Monads

Prof. Clarkson
Fall 2018

Today’s music: Vámanos Pal Monte by Eddie Palmieri
Attendance question

Have you programmed with monads in Haskell?

A. No
B. Yes
C. Yes, and I've written a monad tutorial
Review

Currently in 3110: Advanced data structures
• Streams
• Balanced trees
• Mutability
• Promises

Today:
• Monads
 Monad tutorials

Amount of known monad tutorials

source: https://wiki.haskell.org/Monad_tutorials_timeline

since 2011: another 34 at least
Monad tutorials

"A monad is a monoid object in a category of endofunctors....It might be helpful to see a monad as a lax functor from a terminal bicategory."
Monad tutorial

"A monad is a monoid object in a category of endofunctors....It might be helpful to see a monad as a lax functor from a terminal bicategory."
Monad tutorials

"A monad is a monoid object in a category of endofunctors....It might be helpful to see a monad as a lax functor from a terminal bicategory."

"Monads are burritos." [http://chrisdone.com/posts/monads-are-burritos]
Monad

For our purposes:

```ocaml
module type Monad = sig
  type 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
  val return : 'a -> 'a t
end
```

Any structure that implements the Monad signature is a monad.

What's the big deal???
LOGGABLE FUNCTIONS
Question

let inc_log x =
  (x+1, "incremented " ^ string_of_int x ^ "); ")
let dec_log x =
  (x-1, "decremented " ^ string_of_int x ^ ""); ")

let id_log = inc_log >> dec_log

Why doesn’t that definition work?
A. It doesn’t type check
B. It computes the wrong integer
C. It computes the wrong log message
D. Both B and C
LOGGABLE FUNCTIONS
Upgrading a function

What if we could upgrade a loggable function to accept the input from another loggable function?

upgrade f_log :
    int*string -> int*string

Discussion: how could you implement that?
Another kind of upgrade

• Given $f : \text{int} \to \text{int}$

• How to make it loggable, but with empty log message?

• Need to "lift" a function
  from $\text{int} \to \text{int}$
  to $\text{int} \to \text{int}*\text{string}$
Types

Consider the types:

```plaintext
val upgrade :
    (int -> int * string)
    -> int * string -> int * string

val trivial :
    int -> (int * string)
```
Types

Another way of writing those types:

```fsharp
type 'a t = 'a * string

val upgrade :
    (int  ->  int  t)
  ->  int  t  ->  int  t

val trivial :
    int  ->  int  t
```

Have you seen those types before???
Types

Let’s swap the argument order of upgrade...

```ml
val upgrade :  
  (int -> int t)  
  -> int t  
  -> int t

let upgrade' x f = upgrade f x

val upgrade' :  
  int t  
  -> (int -> int t)  
  -> int t
```
Types

type 'a t = 'a * string

val upgrade' :  
    int t
    -> (int -> int t)
    -> int t

val trivial :  
    int -> int t

Have you seen those types before?
Rewriting types

type 'a t = 'a * string

val bind :
  int t
  -> (int -> int t)
  -> int t

val return :
  int -> int t
Loggable is a monad

```ocaml
module Loggable : Monad = struct
  type 'a t = 'a * string
  let bind (x,s1) f =
    let (y,s2) = f x in
    (y,s1^s2)
  let return x = (x,"")
end

More often called the writer monad
```
Stepping back...

• We took functions
• We made them compute *something more*
  – A logging string
• We invented ways to pipeline them together
  – upgrade, trivial
• We discovered those ways correspond to the Monad signature
FUNCTIONS THAT PRODUCE ERRORS
Functions and errors

A *partial* function is undefined for some inputs

- e.g., `max_list : int list -> int`

- with that type, programmer probably intends to raise an exception on the empty list
  - could also produce an option
  - or could use variant to encode result...
What are the types?

type 'a t = Val of 'a | Err
val value : 'a -> 'a t
val (|>?): 'a t -> ('a -> 'b t) -> 'b t

Have you seen those types before???
Error is a monad

```ocaml
module Error : Monad = struct
  type 'a t = Val of 'a | Err
  let return x = Val x
  let bind m f =
    match m with
    | Val x -> f x
    | Err -> Err
end
```
Option is a monad

```ocaml
module Option : Monad = struct
  type 'a t = Some of 'a | None
  let return x = Some x
  let bind m f =
    match m with
    | Some x -> f x
    | None -> None
end
```
Stepping back...

- We took functions
- We made them compute *something more*
  - A value or possibly an error
- We invented ways to pipeline them together
  - `value, (|>?)`
- We discovered those ways correspond to the *Monad* signature
Lwt is a monad

module Lwt : sig

  type 'a t

  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t

end

• return takes a value and returns an immediately resolved promise
• bind takes a promise, and a callback function, and returns a promise that results from applying the callback
Stepping back...

• We took functions
• The Lwt library made them compute *something more*  
  – a promised result
• The Lwt library invented ways to pipeline them together  
  – `return`, `(>>=)`
• Those ways correspond to the `Monad` signature
• So we call Lwt a *monadic concurrency library*
Another view of Monad

```ocaml
module type Monad = sig
    (* a "boxed" value of type 'a *)
    type 'a t

    (* [m >>= f] unboxes m, *
     * passes the result to f, *
     * which computes a new result, *
     * and returns the boxed new result *)
    val (>>=) : 'a t -> ('a -> 'b t) -> 'b t

    (* box up a value *)
    val return : 'a -> 'a t
end
```

(equate "box" with "tortilla" and you have the burrito metaphor)
SO WHAT IS A MONAD?
Computations

• A *function* maps an input to an output
• A *computation* does that and more: it has some *effect*
  – Loggable computation: effect is a string produced for logging
  – Error computation: effect is a possible error instead of a value
  – Option computation: effect is a possible None instead of a value
  – Promised computation: effect is delaying production of value until later
• A *monad* is a data type for computations
  – *return* has the trivial effect
  – (*>>=*) does the "plumbing" between effects
Phil Wadler

- A designer of Haskell
- Wrote *the paper* on using monads for functional programming

b. 1956

Other monads

- **State**: modifying the state is an effect
- **List**: producing a list of values instead of a single value can be seen as an effect
- **Random**: producing a random value can be seen as an effect
- ...
Monad laws

- We expect data types to obey some algebraic laws
  - e.g., for stacks, \texttt{peek (push x s)} = x
  - We don't write them in OCaml types, but they're essential for expected behavior

- Monads must obey these laws:
  1. \texttt{return x >>= f} is equivalent to \texttt{f x}
  2. \texttt{m >>= return} is equivalent to \texttt{m}
  3. \texttt{(m >>= f) >>= g} is equivalent to \texttt{m >>= (fun x \rightarrow f x >>= g)}

- Why? The laws make sequencing of effects work the way you expect
Monad laws

1. \((\text{return } x >>= f) = f x\)
   Doing the trivial effect then doing a computation \(f\) is the same as just doing the computation \(f\)
   \((\text{return is left identity of bind})\)

2. \((m >>= \text{return}) = m\)
   Doing only a trivial effect is the same as not doing any effect
   \((\text{return is right identity of bind})\)

3. \(((m >>= f) >>= g) \quad = \quad (m >>= (\text{fun } x \rightarrow f x >>= g))\)
   Doing \(f\) then doing \(g\) as two separate computations is the same as doing a single computation which is \(f\) followed by \(g\)
   \((\text{bind is associative})\)
Upcoming events

• N/A

This is effectful.

THIS IS 3110