SCALA MONADS
DECLUTTER YOUR CODE WITH MONADIC DESIGN

Dan Rosen, Marakana, Inc.
San Francisco, CA - February 09, 2012

Monads in Perl
本間 雅洋
hiratara.chira.tara@gmail.com

Yet another presentation on MONADS

Marcin Gryszko
@mgryszko
monads are burritos?

and functors?

(a → b) → 

1. remove astronaut from suit
2. put naked astronaut in station
3. send out whatever the station sends out (well... almost)
trapd in Monad tutorial
plz help

MONADS
MONADS EVERYWHERE
BRACE YOURSELVES

trapd in IO monad
plz help

THE MONAD TUTORIALS ARE COMING
Amount of known monad tutorials

1990s 2000s 2010s
Church and State 1:
Evaluating monads

Philip Wadler

University of Glasgow
Bell Labs, Lucent Technologies
Pure vs. impure

<table>
<thead>
<tr>
<th>Modification</th>
<th>Impure language (Standard ML, Scheme)</th>
<th>Pure language (Miranda*, Haskell†)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error messages</td>
<td>use exceptions</td>
<td>rewrite</td>
</tr>
<tr>
<td>Execution counts</td>
<td>use state</td>
<td>rewrite</td>
</tr>
<tr>
<td>Output</td>
<td>use output</td>
<td>rewrite</td>
</tr>
<tr>
<td>Backwards output</td>
<td>rewrite</td>
<td>modify output</td>
</tr>
</tbody>
</table>

*Miranda is a trademark of Research Software Limited.
†Haskell is not a trademark.
Variations on an evaluator
monad n. 1. Philosophy a. any fundamental entity, esp. if autonomous.

– Collins English Dictionary
Variation zero: The basic evaluator

data  Term  =  Con Int | Div Term Term

eval  ::  Term \rightarrow Int

eval (Con a)  =  a

eval (Div t u)  =  eval t \div eval u
The Term type

trait Term

case class Con(a: Int) extends Term

case class Div(t: Term, u: Term) extends Term
Test data

```scala
val answer: Term =
  Div(
    Div(
      Con(1932),
      Con(2)
    ),
    Con(23)
  )

val error: Term =
  Div(
    Con(1),
    Con(0)
  )
```
Variation zero: The basic evaluator

def eval(t: Term): Int =
    t match {
        case Con(a) => a
        case Div(t, u) => eval(t) / eval(u)
    }
Using the evaluator

scala> val a = eval(answer)
a: Int = 42

scala> val b = eval(error)
java.lang.ArithmeticException: / by zero
   at monads.Eval0$.eval(Term.scala:29)
Variation one: Exceptions

type Exception = String
trait M[A]
case class Raise[A](e: Exception) extends M[A]
case class Return[A](a: A) extends M[A]
Variation one: Exceptions

def eval[A](s: Term): M[Int] =
    s match {
        case Con(a)    => Return(a)
        case Div(t, u) =>
            eval(t) match {
                case Raise(e)   => Raise(e)
                case Return(a) =>
                    eval(u) match {
                        case Raise(e) => Raise(e)
                        case Return(b) =>
                            if (b == 0) Raise("divide by zero")
                            else Return(a / b)
                    }
            }
    }

Variation one: Exceptions

```
scala> val m = eval(answer)
   m: M[Int] = Return(42)

scala> val n = eval(error)
n: M[Int] = Raise(divide by zero)
```
Variation two: State

type State = Int
type M[A] = State => (A, State)

def eval(s: Term): M[Int] =
    s match {
        case Con(a) =>
            x => (a, x)
        case Div(t, u) =>
            x =>
                val (a, y) = eval(t)(x)
                val (b, z) = eval(u)(y)
                (a / b, z + 1)
Variation two: State

scala> val m = eval(answer)(0)
m: (Int, State) = (42,2)
Variation three: Output

type Output = String

type M[A] = (Output, A)

def eval(s: Term): M[Int] =
  s match {
    case Con(a) => (line(s, a), a)
    case Div(t, u) =>
      val (x, a) = eval(t)
      val (y, b) = eval(u)
      (x + y + line(s, a / b), a / b)
  }

def line(t: Term, a: Int): Output =
  t + "=" + a + "\n"
Variation three: Output

```scala
scala> val m = eval(answer)
m: (Output, Int) =
("Con(1932)=1932
Con(2)=2
Div(Con(1932),Con(2))=966
Con(23)=23
Div(Div(Con(1932),Con(2)),Con(23))=42
",42)
```
Monads
monad n. 1. b. (in the metaphysics of Leibnitz) a simple indestructible nonspatial element regarded as the unit of which reality consists.

— Collins English Dictionary
What is a monad?

1. For each type \( A \) of values, a type \( M[A] \) to represent computations.

   In general, \( A \Rightarrow B \) becomes \( A \Rightarrow M[B] \)

   In particular, \( \text{def eval(t: Term): Int} \)

   becomes \( \text{def eval(t: Term): M[Int]} \)

2. A way to turn values into computations.

   \( \text{def pure[A](a: A): M[A]} \)

3. A way to combine computations.

   \( \text{def bind[A, B](m: M[A], k: A \Rightarrow M[B]): M[B]} \)
Monad laws

Left unit

\[ \text{bind}(\text{pure}(a), k) \equiv k(a) \]

Right unit

\[ \text{bind}(m, (a: A) \Rightarrow \text{pure}(a)) \equiv m \]

Associative

\[
\text{bind}(\text{bind}(m, (a: A) \Rightarrow k(a)), (b: B) \Rightarrow h(b)) \\
\equiv \\
\text{bind}(m, (a: A) \Rightarrow \text{bind}(k(a), (b: B) \Rightarrow h(b)))
\]
The evaluator revisited
monad  n. 1. c. (in the pantheistic philosophy of Giordano Bruno) a fundamental metaphysical unit that is spatially extended and psychically aware.

— Collins English Dictionary
Variation zero, revisited: Identity

type M[A] = A

def pure[A](a: A): M[A] = a

def bind[A, B](a: M[A], k: A => M[B]): M[B] = k(a)
def eval(s: Term): M[Int] =
    s match {
        case Con(a) =>
            pure(a)
        case Div(t, u) =>
            bind(eval(t), (a: Int) =>
                bind(eval(u), (b: Int) =>
                    pure(a / b)))
    }
Variation one, revisited: Exceptions

type Exception = String

trait M[A]
case class Raise[A](e: Exception) extends M[A]
case class Return[A](a: A) extends M[A]

def pure[A](a: A): M[A] = Return(a)

def bind[A, B](m: M[A], k: A => M[B]): M[B] =
  m match {
    case Raise(e) => Raise(e)
    case Return(a) => k(a)
  }

def raise[A](e: String): M[A] = Raise(e)
Variation one, revisited: Exceptions

```python
def eval(s: Term): M[Int] =
    s match {
        case Con(a) =>
            pure(a)
        case Div(t, u) =>
            bind(eval(t), (a: Int) =>
                bind(eval(u), (b: Int) =>
                    if (b == 0)
                        raise("divide by zero")
                    else
                        pure(a / b)
                ))
    }
```
Variation two, revisited: State

```scala
type State = Int
type M[A] = State => (A, State)

def pure[A](a: A): M[A] = x => (a, x)

def bind[A, B](m: M[A], k: A => M[B]): M[B] =
  x => {
    val (a, y) = m(x)
    val (b, z) = k(a)(y)
    (b, z)
  }

def tick: M[Unit] = (x: Int) => (() , x + 1)
```
Variation two, revisited: State

def eval(s: Term): M[Int] =
    s match {
      case Con(a) =>
        pure(a)
      case Div(t, u) =>
        bind(eval(t), (a: Int) =>
          bind(eval(u), (b: Int) =>
            bind(tick, (_: Unit) =>
              pure(a / b))))
    }
Variation three, revisited: Output

type Output = String
type M[A] = (Output, A)

def pure[A](a: A): M[A] = ("", a)

def bind[A, B](m: M[A], k: A => M[B]): M[B] = {
    val (x, a) = m
    val (y, b) = k(a)
    (x + y, b)
}

def output[A](s: String): M[Unit] = (s, ())
def eval(s: Term): M[Int] =
    s match {
        case Con(a) =>
            bind(output(line(s, a)), (_, Unit) =>
                pure(a))
        case Div(t, u) =>
            bind(eval(t), (a: Int) =>
                bind(eval(u), (b: Int) =>
                    bind(output(line(s, a / b)), (_, Unit) =>
                        pure(a / b))))
    }

def line(t: Term, a: Int): Output =
    t + "=" + a + "\n"
More monads: Lists and Streams
**Lists**

```scala
type M[A] = List[A]

def pure[A](a: A): M[A] = List(a)

def bind[A, B](m: M[A], k: A => M[B]): M[B] = 
  m match {
    case Nil    => Nil
    case h :: t => k(h) ++ bind(t, k)
  }
```
**Streams**

\[
\text{type } M[A] = \text{Stream}[A]
\]

\[
def \text{pure}[A](a: A): M[A] = \text{Stream}(a)
\]

\[
def \text{bind}[A, B](m: M[A], k: A \Rightarrow M[B]): M[B] = \\
m \text{match } \{ \\
  \text{case Stream}() \Rightarrow \text{Stream}() \\
  \text{case } h \ #:: \ t \Rightarrow k(h) ++ \text{bind}(t, k) \\
\}
\]
Cartesian products

def product[A, B](m: M[A], n: M[B]): M[(A, B)] =
    bind(m, (a: A) =>
        bind(n, (b: B) =>
            pure((a, b))))

// using List or Stream's for comprehension syntax
def productFor[A, B](m: M[A], n: M[B]): M[(A, B)] =
for {
    a <- m
    b <- n
} yield (a, b)
Conclusions
monad  

2. a single-celled organism, especially a flagellate protozoan

— Collins English Dictionary
The Glasgow Haskell compiler

Joint work with

Cordy Hall, Kevin Hammond,
Will Partain, Simon Peyton Jones.

Glasgow Haskell compiler is written in Haskell.

Each phase uses a monad.

Has proved easy to modify in practice.
Monads in the Glasgow Haskell compiler

Type inference phase.
- Exceptions for errors,
- state for current substitution,
- state for fresh variable names,
- read-only state for current location.

Simplification phase.
- State for fresh variable names.

Code generator phase.
- Output for code generated so far,
- state for table mapping variables to addressing modes,
- state for table to cache known state of stack.
Origins


- values (*int*) vs. computations (*T int*)
- call-by-value (*int → T int*)
- call-by-name (*T int → T int*)


- data types (*int*) vs. phrase types (*int exp*)
- call-by-value (*int → int exp*)
- call-by-name (*int exp → int exp*)

But Reynolds missed *unit* and *⊤*. 
monadism or monadology n. (esp. in writings of Leibnitz) the philosophical doctrine that monads are the ultimate units of reality.

– Collins English Dictionary
Monads

(1) \( \text{return } v \ggg \lambda x. k x = k v \)
(2) \( m \ggg \lambda x. \text{return } x = m \)
(3) \( m \ggg (\lambda x. k x \ggg (\lambda y. h y)) = (m \ggg (\lambda x. k x)) \ggg (\lambda y. h y) \)

Arrows

(1) \[ \text{arr id} \ggg f = f \]
(2) \[ f \ggg \text{arr id} = f \]
(3) \[ (f \ggg g) \ggg h = f \ggg (g \ggg h) \]
(4) \[ \text{arr} (g \cdot f) = \text{arr} f \ggg \text{arr} g \]
(5) \[ \text{first} (\text{arr} f) = \text{arr} (f \times \text{id}) \]
(6) \[ \text{first} (f \ggg g) = \text{first} f \ggg \text{first} g \]
(7) \[ \text{first} f \ggg \text{arr} (\text{id} \times g) = \text{arr} (\text{id} \times g) \ggg \text{first} f \]
(8) \[ \text{first} f \ggg \text{arr} \text{fst} = \text{arr} \text{fst} \ggg f \]
(9) \[ \text{first} (\text{first} f) \ggg \text{arr} \text{assoc} = \text{arr} \text{assoc} \ggg \text{first} f \]

Idioms (Applicative Functors)

(1) \( u = \text{pure } id \otimes u \)

(2) \( \text{pure } f \otimes \text{pure } p = \text{pure } (f \ p) \)

(3) \( u \otimes (v \otimes w) = \text{pure } (\cdot) \otimes u \otimes v \otimes w \)

(4) \( u \otimes \text{pure } x = \text{pure } (\lambda f. f \ x) \otimes u \)

Scala translations with help from:

Tony Morris & Jed Wesley-Smith

Typesetting of new slides:

Jed Wesley-Smith

Scala source available:

https://bitbucket.org/jwesley smith/yow-monads