Recitation 21: Monads

Design pattern for function + something more,
"computation"

Examples

- functions that sometimes fail

  2 / 0 → exception

  Alternatively: (no exception) 2/0 = None

  let div a b = if b = 0 then None else Some (a/b)

- functions that produce log strings when called

  let log = ref ""

  let inc x = log := !log ^ "inc called"

  Alternatively (no globals)

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

- functions that "run later" and whose results are used only after they run

  let line = Lwt_io. read_line stdin

"Upgraded" output types

int → int option

int → int * string

Lwt_io.input_channel → string Lwt.t

In general:

"vanilla": 'a → 'b

"upgraded": 'a → 'b t

Problem: how to compose?

(' → ' a) → ' → ' b

(div 4 2) |> div 4

What does div do with None input?

inc x |> inc doesn't typecheck

What does first inc do w/ log of second inc?

→ What we want: propagate None

→ What we wants: concatenate logs

Don't want div/inc to handle this — tons of boilerplate

Have to do this in dec, mult, etc.

Instead, define new pipeline operator >>=
pronounced "bind".

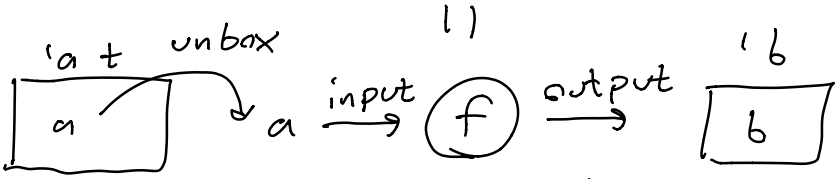
Then: div 4 2 >>= div 4

inc x >>= inc

Recall our upgraded functions look like 'a → 'b t

so (>>=) : 'a t → ('a → 'b t) → 'b t

Think "unbox & apply":



How we define >>= depends on what we want.

For option:

```
let (>>=) m f = match m with
  | None → None   (* propagate None *)
  | Some x → f x
```

For logging:

```
let (>>=) (input, prev_log) f
  let (output, log_entry) = f input in
```

(output, prev_log ^ log_entry)

For promises, complicated, but basically

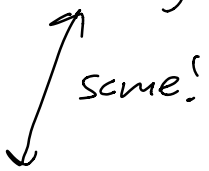
```
let (>>=) m f =
  <Wait for promise m to run>
  match m with
    Resolved input -> f input
```

>>= basically tells how to compose

Can use it to define composition of upgraded funcs.

Recall ordinary composition:

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



For upgraded funcs:

```
let compose' f g = fun x -> f x >>= g
let (>=>) = compose'
```

What is a monad?

Basically, upgraded type plus def of >>=

Definition has to "make sense" (monad laws)

Technically:

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

Examples

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

Most trivial way to "upgrade" a value.

Monad laws

- Any data structure has expected behavior ("laws")
  - pop (push x s) = x
  - option monad: None >>= f = None

That was specific to option monad, but turns out there are some laws that <u>all</u> monads satisfy.

- function composition is associative
  - $f >> (g >> h) \cong (f >> g) >> h$ (means "behaves the same as")

  For upgraded functions

  $f >=> (g >=> h) \cong (f >=> g) >=> h$
- identity

  $f >> id \cong f \cong id >> f$

  For upgraded functions

  $f >=> return \cong f \cong return >=> f$

- In practice: means client of monad
  - doesn't need to worry about order of composition
  - can think of return as a trivial "no-op"
- Can write in terms of bind

$$(m >>= f) >>= g \cong m >>= (\text{fun } x \to f\,x >>= g)$$

$$\text{return } x >>= f = f\,x = f\,x >>= \text{return}$$