Recitation 21: Monads

Design pattern — function + "something more" ("computation")

Examples

2 / 0 ⟶ exception

2 / 0 ⟶ None 2 / 1 ⟶ Some 2

```
let (/) a b = if b = 0 then None else Some ((Stdlib./) a b)
```

```
let log = ref ""
let inc x = log := "inc called" ^ !log; x + 1
```

```
let inc x = (x+1, "inc called")
```

```
let line = Lwt_io.read_line stdin
```

"Upgraded" output types

```
(/) : int -> int -> int option
inc : int -> int * string
read_line : in -> string Lwt.t
```

(2 / 1) / 1

(Some 2) / 1

int -> int

Problem: composition

(4/2) |> (/) 4 typecheck fails!

What to do w/ None input?

inc 3 |> inc

What should 2nd inc do w/ log of first?

→ Propagate None

→ Concat logs

div, inc, mult, dec

Solution: New pipeline op >>= (bind)

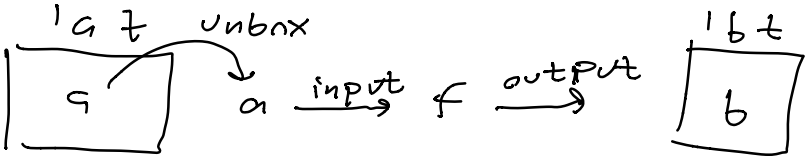
'a → 'b

'a → 'b t

(>>=) : 'a t → ('a → 'b t) → 'b t

m f

Unbox & apply



For option

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

m f

Some 2 / 1

For logging:

```
let (>>=) (v, log) f =
  let (v', entry) = f v in
    (v', entry ^ log)
```

2 / 1 = Some 2

(1 / 0) / 1

None / 1

None

For promises

```
let (>>=) m f =
  <wait for m to resolve>
  match m with
    | Resolved v -> f v
```

(2 / 1) / 1

Some 2 / 1

2 / 1

`>>=` tells us how to compose $g(f(x))$

```
let >> f g = fun x -> f x |> g
let (>=>) f g = fun x -> f x >>= g
```

f: int → int option

let div2 x = safe_divide x 2

What is a monad?

Upgraded type + definition >>=

+ requirements

let div4 = div2 >=> div2

Technically:

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

Return examples

- option: return x = Some x
- logging: = (x, "")
- promises: = Resolved x

Monad laws

Examples of "laws"

pop (push x s) = x

None >>= f = None

>> compose

Ordinary functions

f >> (g >> h) ≅ (f >> g) >> h

id x = x f >> id ≅ f ≅ id >> f

$$f >=> (g >=> h) \cong (f >=> g) >= h$$
$$return >=> f \cong f \cong f >=> return$$

In practice, for client

No worries about order of setting up pipeline

$f \;|> g \;|> h \qquad (f >=> g) >= h$

No worries about weird return behavior

$$(m >>= f) >>= g \cong m >>= (fun\ x \to f\ x >>= g)$$
$$(return\ x) >>= f \cong f\ x \cong f\ x >>= return$$

$$((f >=> g) >=> h)\ x \cong (f >=> (g >=> h))\ x$$

$(f >=> g)\ x \quad h(g(f\ x)) \qquad f\ x \quad (g >=> h)\ f\ x$

$f\ x \quad g(f\ x) \qquad\qquad g\ (f\ x) \quad h\ (g\ (f\ x))$