Monad Transformers

Mads Hartmann Jensen
@mads_hartmann
http://mads379.github.com/

Monad Transformers

"Monads provide a powerful way to build computations with effects. Each of the standard monads is specialized to do exactly one thing. In real code, we often need to be able to use several effects at once"

- ***Real World Haskell*** *by Bryan O'Sullivan, Don Stewart, and John Goerzen*

Agenda

- Two different Monads
  - Implementation
  - Phonebook Example Program using the monad
- Monad Transformers

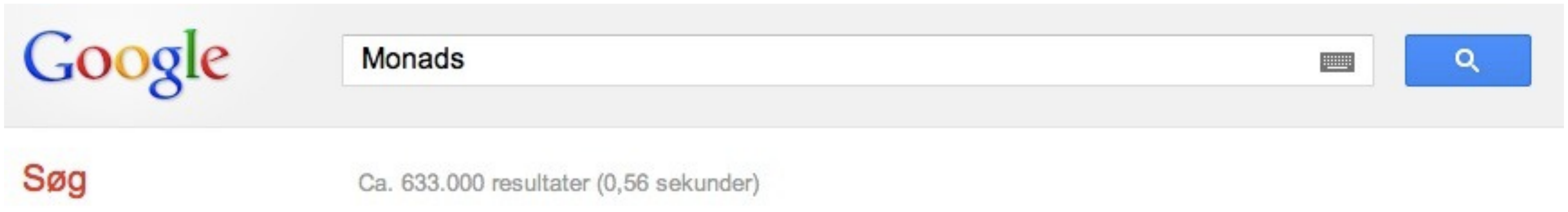
Example Program

- Phonebook with support for insertion, removal and lookup of records.
- We need to deal with state and error handling.

Example Program

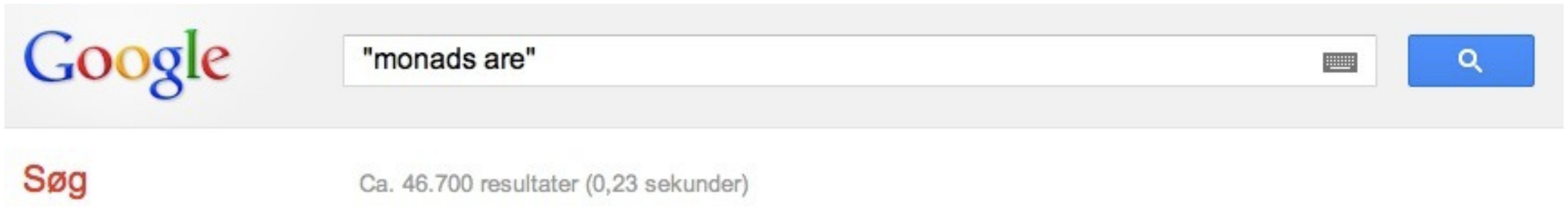
```scala
sealed abstract class Command
case class Add(name: String, info: String) extends Command
case class Remove(name: String) extends Command
case class Lookup(name: String) extends Command
```

Monads


Monads

Søg

Ca. 633.000 resultater (0,56 sekunder)


"monads are"

Søg

Ca. 46.700 resultater (0,23 sekunder)

Monads

Creating an instance of monad M[A]

```
apply[A](a: A): M[A]
```

Interface of monad Instance M[A]

```
map[B](f: A => B): M[B]

flatMap[B](f: A => M[B]): M[B]
```

Monad Laws

```
apply(a).flatMap(f) == f(a)

m.flatMap( apply ) == m

(m.flatMap(f)).flatMap(g) == (m.flatMap(f(_).flatMap(g))
```

Maybe

Maybe

- Monad useful for modeling optional data.

Maybe

```scala
abstract class Maybe[A] {

  def map[B](f: A => B): Maybe[B]
  def flatMap[B](f: A => Maybe[B]): Maybe[B]
  def getOrElse(default: A): A

}

case class Success[A](private val value: A) extends Maybe[A] {

  def map[B](f: A => B) = Success(f(this.value))
  def flatMap[B](f: A => Maybe[B]) = f(this.value)
  def getOrElse(default: A) = this.value

}

case class Fail[A] extends Maybe[A] {

  def map[B](f: A => B) = Fail[B]()
  def flatMap[B](f: A => Maybe[B]) = Fail[B]()
  def getOrElse(default: A) = default

}
```

Maybe

```
Success(10).flatMap( a => Success(20).map( b => a+b ))
// com.sidewayscoding.Maybe[Int] = Success(30)

for {
  a <- Success(10)
  b <- Success(20)
} yield a + b
// com.sidewayscoding.Maybe[Int] = Success(30)


Success(10).flatMap( a => Fail[Int].map( b => a+b ))
// com.sidewayscoding.Maybe[Int] = Fail()

for {
  a <- Success(10)
  b <- Fail[Int]()
} yield a + b
// com.sidewayscoding.Maybe[Int] = Fail()
```

Maybe

```
val x = mightFail()
if (x != null) {
    val y = mightAlsoFail(x)
    if ( y != null) {
      println("some information: %s".format(y))
    } else {
      println("failed :-(")
    }
}

println( mightFail().flatMap{ x =>
  mightAlsoFail(x).map { y =>
    "some information %s".format(y)
  }.getOrElse("failed :-(")
})

println(for {
  x <- mightFail()
  y <- mightAlsoFail()
} yield "some information %s".format(y))
```

Maybe

```scala
object PhonebookOptionApp extends App {

  type Storage = HashMap[String,String]

  def execute(cmd: Command, storage: Storage): Option[(Storage,String)] = cmd match {

    case Add(name, info) => Some(storage + (name -> info), "Added new record: %s".format(name))

    case Remove(name)    => Some(storage - name, "Removed record: %s".format(name))

    case Lookup(name)    => storage.get(name)
                                   .map { info =>
                                            (storage,"Information for %s: %s".format(name, info)) }
  }
}
```

Maybe

```
val initialStorage = HashMap[String,String]()

val rslt1 = for {
  tup1 <- execute(Add("mads","DIKU"), initialStorage)
  tup2 <- execute(Lookup("mads"), tup1._1)
} yield List(tup1._2, tup2._2).mkString("\n")

val rslt2 = for {
  tup1 <- execute(Add("mads","DIKU"), initialStorage)
  tup2 <- execute(Lookup("mads"), tup1._1)
  tup3 <- execute(Remove("mads"), tup2._1)
  tup4 <- execute(Lookup("mads"), tup3._1)
} yield List(tup1._2, tup2._2, tup3._2, tup4._2).mkString("\n")

println("rslt1:\n" + rslt1.getOrElse("failed ;-("))
println("rslt2:\n" + rslt2.getOrElse("failed ;-("))

// rslt1:
// Added new record: mads
// Information for mads: DIKU
// rslt2:
// failed ;-(
```

State

State

- Monad for modeling state

```
case class State[S,R](private val f: (S) => (R,S)) {

  def map[A](g: (R) => A): State[S,A] = State { (s: S) =>
    val (a, s2) = this.f(s)
    (g(a), s2)
  }

  def flatMap[A](g: R => State[S,A]): State[S,A] = State { (s: S) =>
    val (a, s2) = this.f(s)
    val State(f2) = g(a)
    f2(s2)
  }

  def eval(s: S) = this.f(s)._1
  def exec(s: S) = this.f(s)._2
}

object State {
  def get[S]                       = State[S,S]( (s: S) => (s,s) )
  def put[S](newS: S)              = modify( (_: S) => newS)
  def modify[S]( g: (S) => S) = State[S,S] { (s) =>
    val newS = g(s)
    (newS, newS)
  }
}
```

```scala
object PhonebookStateApp extends App {

  type Storage = HashMap[String,String]

  def execute(cmd: Command): State[Storage, String] = cmd match {

    case Add(name, info) => for {
      _ <- modify { (s: Storage) => s.updated(name, info) }
    } yield "Added new record: %s".format(name)

    case Remove(name) => for {
      _ <- modify { (s: Storage) => s - name }
    } yield "Removed record: %s".format(name)

    case Lookup(name) => for {
      s <- get[Storage]
    } yield s.get(name)
             .map{ info => "Information for %s: %s".format(name, info) }
             .getOrElse("No such person in the book")
  }
}
```

```scala
val initialStorage = HashMap[String,String]()

val rslt1 = for {
  str1 <- execute(Add("mads","DIKU"))
  str2 <- execute(Lookup("mads"))
  str3 <- execute(Remove("mads"))
  str4 <- execute(Lookup("mads"))
  str5 <- execute(Add("mads","DIKU"))
} yield List(str1, str2, str3, str4, str5).mkString("\n")

println("rslt1:\n" + rslt1.eval(initialStorage))
// rslt1:
// Added new record: mads
// Information for mads: DIKU
// Removed record: mads
// No such person in the book
// Added new record: mads
```

Maybe & State

Maybe & State

- We want to re-use the monads we already have, hence Monad Transformers

```
object PhonebookMonadTransformerApp extends App {

  import PhonebookData._

  type St = (Int, Map[String, String])                        // The state of the application
  type Failure = String                                       // Make types easier to read
  type PhonebookStateT[A] = StateT[Id, St, A]                 // We want some state
  type PhonebookT[A] = EitherT[PhonebookStateT, Failure, A] // We want some error handling

  def execute(cmd: Command): PhonebookT[String] = cmd match {

    case Lookup(name) => for {
        s           <- tick()
        (cnt, book) =  s
        rslt        =  book.get(name).map( x => Right("information for mads: " + x) )
                                   .getOrElse(Left("Failure executing command %d".format(cnt)))
      } yield rslt

    case Remove(name) => for {
        _    <- tick()
        rslt <- modify { (s: St) => (s._1, s._2 - name)}
      } yield Right("Successfully removed %s to the book".format(name))

    case Add(name, information) => for {
      s     <- tick()
      rslt  <- modify { (s: St) => (s._1, s._2 + (name -> information)) }
    } yield Right("Successfully added %s to the book".format(name))

  })

  def tick(): PhonebookStateT[St] = for {
    s              <- init[St]
    (cnt, storage) =  s
    newS           <- put( (cnt+1, storage))
  } yield newS
}
```

```scala
object PhonebookMonadTransformerApp extends App {

  import PhonebookData._

  type St = (Int, Map[String, String])                        // The state of the application
  type Failure = String                                       // Make types easier to read
  type PhonebookStateT[A] = StateT[Id, St, A]                 // We want some state
  type PhonebookT[A] = EitherT[PhonebookStateT, Failure, A] // We want some error handling

  def execute(cmd: Command): PhonebookT[String] = liftStateTtoEitherT(cmd match {

    case Lookup(name) => for {
        s           <- tick()
        (cnt, book) =  s
        rslt        =  book.get(name).map( x => Right("information for mads: " + x) )
                                     .getOrElse(Left("Failure executing command %d".format(cnt)))
      } yield rslt

    case Remove(name) => for {
        _    <- tick()
        rslt <- modify { (s: St) => (s._1, s._2 - name)}
      } yield Right("Successfully removed %s to the book".format(name))

    case Add(name, information) => for {
      s     <- tick()
      rslt  <- modify { (s: St) => (s._1, s._2 + (name -> information)) }
    } yield Right("Successfully added %s to the book".format(name))

  })

  def liftStateTtoEitherT[A](st: PhonebookStateT[Either[Failure, A]]): PhonebookT[A] =
    EitherT[PhonebookStateT, Failure, A](st)
}
```

```scala
val initial = (0, HashMap[String, String]())

val rslt1 = for {
  msg1 <- execute(Add("mads","21"))
  msg2 <- execute(Lookup("mads"))
} yield List(msg1, msg2).mkString("\n")

println("rslt1:\n" + rslt1.run.eval(initial))

val rslt2 = for {
  msg1 <- execute(Lookup("mads"))
  msg2 <- execute(Add("mads","21"))
} yield List(msg1, msg2).mkString("\n")

println("rslt2:\n" + rslt2.run.eval(initial))

// rslt1:
// Right(Successfully added mads to the book
// information for mads: 21)
// rslt2:
// Left(Failure executing command 1: Not Found in phonebook)
```