Demystifying Monads

Amanda Laucher

ThoughtWorks

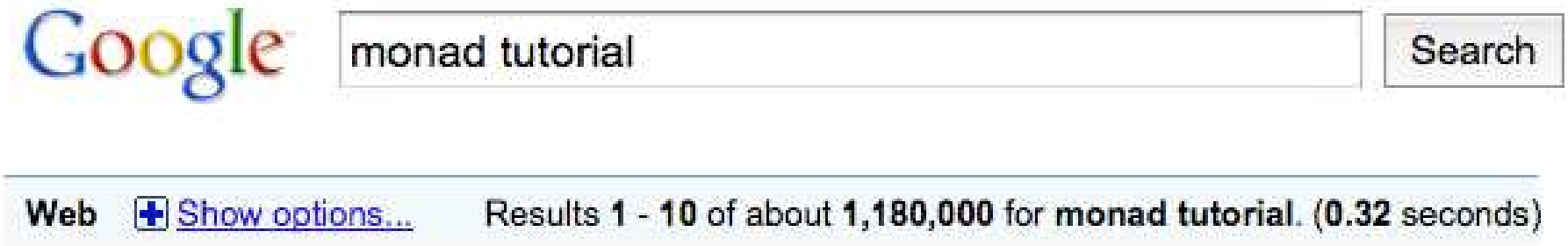
Scientific Fact

Functional programming is the best way to model every computer programming solution

Shared Language

Monads/Gonads, WTH?

Dave Thomas – speakerconf 2010

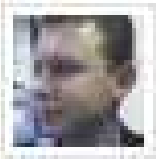

Google
monad tutorial
Search
Web
Show options...
Results 1 - 10 of about 1,180,000 for monad tutorial. (0.32 seconds)

Monad Tutorial Fallacy

What I term the "monad tutorial fallacy," then, consists in failing to recognize the critical role that struggling through fundamental details plays in the building of intuition

Brent Yorgey
http://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/

JoshG Says:
January 13, 2009 at 5:57 pm | Reply



Monads aren't burritos they're vegemite sandwiches! Sheesh! ;-)

The real problem

In mathematical notation, the axioms are:

$$(\mathrm{return}\ x) \gg\!= f \equiv f\ x$$
$$m \gg\!= \mathrm{return} \equiv m$$
$$(m \gg\!= f) \gg\!= g \equiv m \gg\!= \lambda x\ .\ (f\ x \gg\!= g)$$

Categorically unnecessary

It is doubtful that the structuring methods presented here would have been discovered without the insight afforded by category theory

But once discovered they are easily expressed without any reference to things categorical

Phillip Wadler
Monads For Functional Programming

A monad is a design pattern!

- Nothing more
- Monads are not a feature of the language
- They are a pattern that can be used in any language

We don’t need any complex mathematics

Let’s break down what we need...

Type Signatures

`Plus` might have the following signature

`int -> int -> int`

`ToString` might have the following signature

`int -> string`

Type Signatures

`Plus` might have the following signature

```
int -> int -> int
```

`ToString` might have the following signature

```
int -> string
```

`Plus` could also have this signature

```
'a -> 'a -> 'a
```

`ToString` could also have this signature

```
'a -> string
```

Map's type signature

```
('a -> 'b) -> 'a list -> 'b list
```

- For example:

```
map squarestring ["1", "2", "3"]
-> [1, 4, 9]
```

Abstract Data Types

- Just a type
- Built from intrinsic types and other abstract data types
- Type constructor and perhaps some functions

Functions of monads

- Bind
  - adds context to the expression
- Return
  - wraps the input with the monad type
- Others
  - helpers, zero, plus…

So what does a monad do?

- Ask what is happening in the bind!
- The bind adds context to the computation

Vocabulary

- Computational expression
- Workflow
- Monad
  - A type that represents a computation
  - Allows the programmer to chain actions together
  - Each action is decorated with additional processing rules

All monads have 3 things in common

- Type Constructor
  - to define the monad type

All monads have 3 things in common

- Type Constructor
- Return function
  - to wrap the value of the underlying type in the monad type
    `a -> M a`

All monads have 3 things in common

- Type Constructor
- Return function
- Bind function
  - to add context to the expression
    `M(a) -> (a -> M(b)) -> M(b)`
  - Also seen as…
    `let!`
    `>>=`
    `do`

Common monads

- Identity
- Maybe
- List
- Continuation
- Error
- State
- I/O

Identity

- Computation type
  - Simple function application
- Binding strategy
  - The bound function is applied to the input value
- Useful for
  - Passing in an expression or value where a function is required
- Example type
  `identity(a)`

http://www.haskell.org/all_about_monads/html/identitymonad.html

What does Identity look like?

```
type Identity<'a> = |Identity of 'a

type IdentityBuilder() =
    member x.Bind((Identity v), f) = f(v)
    member x.Return v = Identity v

let identity = new IdentityBuilder()
```

How do we use Identity?

```
let calcs() = identity {
                let! a = 1
                let! b = 2
                return a + b
                }
```

Maybe

- Computation type
  - Computations which may return Nothing
- Binding strategy
  - "Nothing" values are bypassed
  - Other values are used as inputs
- Useful for
  - Building computations from sequences of functions that may return Nothing
  - Complex database queries or dictionary lookups are good examples
  - Removing lots of "null checks"
- Example
  - `maybe(person.manager.permission_level)`

http://www.haskell.org/all_about_monads/html/maybemonad.html

Maybe

```
type Maybe<’a> = option<’a>

type MaybeBuilder() =
       member x.Return(x) = succeed x
       member x.Bind(p, rest) = match p with
                                    | None -> fail
                                    | Some r -> rest r

let maybe = MaybeBuilder()
```

Maybe

```
let safesum (x, y) = maybe {
                                let! n1 = failIfBig x
                                let! n2 = failIfBig y
                                let sum = n1 + n2
                                return sum }
```

Or, desugared…

```
let safesum(x, y) =
       maybe.Bind(failIfBig x,
              (fun n1 -> maybe.Bind(failIfBig y,
                     (fun n2 -> maybe.Let(n1 + n2,
                            (fun sum -> maybe.Return(sum)))))))
```

List

- Computation type
  - Those which may return 0, 1, or more possible results
- Binding strategy
  - Applied to all possible values in the input list
  - The resulting lists are concatenated to produce a list of all possible results
- Useful for
  - Building computations from sequences of non-deterministic operations
  - Parsing ambiguous grammars is a common example

http://www.haskell.org/all_about_monads/html/listmonad.html

What does List look like?

```
type ListBuilder() =
    member x.Bind(list, function) = List.concat (List.map function list)
    member x.Return(l) = [l]
    member x.Zero() = []
let listMonad = new ListBuilder()
```

How do we use List?

```
let guarded (b:bool) (xs:'a list)  =
  match b with
  | true  -> xs
  | false -> []

let multiplyTo n =
  listMonad { let! x = [1..n]
              let! y = [x..n]
              return! guarded (x * y = n) [x, y]
             }

let mResult =
  multiplyTo 45 |>
    List.iter (fun t -> printfn "%d-%d" (fst t) (snd t))
```

http://codebetter.com/blogs/matthew.podwysocki/archive/2009/02/20/much-ado-about-monads-list-edition.aspx

Continuation

- Computation type
  - Can be interrupted and resumed
- Binding strategy
  - Creates a new continuation which uses the function as the rest of the computation
- Useful for
  - Complex control structures
  - Workflow processes
  - Error handling
  - Creating co-routines

http://www.haskell.org/all_about_monads/html/continuationmonad.html

What does Continuation look like?

```
type ContinuationMonad() =
    member this.Bind (m, f) = fun c -> m (fun a -> f a c)
    member this.Return x = fun k -> k x

let cont = ContinuationMonad()
```

How do we use Continuation?

```
let fac n =
   let rec loop n =
    cont {
          match n with
          | n when n = 0 -> return 1
          | _ -> let! x = fun f -> f n
                    let! y = loop (n - 1)
                return x * y
        }
   loop n (fun x -> x)

printf "%A" (fac 100000I)
```

Error

- Computation type:
  - Those which may fail or throw exceptions
- Binding strategy
  - Failure records information about the cause/location of the failure
  - Failure values bypass the bound function
  - Other values are used as inputs to the bound function
- Useful for
  - Building computations from sequences of functions that may fail
  - Using exception handling to structure error handling
- Motivation
  - Combine computations that can throw exceptions by bypassing bound functions from the point an exception is thrown to the point that it is handled

http://www.haskell.org/all_about_monads/html/errormonad.html

State

- Computation type
  - Those which maintain state
- Binding strategy
  - Weaves a state parameter through the sequence of bound functions so that the same state value is never used twice, giving the illusion of in-place update
- Useful for
  - Building computations from sequences of operations that require a shared state
- Motivation
  - Avoiding violations of referential transparency

http://www.haskell.org/all_about_monads/html/statemonad.html

IO

- Computation type
  - Those which request input or provide output
- Binding strategy
  - Actions are executed in the order in which they are bound
  - Failures throw I/O errors which can be caught and handled
- Useful for
  - Encapsulating sequential I/O actions
- Motivation:
  - Avoids violations of referentially transparent
  - Confines side-effects

http://www.haskell.org/all_about_monads/html/iomonad.html

Three simple laws

- Left identity
- Right identity
- Associativity

Identity

For all elements in a set:

$e * a = a$ (left identity)

and

$a * e = a$ (right identity)

*where * is any binary operation*

Associativity

$$A * (B * C) = (A * B) * C$$

*where * is any binary operation*

Why use monads?

- Wrap up some code
- Do stuff to it without having to unwrap it
- Eventually use the result

Why use monads?

- Composition is the key to controlling complexity
- Gets rid of boilerplate code
- Helps to focus on the problem domain without getting lost in the ceremony

Questions?

http://manning.com/laucher
pcprogrammer@gmail.com
http://pandamonial.grahamis.com/