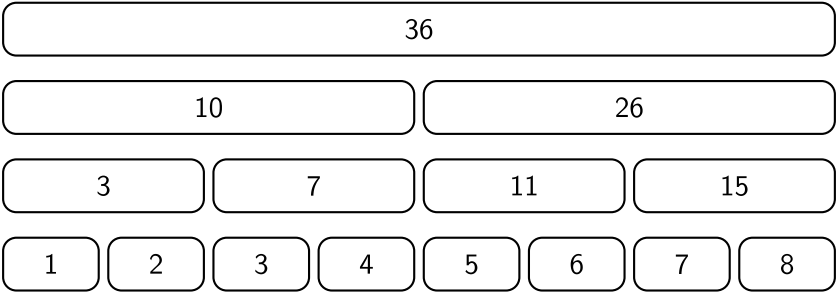
36

1	2	3	4	5	6	7	8

3	7	11	15

10	26

36


36
10
26
3
7
11
15
1
2
3
4
5
6
7
8

Associative composition allows for modular decomposition and reasoning.

Composing programs

`A`

Composing programs

`F[A]`

Composing programs

```
(F[A], F[A]) => F[A]
```

Composing programs

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

Composing programs

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

Composing programs

```scala
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

```scala
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, f)
  }
```

Composing programs

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

  (x: X) => (f(x), g(x))
```

```scala
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

- $F(\_)$: 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
- $\otimes : F(A) \times F(B) \to F((A, B))$
- $\eta : A \to F(A)$

Composing programs

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

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

Composing programs

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

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

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

Composing programs

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

Composing independent programs

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

Composing programs

`F[A]`

Composing dependent programs

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

Composing dependent programs

```scala
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

```scala
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)
```

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

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

Composing dependent programs

- $F(\_)$: type of program

[1] $>=>: (A \to F(B) \times B \to F(C)) \to A \to F(C)$

Composing dependent programs

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

[1] $>=>$: $(A \rightarrow F(B) \times B \rightarrow F(C)) \rightarrow A \rightarrow F(C)$

Composing dependent programs

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

[1] $>=>: (A \to F(B) \times B \to F(C)) \to A \to F(C)$

Composing dependent programs

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

$$(fa \gg\!= f) \gg\!= g \quad = \quad fa \gg\!= (f \gg\!=\!> g)^1$$

[1] $>\!=\!>: (A \to F(B) \times B \to F(C)) \to A \to F(C)$

Composing dependent programs

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

$$(fa \gg= f) \gg= g \quad = \quad fa \gg= (f >=> g)^1$$
$$f(x) \quad = \quad \eta(x) \gg= f$$

[1] $>=>: (A \to F(B) \times B \to F(C)) \to A \to F(C)$

Composing dependent programs

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

$$
\begin{aligned}
(fa \gg\!= f) \gg\!= g &= fa \gg\!= (f \gg\!=\!> g)^1 \\
f(x) &= \eta(x) \gg\!= f \\
fa &= fa \gg\!= \eta
\end{aligned}
$$

[1] $\gg\!=\!>: (A \to F(B) \times B \to F(C)) \to A \to F(C)$

A nicer monad

$$f : A \to F(B)$$
$$g : B \to F(C)$$
$$h : C \to F(D)$$

A nicer monad

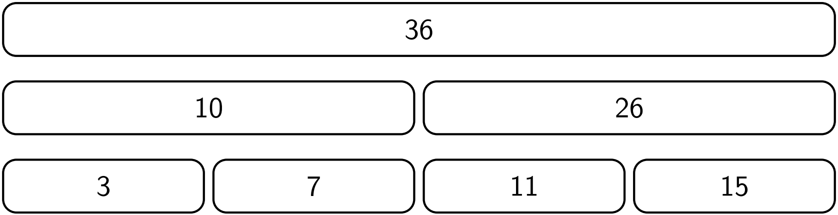
$$f : A \to F(B)$$
$$g : B \to F(C)$$
$$h : C \to F(D)$$

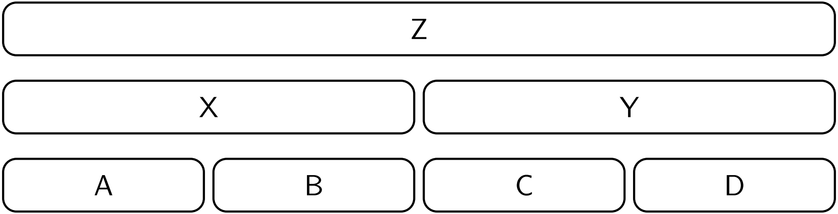
$$(f >=> g) >=> h \quad = \quad f >=> (g >=> h)$$

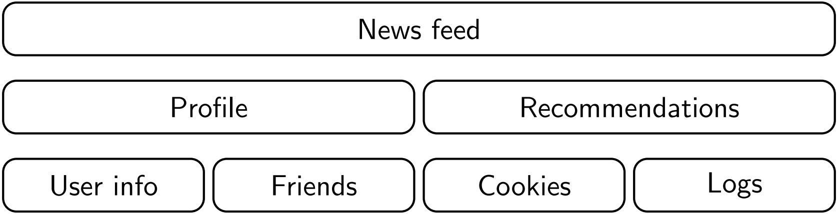
A nicer monad

$$f : A \to F(B)$$
$$g : B \to F(C)$$
$$h : C \to F(D)$$

$$(f >=> g) >=> h \quad = \quad f >=> (g >=> h)$$
$$f \quad = \quad f >=> \eta \quad = \quad \eta >=> f$$


36
10
26
3
7
11
15


Z
X
Y
A
B
C
D


News feed
Profile
Recommendations
User info
Friends
Cookies
Logs

Category theory studies the algebra of composition.

Category theory

- objects: A, B, C ...

Category theory

- objects: A, B, C ...
- arrows: $f : A \to B$, $g : B \to C$ ...

Category theory

- objects: A, B, C ...
- arrows: $f : A \to B$, $g : B \to C$ ...
- $1_A : A \to A$

Category theory

- objects: A, B, C ...
- arrows: $f : A \to B$, $g : B \to C$ ...
- $1_A : A \to A$

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

Category theory

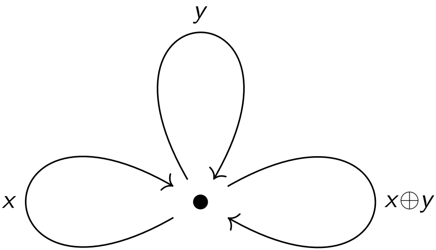
- objects: A, B, C ...
- arrows: $f : A \to B$, $g : B \to C$ ...
- $1_A : A \to A$

$$(f \circ g) \circ h = f \circ (g \circ h)$$
$$f = f \circ 1_A = 1_A \circ f$$

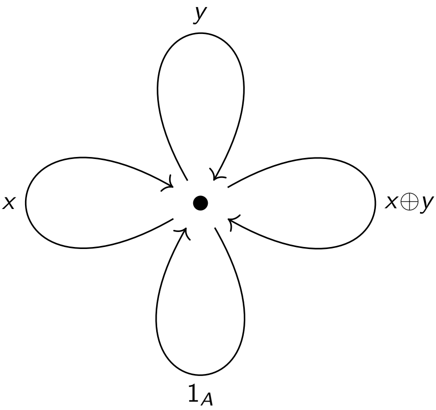
Category theory

$$\begin{tikzcd} & B \arrow[dr, "g"] & \\ A \arrow[ur, "f"] \arrow[rr, "g \circ f"'] & & C \end{tikzcd}$$

Category theory: monoids



Category theory: monoids



Category theory: monads

Category theory: monads

$$\begin{tikzcd} & F(B) & \\ A \arrow[ur, "f"] \arrow[rr, "? \circ f"'] & & F(C) \end{tikzcd}$$

Category theory: monads

$$\begin{tikzcd} & F(B) \\ A \arrow[ur, "f"] \arrow[rr, "? \circ f"'] & & F(C) \end{tikzcd}$$

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

Category theory: monads

$$\begin{tikzcd} & F(B) & \\ A \arrow[ur, "f"] \arrow[rr, "? \circ f"'] & & F(C) \end{tikzcd}$$

$$\gg\!=: (A \to F(B)) \to F(A) \to F(B)$$

Category theory: monads

$$
\begin{tikzcd}
 & F(B) \arrow[dr, "g^*"] & \\
A \arrow[ur, "f"] \arrow[rr, "g^* \circ f"'] & & F(C)
\end{tikzcd}
$$

$$\gg\!=: (A \to F(B)) \to F(A) \to F(B)$$

Category theory: monads

$$\begin{tikzcd} & B \arrow[dr, dashed, "g'"] & \\ A \arrow[ur, dashed, "f'"] \arrow[rr, dashed, "g' \circ f'"'] & & C \end{tikzcd}$$

Category theory: monads

$$
\begin{tikzcd}
 & B \arrow[dr, dashed, "g'"] \arrow[loop above, dashed, "1_B"] & \\
A \arrow[ur, dashed, "f'"] \arrow[rr, dashed, "g' \circ f'"'] \arrow[loop below, dashed, "1_A"] & & C \arrow[loop below, dashed, "1_C"]
\end{tikzcd}
$$

Category theory: monads

$$
\begin{tikzcd}
 & B \arrow[loop above, dashed, "B \to T(B)"] \arrow[dr, dashed, "g'"] & \\
A \arrow[ur, dashed, "f'"] \arrow[rr, dashed, "g' \circ f'"'] \arrow[loop below, dashed, "A \to T(A)"] & & C \arrow[loop below, dashed, "C \to T(C)"]
\end{tikzcd}
$$

Category theory: monads

$$
\begin{tikzcd}
& B \arrow[dr, dashed, "g'"] \arrow[loop above, dashed, "B\to T(B)"] & \\
A \arrow[ur, dashed, "f'"] \arrow[rr, dashed, "g'\circ f'"'] \arrow[loop below, dashed, "A\to T(A)"] & & C \arrow[loop below, dashed, "C\to T(C)"]
\end{tikzcd}
$$

$$\eta : A \to F(A)$$

Can we compose the composers?

```
(A, A) => A
```

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

Product composition: monoids

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

Product composition: monoids

```scala
def append(x: (A, B), y: (A, B)): (A, B) = {
  val a = x._1 append y._1
  val b = x._2 append y._2

  (a, b)
}
```

[1] Assuming `Monoid[A]` and `Monoid[B]`

Product composition: monoids

```scala
def append(x: (A, B), y: (A, B)): (A, B) = {
  val a = x._1 append y._1
  val b = x._2 append y._2

  (a, b)
}

def empty: (A, B) = (Monoid[A].empty, Monoid[B].empty)
```

[1] Assuming `Monoid[A]` and `Monoid[B]`

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

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

Product composition: lax monoidal functors

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

Product composition: lax monoidal functors

```scala
def zip[A, B](x: (F[A], G[A]),
              y: (F[B], G[B])): (F[(A, B)], G[(A, B)]) = {

  val f = x._1 zip y._1
  val g = x._2 zip y._2

  (f, g)
}
```

[1] Assuming `Monoidal[F]` and `Monoidal[G]`

Product composition: lax monoidal functors

```
def zip[A, B](x: (F[A], G[A]),
              y: (F[B], G[B])): (F[(A, B)], G[(A, B)]) = {

  val f = x._1 zip y._1
  val g = x._2 zip y._2

  (f, g)
}

def pure[A](a: A): (F[A], G[A]) =
  (Monoidal[F].pure(a), Monoidal[G].pure(a))
```

[1] Assuming `Monoidal[F]` and `Monoidal[G]`

Product composition: monads

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

Product composition: monads

```
def flatMap[A, B]
  (xa: (F[A], G[A]))(ff: A => (F[B], G[B])):
    (F[(A, B)], G[(A, B)]) = {

  val f = xa._1.flatMap(a => ff(a)._1)
  val g = xa._2.flatMap(a => ff(a)._2)

  (f, g)
}
```

[1] Assuming `Monad[F]` and `Monad[G]`

Nested composition: lax monoidal functors

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

Nested composition: lax monoidal functors

```scala
def zip[A, B](fga: F[G[A]], fgb: F[G[B]]): F[G[(A, B)]] = {
  val fp: F[(G[A], G[B])] = fga zip fgb
  fp.map { case (ga, gb) => ga zip gb }
}
```

[1] Assuming `Monoidal[F]` and `Monoidal[G]`

Nested composition: lax monoidal functors

```scala
def zip[A, B](fga: F[G[A]], fgb: F[G[B]]): F[G[(A, B)]] = {
  val fp: F[(G[A], G[B])] = fga zip fgb
  fp.map { case (ga, gb) => ga zip gb }
}

def pure[A](a: A): F[G[A]] =
  Monoidal[F].pure(Monoidal[G].pure(a))
```

[1] Assuming `Monoidal[F]` and `Monoidal[G]`

Nested composition: monads

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

Nested composition: monads

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

Nested composition: monads

```
def flatMap[A, B](fga: F[G[A]](f: A => F[G[B]]): F[G[B]] =
```

[1] Assuming Monad[F] and Monad[G]

Nested composition: monads

```scala
def flatMap[A, B](fga: F[G[A]](f: A => F[G[B]]): F[G[B]] =
  fga.flatMap { (ga: G[A]) =>
    ???
  }
```

[1] Assuming `Monad[F]` and `Monad[G]`

Review

- Associative composition allows for modular decomposition and reasoning

Review

- Associative composition allows for modular decomposition and reasoning
- Monoids compose A's, lax monoidal functors compose independent F[A]'s, monads compose dependent F[A]'s

Review

- Associative composition allows for modular decomposition and reasoning
- Monoids compose A's, lax monoidal functors compose independent F[A]'s, monads compose dependent F[A]'s
- Monoids can be composed

Review

- Associative composition allows for modular decomposition and reasoning
- Monoids compose A's, lax monoidal functors compose independent F[A]'s, monads compose dependent F[A]'s
- Monoids can be composed
- Lax monoidal functors can be composed

Review

- Associative composition allows for modular decomposition and reasoning
- Monoids compose A's, lax monoidal functors compose independent F[A]'s, monads compose dependent F[A]'s
- Monoids can be composed
- Lax monoidal functors can be composed
- Monads in general cannot be composed

References

- Equational reasoning at scale - Gabriel Gonzalez http://www.haskellforall.com/2014/07/equational-reasoning-at-scale.html
- Invariant Shadows - Michael Pilquist http://mpilquist.github.io/blog/2015/06/22/invariant-shadows-part-2/
- All About Applicative - Me! https://www.youtube.com/watch?v=Mn7BtPALFys
- A Categorial View of Computational Effects - Prof. Emily Riehl https://www.youtube.com/watch?v=6t6bsWVOIzs
- Conceptual Mathematics - F. William Lawvere and Stephen H. Schanuel