CS 456

Programming Languages
Fall 2024

Week 8
Monads and Effects

Type Amplifiers

- Values are often specialized or encapsulated:
  - An option type specializes a value to Some or None
  - A ref type encapsulates a value within a memory container
  - An exception type wraps a value around a computational effect
  - A list type specializes a set of values around a choice action defined by a list index
  - An I/O operation consumes and returns a value in the context of actions that modify a input/output stream
  - ….

- Would like to reason about these types in the same way we reason about types that are not container-ized

Maybe

A "safe" division operation:

```
let div x y = if y = 0 then None else Some (x / y)
```

But, can't use this in the following:

```
let r = 1 + (4 div 2)
```

- The signature for "+" expects an int not an option
- Could change all arithmetic operations to accept an option type as input.

Maybe


```
let plus_opt (x:int option) (y:int option) : int option =
  match x,y with
  | None, _ | _, None -> None
  | Some a, Some b -> Some (Stdlib.( + ) a b)

let ( + ) = plus_opt

let minus_opt (x:int option) (y:int option) : int option =
  match x,y with
  | None, _ | _, None -> None
  | Some a, Some b -> Some (Stdlib.( - ) a b)

let ( - ) = minus_opt

...
```

Better Approach

- Can we define an abstraction that refactors patterns common to these definitions?

```
let propagate_none (op : int -> int -> int) (x : int option)
                   (y : int option) =
  match x, y with
  | None, _ | _, None -> None
  | Some a, Some b -> Some (op a b)

let ( + ) = propagate_none Stdlib.( + )
let ( - ) = propagate_none Stdlib.( - )
let ( * ) = propagate_none Stdlib.( * )

val ( + ) : int option -> int option -> int option = <fun>
val ( - ) : int option -> int option -> int option = <fun>
```

A Better Approach

- Not quite right: abstraction doesn't account for division which must check the value of its second argument before applying the "unsafe" division operator

```
let propagate_none
  (op : int -> int -> int option) (x : int option) (y : int option)
=
  match x, y with
  | None, _ | _, None -> None
  | Some a, Some b -> op a b

let wrap_output (op : int -> int -> int) (x : int) (y : int) : int option
=  Some (op x y)

let div (x : int) (y : int) : int option =
  if y = 0 then None else wrap_output Stdlib.( / ) x y

let ( / ) = propagate_none div
```

Intuition

- Transformed operations on “unboxed” integer values to operate over “boxed” Maybe objects
- Employed two basic transforms:
  - Taking a regular unboxed integer and turning it into a `Maybe` (wrapped with `Some`) - this is what `wrapped_output` does
  - Factoring code to handle pattern-matching against `None`. This involved upgrading/specializing functions that operate over integers to instead accept inputs of type `int option`.

Monad

- Conversion from ordinary to/from option types is tedious
- Would like to wrap (i.e, amplify) computed values with the option they are associated with
- Build a type constructor for this purpose:

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

- A monad defines a container
- `return` puts a value in that container
- `bind` takes a container that contains a value of type `'a`, a function that takes a value of type `'a` and returns a container containing values of type `'b` and returns that container

The Maybe Monad


```
module Maybe : Monad =
struct
  let return (x : int) : int option = Some x
  val return : int -> int option

  val bind : int option -> (int -> int option) -> int option
  let bind (x : int option) (op : int -> int option) : int option =
     match x with
      | None -> None
      | Some a -> op a

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

Maybe Monad


```
let ( + ) (x : int option) (y : int option) : int option =
  x >>= fun a -> y >>= fun b -> return (Stdlib.( + ) a b)

let ( - ) (x : int option) (y : int option) : int option =
  x >>= fun a -> y >>= fun b -> return (Stdlib.( - ) a b)

let ( * ) (x : int option) (y : int option) : int option =
  x >>= fun a -> y >>= fun b -> return (Stdlib.( * ) a b)


let ( / ) (x : int option) (y : int option) : int option =
  x >>= fun a -> y >>= fun b ->
  if b = 0 then None else return (Stdlib.( / ) a b)
```

Maybe Monad

- Further simplification:

```
let upgrade_binary op x y =
  x >>= fun a ->
  y >>= fun b ->
  op a b

let return_binary op x y = return (op x y)

let ( + ) = upgrade_binary (return_binary Stdlib.( + ))
let ( - ) = upgrade_binary (return_binary Stdlib.( - ))
let ( * ) = upgrade_binary (return_binary Stdlib.( * ))
let ( / ) = upgrade_binary div
```

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

Maybe Monad


```
module Maybe : Monad = struct
  type 'a t = 'a option

  let return x = Some x

  let (>>=) m f =
    match m with
    | None -> None
    | Some x -> f x
end
```

The State Monad

Consider the function:

```
let f v s = let (b, x) = g v s in
            let (c, y) = h (b + 1) x in
            let (d, z) = i (c + 1) y
in (d, z)
```

Suppose we model a state as a record: { s1 : int; s2 : int } and

```
- g1 = fun v s -> let {s1 = s1; s2 } = s in (s1, {s1 = s1 + v, s2})

- h1 = fun v s -> let {s1;  s2 = s2} = s in (s2, {s1;  s2 = s2 + v})

- i1 = fun v s -> let {s1 = s1;  s2 = s2} = s in
           (s1 + s2,  {s1 = s1 + v;  s2 = s2 + v})
```

Then `f1 0 { s1 = 0;  s2 = 0}` yields `(2, {s1 = 2;  s2 = 2})`

`g1`, `h1`, and `i1` given a value and a state, returns a new value, and a new state. In other words, they encapsulate a state transformer.

The State Monad

So,

```
let f v s = let (b, x) = g v s in
            let (c, y) = h (b + 1) x in
            let (d, z) = i (c + 1) y
        in (d, z)
```

following the design pattern we used for the Maybe monad, we can express this function monadically as:

```
let f v = (g v) >>= fun b ->
          (h (b + 1)) >>= fun c ->
          (i (c + 1)) >>= fun z -> return z
```

What does `(f 0)` return? It returns a computation that when applied to an initial state, executes the sequence of calls to `g`, `h`, and `i`, threading the state appropriately.

The State Monad


```
module State : Monad = struct
  type state     (* the record {s1; s2} *)
  type 'a t = state -> ‘a * state
   (* a state monad is a container over a state transition function *)
   (* in our example, these are the functions g, h, and i after they have
      been applied to an initial value. *)

  val return: ‘a -> ‘a t
  let return x = fun s -> (x, s)

  val bind: ‘a t -> (‘a -> ‘b t) -> ‘b t
  let bind s f =
    fun state ->
      (* apply the supplied state transition function *)
      let (a, s’) = s state in
      (* generate a new state transition function and value *)
      let (b, s’’) = f a s’ in
      (b, s’’)
end
```

The State Monad


```
let f v = (g v) >>= fun b ->
          (h (b + 1)) >>= fun c ->
          (i (c + 1)) >>= fun z -> return z
```

```
val bind: ‘a t -> (‘a -> ‘b t) -> ‘b t
  let bind s f =
    fun state ->
      (* apply the supplied state transition function *)
      let (a, s’) = s state in
      (* generate a new state transition function and value *)
      let (b, s’’) = f a s’ in
      (b, s’’)
end
```

`(g v) >> fun b -> <rest of computation> ==>`

`bind (g v) (fun b -> <rest of computation> ==>`

returns a function that when applied to `state`, applies `(g v)`

`(i.e, fun s -> let {s1 = s1; s2 } = s in (s1, {s1 = s1 + v, s2}))`

to `state`, and then applies `(fun b -> <rest of computation>)` to `s1` and the new state `{s1 = s1 + v, s2}`

The State Monad


```
let f v = (g v) >>= fun b ->
          (h (b + 1)) >>= fun c ->
          (i (c + 1)) >>= fun z -> return z
```

The effect of bind in the state monad is to return a computation that when supplied an initial state, performs the effects on that state as defined by `g`, `h`, and `i`. If we define:

```
let run comp = comp {s1 = 0;  s2 = 0}
```

then

```
run (f 0)
```

executes the computation. In other words, bind allows us to compose a sequence of state-manipulating computations and returns a function that executes these computations when given an initial state.

Functors

- Ordinary computations operate over values (e.g., 2 + 3 = 5)
- Values often reside in containers or boxes (e.g., an option box)
- Cannot directly apply a value that is wrapped in a context
- First step:
  - An operation that applies a function to values wrapped in a context

```
module type Functor =  sig
    type ‘a t
    val fmap : (‘a -> ‘b) -> ‘a t -> ‘b t
  end
```

An instance of this structure:

```
module MaybeFunctor : Functor = struct
     type ‘a t = ‘a option
     let fmap f x = match x with
                    | None -> None
                    | Some y -> Some (f y)
 end
```

Applicative Functors

- Both functions and values can be wrapped in a context (e.g., a state transition function)
- An applicative functor handles the application of a function wrapped in a context to a value wrapped in a context

```
module type Applicative = sig
   include Functor
   val pure : ‘a  -> ‘ a t   (* wraps a value into a context *)
   val apply : (‘a -> ‘b) t -> ‘a t -> ‘b t
 end
```

Applicative Functors


```
module OptionApplicative : Applicative =
struct
  type ‘a t = ‘a option
  let pure x = Some x

  let apply fo xo =
     match fo, xo with
     | Some f, Some x -> Some (f x)
     | _ -> None
end
```

Monads

- Apply a function that returns a wrapped value to a wrapped value.
- The bind operator provides this functionality

Example:

```
let half x = if (even x)
             then Some (x / 2)
             else None
```

```
Now,
  (Some 10) Maybe.>>= half --> Some 5
  (Some 10) Maybe.>>
  half      Maybe.>>=
  half                   —-> None
```

References

OCaml Programming:

https://cs3110.github.io/textbook/chapters/ds/monads.html

Of Course ML Has Monads:

https://existentialtype.wordpress.com/2011/05/01/of-course-ml-has-monads/

Understanding Monads (Haskell)

https://en.wikibooks.org/wiki/Haskell/Understanding_monads