Monads

Prof. Clarkson
Fall 2015

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

Previously in 3110:
• Imperative programming
• Async: deferreds, return, bind

Today:
• Monads
Monad tutorials

since 2011: another 34 at least

source: https://wiki.haskell.org/Monad_tutorials_timeline
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: a monad is a data type:

```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. (Just like any structure that implements the Queue signature is a queue, etc.)

What's the big deal???
DEBUGGABLE FUNCTIONS
Recall A1

cipher : string -> ... -> string
simulate : string -> ... -> unit

What was the difference between these two functions?

• **cipher** returned the ciphered string and didn't print anything

• **simulate** printed debugging output but didn't print the ciphered string
Recall A1

cipher : string \rightarrow \ldots \rightarrow \text{string}
simulate : string \rightarrow \ldots \rightarrow \text{unit}

If you want to avoid duplicating code, one possibility would be to implement a helper:

\text{h} : \text{string} \rightarrow \ldots \rightarrow \text{string*string}

• In the pair \text{h} returns:
  – the first string would be the enciphered message
  – the second string would be the debug output as a single string
• \text{cipher} would ignore the second string and return the first
• \text{simulate} would ignore the first string and print the second
Debuggable functions

Suppose you're implementing two helper functions:

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

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

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

But your implementations have bugs, so you'd like to make them debuggable:

- \( \text{fd}: \text{int} \rightarrow \text{int} \times \text{string} \)
- \( \text{gd}: \text{int} \rightarrow \text{int} \times \text{string} \)

And you'd like to debug their composition:

```plaintext
let hd x = ???
(\* NOT: x |> fd |> gd \*)
```

Q: Why not?

A: \( \text{gd} \) takes an \text{int} as input not an \text{int} \times \text{string}
Debuggable functions

```plaintext
let hd x =
  let (y,s1) = fd x in
  let (z,s2) = gd y in
  (z,s1^s2)
```

Critique:
• Hard to infer from that code that it's doing composition!
• Ugly compared to
  ```plaintext
  let h x = x |> f |> g
  ```
Upgrading a function

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

```plaintext
upgrade gd :
  int*string -> int*string
```

How would you implement `upgrade`?
Upgrading a function

```ocaml
define upgrade f (x,s1) = 
  let (y,s2) = f x in 
  (y,s1^s2)
define hd x = x |> fd |> upgrade gd
```

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

• Suppose we have a function \( e : \text{int} \rightarrow \text{int} \) that we want to include in a debuggable pipeline of functions, but we're not interested in debugging \( e \) itself
  
  – won't typecheck: 
    \[ x \rightarrow \text{fd} \rightarrow e \rightarrow \text{upgrade} \ \text{gd} \]
  
  – won't typecheck:
    \[ x \rightarrow \text{fd} \rightarrow \text{upgrade} \ e \rightarrow \text{upgrade} \ \text{gd} \]

• We need a way to "lift" a function from \( \text{int} \rightarrow \text{int} \) to \( \text{int} \rightarrow \text{int}*\text{string} \)
Another kind of upgrade

That's easy:

```haskell
let trivial x = (x, "")
let lift f x = x |> f |> trivial
```

Now we can write:

```haskell
x |> fd
    |> upgrade (lift e)
    |> upgrade gd
```
Upgrades

Consider the types of two of our upgrade functions:

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

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

Another way of writing those types:

```ml
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???
Rewriting types

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

let upgrade' m f = upgrade f m
val upgrade' :
  int t
  -> (int -> int t)
  -> int t

val trivial :
  int -> int t
```

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

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
Debuggable is a monad

```ocaml
module Debuggable : 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
```
Stepping back...

• We took functions
• We made them compute *something more*
  – A debug string
• We invented ways to pipeline them together
  – *upgrade, trivial*
• We discovered those ways correspond to the *Monad* signature
FUNCTIONS THAT PRODUCE ERRORS
Consider A4

- You're implementing an interpreter
- The type for values contains VError
- But suppose it didn't...
  - then sometimes eval would get stuck and be unable to produce a value
    - e.g., eval "1/0"
  - one way to handle this would be to return a variant: one constructor for real values, another constructor for errors
Consider partial functions

• A partial function (in math) is undefined for some inputs
  – e.g., `max_list : int list → int`
  – what should it do for empty list?
  – could produce an error
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...

```plaintext
let neg' = function
  | Err -> Err
  | Val x -> Val (-x)

let div' x y =
  match (x,y) with
  | (Err,_) | (_,Err) -> Err
  | (Val a,Val b) -> 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

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

let (|>?) = rev_app_noerr

let neg' x =
    x |?> fun a -> Val (-a)

let div' x y =
    x |?> fun a -> (y |?> fun b -> Val (a / b))
Another way to write that code

```
let ( |>?) = rev_app_noerr
let value x = Val x

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

let div' x y =
  x |>?
  fun a ->
  y |>?
  fun b ->
  value (a / b)
```
What are the types?

```ocaml
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???
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
datatype 'a t = Some of 'a | None

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
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

```
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
  – Debuggable computation: effect is a string produced for examination
  – 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
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

- Every data type obeys some algebraic laws
  - e.g., for stacks, `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. `return x >>= f` is equivalent to `f x`
  2. `m >>= return` is equivalent to `m`
  3. `(m >>= f) >>= g` is equivalent to `m >>= (fun x -> f x >>= g)`

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

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

3. \( (m >>= f) >>= g \) is equivalent to
   \( m >>= (\text{fun } x \rightarrow f \ x >>= g) \)
   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

• [today] A4 due
• [Monday] project charter due

This is monadic.

THIS IS 3110