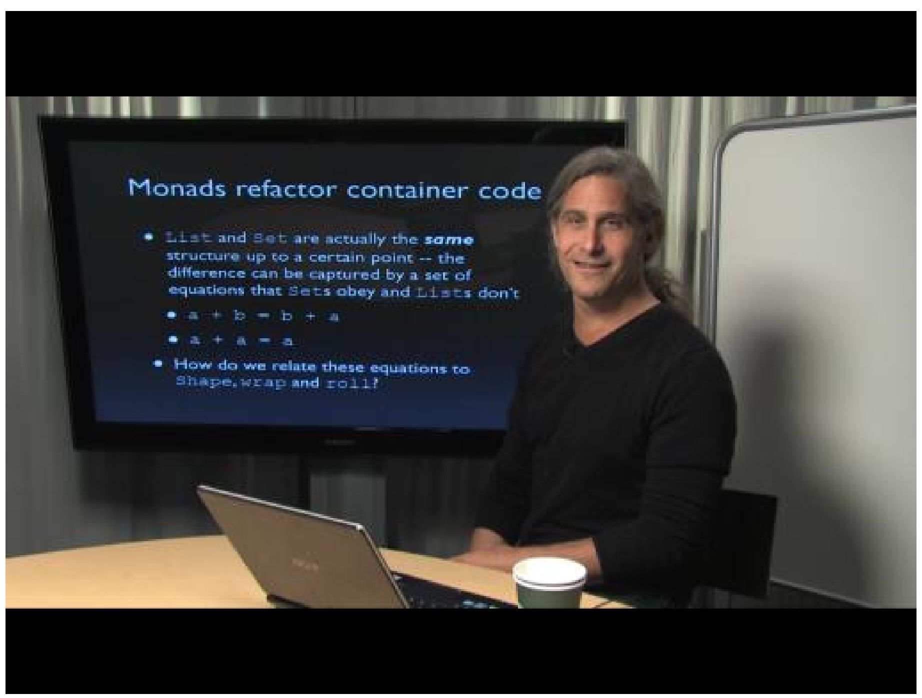
The First Monad Tutorial

Philip Wadler
University of Edinburgh

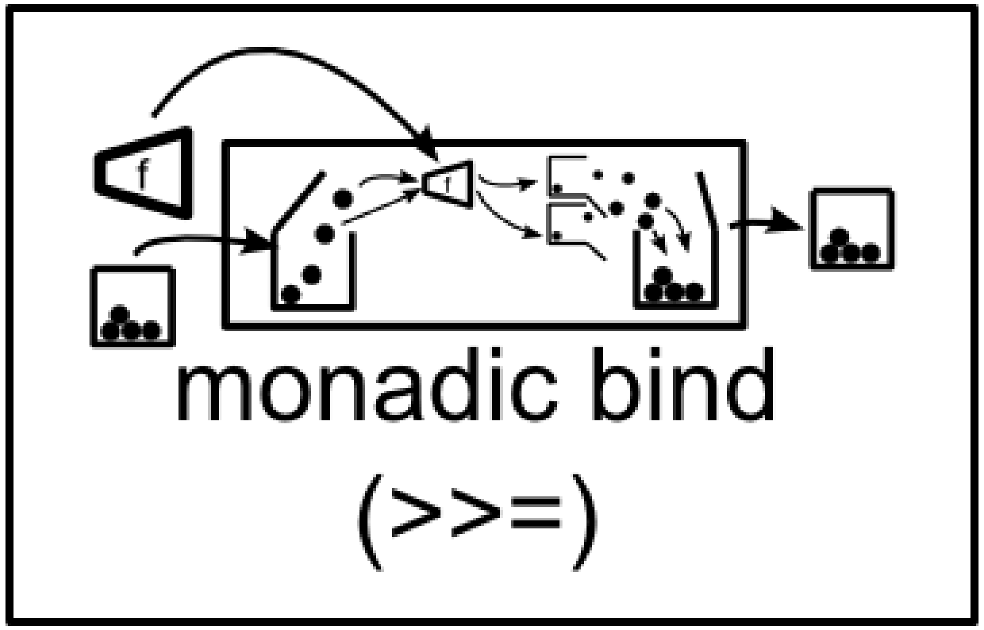
YOW! Melbourne, Brisbane, Sydney
December 2013

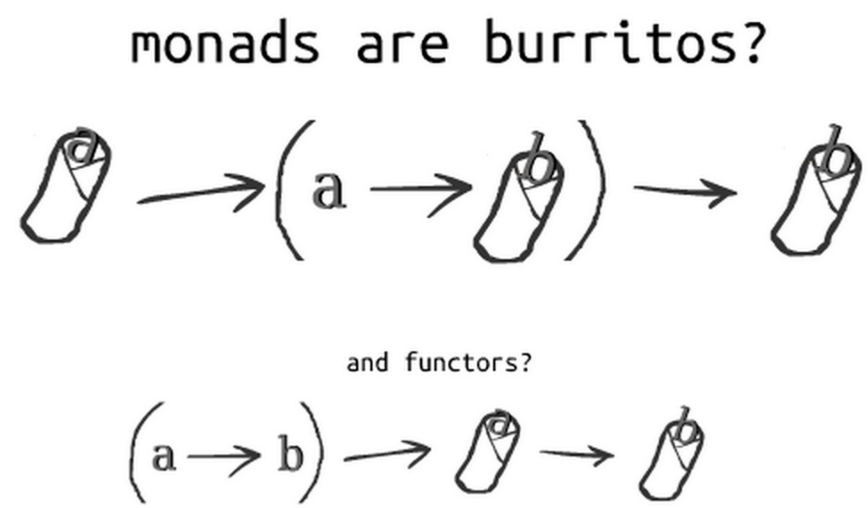

SCALA MONADS
DECLUTTER YOUR CODE WITH MONADIC DESIGN
SC
Dan Rosen, Marakana, Inc.
San Francicso, CA - February 09, 2012

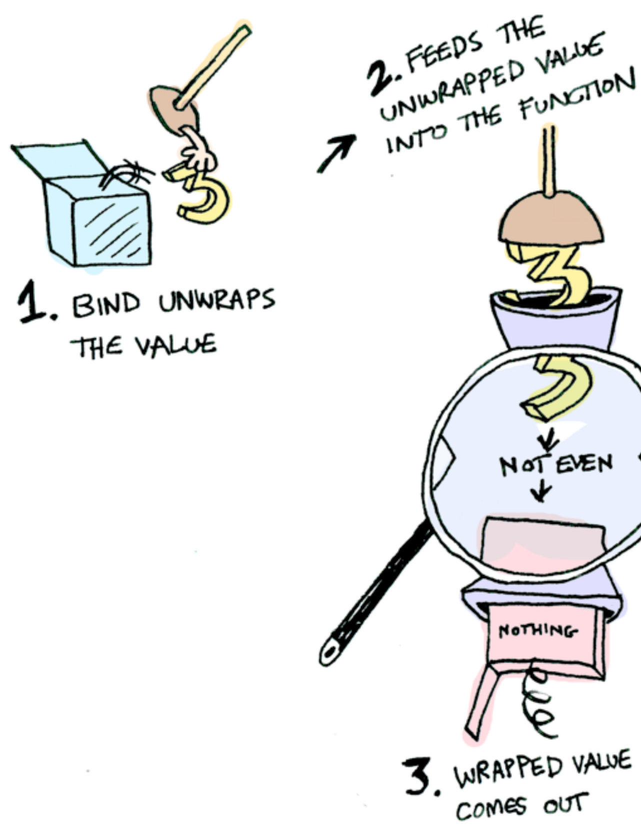

Monads in Perl
本間　雅洋
hiratara <hira.tara@gmail.com>

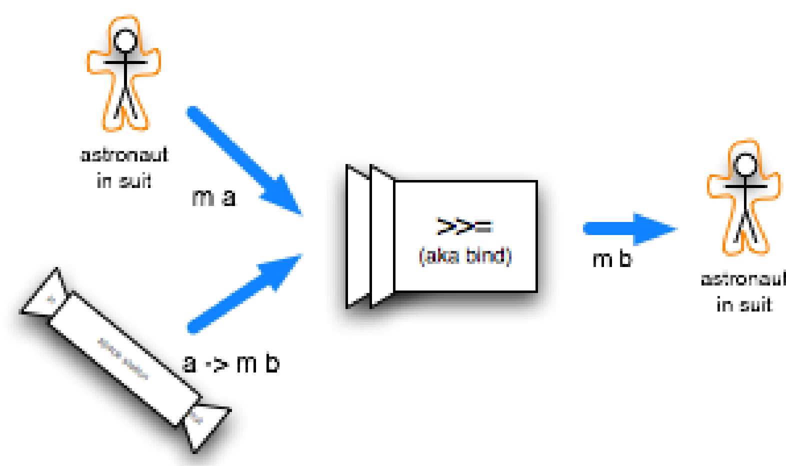

Monads refactor container code
List and Set are actually the same structure up to a certain point -- the difference can be captured by a set of equations that Sets obey and Lists don't
a + b = b + a
a + a = a
How do we relate these equations to Shape, wrap and roll?


Yet another presentation on
MONADS
Marcin Gryszko
@mgryszko

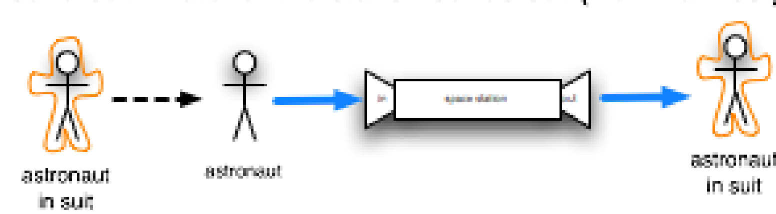








1. remove astronaut from suit
2. put naked astronaut in station
3. send out whatever the station sends out (well... almost)



Haskell

Clojure

Microsoft Visual F#

Microsoft C#.net

Scala

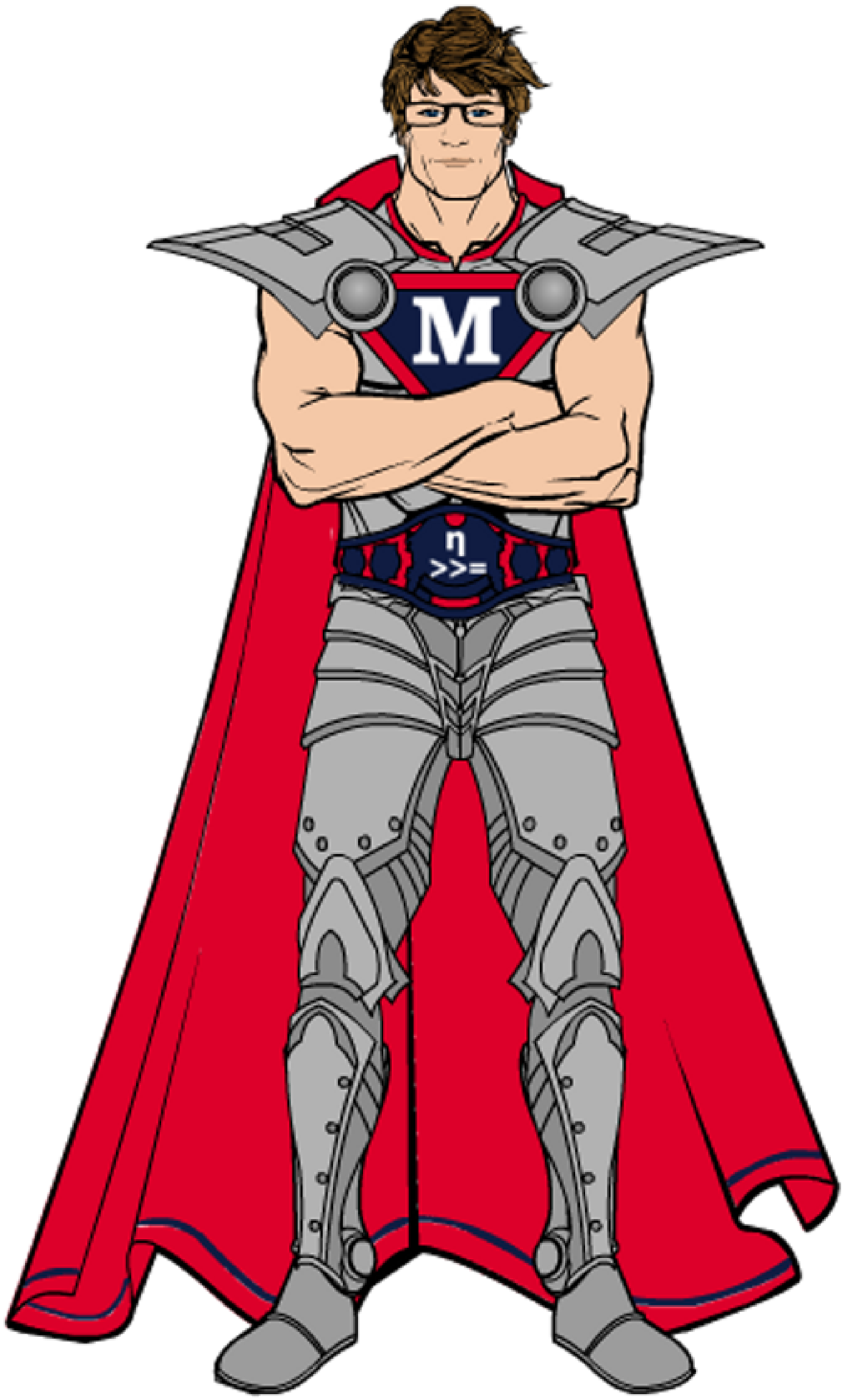
python™

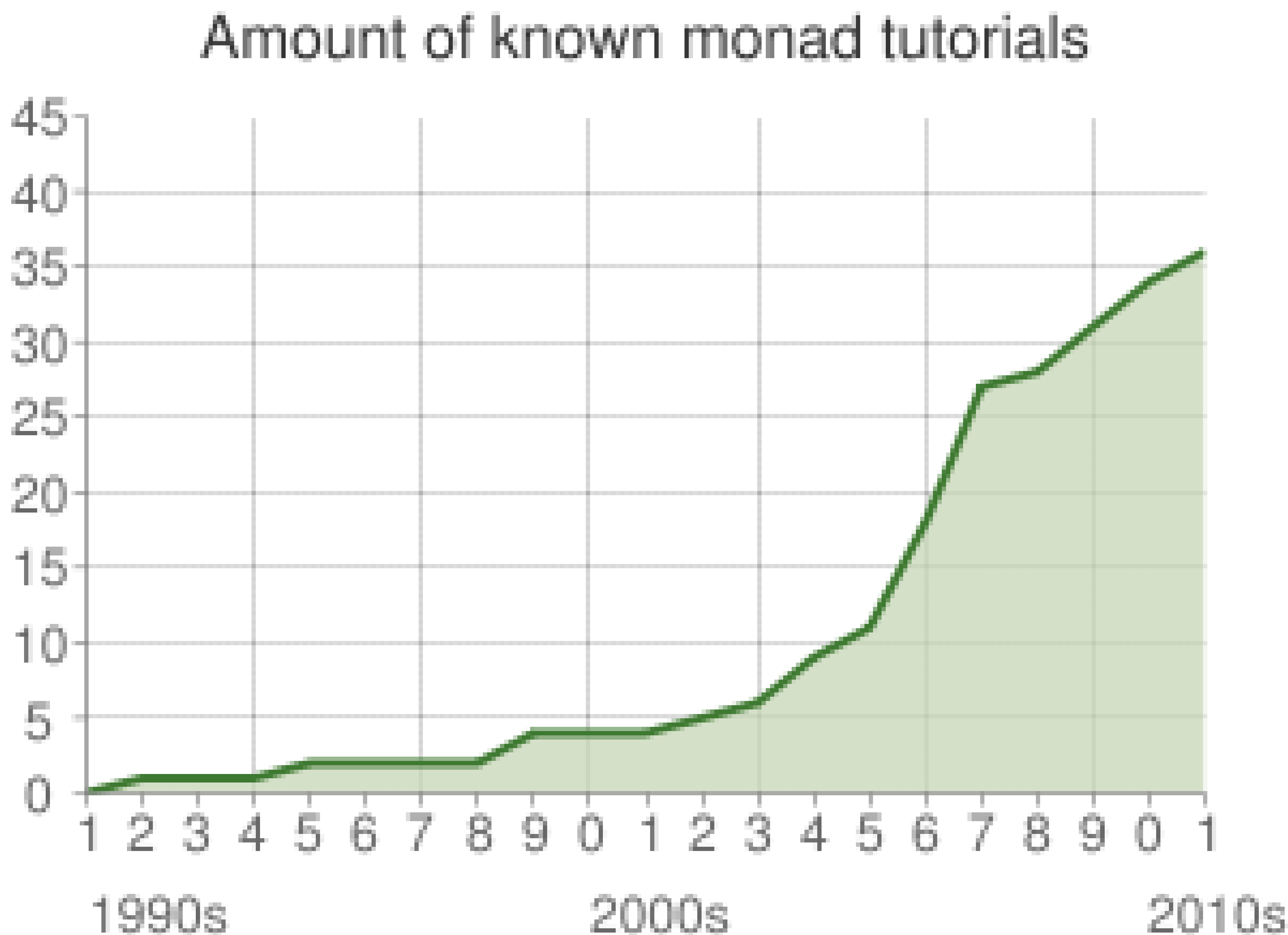

trapd in Monad tutorl
plz help


MONADS
MONADS EVERYWHERE
makeameme.org


trapd in IO monad
plz help


BRACE YOURSELVES
THE MONAD TUTORIALS ARE
COMING


M
η
>>=



Amount of known monad tutorials
45
40
35
30
25
20
15
10
5
0
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
1990s
2000s
2010s

Church and State 1: Evaluating monads

Philip Wadler

~~University of Glasgow~~

Bell Labs, Lucent Technologies

Pure vs. impure

Modification	Impure language (Standard ML, Scheme)	Pure language (Miranda*, Haskell†)
Error messages	use exceptions	rewrite
Execution counts	use state	rewrite
Output	use output	rewrite
Backwards output	rewrite	modify output

*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

$$
\begin{array}{lcl}
\mathbf{data}\ \ \mathit{Term} & = & \mathit{Con}\ \mathit{Int} \mid \mathit{Div}\ \mathit{Term}\ \mathit{Term} \\
\\
\mathit{eval} & :: & \mathit{Term} \rightarrow \mathit{Int} \\
\mathit{eval}\,(\mathit{Con}\ a) & = & a \\
\mathit{eval}\,(\mathit{Div}\ t\ u) & = & \mathit{eval}\ t \div \mathit{eval}\ u
\end{array}
$$

The Term type

```
trait Term
case class Con(a: Int) extends Term
case class Div(t: Term, u: Term) extends Term
```

Test data

```
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

```scala
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> 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 => B` becomes `A => M[B]`
   In particular, `def eval(t: Term): Int`
   becomes `def eval(t: Term): M[Int]`

2. A way to turn values into computations.

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

3. A way to combine computations.

   ```
   def bind[A, B](m: M[A], k: A => M[B]): M[B]
   ```

Monad laws

Left unit

```
bind(pure(a), k) ≡ k(a)
```

Right unit

```
bind(m, (a: A) => pure(a)) ≡ m
```

Associative

```
bind(bind(m, (a: A) => k(a)), (b: B) => h(b))
≡
bind(m, (a: A) => bind(k(a), (b: B) => 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

```scala
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)
```

Variation zero, revisited: Identity

```
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

```scala
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

```
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

```
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

```scala
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, ())
```

Variation three, revisited: Output

```
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

```
type M[A] = Stream[A]

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

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

Cartesian products

```scala
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 *n.* **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

Eugenio Moggi, *Computational λ-calculus and monads*, 1989.

values (*int*) vs. computations (T *int*)

call-by-value ($int \to T\ int$)

call-by-name ($T\ int \to T\ int$)

Michael Spivey, *A functional theory of exceptions*, 1990.

John Reynolds, *The essence of Algol*, 1981.

data types (*int*) vs. phrase types (*int exp*)

call-by-value ($int \to int\ exp$)

call-by-name ($int\ exp \to int\ exp$)

But Reynolds missed *unit* and $\star$.

monadism *or* **monadology** *n.* (esp. in writings of Leibnitz) the philosophical doctrine that monads are the ultimate units of reality.

– *Collins English Dictionary*

Monads

$$
\begin{array}{lrcl}
(1) & \textit{return}\; v \gg\!= \lambda x.\, k\; x & = & k\; v \\
(2) & m \gg\!= \lambda x.\, \textit{return}\; x & = & m \\
(3) & m \gg\!= (\lambda x.\, k\; x \gg\!= (\lambda y.\, h\; y)) & = & (m \gg\!= (\lambda x.\, k\; x)) \gg\!= (\lambda y.\, h\; y)
\end{array}
$$

- Eugenio Moggi, Computational Lambda Calculus and Monads, *Logic in Computer Science*, 1989.
- Philip Wadler, Comprehending Monads, *International Conference on Functional Programming*, 1990.
- Philip Wadler, The Essence of Functional Programming, *Principles of Programming Languages*, 1992.
- Philip Wadler, Monads for functional programming, *Marktoberdorf Summer School on Program Design Calculi*, M. Broy, editor, Springer Verlag, 1992.

Arrows

$$
\begin{array}{lrcl}
(1) & \mathit{arr}\ \mathit{id} \ggg f & = & f \\
(2) & f \ggg \mathit{arr}\ \mathit{id} & = & f \\
(3) & (f \ggg g) \ggg h & = & f \ggg (g \ggg h) \\
(4) & \mathit{arr}\ (g \cdot f) & = & \mathit{arr}\ f \ggg \mathit{arr}\ g \\
(5) & \mathit{first}\ (\mathit{arr}\ f) & = & \mathit{arr}\ (f \times \mathit{id}) \\
(6) & \mathit{first}\ (f \ggg g) & = & \mathit{first}\ f \ggg \mathit{first}\ g \\
(7) & \mathit{first}\ f \ggg \mathit{arr}\ (\mathit{id} \times g) & = & \mathit{arr}\ (\mathit{id} \times g) \ggg \mathit{first}\ f \\
(8) & \mathit{first}\ f \ggg \mathit{arr}\ \mathit{fst} & = & \mathit{arr}\ \mathit{fst} \ggg f \\
(9) & \mathit{first}\ (\mathit{first}\ f) \ggg \mathit{arr}\ \mathit{assoc} & = & \mathit{arr}\ \mathit{assoc} \ggg \mathit{first}\ f
\end{array}
$$

- John Hughes, Generalising Monads to Arrows, *Science of Computer Programming*, 2000.

Idioms (Applicative Functors)

$$
\begin{array}{lrcl}
(1) & u & = & \mathit{pure}\ \mathit{id} \otimes u \\
(2) & \mathit{pure}\ f \otimes \mathit{pure}\ p & = & \mathit{pure}\ (f\ \ p) \\
(3) & u \otimes (v \otimes w) & = & \mathit{pure}\ (\cdot) \otimes u \otimes v \otimes w \\
(4) & u \otimes \mathit{pure}\ x & = & \mathit{pure}\ (\lambda f.\ f\ \ x) \otimes u
\end{array}
$$

- Conor McBride and Ross Patterson, Applicative Programming with Effects, *Journal of Functional Programming*, 2008.

Scala translations with help from:

Tony Morris & Jed Wesley-Smith

Typesetting of new slides:

Jed Wesley-Smith

Scala source available:

https://bitbucket.org/jwesleysmith/yow-monads