Simplicity in Composition

Adelbert Chang

Scala World 2017

Composition

- A: type (set) of values

Composition

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

Composition

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

Composition

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

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

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

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

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

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

```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

$\boxed{36}$

1 2 3 4 5 6 7 8

3 7 11 15

10	26

36

36

10 26

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

```
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))
```

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

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

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

```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[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

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

Composing dependent programs

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

Composing dependent programs

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

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

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

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

```scala
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

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

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

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

A nicer monad

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

A nicer monad

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

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

Category theory studies the algebra of composition.