Categories in the wild

Wouter Stekelenburg

January 26, 2018

Naked interface

```
trait Cassandra {
  def prepare(statement: String): PreparedStatement
  def prepare(statement: RegularStatement):
      PreparedStatement
  def execute(statement: String): Unit
  def execute(statement: Statement): Unit
}
```

Wrapped interface

```
trait Cassandra {
  def prepare(statement: String): F[PreparedStatement]
  def prepare(statement: RegularStatement):
      F[PreparedStatement]
  def execute(statement: String): F[Unit]
  def execute(statement: Statement): F[Unit]
}
```

Map

```
map: (X => Y) => F[X] => F[Y]

// map((x: X) => x)(y) === y
// map(f)(map(g)(y)) === map((x: X) => f(g(x)))(y)
```

Functor

Zip

```
zip: (F[X],F[Y]) => F[(X,Y)]

// map(_._1)(zip(fx, fy)) === fx
// map(_._2)(zip(fx, fy)) === fy
// zip(map(_._1)(z),map(_._2)(z)) === z
```

Unit

```
unit: X => F[X]

// map(f)(unit(x)) === unit(f(x))
```

Traverse

```
traverse: (X => F[X]) => List[X] => F[List[X]]

// traverse(f)(Nil) === unit(Nil)
// traverse(f)(h :: t) === map{
//   case (x,y) => x :: y
// }(zip(f(h),traverse(f)(t)))
```

Applicative functor

```
map: (X => Y) => F[X] => F[Y]
zip: (F[X],F[Y]) => F[(X,Y)]
unit: X => F[X]
traverse: (X => F[X]) => List[X] => F[List[X]]
```

Bind

```
bind: (X => F[Y]) => F[X] => F[Y]

// bind(f)(bind(g)(x)) === bind((y: X) =>
    bind(f)(g(y))(x))
// bind(unit)(x) === x
// bind(f)(unit(x)) === unit(f(x))

// map(f)(x) === bind((y: X) => unit(f(y)))(x)
// zip(x,y) === bind((x0: X) => map((y0: Y) => (x0, y0)))
```

ReduceM

```
reduceM: F[X] => (X => X => F[X]) => List[X] => F[X]

// reduceM(a)(b)(Nil) === unit(Nil)
// reduceM(a)(b)(h :: t) === bind(b(h))(reduceM(a)(b)(t))
```

Monad

```
unit: X => F[X]
bind: (X => F[Y]) => F[X] => F[Y]
reduceM: ((X,X) => F[X]) => List[X] => F[X]
```

Motivation

- used in calendars api, savings me api
- Scala standard library

Asynchronous interface

```
abstract class Cassandra(implicit ec:ExecutionContext) {
  def prepare(statement: String):
      Future[PreparedStatement]
  def prepare(statement: RegularStatement):
      Future[PreparedStatement]
  def execute(statement: String): Future[Unit]
  def execute(statement: BoundStatement): Future[Unit]
}
```

- `Future.successful` and `Future.apply` for `unit`
- `flatMap` for `bind`
- `Future.traverse` exists

Motivation

Continuation passing style

- abstracts over the notion of registering callbacks
- can mimic any monad
- reasonable solution in Java

Continuation passing style monad

```
object CPSMonad {

  type CPS[X] = (X => Unit) => Unit

  def unit[X](x: => X): CPS[X] = ((c: X) => Unit) => c(x)

  def bind[X, Y](f: X => CPS[Y])(cx: CPS[X]): CPS[Y] =
      ((cy: Y) => Unit) => cx((x: X) => f(x)(cy))

  //...
}
```

Stacksafe traverse

```
def traverse[X, Y](f: X => CPS[Y])(lx: List[X]):
    CPS[List[Y]] = {
  @tailrec def helper(lx: List[X], cy: List[Y] =>
      Unit): Unit = lx match {
    case Nil => cy(Nil)
    case hx :: tx => helper(tx, (ty: List[Y]) =>
        f(hx)((hy: Y) => cy(hy :: ty)))
  }
  helper(lx,_)
}
```

Motivation

- Reify everything
- Build and manipulate ASTs for later interpretation

Free monad constructors

```
sealed trait Free[F[_], X]

case class Return[F[_], X](x: X) extends Free[F, X]

case class Effect[F[_], X](fx: F[X]) extends Free[F, X]

case class Bind[F[_], X](f: X => Free[F, X], frx: Free[F,
    X]) extends Free[F, X]

case class Traverse[F[_], X](f: X => Free[F, X], lx:
    List[X]) extends Free[F, List[X]]
```

Effect constructors

```
sealed trait CassandraEffect[X]

case class PrepareString(statement: String) extends
    CassandraEffect[PreparedStatement]

case class PrepareRegular(statement: RegularStatement)
    extends CassandraEffect[PreparedStatement]

case class ExecuteString(statement: String) extends
    CassandraEffect[ResultSet]

case class ExecuteStatement(statement: Statement) extends
    CassandraEffect[ResultSet]
```

Motivation

- abstract over monads themselves
- dependency injection applied to monads

Monad type class

```
trait Monad[M[_]] {

  def unit[X](x: X): M[X]

  def bind[X, Y](f: X => M[Y])(mx: M[X]): M[Y]

  def traverse[X, Y](f: X => M[Y])(lx: => List[X]):
      M[List[X]]
}
```

Generically wrapped interface

```
abstract class Cassandra[F: Monad] {
  def prepare(statement: String): F[PreparedStatement]
  def prepare(statement: RegularStatement):
      F[PreparedStatement]
  def execute(statement: String): F[Unit]
  def execute(statement: Statement): F[Unit]
}
```

Summary

- ‘functor’, ‘applicative functor’ and ‘monad’ specify interfaces that handle callbacks, in increasing power
- ‘Future’, ‘continuation passing style’, ‘free monads’ and ‘generic monads’ show a range of technical implementations that can hide behind those interfaces