Scala type classes (Cats)

Ramy Tanios

December 18, 2022

1 Cats type classes

Sample implementations are presented below for the most important type classes supported by scala and Cats library.

1.1 Semigroup

A **Semigroup** is a type class that provides the `combine` method. The following is the trait implementation of a **Semigroup**:

```
trait Semigroup[F]{
  def combine(l: F, r: F): F
}
```

The `combine` method satisfies the associativity propery $f(f(x,y),z) = f(x,f(y,z))$.

1.2 Monoid

A **Monoid** is a type class that **extends** a **Semigroup**, providing in addition to the `map` method, a `pure` or `unit` method. The following is a trait implementation of a **Monoid**:

```
trait Monoid[F]{
  def unit: F
  def combine(l: F, r: F): F
}
```

The former methods should follow the follow properties:

- $f(f(x,y),z) = f(x,f(y,z))$ (associativity).
- $f(x,0) = f(0,x) = x$.

1.3 Functor

A **Functor** is a type class that provides the `map` method. The `map` serves as an **ETW** pattern: Extract, Transform and Wrap. The following is a trait implementation of a **Functor**:

```
trait Functor[F[_]]{
  def map[A, B](ma: F[A])(f: A => B): F[B]
}
```

The `map` method should have the following properties:

- $map(x)(identity) = x$

1.4 Monad

A **Monad** is a type class that provides the `pure`, `map` and `flatMap` methods. A monad **extends** a **Functor**. The following is a trait implementation of a **Monad**:

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

```
  def flatMap[A, B](ma: F[A])(f: A => F[B]): F[B]
}
```

1.5 Semigroupal

A **Semigroupal** is a type class that provides the `product` method. The following is a trait implementation of a **Semigroupal**:

```
trait Semigroupal[F[_]]{
  def product[A, B](ma: F[A], mb: F[B]): F[(A,B)]
}
```

Note that a **Monad** extends a **Semigroupal** as the `product` can be implemented using the `map` and `flatMap` methods as follows:

```
def product[A, B](ma: F[A], mb: F[B]): F[(A,B)] =
  ma.flatMap(a => mb.map(b => (a, b)))
}

def product[A, B](ma: F[A], mb: F[B]): F[(A,B)] = // for comprehension
  for {
    a <- ma
    b <- mb
  } yield (a,b)
```

1.6 Applicative

An **Applicative** is a type class that implements the `pure` and `map` methods. Note that an **Applicative** extends a **Functor**.

```
trait Applicative[F[_]]{
  def pure[A](a: A): F[A]
  def map[A, B](ma: F[A], mb: F[B])(f: A => B): F[B]
  def ap[A, B](ma: F[A])(f: F[A => B]): F[B] // provided by scala
}
```

Note that an **Applicative** extends a **Semigroupal** as the former can implement the `product` through the `ap` method as follows:

```
def product[A, B](ma: F[A], mb: F[B]): F[(A,B)] =
  ap(ma)(mb.map(b => ((a: A) => b)))
```