Monads

Peter Potts

24th November 2011

http://peterpotts.com/pjp.zip

Informal Definition of Monad

A monadic type is an embellished result type that encapsulates cross cutting program logic such as side-effects without breaking the functional model.

$$A \xrightarrow{f} B \xrightarrow{g} C$$

$$M[A] \Rightarrow M[B] \Rightarrow M[C]$$

Built-in Java Monads

- No object handling:

  ```
  public B f(A a) { ... }
  ```

  M[B] = B or null
- Exception handling:

  ```
  public B f(A a) throws E { ... }
  ```

  M[B] = B or E
- Standard output handling:

  ```
  public B f(A a) { ... }
  ```

  M[B] = B and Screen

Formal Definition of Monad

- Type constructor M:
  For every underlying type A,
  M[A] is the corresponding monadic type.
- Unit function:
  The **simplest reversibly** map
  from a value of type A,
  to a value of type M[A].
- Binding operation:
  An infix operator (>>=) between
  a value of type M[A] and
  a function of type A → M[B]
  to give a result of type M[B].

Axioms of a Monad

unit : A => M[A]

>>= : M[A] => (A => M[B]) => M[B]

Simplest: unit(x) >>= f $\equiv$ f(x)

Reversible: m >>= unit $\equiv$ m

Associative: (m >>= f) >>= g $\equiv$ m >>= (x => (f(x) >>= g))

No Object Handling

- M[A] = Option[A]
- Option is Some(x) or None
- unit(x) = Some(x)
- bind(m, f) = m flatMap f

```
≡ m match {
    case None => None
    case Some(x) => f(x)
}
```

Exception Handling

- M[A] = Either[Throwable, A]
- Either is Left(exception) or Right(x)
- unit(x) = Right(x)
- bind(m, f) = m fold (Left(_), f(_))

```
≡ m match {
    case Left(e) => Left(e)
    case Right(x) => f(x)
}
```

Standard Output Handling

- M[A] = (String, A)
- Tuple is (output, x)
- unit(x) = (“”, x)
- bind(m, f) = {

```
    val (screen, x) = monad
    val (output, y) = f(x)
    (screen + output, y)
}
```

Monad Generalizations

• Identity	M[A] = A
• Option	M[A] = Option[A]
• Collection	M[A] = List[A]
• State	M[A] = S => (A, S)
• Reader	M[A] = C => A
• Writer	M[A] = (D, A) for Monoid D
• Continuation	M[A] = (A => R) => R

Scala Monad

```
trait Monad {
    type M[A]

    def unit[A](value: A): M[A]

    def bind[A, B](monad: M[A], function: A => M[B]): M[B]

    class Features[A](val monad: M[A]) {
        ...
    }

    implicit def features[A](monad: M[A]) = new Features(monad)
}
```

Infix Pipeline

m >>= f bind(m, f)

```
def unit[A](x: A) = Some(x)

implicit def bind[A](m: Option[A]) = new {
    def >>=[B](f: A => Option[B]) = m flatMap f
}

def halve(x: Int) = if (x % 2 == 0) Some(x / 2) else None

unit(12) >>= halve >>= halve → Some(3)
unit(5) >>= halve >>= halve → None
```

Bind Alternatives

bind = >>= = flatMap

unit(12) >>= halve >>= halve → Some(3)

unit(5) >>= halve >>= halve → None

Some(12) flatMap halve flatMap halve → Some(3)

Some(5) flatMap halve flatMap halve → None

For Comprehension

for (a <- p; b <- q; c <- r) yield s

p.flatMap(a => q.flatMap(b => r.map(z => s)))

for (a <- p; b <- q(a); c <- r(b)) yield c

p flatMap q flatMap r

p >>= q >>= r

Monadic Features

```
class Features[A](val m: M[A]) {
  // Infix Pipeline
  def >>=[B](f: A => M[B]) = bind(m, f)

  // For Comprehension
  def flatMap[B](f: A => M[B]) = bind(m, f)
  def map[B](f: A => B) = bind(m, (x: A) => unit(f(x)))
}
```

Continuation Monad

- class ContinuationMonad[R] extends Monad
- M[A] = (A => R) => R
- unit(x) = p => p(x)
- bind(m, f) = p => m(x => f(x)(p))

Lazy Evaluation Example

```
def evaluate[A](precision: Int)(stream: Stream[A]) = stream.take(precision).mkString(",")

val monad = new ContinuationMonad[String]

import monad._

def from(n: Int): Stream[Int] = Stream.cons(n, from(n + 1))

def sieve(s: Stream[Int]): Stream[Int] = Stream.cons(
      s.head,
      sieve(s.tail filter { _ % s.head != 0 }))

def except(n: Int)(s: Stream[Int]) = s filter { _ != n }

def example1 = unit(Stream.continually(1))(evaluate(3)) mustEqual "1,1,1"

def example2 = (from(2) >>= sieve >>= except(7))(evaluate(4)) mustEqual "2,3,5,11"
```