Monads for Real People

Will Pleasant-Ryan
Atomic Object
@KreliAnodos

What is a Monad?

- Monads define methods for chaining similar operations together and reducing boilerplate

You're Already Using Monads

- Promise -> Javascript
  - ```
    $.get("/some/resources").then(function (val1) {
        return $.get("/some/resources/" + val1[0]).then(function (val2) {
            console.log("result of /some/resources/val1[0]", val2);
        });
    })
    ```
  - ```
    $.get("/some/resources").then(function (val1) {
        return $.get("/some/resources/" + val1[0]);
    }).then(function (val2) {
        console.log("result of /some/resources/val1[0]", val2);
    })
    ```

You're Already Using Monads

- Async -> C#
  - ```
    public Task<string> DoStuff()
    {
        return Http.Get("/some/resources").ContinueWith(task =>
        {
            if (task.Failure) return task;
            return Http.Get("/some/resources/" + task.Result);

        });
    }
    ```
  - ```
    public async Task<string> DoStuff()
    {
        var result = await Http.Get("/some/resources");
        return await Http.Get("/some/resources/" + result);
    }
    ```

You're Already Using Monads

- Try / ?. -> C#, Ruby
  - C#:
    - ```
      if (foo != null && foo.Bar != null) {
          return foo.Bar.Baz;
      }
      ```
    - `foo?.Bar?.Baz`
  - Ruby:
    - ```
      if foo.present? && foo.bar.present?
        foo.bar.baz
      ```
    - `foo.try(:bar).try(:baz) // try is from Rails`

You're Already Using Monads

- List -> C#, Ruby, jQuery
  - C#:
    - ```
      new [] { 1, 2, 3 }.ForEach(x => {
        yield return x;
        yield return x * 2;
      });
      ```
    - `new [] { 1, 2, 3 }.SelectMany(x => new [] { x, x * 2 });`
  - Ruby:
    - `[1, 2, 3].map {|x| [x, x * 2]}.flatten`
    - `[1, 2, 3].flat_map {|x| [x, x * 2]}`
  - jQuery: `$(".multiple-matches").find(".more-matches");`
      ```

You're Already Using Monads

- The patterns we've just seen are all ad hoc or "one off" attempts to deal with types of computation chaining
- These types of problems are pretty common in programming, so it would be good to have a general solution
- *Monads* are one solution
- *Monads* provide a common pattern for handling computation chaining

Uptown Functor

- A functor generally provides “context” around value types
  - Lists
  - Promises
  - Streams
  - Function objects (e.g. Func<int,T>)
  - Really any type which can support a generic parameter
- Technically, a functor must support *map*:
  - **F<a>** -> **(a -> b)** -> **F<b>**
  - e.g., List<int> -> (int -> string) -> List<string>
    - `new List<int>{ 1, 2, 3 }.Select(i => (i * 2).ToString())`

Call Me Maybe

- **Maybe:** a way of representing a possible value or the absence of a value
  - Similar to Nullable<> generic type in C#
  - in Haskell, **Just 5** or **Nothing**
  - in F#, **Some 5** or **None** (*Maybe* is called the *Option* type in F#)
  - a better way of dealing with *null*

Call Me Maybe

How many times have you wanted to write this:

- ```
  var x = mightReturnNull(1);
  var y = mightAlsoReturnNull(x);
  var z = couldBeNullToo(y);
  return finalComputationOn(x, y, z);
  ```

But to be safe you have to write this:

- ```
  var x = mightReturnNull(1);
  if (x == null) return null;
  var y = mightAlsoReturnNull(x);
  if (y == null) return null;
  var z = couldBeNullToo(y);
  if (z == null) return null;
  return finalComputationOn(x, y, z);
  ```

Call Me Maybe

Can we have all the safety of

- ```
  var x = mightReturnNull(1);
  if (x == null) return null;
  var y = mightAlsoReturnNull(x);
  if (y == null) return null;
  var z = couldBeNullToo(y);
  if (z == null) return null;
  return finalComputationOn(x, y, z);
  ```

without the boilerplate?

Yes!

- ```
  maybe {
    let! x = mightReturnNone 1
    let! y = mightAlsoReturnNone x
    let! z = couldBeNoneToo y
    return finalComputationOn x y z
  }
  ```
- This looks like the original example, but has all the safety of the second - <u>without the clutter</u>
- Many monads are great at drawing out the intent of code and suppressing the necessary “plumbing”

The Choice of a New Generation

- Choice:
  - One of two (or more) values, for example a function might return a data structure or an error message
  - called Choice in F#, Either in Haskell
  - you can think of it like *Option*, but with an error value instead of *None*
  - For our purposes:
    - ```
      public struct Choice<TFail, TSuccess>
      {
          public bool IsSuccess { get; }
          public TSuccess SuccessValue { get; }
          public TFail FailValue { get ; }
      }
      ```
      ```

The Choice of a New Generation

- ```
  public Choice<string, SuccessStructure> DoSomeStuff() {
      Choice<string, int> x = mightFail(1);
      if (!x.IsSuccess) return Choice.Failure<string,SuccessStructure>(x.FailValue);
      Choice<string, bool> y = mightAlsoFail(x);
      if (!y.IsSuccess) return Choice.Failure<string,SuccessStructure>(y.FailValue);
      Choice<string, string> z = couldBeFailureToo(y);
      if (!z.IsSuccess) return Choice.Failure<string,SuccessStructure>(z.FailValue);
      return finalComputationOn(x.SuccessValue,y.SuccessValue,z.SuccessValue);
  }
  ```
- ```
  let doSomeStuff() =
    choice {
      let! x = mightFail 1
      let! y = mightAlsoFail x
      let! z = couldBeFailureToo y
      let! ret = finalComputationOn x y z
      return ret
    }
  ```

Okay, But What's a Monad?

- A Functor can be a monad by supporting two functions
  - bind
  - return
- **return** - take a raw value and put it in a Functor "context"
- **a -> M<a>**
  - return 5 -> Some 5 // Option (Maybe)
  - return "hi" -> Choice2Of2 "hi" // Choice (Either)
  - return true -> [true] // List

Monad is as Monad does

- **bind:** connects one operation to the next
  - very similar to functor map, but where the transform can produce its own context
- **bind: M<a>** -> **(a -> M<b>)** -> **M<b>**
  - all functors involved must be the same type of functor (e.g. List, Option, Choice)
- e.g. **Option<int>** -> **(int -> Option<float>)** -> **Option<float>**
- shorthand operator is **>>=** in Haskell
- So how do we define **bind** for *Option*?
  - *let optionBind (opt:Option<’a>) (func:(’a -> Option<’b>)) : Option<’b> = ???*
    - *’ means generic type in F#*

Monad is as Monad does

- ```
  let optionBind opt funcValToOpt =
    if opt.IsNone then
      None
    else
      funcValToOpt opt.Value
  ```

- ```
  let printIfOdd x =
    if x % 2 = 1 then Some (x.ToString()) else None
  ```

- `optionBind (Some 3) printIfOdd -> Some “3”`
- `optionBind (Some 8) printIfOdd -> None`
- `optionBind None printfIfOdd -> None`

Monad is as Monad does

- Let's return to our Option example:
  - ```
    maybe {
      let! x = mightReturnNone 1
      let! y = mightAlsoReturnNone x
      let! z = couldBeNoneToo y
      return finalComputationOn x y z
    }
    ```
- What does the "unrolled" version look like?
  - ```
    optionBind mightReturnNone(1) (fun x ->
        optionBind mightAlsoReturnNone(x) (fun y ->
            optionBind couldBeNoneToo(y) (fun z ->
                optionReturn (finalComputationOn x y z)
    )))
    ```

I Fought the Law(s) and The Law(s) Won

- Identity laws establish a symmetrical relationship between **bind** and **return**
- Associativity law guarantees predictable composition of operations

I Fought the Law(s) and The Law(s) Won

Left Identity

- **return x bind f** equals **f x**
- given:
  - ```
    let printIfOdd x =
      if (x % 2) == 1 then Some (x.ToString()) else None
    ```
- *return 5 (i.e. Some 5) bind printIfOdd* equals *printIfOdd 5*
- as a counter example, if we
  - `let return v = Some (v+1) // for numeric types`
  - then *return 5 (i.e Some 6) bind printIfOdd* would equal *printIfOdd 6*
    ```

I Fought the Law(s) and The Law(s) Won

Right Identity

- **m bind return** equals **m**
- *Some 5 bind return* equals *Some 5*
- again if we
  - `let return v = Some (v+1) // for numeric types`
- *Some 5 bind return* would equal *Some 6*

I Fought the Law(s) and The Law(s) Won

Associativity - monadic composition

- say we have **a -> m<b>** and **b -> m<c>** which together produce **a -> m<c>**
  - e.g., **int -> Option<string>** and **string -> Option<boolean>** combines to form **int -> Option<boolean>**
    - ```
      let f1 i : string =
        if i > 5 then Some (i.ToString()) else None
      let f2 s : bool =
        if s.Length > 2 Some true else None
      let f1And2 = (fun a -> (f1 a) |> Option.bind f2)
      ```
    - f1And2 is written as **f1 >=> f2**
      ```

I Fought the Law(s) and The Law(s) Won

Associativity

- to satisfy associativity,
  (aToMb >=> bToMc) >=> cToMd is equal to
  aToMb >=> (bToMc => cToMd)
- **>=>** - Kleisli composition of monadic functions
  - *not to be confused with Khaleesi composition*

The State That I Am In

- The state monad allows the consumption / mutation of state to produce a series of outputs and a final state
- Example: given random number generation *randGen*
  - *maxValue -> seed -> randomValue * newSeed*
    - *A * B* means a tuple of A and B
    - because our data is immutable, we will output a new seed value with each invocation

The State That I Am In

- Let's say we need three random values with different bounds
  - ```
    let rand1, seed1 = randGen 10 initialSeed
    let rand2, seed2 = randGen 25 seed1
    let rand3, _ = randGen 400 seed2  // we don't need the final seed
    [ rand1 ; rand2 ; rand3 ]
    ```
- Using the state monad:

```
state {
  let! rand1 = randGen 10
  let! rand2 = randGen 25
  let! rand3 = randGen 400
  return [rand1 ; rand2 ; rand3]
} initialSeed
```
```

The State That I Am In

- Let’s take a closer look at what is happening here:

```
state {
  let! rand1 = randGen 10
  let! rand2 = randGen 25
  let! rand3 = randGen 400
  return [rand1 ; rand2 ; rand3]
} initialSeed
```

- You’ll notice that if randGen is
  - `let randGen (maxValue: int) -> (seed:int) -> double * int`
  - we did not provide the final argument in the monad expression
  - bound expressions always produce the monadic type

The State That I Am In

- The state monad is a function!
  - **State<T> = state -> T*state**
  - state builds a pipeline somewhat similar to Kleisli composition
  - state value is replaced continuously as it threads through pipeline
  - `let return v = (fun s -> v*s)`
  - ```
    let stateBind m:('s -> 'a*'s) f:('a -> ('s -> 'b*'s)) : ('s -> 'b*'s) =
      (fun origState ->
        let value, newState = m origState
        f value newState
      )
    ```
  - ```
    // randGen(10) will receive initialSeed
    stateBind randGen(10) (fun rand1 ->
        // randGen(25) will receive seed1 from randGen 10 initialSeed
        stateBind randGen(25) (fun rand2 ->
            // randGen(400) will receive seed2 from randGen 25 seed1
            stateBind randGen(400) (fun rand3 ->
                stateReturn [rand1 ; rand2 ; rand3]
    ))) initialSeed
    ```

The State That I Am In

- Using partial application we can produce a State<T> function from a normal function
  - `let stateBind m:(s -> a*s) f:(a -> (s -> b*s)) : (s -> b*s)`
  - note that you can normally define your state functions **a -> (s -> b*s)** as **a -> s -> b*s** (i.e. two arguments and a tuple for an output)
  - that is, this declaration:
    - `let doStuff x = (fun y -> …)`
  - *equals* this declaration:
    - `let doStuff x y = ...`

The More You Know (Reader)

- What if we need to have access to “global” data that doesn’t change?
- **env -> T**
- lookup: name -> phonebook -> number
- reader {
```
  let! num1 = lookup “Joe”
  let! num2 = lookup “Jenny”
  let! num3 = lookup “Phil”
  return [num1 ; num2 ; num3]
} phonebook
```
- `let return v = (fun env -> v)`
- `let readerBind m:(‘env -> ‘a) f:(‘a -> (‘env -> ‘b)) : (‘env -> ‘b) =`
  - ```
    (fun env ->
        let aVal = m env
        f aVal
    )
    ```
    ```

Lists (or, the world is flat)

- List:
  - **bind:** List<T> -> (T -> List<U>) -> List<U>
  - flat_map / SelectMany
  - can be chained repeatedly:
  - list {

```
    let! eachElem1 = generateAList
    let! eachElem2 = aListForEach eachElem
    let! eachElem3 = anotherListForEach eachElem2
    // if not for monads, we would have a List<List<List<x>>>
    return (eachElem3 * 10)
}
```

Asynchronicity

- Async:
  - **bind:** Async<T> -> (T -> Async<U>) -> Async<U>
  - C# 4.5’s *async*
  - Javascript Promise chaining
  - ```
    let downloadAndExtractLinks url =
        async {
            let webClient = new System.Net.WebClient()
            let html = webClient.DownloadString(url : string)
            let! links = extractLinksAsync html
            return url, links.Count
        }
    ```
    ```

Mixing It Up: Deserialization

- Fundamentally, deserialization is
  - sequence of bytes -> DataStructure
  - for these examples, let's assume DataStructure =

```
{
  Foo : int
  Bar : string
  Baz : Point
}
```

- consuming the bytes as we read them sounds like a job for "state"

Mixing It Up: Deserialization

- assume *unpack<T>: byte[] -> Tuple(T * byte[])*
  - ```
    state {
      let! fooValue = unpack
      let! barValue = unpack
      let! bazValue = unpack
      return {
        Foo = fooValue
        Bar = barValue
        Baz = bazValue
      }
    } initialBytes
    ```
- notice the high degree of type inference
    ```

Mixing It Up: Deserialization

- What if something goes wrong?
  - assume `unpack<T>: byte[] -> Choice<Tuple(T * byte[]),Error>`
  - ```
    protectedState {
      let! fooValue = unpack
      let! barValue = unpack
      let! bazValue = unpack
      return {
        Foo = fooValue
        Bar = barValue
        Baz = bazValue
      }
    } initialBytes
    ```
    ```

Mixing It Up: Deserialization

- What if unpack requires a lookup table for deserializing subtypes?
  - assume *unpack<T>: table -> byte[] -> Choice<Tuple(T * byte[]),Error>*
  -
```
readerProtectedState {
  let! fooValue = unpack
  let! barValue = unpack
  let! bazValue = unpack
  return {
    Foo = fooValue
    Bar = barValue
    Baz = bazValue
  }
} table initialBytes
```

(safe) Magic!

Real World Example 1: Multi-Layered Functor Context

- Imagine that a program maintains many simultaneous communication channels, and logs each channel into its own file
- Each log file contains several lines
- Each line contains a timestamp and a JSON payload
- *grep*'ing to find interesting data works, but it has limitations
- merely extracting matching substrings can make it easy to lose context

Real World Example 1: Multi-Layered Functor Context

- instead, use a functor Log<data>
- transforms that do functor mapping will maintain filename and timestamps
  - **filter**: keep all lines where payload matches filter
  - **timespan**: keep all times in time window
  - **select**: JSON path transform of payload
  - **merge**: combine two Logs
  - **mergeSingleton**: intersperse a Log file into other Log files
- also support writing out altered Log to a directory
- https://github.com/willryan/TimestampedJsonLogFilter

Real World Example 2: Unhappy Paths

- Imagine going through a long, asynchronous process of building up a complicated data structure
- failures could happen at multiple points
- in C#, use a Task<Choice<success,error>> return type
- C# helps you with async syntactic sugar, but you’re on your own for Choice type
    - I did add a “BindFail” for my choice type, for the rare case where you want to do something interesting on the fail case
        - technically not monadic, but oh well

Real World Example 3: Stateful Reboot

- For another case, imagine code which must reboot multiple devices in a complicated order
  - rebooting is asynchronous
  - order is complex and next steps require analysis of current state
  - sounds like a job for async + state!
- State object has fluent methods for transform
  - indicate device was rebooted
    - `var state3 = state2.TriedReboot(new [] { device });`
  - add devices expected to be present after reboot
  - add which devices have been re-discovered after a reboot
    - `var state4 = state3.AddReady(newDevices);`
  - track number of tries to reboot a particular device
- I called Bind() “ThenWith()” to simplify nomenclature
- Still looks like callback hell in C# though

Bonus: CSharpMonad!

- Use *LINQ* syntatic sugar to achieve “pretty” monads in C#
- Remember this?
  - ```
    public Choice<SuccessStructure, string> DoSomeStuff() {
        int x = mightFail(1);
        if (x.Fail) return Choice.Failure<SuccessStructure,string>(x.FailValue);
        bool y = mightAlsoFail(x);
        if (y.Fail) return Choice.Failure<SuccessStructure,string>(y.FailValue);
        string z = couldBeFailureToo(y);
        if (z.Fail) return Choice.Failure<SuccessStructure,string>(z.FailValue);
        return finalComputationOn(x,y,z);
    }
    ```
- How about:
  - ```
    public Either<Output,string> doSomeStuff()
    {
        from x in mightFail(1)
        from y in mightAlsoFail(x)
        from z in couldBeFailureToo(y)
        select finalComputation(x,y,z);
    }
    ```

Bonus: CSharpMonad!

- Because partial application is not available, state functions must be declared like
  - `public Func<state,Tuple<U,state>> stateFunc(T,OtherInputs)`
  - but you can write

    ```
    public Tuple<U,state> stateFunc(T t, OtherInputs o, state s) and
    Func<state,Tuple<U,state>> stateFuncPrime(T t, OtherInputs o) {
        return (st => stateFunc(t, o, st));
    }
    ```
  - or use a technique such as in MethodToDelegate :)

Resources / Further Reading

https://github.com/willryan/MonadsForRealPeople

http://learnyouahaskell.com/

http://fsharpforfunandprofit.com/posts/monadster/

```haskell
main = do
  questions <- getLine
  putStrLn "Thanks!"
```