Monads: The Essence of Functional Programming

Bruce Cota

March 4, 2014

Monads: The Essence of Functional Programming

- Title poached from a 1992 paper oby Phillip Wadler about the *Glorious Glasgow Haskell Compiler*
- I didn't understand monads until I found that paper

Prologue

In Scala you often see code like this

```
val l1 = List(1,2,3,4)
val l2 = List(8,10,100,92)
for (x <- l1; y<-l2) yield x*y
```

It looks like syntactic sugar for nested iteration over two lists.

Prologue

Then you see code that looks like this

```
val usdQuote = future { connection.getVal(USD) }
val chfQuote = future { connection.getVal(CHF) }
val purchase = for {
  usd <- usdQuote
  chf <- chfQuote
  if isProfitable(usd, chf)
} yield connection.buy(amount, chf)
```

We're iterating over futures??

It seems like you can "iterate" over anything in Scala

Prologue

- `for` statement is doing *for comprehension over Monads*
- Based on the `do` statement from Haskell
- Scala people say "Monad" about every 10 minutes
- What the heck is a Monad?
- Should I care?

Monads in Greek Philosophy

> *According to Hippolytus, this view was inspired by the Pythagoreans, who called the first thing that came into existence the* monad, *which begat the dyad, which begat the numbers, which begat the point, begetting lines or finiteness,*

(from Wikipedia)

Monads in Category Theory

- A *monad* is a certain construction (a *monoid* on the category of *functors*), that can (supposedly) be used to define most other constructions of Category Theory.
- See final chapter of online book: *Learn You a Haskell for a Greater Good*

Monads in Functional Programming

- In Wadell[92], a Monad is a design pattern for solving problems in Haskell that are utterly trivial in any other language.
- If can be formally described as a Monad from Category Theory (almost).
- Used as a fundamental control structure in Haskell.
- Heavily used in Scala.

Prologue

- In Wadell[92], a Monad is a design pattern for solving problems in Haskell that are utterly trivial in any other language.
- But they generalize to a different (and better!) way of solving many fundamental problems that were "settled law"!
  - Compile time checking for Null Pointer Exceptions
  - Error Handling without Exceptions
  - Iteration without nested loops
  - Composing futures from futures

And Monads are actually very simple!

Monads: Simple but Hard

Monads are simple, but were hard (for me) to understand. Why?

1. Motivation (I don't use Haskell)
2. Category Theory
3. Inconsistent terminology
4. Monads in Scala are extra confusing.
5. Monads in Functional Programming are not a metaphor for anything (except, well Monads from Category Theory).

Category Theory

Category Theory is a branch of Mathematics also called

- Universal Mathematics (by Category Theorists)

Category Theory

Category Theory is a branch of Mathematics also called

- Universal Mathematics (by Category Theorists)
- Abstract Nonsense (by other Mathematicians)

Category Theory

Category Theory is a branch of Mathematics also called

- Universal Mathematics (by Category Theorists)
- Abstract Nonsense (by other Mathematicians)
- Intellectual Terrorism (by Computer Scientists)

The Category of Types

I think a tiny bit of category theory (three terms) makes things clearer, so ..

- Imagine an infinite directed graph
- Nodes are types (`Int`, `String`, `List[String]`, etc).
- Arrows are functions that you can write in code, e.g. `toString` is an arrow from `Int` to `String`

There are two properties that make this graph a category.

Function Composition

First, a function from $g : X => Y$ and a function $f : Y => Z$ can be composed to give a new function $g \circ f : X => Z$.

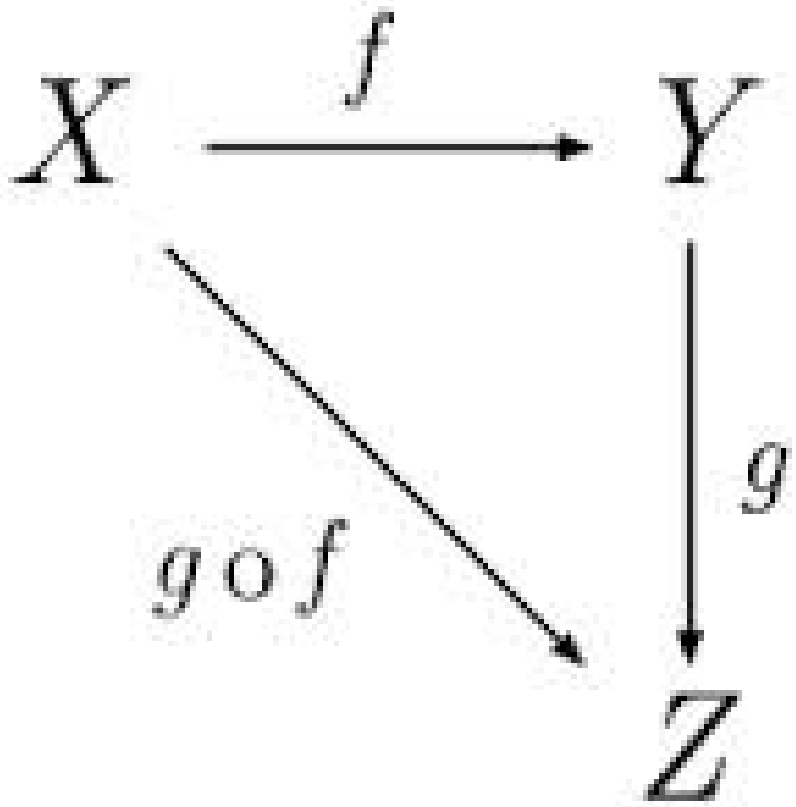


Figure : This diagram *commutes*

Identity Arrow

Second, every node has an identity arrow

```
def id[T](x:T) = x
```

such that for any functions $f : X => Y$

```
f(id[X](x)) = f(x)
```

and

```
id[X](f(x)) = f(x)
```

In category theory: $f \circ id_X = id_y \circ f = f$

A Simple Category

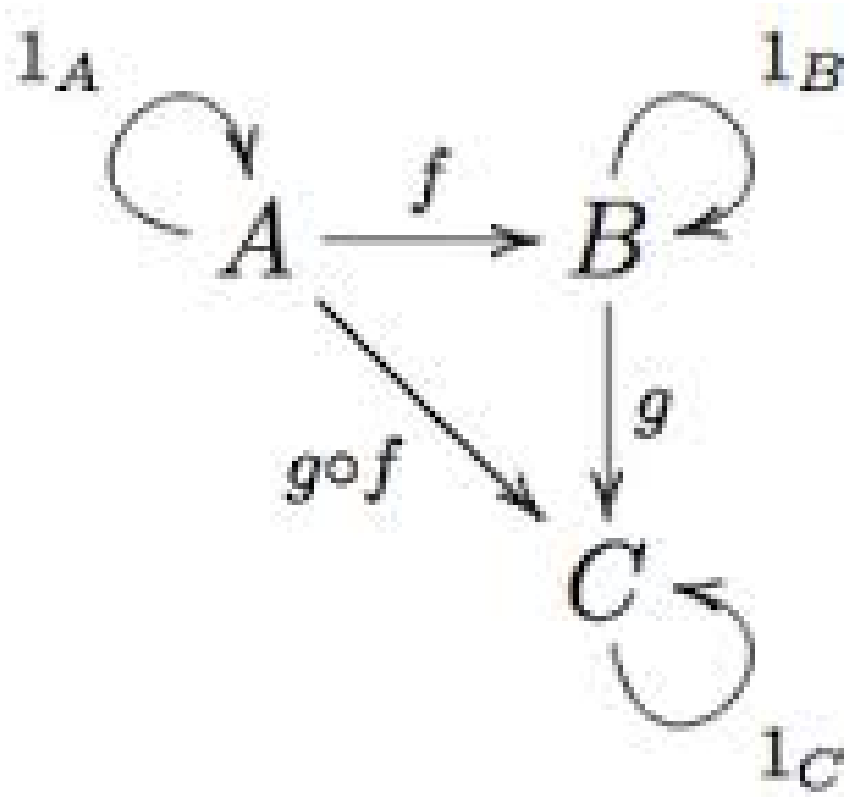
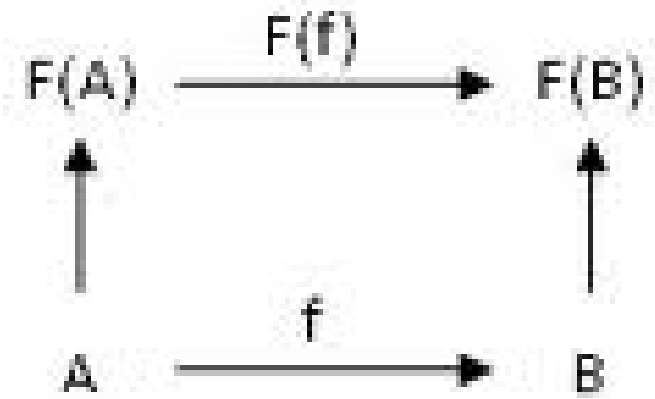


Figure : This diagram *commutes*

Functors

A *functor* is a mapping between categories that preserves composition.



Functors

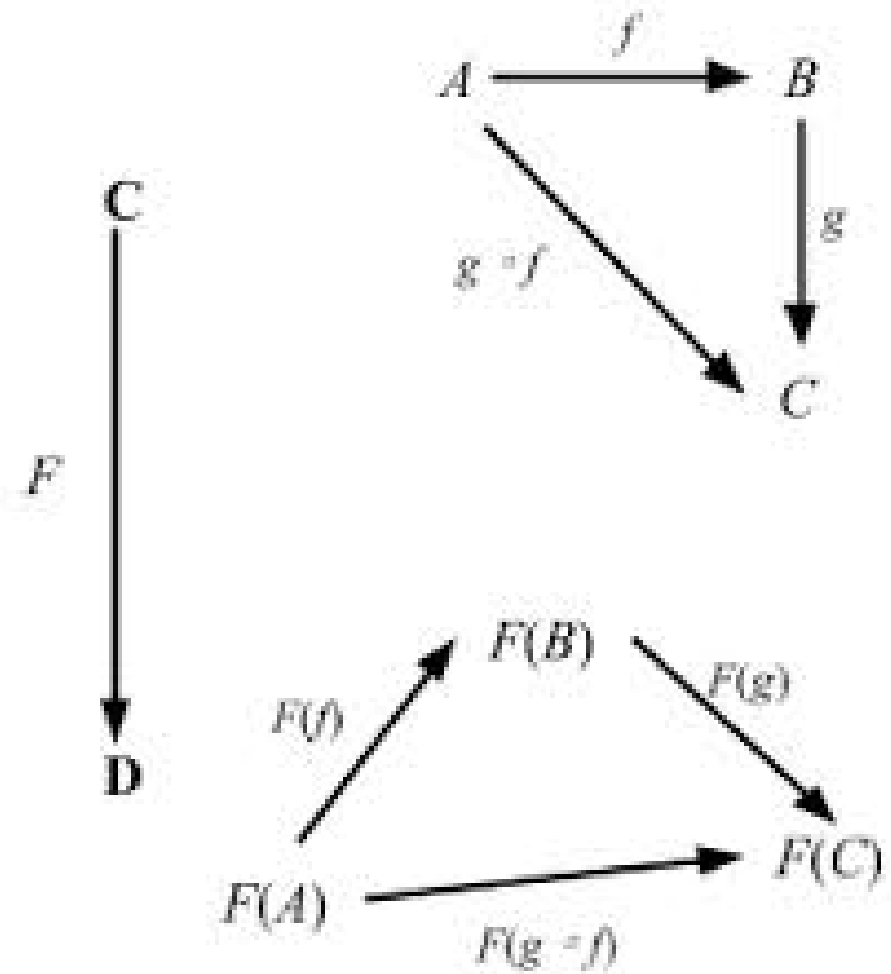


Figure : F is a functor from **C** to **D**

Functors in Category of Types

A functor from the category of types (an *endofunctor*) has three parts

1. A Generic Type `F[X]` – the *Type Constructor*.
2. A generic *unit function* that maps `X` to `F[X]`.
3. A (generic) higher order *lifting* function that maps every function to a function `F[A] => F[B]`.

Functor Example

1. Type Constructor:

   ```
   List[X]
   ```

   (that's Scala syntax – same as `List<X>`)

2. unit function

   ```
   def unit[X](x:X) = List(x)  //singleton list
   ```

3. lifting function

   ```
   //iterate over a list of A.  Apply f to each a,
   def lift[A,B](f:A=>B) = {
     la:List[A] => for (a<-la) yield f(a)
   }
   ```

   (A real functional programmer would use recursion.)

Monads in Category Theory

- A Monad is essentially a flattening operation on an (endo)functor.
- Consider flattening a List[List[T]] to a List[T]:
  - `List( List(a,b), List(c,d), List(e,f) )` => `List(a,b,c,d,e,f)`
- Flattening operation must be associative – it doesn't matter whether List[List[List[T]]] is flattened from "outside in" or "inside out".

Monads – who cares?

Motivating Example from Wadell:

- Imagine adding a counter to a **pure** functional program – with **no state**..
- You can't use a variable, it has to be part of the argument returned by the functions.
- Probably only a few functions care about this counter
- How can we do this without adding the count to the input/output of all functions.
- (Pretty soon you're carrying a giant Hash that acts like state.)

Monads – who cares?

Motivating Example from Wadell:

> Suppose you have $f : X => Y$ and $g : Y => Z$ and you want to change f $f' : X => \langle Y, \mathrm{Int} \rangle$.
>
> How can we compose f' and g without also modifying g?

The $\langle Y, \mathrm{Int} \rangle$ is a *tuple* – an ordered pair, like a record without names for the members.

Monads – who cares?

Consider the following functor $\mathrm{F[T]}$

- Type Constructor: $\langle$ `T, Int` $\rangle$
- Unit Function: `def u[T](x:T) = (x,0)`
- Lifting Function: Given a function $\mathrm{f : A => B}$, create a new function that takes a $\langle \mathrm{a, i} \rangle$ and returns $\langle \mathrm{f(a), i} \rangle$ – just ignore the `Int`

Monads – who cares?

Nowe can use the lifting function to compose $g : Y => Z$ with $\mathrm{f}' : \mathrm{X} => \mathrm{F[Y]}$ to get

$$\mathrm{lift}(g) : \mathrm{F[Y]} => \mathrm{F[Z]}$$

so

$$(\mathrm{lift}(g))(f') : \mathrm{X} => \mathrm{F[Z]}$$

Monads – who cares?

Halfway there, but now we've just moved the problem

- Wind up with a lot of functions of the form $f : \mathrm{A} => \mathrm{F}[\mathrm{B}]$.
- how to compose $f : \mathrm{A} => \mathrm{F}[\mathrm{B}]$ with $g : \mathrm{B} => \mathrm{F}[\mathrm{C}]$?
- If we lift $g : \mathrm{B} => \mathrm{F}[\mathrm{C}]$ we get $\mathrm{lift}(g) : \mathrm{F}[\mathrm{B}] => \mathrm{F}[\mathrm{F}[\mathrm{C}]]$ which is probably not the right thing.

But wait, we can flatten these things!

Monads – who cares?

An object of type $\mathrm{F[F[T]]}$ is of the form $\langle\langle \mathrm{t}, \mathrm{i}\rangle, \mathrm{j}\rangle$.

The flattened version is just $\langle \mathrm{t}, \mathrm{i} + \mathrm{j}\rangle$.

If we define a `flatten` function, then we can compose $f : \mathrm{A} => \mathrm{F[B]}$ with $g : \mathrm{B} => \mathrm{F[C]}$ by $\mathrm{flatten}((\mathrm{lift}(g))(f)) : \mathrm{A} => \mathrm{F[C]}$

Monads in Haskell

Haskell has a Monad *Typeclass* – essentially an interface with three methods

- A type constructor `T[X]`
- A generic function $\mathrm{unit[X] : X => T[X]}$
- A generic function
  $\mathrm{bind[A, B] : (A => T[B]) \Rightarrow (T[A] => T[B])}$

The bind function "lifts on one side". It can be derived from the flattening operation.

Comprehension over Monads

If T is a Monad, the Haskell do statement composes functions of type $A => T[B]$ with functions of type $B => T[C]$. It can compose a whole list of functions, in fact.
A stupid example:

```
let f x = [1..x] //list of int from 1 to x
let g x = [x,x,x] //list of 3 x's
do a<-[1,2,3]; b<-f a; g b
[1,1,1,1,1,1,2,2,2,1,1,1,2,2,2,3,3,3]
```

Monads in Scala

In Scala, `for` is used instead of `do`.

```
def f(x:Int) = 1 until x
def g(x:Int) = List(x,x,x)
for (a<-List(1,2,3); b<-f(a); c<-g(b)) yield c
res1: List[Int] = List(1, 1, 1, 1, 1, 1, 2, 2, 2)
```

Monads in Scala

Scala has `FilterMonadic trait` which is sort of a functor/monad combination, but a bit more complicated.

- The role of the lifting function is played by a *map* method.
- The `bind` is called *flatmap* (which is a much better name).
- *for* is used instead of `do`.

Collection Monads

What good is this?
First, we have syntactic sugar for nested iteration. (Collections are Monadic.)

```
//Generate a list of all three letter words
val alphabet = 'a' until 'z'
for (c1 <- alphabet;
     c2 <- alphabet;
     c3 <- alphabet)
  yield c1.toString + c2.toString + c3.toString
```

Nice, but not yet worth the price of admission.

Scala for Comprehension

Scala converts this loop

```
for (c1 <- alphabet;
     c2 <- alphabet;
     c3 <- alphabet)
  yield c1.toString+c2.toString+c3.toString
```

into this code

```
alphabet.flatmap {
  c1 => alphabet.flatmap {
    c2 => alphabet.map {
      c3 => c1.toString+c2.toString+c3.toString
    }
  }
}
```

Scala for Comprehension

Let's understand this from the inside out:

```
{
    c3 => c1.toString+c2.toString+c3.toString
}
```

A function (closure) that maps a character `c3` to a `String`.

For Comprehension Inside Out

Let's understand this from the inside out:

```
alphabet.map {
    c3 => c1.toString+c2.toString+c3.toString
}
```

Changes a function `Char => String` into a function `List[Char] => List[String]` and applies this new function to `alphabet`, yielding a `List[String]`.

For Comprehension Inside Out

```
    c2 => alphabet.map {
        c3 => c1.toString+c2.toString+c3.toString
    }
```

A function `Char => List[String]`

For Comprehension Inside Out

```
alphabet.flatmap {
 c2 => alphabet.map {
     c3 => c1.toString+c2.toString+c3.toString
 }
}
```

Changes a function `Char => List[String]` into a function `List[Char] => List[String]` and applies this new function to `alphabet`, yielding a

For Comprehension Inside Out

```
alphabet.flatmap {
 c2 => alphabet.map {
     c3 => c1.toString+c2.toString+c3.toString
 }
}
```

Changes a function `Char => List[String]` into a function `List[Char] => List[String]` and applies this new function to `alphabet`, yielding a `List[String]`

For Comprehension Inside Out

```
alphabet.flatmap {
  c1 => alphabet.flatmap {
    c2 => alphabet.map {
      c3 => c1.toString+c2.toString+c3.toString
    }
  }
}
```

Another flatmap application: changes a function `Char => List[String]` into a function `List[Char] => List[String]` and apply to `alphabet`, yielding a `List[String]`

Futures

- A Future is a value that you have to wait for
- Scala Example:

```
val fval f: Future[List[String]] = future {
  session.getRecentPosts
}
f onComplete {
  case Success(posts) =>
    for (post <- posts) println(post)
  case Failure(t) => println("Error: " + t.getMe
}
```

Futures as Monads

- How would we write `flatmap` for a future?
- Given a function `f:A => Future[B]`, and a value that is a `Future[A]`, how do we produce a `Future[B]`?

Futures as Monads

- It's clear how to flatten a `Future[Future[A]]` – just wait once.
- How would we write `flatmap` for a future?
- Given a function `f:A => Future[B]`, and a value that is a `Future[A]`, how do we produce a `Future[B]`?
- We wait for a value from the `Future[A]` and, if it succeeds, feed it to `f` to get a `Future[B]`
- (The actual Scala code is a little more than I want to paste here.)

For Comprehension over Futures

```
val usdQuote = future {
  connection.getCurrentValue(USD)
}
val chfQuote = future {
  connection.getCurrentValue(CHF)
}
val purchase:Future[Int] = for {
  usd <- usdQuote
  chf <- chfQuote
  if isProfitable(usd, chf)
} yield connection.buy(amount, chf)
```

Composes Futures without any explicit waiting, callbacks, or pattern matching. It's all handled in `flatmap` and `map`.

My Favorite Monad

Figure : My Favorite Monad: `Option`

Nulls

67% of all production stack traces are caused by Null Pointer Exceptions.
(Based on a statistical analysis of 3 stack traces.)

Nulls

67% of all production stack traces are caused by Null Pointer Exceptions.
(Based on a statistical analysis of 3 stack traces.)

Q. Then why do we have `Null`?

Nulls

67% of all production stack traces are caused by Null Pointer Exceptions.
(Based on a statistical analysis of 3 stack traces.)

Q. Why do we have `Null`?

A. To separate declaration from assignment.

```
Int someInt;
//what is the value of  someInt right now?
...
someInt = 17
```

Nulls

67% of all production stack traces are caused by Null Pointer Exceptions.
(Based on a statistical analysis of 3 stack traces.)

Q. Why do we have `Null`?

A. To separate declaration from assignment.

```
Int someInt;
//what is the value of  someInt right now?
...
someInt = 17
```

In (pure) functional programming you can't declare a variable without a value, so Haskell has no nulls.

Nulls

- Java needs `Null` because it needs a value that means "uninitialized".
- In (pure) functional programming there is no such value.
- But `Null` is sometimes used to mean things, like "unknown", "ignore this parameter", etc.
- For this purpose Haskell uses `Maybe` type constructor, which is monadic.
- Same thing in Scala, but it's called `Option`.
- Big advantage over Nulls, because you can't have runtime NPEs.

Option Type

- `Option[T]` is an example of a *union type*
- Concept: A value of type `Option[T]` is either of type `T` or the special value `None`.

Q. How can the compiler tell if, say `100` is a value of type `Int` or of type `Option[Int]`?

Option Type

- `Option[T]` is an example of a *union type*
- Concept: A value of type `Option[T]` is either of type `T` or the special value `None`.

Q. How can the compiler tell if, say `100` is a value of type `Int` or `Option[Int]`?

A. Introduce another type constructor `Some` as a tag signalling to the compiler that the type of an expression is `Option`.

- `val x: Option[Int] = Some(100)` //compiles
- `val x: Option[Int] = None` //compiles
- `val x: Option[Int] = 100` //does not compile
- `val x: Int = Some(100)` //does not compile

Working with Option

`Options` values are "deconstructed" using the pattern matching `match` statement

```
def upperCase(maybeName: Option[String]) =
maybeName match {
   case Some(name) =>  name.trim.toUpperCase
   case None => None
}
scala> upperCase(Some("abc"))
res1: java.io.Serializable = ABC
scala> upperCase(None)
res2: java.io.Serializable = None
scala> upperCase("abc")
<console>:9: error: type mismatch;
```

Also the function won't compile if we leave out the `case None` line.

Benefits of Option

Using `Option.None` to mean "unknown" is better than using `Null` because the following code doesn't compile

```
scala> def upperCase(name: String) =
       name.trim.toUpperCase
scala> upperCase("abc")
res13: String = ABC
scala> upperCase(None)
<console>:9: error: type mismatch;
 found   : None.type
             upperCase(None)
```

Say goodbye to run-time NPE!!

Option as a Monad

`Option[T]` is a natural functor/monad:

- The unit function is just `Some(x)`
- ```
  def map[A,B](f:A=>B) = {
    (x:Option[A]) =>
    x match {
      case None => None
      case Some(a) => Some(f(a))
    }
  }
  ```
- ```
  def flatmap[A,B](f:A=>Option[B]) = {
    (x:Option[A]) =>
    x match {
      case None => None
      case Some(a) => f(a)
    }
  }
  ```

Monads in Scala

The previous slide oversimplified (lied). The actual definitions are *methods* on the `Option` class.

```
class Option[+A] ..
..
def map[B](f: A => B): Option[B] =
this match {
  case Some(a) => Some(f(a))
  case None => None
}
...
def flatMap[B](f: A => Option[B]): Option[B] =
this match {
  case Some(a) => f(a)
  case None => None
}
```

Monads in Scala

The actual definitions from the Scala source code:

```
@inline final def map[B](f: A => B): Option[B] =
  if (isEmpty) None else Some(f(this.get))
```

and

```
@inline final def flatMap[B](f: A => Option[B]):
  if (isEmpty) None else f(this.get)
```

- `isEmpty` returns `true` if and only if `this == None`
- `get` pulls the value out of a `Some(_)`

Option as a Monad

Monadic properties of `Option` can automate the `None` checking.

```
for {
  name <- request.getParameter("name")
  trimmed <- Some(name.trim)
  upper <- Some(trimmed.toUpperCase)
} yield upper
```

Scala compiler translates to:

```
request.getParameter("name").flatmap {
  name => Some(name.trim).flatmap {
    trimmed => {
      Some(trimmed.toUpperCase).map {
        upper => upper
      }
    }
  }
}
```

Option Monad

```
request.getParameter("name").flatmap {
  name => Some(name.trim).flatmap {
    trimmed => {
      Some(trimmed.toUpperCase).map {
        upper => upper
      }
    }
  }
}
```

If `request.getParameter("name")` is `None`, the whole expression returns `None`.

The boilerplate of checking for `None` is captured in the `map` and `flatmap` methods.

Option Monad

But a sequence of calls in `for` comprehension is easier to read than a nested sequence of calls.

```
for {
  name <- request.getParameter("name")
  trimmed <- Some(name.trim)
  upper <- Some(trimmed.toUpperCase)
} yield upper
```

More about Scala for comprehension

For good or ill, the for comprehension in Scala is more complicated than just monadic comprehension.
The good: We can do filtering inside the for loop

```
for {
  name <- request.getParameter("name")
  trimmed <- Some(name.trim)
  upper <- Some(trimmed.toUpperCase)
    if trimmed.length != 0
} yield upper
```

More about Scala for comprehension

More filtering

```
for (c1 <- alphabet;
     c2 <- alphabet if c2 != c1;
     c3 <- alphabet)
  yield c1.toString + c2.toString + c3.toString
```

More about Scala for comprehension

More good: We can put computation in the "body" of the for comprehension

```
for {
  name <- request.getParameter("name");
  trimmed <- Some(name.trim);
  upper <- Some(trimmed.toUpperCase)
           if trimmed.length != 0
} println(upper)
```

More about Scala for comprehension

The bad: To participate in for comprehension, a class must implement four methods

```
abstract def flatMap[B, That]
    (f: (A) => GenTraversableOnce[B])
    (implicit bf: CanBuildFrom[Repr, B, That]): Tha

abstract def foreach[U](f: (A) => U): Unit

abstract def map[B, That](f: (A) => B)
      (implicit bf: CanBuildFrom[Repr, B, That]): T

abstract def withFilter(p: (A) => Boolean):
     FilterMonadic[A, Repr]
```

Probably too complicated for the average Scala developer.

Some Other Monads

- `Try` — clean error transmission without throwing exceptions.
- `Future` — Futures can be composed with for comprehension to yield a new Future.
- `State Monad` –Of interest to purists. Implement state without explicitly adding it to the input of every function.
- `IO Monad` – A function produces a result plus a list of outputs. The for comprehension concatenates all the outputs to an output stream.

Try Monad in Scala

- Functional programming technique for transmitting errors.
- `Try` is a union type like `Option`. A `Try[T]` is either a `Success[T]` or a `Failure`.
- `Failure` constructor takes a `Throwable` argument. (Holds a stack trace or a caught Exception from non-functional code).
- `map` and `flatmap` are similar to `Option[T]` – a `Failure` is always mapped to a `Failure`.
- A `Failure` in a for comprehension effectively "short circuits".

Try Monad Example

If any method inside the `for` returns a `Failure`, the result will be a `Failure`

```
def getURLContent(url: String):
Try[Iterator[String]] =
  for {
    url <- parseURL(url)
    connection <- Try(url.openConnection())
    is <- Try(connection.getInputStream)
    source = Source.fromInputStream(is)
  } yield source.getLines()
```

Note that the `Try(..)` constructs a `Try` by trapping Exceptions and converting them to `Failure` objects.

Try Monad in Scala

- `Try` is new to Scala 2.10.
- `Either` (a symmetric union type) was intended to be used for error transmission, but it's not Monadic so it's not useful for this purpose.
- Other people wrote their own error handling monads, and Scala finally incorporated one into it's standard library.

Summary

- Monads are a simple functional programming technique, but may be hard to understand without a little category theory.
- Invented for pure functional programming
  - StateMonad
  - IOMonad
- New (and better?) way to do old things
  - Nested iteration without nested loops.
  - Null pointer checking with type safe Nulls.
  - Error transmission without throwing/catching exceptions
- Good way to do some new things
  - Composing Futures to create a new Future.