Monads

Prof. Clarkson
Fall 2017

Today’s music: *Vámanos Pal Monte* by Eddie Palmieri
Review

Currently in 3110: Advanced topics
• Futures: Async: deferreds, return, bind

Today:
• Monads
Monad tutorials

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

since 2011: another 34 at least
Question

Have you programmed with monads in Haskell?

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

Suppose you're implementing two functions:

- \( f : \text{int} \rightarrow \text{int} \)
- \( g : \text{int} \rightarrow \text{int} \)

And you'd like to compute their \textit{composition}:

\[
\text{let } h \ x = g \ (f \ x) \quad (* = x \ |> f \ |> g *)
\]

\[
\text{let } (>>) \ f \ g \ x = x \ |> f \ |> g
\]

\[
\text{let } h \ x = (f \ >> g) \ x
\]

\[
\text{let } h = f \ >> g
\]
Loggable functions

You’d like also **log** some additional information each time function is called:

- **f_log**: `int -> int * string`
- **g_log**: `int -> int * string`
Loggable functions

```plaintext
let inc x = x+1
let dec x = x-1
let id = inc >> dec

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 *)
```

Q: Why doesn’t that work?
A: `dec_log` takes an `int` as input not an `int * string`
Loggable functions

```plaintext
let id_log x =
    let (y,s1) = inc_log x in
    let (z,s2) = dec_log y in
    (z,s1^s2)
```

Critique:
• Hard to infer from that code that it's doing composition!
• Ugly and verbose compared to
  ```plaintext
  let id_log = inc_log >> dec_log
  ```
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
Upgrading a function

```ocaml
let upgrade f_log (x,s1) =
  let (y,s2) = f_log x in
  (y,s1^s2)

let id_log =
  inc_log >>= upgrade dec_log
```

Nice separation of concerns!
- `upgrade` handles the "plumbing" with the strings
- the definition of `id_log` is clearly about composition
Another kind of upgrade

• Given \( f : \text{int} \rightarrow \text{int} \)
• How to make it loggable, but with empty log message?
• Need to "lift" a function from \( \text{int} \rightarrow \text{int} \) to \( \text{int} \rightarrow \text{int} \ast \text{string} \)
• That's easy:

\[
\text{let trivial } x = (x, "")
\]
\[
\text{let lift } f = f >> \text{trivial}
\]
Types

Consider the types:

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

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

Another way of writing those types:

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

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

```ocaml
val trivial :  
    int -> int t
```
Types

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

```ocaml
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

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

• A4: you implemented an interpreter
  – Results could be either values or exceptions
  – So evaluation produced a variant with constructor for either possibility

• A partial function (in math) is undefined for some inputs
  – e.g., \texttt{max_list : int list \rightarrow int}
  – with that type, programmer probably intends to raise an exception on the empty list
    • could also produce an option
    • or like A4, could use variant to encode result...
A type for possible errors

type 'a t = Val of 'a | Err

let div (x:int) (y:int) =
  if y=0 then Err
  else Val (x / y)

let neg (x:int) = Val (-x)
Error handling

Lifting those function to handle inputs that might be errors...

```ocaml
let neg_err = function
| Err    -> Err
| Val x  -> Val (-x)
```

```ocaml
let div_err x y =
  match (x,y) with
  | (Err,_) | (_,Err)    -> Err
  | (Val a,Val b)    -> if b=0 then Err else Val (a/b)
```

And any other functions you write have to pattern match to handle errors... Could we get rid of all that boilerplate pattern matching?
Eliminating boilerplate matching

(* [rev_app_err m f] applies f to m, like [x |> f], but handling Err as necessary. *)

let rev_app_err m f =
  match m with
  | Val x -> f x
  | Err -> Err

let (|>?) = rev_app_err
Eliminating boilerplate matching

let neg_err = function
    | Err -> Err
    | Val x -> Val (-x)

let neg_err x =
    x |->? (fun a -> Val (-a))
Eliminating boilerplate matching

```oca
define div_err (x : T a, y : T b)
  let div_err x y =
    match (x,y) with
    | (Err,_) | (_,Err) -> Err
    | ( Val a, Val b ) ->
      if b=0 then Err else Val (a/b)
  
define div_err (x : T a, y : T b)
  x |> fun a ->
  y |> fun b ->
  if b=0 then Err else Val (a/b)
```
Another way to write that code

```ocaml
let value x = Val x

let neg_err x =
    x |?>? fun a ->
    value (-a)

let div_err x y =
    x |?>? fun a ->
    y |?>? fun b ->
    if b=0 then Err else value (a/b)
```
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
ASYNC
**Deferred is a monad**

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

- `return` takes a value and returns an immediately determined deferred
- `bind` takes a deferred, and a function from a non-deferred to a deferred, and returns a deferred that result from applying the function
Stepping back...

- We took functions
- The Async library made them compute *something more*
  - a deferred result
- The Async library invented ways to pipeline them together
  - `return`, `(>>=)`
- Those ways correspond to the **Monad** signature
- So we call Async 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
  – Deferred computation: effect is delaying production of value until scheduler makes it happen

• 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

• As you've seen in Coq, data types must 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} >>= \texttt{return} is equivalent to \texttt{m}
  3. \texttt{(m >>= f) >>= g} is equivalent to \texttt{m >>= (fun x -> f x >>= g)}

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

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

3. \(( (m \gg= f) \gg= g) = (m \gg= (\text{fun } x \rightarrow f x \gg= g))\)
   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

- Happy Thanksgiving Break!
- [Wed and Thur] no class
- [Mon] recitations do meet

This is effectful.

THIS IS 3110