Functional Programming Techniques for Philosophy and Linguistics
Chris Barker and Jim Pryor, NASSLLI 2016
JP's Thursday Handout

Remember Polymorphic Types?
id : $\forall\alpha.\ \alpha \to \alpha$ (we'll usually suppress prenex $\forall\alpha$ in type signatures)
id = $\Lambda\alpha.\ \lambda x{:}\ \alpha.\ x$ (will also suppress initial $\Lambda\alpha$, and the [type] applications)

Schematic Type Expressions
$\text{Int} \to \alpha \ \equiv\ \boxed{\alpha}_{\text{Reader Int}}$
$\text{Set}\ \alpha \ \equiv\ \boxed{\alpha}_{\text{Set}}$
I'll use xx and yy as variables for these.
(At one point I'll use xxx as a variable for a $\boxed{\boxed{\alpha}}$, with the boxes understood univocally.)

Kleisli arrow types for a given $\boxed{}$ are: $\alpha \to \boxed{\beta}$
Contrast ordinary arrow types: $\alpha \longrightarrow \beta$
I'll use j and k as variables for Kleisli arrows, and f and g for functions with ordinary types.

Endofunctors
some type operation $\boxed{}$
and a paired function *map* : $\forall\alpha\ \beta.\ (\alpha \longrightarrow \beta) \longrightarrow \boxed{\alpha} \longrightarrow \boxed{\beta}$
obeying the laws:
map (id : $\alpha \to \alpha$) xx = (id : $\boxed{\alpha} \to \boxed{\alpha}$) xx = xx
map (g $\circ$ f) = (map g) $\circ$ (map f)

Example1: $\boxed{\alpha}_{\text{Set}} = \text{Set}\ \alpha$
$\text{map}_{\text{Set}} : (\alpha \longrightarrow \beta) \longrightarrow \boxed{\alpha}_{\text{Set}} \longrightarrow \boxed{\beta}_{\text{Set}}$, that is:
$\text{map}_{\text{Set}} : (\alpha \to \beta) \to \text{Set}\ \alpha \to \text{Set}\ \beta$
map_{Set} f xx = { f x | x $\in$ xx }
So map_{Set} succ {2, 3, 10} = {3, 4, 11}

Example 2: $\boxed{\alpha}_{\text{Intensionality}} = \text{World} \to \alpha$
$\text{map}_{\text{Intensionality}} : (\alpha \longrightarrow \beta) \longrightarrow \boxed{\alpha}_{\text{Intensionality}} \longrightarrow \boxed{\beta}_{\text{Intensionality}}$, that is:
$\text{map}_{\text{Intensionality}} : (\alpha \to \beta) \to (\text{World} \to \alpha) \to (\text{World} \to \beta)$
$\text{map}_{\text{Intensionality}}$ f xx = λw. f (xx w)

Other names for map: fmap, <$>, liftA, liftM

Monads

some type operation $\Box$

and a paired function *map* : $\forall\alpha\,\beta.\ (\alpha \longrightarrow \beta) \longrightarrow \boxed{\alpha} \longrightarrow \boxed{\beta}$ (as above)

also a paired function *join* : $\forall\alpha.\ \boxed{\boxed{\alpha}} \longrightarrow \boxed{\alpha}$

(e.g., for $\Box_{Set}$, this is $\bigcup$)

also a paired function ⇧("up" or map0): $\forall\alpha.\ \alpha \longrightarrow \boxed{\alpha}$

(e.g., for $\Box_{Set}$, this is singleton)

instead of map + join, you could have a single function

$<=< \ : \forall\alpha\,\beta\,\gamma.\ (\beta \longrightarrow \boxed{\gamma}) \longrightarrow (\alpha \longrightarrow \boxed{\beta}) \longrightarrow (\alpha \longrightarrow \boxed{\gamma})$

compare the type of the ordinary composition operator

$\circ : \forall\alpha\,\beta\,\gamma.\ (\beta \longrightarrow \gamma) \longrightarrow (\alpha \longrightarrow \beta) \longrightarrow (\alpha \longrightarrow \gamma)$

<=< is called "Kleisli composition." It plays the role for Kleisli arrow types ($\alpha \longrightarrow \boxed{\beta}$) that $\circ$ plays for ordinary arrow types ($\alpha \longrightarrow \beta$).

Example:

duplicate 3 = $\{3, 3, 3\}_{multi}$

upto 4 = $\{0, 1, 2, 3\}_{multi}$

(duplicate <=< upto) 4 = join $\{\{\}, \{1\}, \{2, 2\}, \{3, 3, 3\}\}_{multi} = \{1, 2, 2, 3, 3, 3\}_{multi}$

These functions (map + join + ⇧, or <=< + ⇧) have to obey laws, best stated as:

k′ <=< (k <=< j) = (k′ <=< k) <=< j

⇧ <=< j = j = j <=< ⇧

In summary, <=< is associative and ⇧ is its identity. So *monads* are a generalization with polytypes of the algebraic notion of a *monoid*.

Interdefinitions:

j >=> k ≡ k <=< j

xx >>= ("bind") k ≡ (k <=< id) xx ≡ (k <=< const xx) anything ≡ join (map k xx)

k <=< j ≡ join ∘ map k ∘ j ≡ λx. (j x >>= k)

join xxx ≡ xxx >>= id

map f xx ≡ xx >>= λx. ⇧ (f x)

map2 f xx yy ≡ xx >>= λx. yy >>= λy. ⇧ (f x y)

Compare types:

⇧/map0 : $\forall\alpha.\ \alpha \rightarrow \boxed{\alpha}$ lifts a value (nullary function) into $\Box$

map : $\forall\alpha\,\beta.\ (\alpha \rightarrow \beta) \rightarrow \boxed{\alpha} \rightarrow \boxed{\beta}$ lifts a unary function into $\Box$

map2 : $\forall\alpha\,\beta\,\gamma.\ (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \boxed{\alpha} \rightarrow \boxed{\beta} \rightarrow \boxed{\gamma}$ lifts a binary function into $\Box$

Other names for ⇧/map0: η, pure, return, unit (≠ our Boring type), monadic id, singleton

Other names for join: μ

Other names for >>=/bind: ★

Monadic Layers

$\boxed{\alpha}_{\text{Set}}$ is really $\boxed{\alpha}_{\text{SetT (Identity)}}$ = Set α

$\boxed{\alpha}_{\text{Intensionality}}$ is really $\boxed{\alpha}_{\text{Reader World}}$ is really $\boxed{\alpha}_{\text{ReaderT World (Identity)}}$ = World → α

There are also monadic types like $\boxed{\alpha}_{\text{ReaderT World (SetT (Identity))}}$ = World → Set α

$\boxed{\alpha}_{\text{StateT S (SetT (Identity))}}$ = S → Set (α × S)

$\boxed{\alpha}_{\text{SetT (StateT S (Identity))}}$ = S → (Set α) × S

Definitions for Identity Monad

$\boxed{\alpha}_{\text{Identity}}$ = α

⇧ = id

k <=< j = k ∘ j

xx >>= k = k xx

map = λf xx. f xx

(Note: map and >>= won't have the same definition *in general*: usually their types differ.)

Definitions for MaybeT Monadic Layer

type Maybe/Shortlist α = None ( ) + One (α)

$\boxed{\alpha}_{\text{MaybeT (M)}} = \boxed{\text{Maybe } \alpha}_{\text{M}}$

liftT = λxx. map_M One xx

⇧ = liftT ∘ $⇧_M$

xx >>= k = xx $>>=_M$ λxs. case xs of { None ⟶ $⇧_M$ None | One x ⟶ k x }

Auxiliary functions for MaybeT: *zero* : $\boxed{\alpha}$; *zero* = $⇧_M$ None

When M = Identity: **xx $>>=_{\text{Maybe}}$ k = case xx of { None ⟶ None | One x ⟶ k x }**

$⇧_{\text{Maybe}}$ = λx. One x

$\text{map2}_{\text{Maybe}}$ = λf xx yy. case (xx, yy) of { (One x, One y) ⟶ One (f x y) | else None }

Definitions for SetT Monadic Layer

$\boxed{\alpha}_{\text{SetT (M)}} = \boxed{\text{Set } \alpha}_{\text{M}}$

liftT = λxx. map_M singleton xx

⇧ = liftT ∘ $⇧_M$

xx >>= k = xx $>>=_M$ λxs. union_M { k x | x ∈ xs }

where union_M : Set $\boxed{\text{Set } \beta}_{\text{M}} \rightarrow \boxed{\text{Set } \beta}_{\text{M}}$

union_M { } = $⇧_M$ { }

union_M {bb} = bb

union_M {bb, bb′} = map2_M (∪) bb bb′

union_M {bb, bb′, bb″} = map2_M (∪) (map2_M (∪) bb bb′) bb″

...

Auxiliary functions for SetT: *zero* : $\boxed{\alpha}$; *zero* = $⇧_M$ { }

plus : $\boxed{\alpha} \rightarrow \boxed{\alpha} \rightarrow \boxed{\alpha}$; *plus* = λxx yy. map2_M (∪) xx yy

When M = Identity: **xx $>>=_{\text{Set}}$ k = ⋃ { k x | x ∈ xx }**

$⇧_{\text{Set}}$ = λx. {x}

map2_{Set} = λf xx yy. { f x y | x ∈ xx, y ∈ yy }

Definitions for ReaderT Monadic Layer

$\boxed{\alpha}_{\text{ReaderT R (M)}} = R \to \boxed{\alpha}_M$
liftT = $\lambda xx.\ \lambda r.\ xx$
$\Uparrow\ = \text{liftT} \circ \Uparrow_M$
$xx >>= k = \lambda r.\ xx\ r >>=_M \lambda x.\ k\ x\ r$
Auxiliary functions for ReaderT: *ask* : $\boxed{R}$; *ask* = $\Uparrow_M$

localshift : $(R \to R) \to \boxed{\alpha} \to \boxed{\alpha}$; *localshift* = $\lambda f\ xx.\ xx \circ f$

When M = Identity: $\mathbf{xx >>=_{Reader\ R} k = \lambda r.\ let\ x = xx\ r;\ yy = k\ x\ in\ yy\ r}$
$\mathbf{\Uparrow_{Reader\ R} = \lambda x.\ \lambda r.\ x}$
$\mathbf{map2_{Reader\ R} = \lambda f\ xx\ yy.\ \lambda r.\ f\ (xx\ r)\ (yy\ r)}$

Definitions for StateT Monadic Layer

$\boxed{\alpha}_{\text{StateT S (M)}} = S \to \boxed{\alpha \times S}_M$
liftT = $\lambda xx.\ \lambda s.\ \text{map}_M\ (\lambda x.\ (x, s))\ xx$
$\Uparrow\ = \text{liftT} \circ \Uparrow_M$
$xx >>= k = \lambda s.\ xx\ s >>=_M \lambda(x, \mathbf{s'}).\ k\ x\ \mathbf{s'}$
Auxiliary functions for StateT: *get* : $\boxed{S}$; *get* = $\lambda s.\ \Uparrow_M (s, s)$

modify : $(S \to S) \to \boxed{\text{Boring}}$; *modify* = $\lambda f.\ \lambda s.\ \Uparrow_M (\,(\,),\ f\ s)$

When M = Identity: $\mathbf{xx >>=_{State\ S} k =}\ \lambda s.\ \text{let } (x,\mathbf{s'}) = xx\ s;\ yy = k\ x \text{ in } yy\ \mathbf{s'}$
$\mathbf{\Uparrow_{State\ S} = \lambda x.\ \lambda s.\ (x, s)}$

Definitions for WriterT Monadic Layer

$\boxed{\alpha}_{\text{WriterT W (M)}} = \boxed{\alpha \times W}_M$, where W is e.g., a list of logged messages
liftT = $\lambda xx.\ \text{map}_M\ (\lambda x.\ (x, [\,]))\ xx$
$\Uparrow\ = \text{liftT} \circ \Uparrow_M$
$xx >>= k = xx >>=_M \lambda(x, ws).\ k\ x >>=_M \lambda(y, ws').\ \Uparrow_M (y, ws \lhd\rhd ws')$
Auxiliary functions for WriterT: *tell* : $W \to \boxed{\text{Boring}}$; *tell* = $\lambda ws.\ \Uparrow_M (\,(\,),\ ws)$

listen : $\boxed{\alpha} \to \boxed{\alpha \times W}$; *listen* = $\lambda xx.\ xx >>=_M \lambda(x, ws).\ \Uparrow_M ((x, ws), ws)$

censor : $(W \to W) \to \boxed{\alpha} \to \boxed{\alpha}$; *censor* = $\lambda f\ xx.\ xx >>=_M \lambda(x, ws).\ \Uparrow_M (x, f\ ws)$

When M = Identity: $\mathbf{xx >>=_{Writer\ W} k = let\ (x,ws) = xx;\ (y,ws') = k\ x\ in\ (y,\ ws \lhd\rhd ws')}$
$\mathbf{\Uparrow_{Writer\ W} = \lambda x.\ (x, [\,])}$

Examples of Using (Simple, Single-layered) Monads

1. Safe division (CB, using Maybe monad)

   …

2. ± (JP, using Set monad)

   * What is: *(3 * √4) - √25*, interpreting that as: *(3 * ±2) - ±5*?

```
> plusMinus x = [x, -x] :: Set Int
> :type plusMinus
plusMinus :: Int -> Set Int

> map2 (*) (up 3) (plusMinus 2)
Set [-6,6]

> map2 (-) (map2 (*) (up 3) (plusMinus 2)) (plusMinus 5)
Set [-1,-11,11,1]
```

3. Variable binding (CM, using Reader monad)

   …

4. Running tally (JP, using State monad)

   * Suppose you're trying to use the State monad to keep a running side-tally of how often certain arithmetic operations have been used in computing a complex expression. You've settled upon the design plan of using the State monad, and defining a function like this:

   let counting_plus xx yy = tick >>= λ_. map2 (+) xx yy

   How should you define the operation *tick* to make this work? The intended behavior is that after running:

   let zz = counting_plus (up 1) (counting_plus (up 2) (up 3))
   in runState zz 0

   you should get a payload/at-issue result of 6 (that is, 1+(2+3)) and a final side-tally of 2 (because + was used twice).

```
> let -- xx >> yy = xx >>= \_ -> yy
      tick :: State Int ()
      tick = modify succ
      counting_plus xx yy = tick >> map2 (+) xx yy
      zz :: State Int Float
      zz = counting_plus (up 1) (counting_plus (up 2) (up 3))
  in runState zz 0
(6.0, 2)
```

   * Instead of the design in the previous problem, suppose you had instead chosen to do things this way:

   let counting_plus' xx yy = map2 (+) xx yy >>= tock

   How should you define the operation *tock* to make this work, with the same behavior as before?

```
> let tock :: Float -> State Int Float
      tock = \z -> modify succ >> up z
      counting_plus' xx yy = map2 (+) xx yy >>= tock
      zz' = counting_plus' (up 1) (counting_plus' (up 2) (up 3))
  in runState zz' 0
(6.0, 2)
```