POPL Classes

This week submissions were great: Keep the hard work up!

- We will focus on the difficult bits on the submissions.

Exercise 2.

- Linked list implementation in Fun.
- No need to change interpreter if we change 'head/tail'.

Exercise 3.

- Implementation of repeat in let-style.
- We could also do monadic style.

Exercise 6.

- Combining monads is usually not that easy.
- E.g. probability and non-determinism.

POPL Classes

Exercise 7

- All solutions will use the effects (memory) to track evaluation.
- For most monads, the order of evaluation does matter.
- For some monads, it does not matter: these are commutative monads.

$$f\ x \triangleright (\lambda y.\ g\ u \triangleright (\lambda v.\ h\ y\ v)) = f\ x \triangleright (\lambda y.\ g\ u \triangleright (\lambda v.\ h\ y\ v))$$

- Exceptions (Maybe) form a commutative monad.
- Non-determinism with lists is non-commutative; it is commutative with Data.Set.
- Probability (should be) commutative.

POPL Classes

Exercise 10.

- Submissions had great formal proofs.
- This is the equivalence between monads and monads in relative form.
- If curious: Monads need not be endofunctors.
- In Haskell: rebinable syntax, or ($>) combinators with constraints.

Exercise 5.

- We use associativity of let.

$$\text{let } x \overset{\varepsilon}{=} f \text{ in } (\text{let } y \overset{\varepsilon'}{=} g \text{ in } h) \equiv \text{let } y \overset{\varepsilon'}{=} (\text{let } x \overset{\varepsilon}{=} f \text{ in } g) \text{ in } h$$

- Associativity of let holds in CBV and CBN, but not if we mix both.
  - ε = CBV but ε' = CBN breaks associativity.
  - ε' = CBN but ε = CBV does not break.

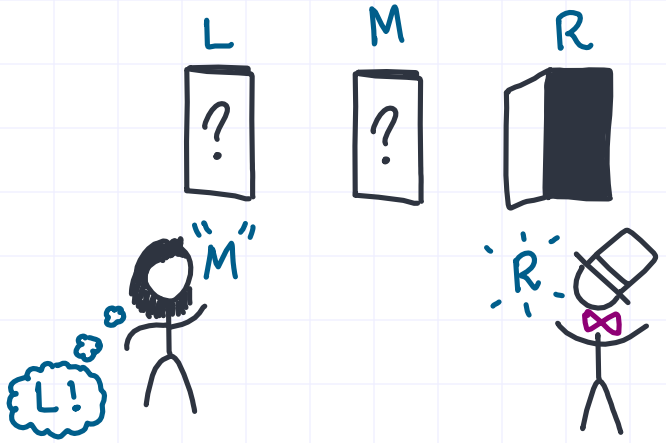
Monty Hall Problem

(i) $\rightsquigarrow \frac{1}{3}|L\rangle + \frac{1}{3}|M\rangle + \frac{1}{3}|R\rangle$

(ii) $\rightsquigarrow \frac{1}{3}|LR\rangle + \frac{1}{6}|ML\rangle + \frac{1}{6}|MR\rangle + \frac{1}{3}|RL\rangle$

(iii) $\rightsquigarrow \frac{1}{3}|\cancel{LR}\rangle + \frac{1}{6}|ML\rangle + \frac{1}{6}|\cancel{MR}\rangle + \frac{1}{3}|RL\rangle$

(iv) $\rightsquigarrow \frac{1}{3}|ML\rangle + \frac{2}{3}|RL\rangle$



FunProb, and probabilistic programming.

POPL Classes

Three covid tests problem (from Jacobs, 2023).

> The incidence of an illness is 1 in 20.
> Sensitivity of the test is 9/10.
> Specificity of the test is 3/5.
> A patient got two positive and one negative test.
> What is the probability the patient is ill?

We all know how to solve these problems (by many Bayes updates). But this is tedious and prone to error.

> Idea: carry Bayes update inside a monad
> $M\,a = \mathrm{Set}\,(a, \mathrm{Double})$, with some restrictions.
> We will be able to describe the problem and have it solved in Fun.

POPL Classes

- Submissions : very good, very few.
- Perhaps more useful to do exercises you have already tried.
- In any case, let us just go through the exercises.
- I wrote the quick implementation of Ex3 and Ex4; we can test Ex5.

POPL - Class 3

Exercise 1. We start by recalling the definitions.

```
result x = Ok x ;        Ok x ▷ f = f x ;        orelse (Ok x) ym = Ok x ;        failure = Fail ;
                         Fail ▷ f = Fail ;       orelse Fail ym = ym ;
```

Let us prove associativity. We proceed by structural induction (case analysis)

```
(Fail ▷ f) ▷ g = Fail ▷ g = Fail = Fail ▷ (λx → f x ▷ g) ;
(Ok x ▷ f) ▷ g = f x ▷ g  = Ok x ▷ (λu → f u ▷ g) ;
```

Let us prove the two unit laws.

```
Fail ▷ Ok = Fail ;        Ok x ▷ Ok = Ok x ;        (for left unitality)
Ok x ▷ f = f x ;                                    (for right unitality)
```

POPL - Class 3

Exercise 2.a. We define `result x = Ok x ;` `Ok x ▷ f = f x ;` `Fail v ▷ f = Fail v .`

Exercise 2.b. We define `failure v = Fail v ;` `orelse (Ok x) f = Ok x ;` `orelse (Fail v) f = f v ;`

Exercise 2.c. We do not need to modify the AST constructors to define *fail*, but we need to for *orelse*.

```
eval (Orelse e1 e2) env = orelse (eval e1 env) (λv → eval e2 env ▷ (λf → apply f [v]))
primitive "fail" (λx → failure x)
```

Exercise 2.d. Let us first reason with the interpreter. Let us propose changing evaluation order.

```
eval [| 2 orelse fail(3) |] ρ ≡
orelse (eval [|2|] ρ) _       ≡
orelse (Ok (IntVal 2)) _      ≡
Ok (IntVal 2).
```

```
eval (Orelse e1 e2) env =
  eval e2 env ▷ (λf →
    orelse (eval e1 env) (λv →
      apply f [v]))
```

POPL - Class 3

Exercise 3. We define $M\alpha \equiv Mem \to Maybe\ (\alpha, Mem)$.

```
result x m = Just (x,m)

(xm ▷ f) m = case (xm m) of
               (Just (x,m')) → f x m'
               Nothing → Nothing
```

We define the following operations.

```
new m = Just (fresh m)   -- note that fresh gives (Value, Mem).
get l m = Just (contents m l, m)
put l v m = Just ((), update m l v)
failure m = Nothing
orelse xm ym m = case xm m of
                   Just (x,m') → Just (x,m')
                   Nothing → ym m
```

POPL - Class 3

Exercise 4. We define $M\,\alpha \equiv Mem \to (Maybe\ \alpha, Mem)$.

```
result x m = (Just x, m)

(xm ▷ f) m = case (xm m) of
               (Just (x, m')) → f x m'
               (Nothing, m')  → (Nothing, m')
```

We define the following operations.

```
new m = let (a, m') = fresh m in (Just a, m')
get l m = (Just (contents m l), m)
put l v m = (Just (), update m l v)
failure m = (Nothing, m)
orelse xm ym m = case xm m of
                   (Just x, m')  → (Just x, m')
                   (Nothing, m') → ym m'   -- ym m also valid, would miss unitality of orelse
```

POPL - Class 3

Exercise 5.a. With $Mem \to Maybe(\alpha, Mem)$, any error prevents changes to memory. With $Mem \to (Maybe\ \alpha, Mem)$, changes to memory are allowed even on errors.

Exercise 5.b. The latter allows us to carry a single copy of memory.

Exercise 5.c. The result depends on whether changes to memory are reverted on failure.

```
val x = new();; x := 0;; x := 1; fail();; !x ;;
```

POPL - Class 3

Exercise 8.

```
fib :: Int → (Int → α) → α
fib n k = if n ≤ 1 then k n
          else fib (n-1) (λx →
               fib (n-2) (λy →
                    k (x+y)))
```

POPL - Class 3

Exercise 9. Recall that the definition is

$$\mathsf{result}\ x\ k = k\ x \qquad (xm \triangleright f)\ k = xm\ (\lambda x \to f\ x\ k)$$

Let us prove the monad axioms.

$$\begin{aligned}
&(\mathsf{result}\ x \triangleright f)\ k && \equiv \langle \text{def } \triangleright \rangle \\
&\mathsf{result}\ x\ (\lambda y \to f\ y\ k) && \equiv \langle \text{def result} \rangle \\
&f\ x\ k
\end{aligned}$$

$$\begin{aligned}
&(xm \triangleright \mathsf{result})\ k && \equiv \langle \text{def } \triangleright \rangle \\
&xm\ (\lambda x \to \mathsf{result}\ x\ k) && \equiv \langle \text{def result} \rangle \\
&xm\ k
\end{aligned}$$

$$\begin{aligned}
&((xm \triangleright f) \triangleright g)\ k && \equiv \langle \text{def } \triangleright \rangle^{\to} \\
&(xm \triangleright f)\ (\lambda x \to g\ x\ k) && \equiv \langle \text{def } \triangleright \rangle^{\to} \\
&xm\ (\lambda y \to f\ y\ (\lambda x \to g\ x\ k)) && \equiv \langle \text{def } \triangleright \rangle^{\leftarrow} \\
&xm\ (\lambda y \to (f\ y \triangleright g)\ k) && \equiv \langle \text{def } \triangleright \rangle^{\leftarrow} \\
&(xm \triangleright (\lambda x \to f\ x \triangleright g))\ k
\end{aligned}$$

POPL - Class 3

EXTRA. Continuation passing is cocartesian coclosed.

```
type Cont x = (x → A) → A
type Minus x y = (x, y → A)

cocurry :: (x → Cont (y+z)) → ((x-z) → Cont y)
cocurry f (x,w) q = f x (\case
                           Left y → q y
                           Right z → w z)

councurry :: ((x-z) → Cont y) → (x → Cont (y+z))
councurry f x v = f (x , λz → v (Right z)) (λy → v (Left y))
```

POPL - Class 4

- Please remind me to take attendance.
- And to give you time to fill feedback forms.

- Submissions were just excellent. And late submissions were very useful and mostly great.
- Some new techniques (CEK, CPS) may be useful to detail.

POPL - Class 4

Exercise 1. Let us write the CPS translation.

```
revk [] K = K []
revk (x:xs) K = revk xs (λ rs → K(rs ++[x]))
```

By induction, let us rewrite continuations in `revk xs id` as appending.

Base: `id = (++[])`
Given `K = (++zs)`, then:

```
(λ rs → K(rs ++[x])) =
(λ rs → (++zs)(rs ++[x]))
(λ rs → (rs ++[x])++zs)
(λ rs → rs ++ (x:zs))
(++(x:zs)).
```

Inspired by this, we can write revk in "appendix-passing style".

```
revx :: [Int] → [Int] → [Int]
revx [] apx = apx
revx (x:xs) apx = rev xs (x:apx)
```

POPL - Class 4

Exercise 2.

```
fib :: Int → (Int → α) → α
fib n k = if n ≤ 1 then k n
          else fib (n-1) (λx →
               fib (n-2) (λy →
               k (x+y)))
```

F1 → `fib (n-1) (λx →`

F2 → `fib (n-2) (λy →`

Let us define a datatype of continuations. With it, we defunctionalize.

```
data Cont = F0 | F1 Int Cont | F2 Int Cont
```

(F0: (id); F1: (n) (k); F2: (x) (k))

```
fib :: Int → Cont → Int
fib n k =
    if n ≤ 1 then apply k n
             else fib (n-1) (F1 n k)
```

```
apply :: Cont → Int → Int
apply F0 a = a
apply (F1 n k) x = fib (n-2) (F2 x k)
apply (F2 x k) y = apply k (x+y)
```

POPL - Class 4

Exercise 4. All strings are of the form ab^n for some $n \in \mathbb{N}$

- $ab = ab^1$
- $ax = ab^n \Rightarrow x = b^n \Rightarrow axx = ab^{2n}$
- $abbbx = ab^n \Rightarrow x = b^{n-3} \Rightarrow ax = ab^{n-3}$

We know that $5 = 1*2*2*2-3$. We prove that all strings are of the form ab^n with $3 \nmid n$.

- $ab = ab^1$, and $3 \nmid 1$
- $ax = ab^n \Rightarrow x = b^n \Rightarrow axx = ab^{2n}$, and $3 \nmid n$ implies $3 \nmid 2n$
- $abbbx = ab^n \Rightarrow x = b^{n-3} \Rightarrow ax = ab^{n-3}$, and $3 \nmid n$ implies $3 \nmid n-3$

Now, $abbb = ab^n$ implies $n = 3$, leading to contradiction.

POPL - Class 4

Exercise 6.a. The integers, $\mathbb{Z}$, have no upper bound for the chain $\{n\}_{n\in\mathbb{Z}}$: if u were an upper bound, $u+1 \in \mathbb{Z}$, so $u \geq u+1$ and $0 \geq 1$. Negation, $(\mathbb{Z},\leq)\to(\mathbb{Z},\geq)$ is monotone: any bottom element would imply a top one.

Exercise 6.b. Let $\mathbb{N} \cup \{\omega_1, \omega_2\}$ with $\omega_1 \geq n$ and $\omega_2 \geq n$. A lower upper bound must be lower than both ω_1 and ω_2, so none of them. It should be a natural, so not an upper bound.

Exercise 6.c. Any chain $\{S_i\}_i$ has a least upper bound, $\bigcup_i S_i$. Let us show F preserves these.

$$
\begin{aligned}
F(\textstyle\bigcup_i S_i) &= \{1\} \cup \{n+2 \mid n \in \textstyle\bigcup_i S_i\} && \text{(by def. of set comprehensions)}\\
&= \{1\} \cup \textstyle\bigcup_i \{n+2 \mid n \in S_i\} && \text{(by distributivity)}\\
&= \textstyle\bigcup_i \{1\} \cup \{n+2 \mid n \in S_i\} && \text{(by def.)}\\
&= \textstyle\bigcup_i F(S_i).
\end{aligned}
$$

The least fixed point is, by Kleene-Tarski, the least upper bound of $\bot \leq F\bot \leq FF\bot \leq \dots$

$$\emptyset \subseteq \{1\} \subseteq \{1,3\} \subseteq \{1,3,5\} \subseteq \dots$$

which are the odd natural numbers.

POPL - Class 4

Exercise 5.

Recall the CEK machine so far.

$$\langle e_1(e_2),\ env,\ K\rangle \longrightarrow \langle e_1,\ env,\ (\square(e_2),env):K\rangle$$
$$\langle \text{lambda}\ (x)\ e,\ env,\ K\rangle \longrightarrow [K, (\text{lambda}\ (x)\ e, env)]$$
$$\langle n, env, K\rangle \longrightarrow [K, n]$$
$$\langle x, env, K\rangle \longrightarrow [K, env(x)]$$
$$[(\square(e), env):K, v] \longrightarrow \langle e, env, (v\square):K\rangle$$
$$[((\text{lambda}\ (x)\ e, env)(\square)):K, v] \longrightarrow \langle e, env[x\mapsto v], K\rangle$$

We add rules to evaluate succ and pred, and their continuations.

$$\langle \text{succ}, env, K\rangle \longrightarrow [K, \text{succ}]$$
$$\langle \text{pred}, env, K\rangle \longrightarrow [K, \text{pred}]$$
$$[\text{succ}\,\square : K, n] \longrightarrow [K, n+1]$$
$$[\text{pred}\,\square : K, n] \longrightarrow [K, n-1]$$

POPL - Class 4

Exercise 5.

$\langle succ(pred(x)),\ x \mapsto 1,\ Show \rangle$
$\langle succ,\ x \mapsto 1,\ (\square(pred(x)), x \mapsto 1) : Show \rangle$
$[(\square(pred(x)),\ x \mapsto 1) : Show,\ succ]$
$\langle pred(x),\ x \mapsto 1,\ succ\ \square : Show \rangle$
$\langle pred,\ x \mapsto 1,\ (\square(x), x \mapsto 1) : succ\ \square : Show \rangle$
$[(\square(x), x \mapsto 1) : succ\ \square : Show,\ pred]$
$\langle x,\ x \mapsto 1,\ pred\ \square : succ\ \square : Show \rangle$
$[pred\ \square : succ\ \square : Show,\ 1]$
$[succ\ \square : Show,\ 0]$
$[Show,\ 1]$

$\langle e_1(e_2),\ env,\ K \rangle \longrightarrow \langle e_1,\ env,\ (\square(e_2), env) : K \rangle$
$\langle lambda\ (x)\ e,\ env,\ k \rangle \longrightarrow [K, (lambda\ (x)\ e, env)]$
$\langle n, env, k \rangle \longrightarrow [k, n]$
$\langle x, env, K \rangle \longrightarrow [K, env(x)]$
$[(\square(e), env) : k, v] \longrightarrow \langle e, env, (v\square) : k \rangle$
$[((lambda\ (x)\ e,\ env)(\square)) : K,\ v] \longrightarrow \langle e,\ env[x \mapsto v],\ K \rangle$
$\langle succ, env, k \rangle \longrightarrow [K, succ]$
$\langle pred, env, k \rangle \longrightarrow [K, pred]$
$[succ\ \square : k,\ n] \longrightarrow [K, n+1]$
$[pred\ \square : k,\ n] \longrightarrow [K, n-1]$