Principes de Programmation
TD4

28 mars 2018

Exercice 1 (Une monade est un foncteur).
On rappelle que la classe `Functor` est définie par :

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

La classe `Monad` est définie par : [1]

```
class Monad m where
  return :: a -> m a
  (=<<)  :: (a-> m b) -> m a -> m b
```

1. Encodez les foncteurs dans les monades. C'est à dire, supposez que `m` est une monade, et implémentez `Functor m`.
   ⇝ L'idée est de regarder le type de `fmap::(a->b) -> m a -> m b`, de voir que celà resemble beaucoup à `(=<<):: (a-> m b) -> m a -> m b` sauf que la fonction en entré ne retourne pas dans la monade... pour ca il sufit de composer avec `return` :

   ```
   instance Monad m => Functor m where
     fmap f x = (return.f) =<< x
   ```

Cela ne suffit pas pour dire qu'une monade est toujours un foncteur, il faut vérifier aussi les axiomes de foncteur.

On rappelle qu'un foncteur doit vérifier les axiomes suivants :

```
fmap id     ~=   id                  -- eq. 1f
fmap (f.g)  ~=   fmap f . fmap g     -- eq. 2f
```

De la même façon, une monade doit vérifier les axiomes suivants :

```
return =<< x       ~=   x                              -- eq. 1m
f =<< (return x)   ~=   f x                            -- eq. 2m
f =<< (g =<< x)    ~=   (\y -> f =<< (g y) ) =<< x     -- eq. 3m
```

1. Attention, pour des raisons historiques, il faut définir $(>>=) :: ma- > (a- > mb)- > mb$ plutôt que =<<...

2. (dificile) Faites ces vérifications. Càd, supposez que `m` vérifie les axiomes de monade, et vérifiez que votre encodage vérifie les axiomes de foncteur.
   ⇝ Equation 1f, for all `x` :

```
fmap id x    ~=  (return.id) =<< x                -- encodage fmap
             ~=  (\y -> return (id y)) =<< x      -- def (.)
             ~=  (\y -> return ((\z->z) y)) =<< x -- def id
             ~=  (\y -> return y) =<< x           -- beta
             ~=  return =<< x                     -- eta
             ~=  x                                -- eq. 1m
```

   Equation 2f, for all `x` :

```
fmap (f.g) x  ~= (return.(f.g)) =<< x                          -- encodage fmap
              ~= ((return.f).g) =<< x                          -- cf. TD3
              ~= (\y -> (return.f).g) y) =<< x                 -- eta
              ~= (\y -> (return.f) (g y)) =<< x                -- def (.)
              ~= (\y -> (return.f) =<< (return (g y))) =<< x   -- eq. 2m
              ~= (\y -> (return.f) =<< ((return.g) y)) =<< x   -- def (.)
              ~= (return.f) =<< ((return.g) =<< x)             -- eq. 3m
              ~= (return.f) =<< (fmap g x)                     -- encodage fmap
              ~= fmap f (fmap g x)                             -- encodage fmap
              ~= (fmap f . fmap g) x                           -- def .
```

Exercice 2 (Les monades).
On rappelle que `Maybe` est définie ainsi :

```
data Maybe a = Something a  |  Nothing
```

1. Montrez que `Maybe` est une monade.
   ⇝ Equation 1f, for all `x` :

   ```
   instance Monad Maybe where
     return          = Just
     Nothing  >>= f  = Nothing
     (Just x) >>= f  = f x
   ```

   Il faut encore vérifier les équations :
   Eq. 1m, on sépare les cas `x=Nothing` et `x = Just y` :

   ```
   return =<< Nothing   ~= Nothing     -- def (=<<)
   return =<< (Just y)  ~= return y    -- def (=<<)
                        ~= Just y      -- def return
   ```

   Eq. 2m :

   ```
   f =<< (return x)  ~=  f =<< (Just x)  -- def return
                     ~=  f x             -- def (=<<)
   ```

   Eq. 3m, on sépare les cas `x=Nothing` et `x = Just z` :

   ```
   f =<< (g =<< Nothing) ~= f =<< Nothing                     -- def (=<<)
                         ~= Nothing                           -- def (=<<)
                         ~= (\y -> f =<< (g y) ) =<< Nothing  -- def (=<<)

   f =<< (g =<< Just z)  ~= f =<< (g z)                        -- def (=<<)
                         ~= (\y -> f =<< (g y) ) z             -- beta
                         ~= (\y -> f =<< (g y) ) =<< (Just z)  -- def (=<<)
   ```

On définit `Either a b` par

```
data Either a b = Left a | Right b
```

2. Montrez que pour tout `a`, `Either a` est une monade.
   ⇝ Equation 1f, for all `x` :

   ```
   instance Monad Either a where
     return          = Right
     (Left  y) >>= f  = Left y
     (Right y) >>= f  = f y
   ```

   Il faut encore vérifier les équations :
   Eq. 1m, on sépare les cas `x = Left y` et `x = Right y` :

   ```
   return =<< (Left  y)  ~= Left y    -- def (=<<)
   return =<< (Right y)  ~= return y  -- def (=<<)
                         ~= Right y   -- def return
   ```

Eq. 2m :

```
f =<< (return x)  ~=  f =<< (Right x)  -- def return
                  ~=  f x              -- def (=<<)
```

Eq. 3m, on sépare les cas `x = Left z` et `x = Right z` :

```
f =<< (g =<< (Left  z)) ~= f =<< (Left z)                    -- def (=<<)
                        ~= Left z                            -- def (=<<)
                        ~= (\y -> f =<< (g y) ) =<< Left z   -- def (=<<)

f =<< (g =<< (Right z)) ~= f =<< (g z)                         -- def (=<<)
                        ~= (\y -> f =<< (g y) ) z              -- beta
                        ~= (\y -> f =<< (g y) ) =<< (Right z)  -- def (=<<)
```

3. Montrez que les listes forment une monade.
   ⇝ Equation 1f, for all `x` :

```
instance Monad [] where
  return x     = [x]
  []    >>= f  = []
  (x:l) >>= f  = (f x)++(l >>= f)
```

Il faut encore vérifier les équations :
Eq. 1m, on fait une récursion sur la taille de la liste avec les cas `x = []` et `x = (y:l)` :

```
return =<< []     ~= []                          -- def (=<<)
return =<< (y:l)  ~= (return y)++(return =<< l)  -- def (=<<)
                  ~= (return y)++l               -- hypothese de recursion
                  ~= [y]++l                      -- def return
                  ~= y:([]++l)                   -- def ++
                  ~= y:l                         -- def ++
```

Eq. 2m :

```
f =<< (return x)  ~=  f =<< [x]          -- def return
                  ~=  (f x) ++ []        -- def (=<<)
                  ~=  (f x)              -- Axiome 1 (voir plus loin)
```

Eq. 3m, on fait une récursion sur la taille de la liste avec les cas `x = []` et `x = (y:l)` : On a d'abord le cas `x = []` :

```
f =<< (g =<< [])    ~= f =<< []                        -- def (=<<)
                    ~= []                              -- def (=<<)
                    ~= (\y -> f =<< (g y) ) =<< []     -- def (=<<)
f =<< (g =<< (z:l)) ~= f =<< ((g z)++(g=<<l))          -- def (=<<)
                    ~= (f =<< (g z))
                       ++ (f=<<(g=<<l))                -- Axiome 2
                    ~= ((\y -> f =<< (g y)) z)
                       ++ (f=<<(g=<<l))                -- beta
                    ~= ((\y -> f =<< (g y)) z)
```

```
                    ++ ((\y -> f =<< (g y)) =<< l)   -- HR
                ~= (\y -> f =<< (g y)) =<< (z:l)    -- def (=<<)
```

On doit encore prouver nos axiomes :
Axiome 1 ; par récurence sur la taille de `l`, on prouve que `l++[]~=l` :

```
[]    ++ [] ~= []         -- def (++)
(x:l) ++ [] ~= x:(l++[])  -- def (++)
            ~= x:l        -- hypothese de recursion
```

Axiome 2 ; par récurence sur la taille de `l1`, on prouve que
`f =<< (l1++l2) = (f =<< l1)++(f =<< l2)` :

```
f =<< ([]++l2)      ~= f =<< l2                              -- def (++)
                    ~= []++(f =<< l2)                        -- def (++)
                    ~= (f =<< [])++(f =<< l2)                -- def (=<<)
f =<< ((x:l1)++l2)  ~= f =<< (x:(l1++l2))                    -- def (++)
                    ~= (f x) ++ (f =<< (l1++l2))             -- def (=<<)
                    ~= (f x) ++ ((f =<< l1) ++ (f =<< l2))   -- HR
                    ~= ((f x) ++ (f =<< l1)) ++ (f=<< l2)    -- TD3
                    ~= (f =<< (x:l1)) ++ (f =<< l2)          -- def (=<<)
```

On définit les arbres de préfixe ainsi :

```
data PTree a = Noeud (Char -> PTree a)  |  Feuille a
```

4. Montrez que les arbres de préfixe forment une monade.

On définit `Futur` qui encode le `yield` de python :

```
type Tick = ()
data Futur a = Available a | Later (Tick -> Futur a)
```

5. Montrez que `Yield` est une monade.

Un type à état est défini (presque) comme suit :

```
type ST s a = (s -> (a,s))
```

où `s` est le type de l'état courant et `a` est le type sur lequel on travaille.

6. Montrez que `ST s` forme une monade (quelque soit `s`).

La monade de continuation [2] est définie par :

```
type Cont r a = (a -> r) -> r
```

7. Montrez que `Cont r` forme une monade (quelque soit `r`).

2. importée de Control.Monad.Trans.Cont