Monads

CS3100 Fall 2019

Review

Previously

- Streams, laziness and memoization

This lecture

- Monads
  - Dealing with **effects** in a **pure** setting

Whence Monads

- The term "monad" come from **Category Theory**
  - Category theory is the study of mathematical abstractions
  - Out of scope for this course
  - We will focus on **programming with monads.**
- Monads were popularized by the Haskell programming language
  - Haskell is **purely functional** programming languages
  - Unlike OCaml, Haskell separates pure code from side-effecting code through the use of monads.

What is a Monad?

A monad is any implementation that satisfies the following signature:

In [57]:

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

Out[57]:

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

and the **monad laws**.

Example: Interpreter

- All of this seems very abstract (as many FP concepts are).
  - Monad is a design pattern rather than a language feature.
- An example will help us see the pattern.
  - Overtime, you'll spot monads everywhere.
- Let's write an interpreter for artihmetic expressions

Interpreting artihmetic expressions

In [58]:

```
type expr = Val of int | Plus of expr * expr | Div of expr * expr
```

Out[58]:

```
type expr = Val of int | Plus of expr * expr | Div of expr
* expr
```

- Our goal is to make the interpreter a **total function**.
  - Produces a **value** for every arithmetic expression.

In [59]:

```
let rec eval e = match e with
  | Val v -> v
  | Plus (v1,v2) -> eval v1 + eval v2
  | Div (v1,v2) -> eval v1 / eval v2
```

Out[59]:

```
val eval : expr -> int = <fun>
```

Division by zero

This looks fine. But what happens if the denominator in the division is a 0.

In [60]:

```
eval (Div (Val 1, Val 0))
```

```
Exception: Division_by_zero.
Raised by primitive operation at unknown location
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

How can we avoid this?

Interpreting Arithmetic Expressions: Take 2

- Rewrite `eval` function to have the type `expr -> int option`
  - Return `None` for division by zero.

In [61]:

```
let rec eval e = match e with
  | Val v -> Some v
  | Plus (e1,e2) ->
      begin match eval e1 with
      | None -> None
      | Some v1 ->
          match eval e2 with
          | None -> None
          | Some v2 -> Some (v1 + v2)
      end
  | Div (e1,e2) ->
      match eval e1 with
      | None -> None
      | Some v1 ->
          match eval e2 with
          | None -> None
          | Some v2 -> if v2 = 0 then None else Some (v1 / v2)
```

Out[61]:

```
val eval : expr -> int option = <fun>
```

In [62]:

```
eval (Div (Val 1, Val 0))
```

Out[62]:

```
- : int option = None
```

Abstraction

- There is a lot of repeated code in the interpreter above.
  - Factor out common code.

In [63]:

```
let return v = Some v
```

Out[63]:

```
val return : 'a -> 'a option = <fun>
```

In [64]:

```
let bind m f = match m with
  | None -> None
  | Some v -> f v
```

Out[64]:

```
val bind : 'a option -> ('a -> 'b option) -> 'b option = <f
un>
```

Abstraction

Let's rewrite the interpreter using these functions.

In [65]:

```
let rec eval e = match e with
  | Val v -> return v
  | Plus (e1,e2) ->
      bind (eval e1) (fun v1 ->
      bind (eval e2) (fun v2 ->
      return (v1+v2)))
  | Div (e1,e2) ->
      bind (eval e1) (fun v1 ->
      bind (eval e2) (fun v2 ->
      if v2 = 0 then None else return (v1 / v2)))
```

Out[65]:

```
val eval : expr -> int option = <fun>
```

Infix bind operation

Usually `bind` is defined as an infix function `>>=`.

In [66]:

```
let (>>=) = bind
```

Out[66]:

```
val ( >>= ) : 'a option -> ('a -> 'b option) -> 'b option =
<fun>
```

In [67]:

```
let rec eval e = match e with
  | Val v -> return v
  | Plus (e1,e2) ->
      eval e1 >>= fun v1 ->
      eval e2 >>= fun v2 ->
      return (v1+v2)
  | Div (e1,e2) ->
      eval e1 >>= fun v1 ->
      eval e2 >>= fun v2 ->
      if v2 = 0 then None else return (v1 / v2)
```

Out[67]:

```
val eval : expr -> int option = <fun>
```

Modularise

- The `return` and `>>=` we have defined for the interpreter works for any computation on option type.
  - Put them in a module, we get the Option Monad.

In [68]:

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

module OptionMonad : (MONAD with type 'a t = 'a option) = struct
  type 'a t = 'a option
  let return v = Some v
  let (>>=) m f = match m with
  | Some v -> f v
  | None -> None
end
```

Out[68]:

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

Out[68]:

```
module OptionMonad :
  sig
    type 'a t = 'a option
    val return : 'a -> 'a t
    val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
  end
```

Monad Laws

Any implementation of the monad signature must satisfy the following laws:

```
1. return v >>= k  ≡  k v (* Left Identity *)
2. v >>= return  ≡  v (* Right Identity *)
3. (m >>= f) >>= g  ≡  m >>= (fun x -> f x >>= g) (* Associativity *)
```

Option monad satisifies monad laws

Left Identity: `return v >>= k  ≡  k v`

```
  return v >>= k
≡ (Some v) >>= k (* by definition of return *)
≡ match Some v with None -> None | Some v -> k v (* by definitio
n of >>= *)
≡ k v (* by beta reduction *)
```

Exercice: Prove other laws.

State Monad

- Each monad implementation typically extends the signature with additional operations.
- A State Monad introduces a **single, typed mutable cell.**
- Here's a signature for dealing with mutable state, which adds
  - `get` and `put` functions for reading and writing the state, and
  - a `runState` function for actually running computations.

In [69]:

```
module type STATE = sig
  type state
  include MONAD
  val get : state t
  val put : state -> unit t
  val run_state : 'a t -> init:state -> state * 'a
end
```

Out[69]:

```
module type STATE =
  sig
    type state
    type 'a t
    val return : 'a -> 'a t
    val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
    val get : state t
    val put : state -> unit t
    val run_state : 'a t -> init:state -> state * 'a
  end
```

State Monad

Here's an implementation of `State`, parameterised by the type of the state:

In [70]:

```
module State (S : sig type t end)
  : STATE with type state = S.t = struct
  type state = S.t
  type 'a t = state -> state * 'a
  let return v = fun s -> (s, v)
  let (>>=) m f = fun s ->
    let (s', a) = m s in
    f a s'
  let get s = (s, s)
  let put s' _ = (s', ())
  let run_state m ~init = m init
end
```

Out[70]:

```
module State :
  functor (S : sig type t end) ->
    sig
      type state = S.t
      type 'a t
      val return : 'a -> 'a t
      val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
      val get : state t
      val put : state -> unit t
      val run_state : 'a t -> init:state -> state * 'a
    end
```

Using State Monad

In [71]:

```
module IntState = State (struct type t = int end)
open IntState

let inc v =
  get >>= fun s ->
  put (s+v)

let dec v =
  get >>= fun s ->
  put (s-v)

let double =
  get >>= fun s ->
  put (s*2)
```

Out[71]:

```
module IntState :
  sig
    type state = int
    type 'a t
    val return : 'a -> 'a t
    val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
    val get : state t
    val put : state -> unit t
    val run_state : 'a t -> init:state -> state * 'a
  end
```

Out[71]:

```
val inc : int -> unit IntState.t = <fun>
```

Out[71]:

```
val dec : int -> unit IntState.t = <fun>
```

Out[71]:

```
val double : unit IntState.t = <abstr>
```

Using State Monad

In [72]:

```
IntState.run_state ~init:10 (
  inc 5 >>= fun () ->
  dec 10 >>= fun () ->
  double)
```

Out[72]:

```
- : IntState.state * unit = (10, ())
```

In [73]:

```
let module FloatState = State (struct type t = float end) in
let open FloatState in
FloatState.run_state ~init:5.4 (
  get >>= fun v ->
  put (v +. 1.0))
```

Out[73]:

```
- : float * unit = (6.4, ())
```

State monad satisfies monad laws

Right Associativity: `v >>= return  ≡  v`

```
    v >>= return
  ≡ fun s -> let (s', a) = v s in return a s' (* by definition of
 >>= *)
  ≡ fun s -> let (s', a) = v s in (fun v s -> (s,v)) a s' (* by de
finition of return *)
  ≡ fun s -> let (s', a) = v s in (s',a) (* by beta reduction *)
  ≡ fun s -> (fun (s', a) -> (s', a)) (v s) (* rewrite `let` to `f
un` *)
  ≡ fun s -> v s (* by eta reduction *)
  ≡ v
```

Exercise: Prove other laws.

Type of State

- State in the state monad is of a single type
  - In our example, the state was of `int` type
- *Can we change type of state as the computation evolves?*

Parameterised monads

- Parameterised monads add two additional type parameters to `t` representing the start and end states of a computation.
- A computation of type `('p, 'q, 'a) t` has
  - *precondition* (or starting state) `'p`
  - *postcondition* (or ending state) `'q`
  - *produces a result* of type `'a`.

i.e. `('p, 'q, 'a) t` is a kind of Hoare triple `{P} M {Q}`.

Parameterised monads

Here's the parameterised monad signature:

In [74]:

```
module type PARAMETERISED_MONAD =
sig
  type ('s,'t,'a) t
  val return : 'a -> ('s,'s,'a) t
  val (>>=) : ('r,'s,'a) t ->
       ('a -> ('s,'t,'b) t) ->
              ('r,'t,'b) t
end
```

Out[74]:

```
module type PARAMETERISED_MONAD =
  sig
    type ('s, 't, 'a) t
    val return : 'a -> ('s, 's, 'a) t
    val ( >>= ) : ('r, 's, 'a) t -> ('a -> ('s, 't, 'b) t)
-> ('r, 't, 'b) t
  end
```

Parameterised state monad

Here's a parameterised monad version of the `STATE` signature, using the extra parameters to represent the type of the reference cell.

In [75]:

```
module type PSTATE =
sig
 include PARAMETERISED_MONAD
 val get : ('s,'s,'s) t
 val put : 's -> (_,'s,unit) t
 val runState : ('s,'t,'a) t -> init:'s -> 't * 'a
end
```

Out[75]:

```
module type PSTATE =
  sig
    type ('s, 't, 'a) t
    val return : 'a -> ('s, 's, 'a) t
    val ( >>= ) : ('r, 's, 'a) t -> ('a -> ('s, 't, 'b) t)
-> ('r, 't, 'b) t
    val get : ('s, 's, 's) t
    val put : 's -> ('a, 's, unit) t
    val runState : ('s, 't, 'a) t -> init:'s -> 't * 'a
  end
```

Parameterised state monad

Here's an implementation of `PSTATE` .

In [76]:

```
module PState : PSTATE =
struct
  type ('s, 't, 'a) t = 's -> 't * 'a
  let return v s = (s, v)
  let (>>=) m k s = let t, a = m s in k a t
  let put s _ = (s, ())
  let get s = (s, s)
  let runState m ~init = m init
end
```

Out[76]:

```
module PState : PSTATE
```

Computation with changing state

In [77]:

```
open PState

let inc v = get >>= fun s -> put (s+v)
let dec v = get >>= fun s -> put (s-v)
let double = get >>= fun s -> put (s*2)

let to_string = get >>= fun i -> put (string_of_int i)
let of_string = get >>= fun s -> put (int_of_string s)
```

Out[77]:

```
val inc : int -> (int, int, unit) PState.t = <fun>
```

Out[77]:

```
val dec : int -> (int, int, unit) PState.t = <fun>
```

Out[77]:

```
val double : (int, int, unit) PState.t = <abstr>
```

Out[77]:

```
val to_string : (int, string, unit) PState.t = <abstr>
```

Out[77]:

```
val of_string : (string, int, unit) PState.t = <abstr>
```

Computation with changing state

In [78]:

```
let foo = inc 5 >>= fun () -> to_string
let bar = get >>= fun s -> put (s ^ "00")

let baz = foo >>= fun () -> bar
let quz = bar >>= fun () -> foo
```

Out[78]:

```
val foo : (int, string, unit) PState.t = <abstr>
```

Out[78]:

```
val bar : (string, string, unit) PState.t = <abstr>
```

Out[78]:

```
val baz : (int, string, unit) PState.t = <abstr>
```

```
File "[78]", line 5, characters 28-31:
Error: This expression has type (int, string, unit) PState.
t
       but an expression was expected of type (string, 'a,
 'b) PState.t
       Type int is not compatible with type string
   4: let baz = foo >>= fun () -> bar
   5: let quz = bar >>= fun () -> foo
```

A well-typed stack machine

- Let's build a tiny stack machine with 3 instructions
  - `push` pushes a constant on to the stack. Constant could be of any type.
  - `add` adds the top two integers on the stack and pushes the result
  - `_if_` expects a `[b;v1;v2] @ rest_of_stack` on top of the stack.
    - if `b` is true then result stack will be `v1::rest_of_stack`
    - otherwise, `v2::rest_of_stack`.
- Our stack machine will not get stuck!
  - recall the definition from lambda calculus lectures
- This is how WASM operational semantics is defined!

Stack operations

- Because our stack will have values of different types, encode then using pairs.
  - `[]` will be `()`

- `[1;2;3]` will be `(1, (2, (3, ())))`
- `[1;true;3]` (which is not a well-typed OCaml expression) will be `(1, (true, (3, ())))`

Stack Operations

In [79]:

```
module type STACK_OPS =
sig
  type ('s,'t,'a) t
  val add : unit -> (int * (int * 's),
                     int * 's,
                     unit) t
  val _if_ : unit -> (bool * ('a * ('a * 's)),
                      'a * 's,
                      unit) t
  val push_const : 'a -> ('s,
                          'a * 's,
                          unit) t
end
```

Out[79]:

```
module type STACK_OPS =
  sig
    type ('s, 't, 'a) t
    val add : unit -> (int * (int * 's), int * 's, unit) t
    val _if_ : unit -> (bool * ('a * ('a * 's)), 'a * 's, u
nit) t
    val push_const : 'a -> ('s, 'a * 's, unit) t
  end
```

Stack Machine

We can combine the stack operations with the parameterised monad signature to build a signature for a stack machine:

In [80]:

```
module type STACKM = sig
 include PARAMETERISED_MONAD
 include STACK_OPS
   with type ('s,'t,'a) t := ('s,'t,'a) t
 val execute : ('s,'t,'a) t -> 's -> 't * 'a
end
```

Out[80]:

```
module type STACKM =
  sig
    type ('s, 't, 'a) t
    val return : 'a -> ('s, 's, 'a) t
    val ( >>= ) : ('r, 's, 'a) t -> ('a -> ('s, 't, 'b) t)
-> ('r, 't, 'b) t
    val add : unit -> (int * (int * 's), int * 's, unit) t
    val _if_ : unit -> (bool * ('a * ('a * 's)), 'a * 's, u
nit) t
    val push_const : 'a -> ('s, 'a * 's, unit) t
    val execute : ('s, 't, 'a) t -> 's -> 't * 'a
  end
```

Stack Machine

Here is the implementation of the stack machine

In [81]:

```
module StackM : STACKM =
struct
  include PState

  let add ()=
    get >>= fun (x,(y,s)) ->
    put (x+y,s)

  let _if_ () =
    get >>= fun (c,(t,(e,s))) ->
    put ((if c then t else e),s)

  let push_const k =
    get >>= fun s ->
    put (k, s)

  let execute c s = runState ~init:s c
end
```

Out[81]:

```
module StackM : STACKM
```

Using the stack machine

In [82]:

```
let program = let open StackM in
  push_const 4 >>= fun () ->
  push_const 5 >>= fun () ->
  push_const true >>= fun () ->
  _if_ () >>= fun () ->
  add ()
```

Out[82]:

```
val program : (int * '_weak3, int * '_weak3, unit) StackM.t
= <abstr>
```

In [83]:

```
StackM.execute program (20,(10,()))
```

Out[83]:

```
- : (int * (int * unit)) * unit = ((25, (10, ())), ())
```

Using the stack machine

In [84]:

```
StackM.execute (StackM._if_ ()) (false,(10,()))
```

```
File "[84]", line 1, characters 43-45:
Error: This expression has type unit but an expression was
 expected of type
         int * 'a
   1: StackM.execute (StackM._if_ ()) (false,(10,()))
```

In [85]:

```
StackM.execute (StackM.add ()) ()
```

```
File "[85]", line 1, characters 31-33:
Error: This expression has type unit but an expression was
 expected of type
         int * (int * 'a)
   1: StackM.execute (StackM.add ()) ()
```

Fin.