Principes de Programmation
TD3

14 mars 2018

Exercice 1 (Monoïdes).
La classe `Monoid` est définie par :

```
class Monoid m where
  mempty :: m
  (<>)   :: m -> m -> m
```

Un monoid `m` doit vérifier les axiomes suivants :

```
mempty <> x        ~=   x
x <> mempty        ~=   x
(x <> y) <> z      ~=   x <> (y <> z)
```

1. Donnez des exemples de monoides.
   ⇝ `Int` avec `+`, `Int` avec `*`, `Float` avec `+`, `Float` avec `*`, en général n'importe quel `Num` avec `+` ou `*`,
   `[a]` avec `++`,
   `Bool` avec `||` ou `&&`.
   ...
   Aparté : `Num a` est doublement un monoïde (sommes et produit). En haskell on les distingue grâces à un wraper : [1]

   ```
   newtype Sum a = Sum a

   instance Num a => Monoid (Sum a) where
     mempty              = Sum 0
     (Sum x) <> (Sum y)  = Sum (x + y)

   data Product a = Product a

   instance Num a => Monoid (Product a) where
     mempty                      = Product 1
     (Product x) <> (Product y)  = Product (x * y)
   ```

1. "newtype" est comme "data" mais avec un seul constructeur : c'est un wraper

2. Montrez que `[a]` est un monoid. [2]

   ⇝

```haskell
instance Monoid [a] where
  mempty        = []
  l <> ll       = l ++ ll
```

   ou encore (équivalent) :

```haskell
instance Monoid [a] where
  mempty        = []
  []    <> ll   = ll
  (x:l) <> ll   = x : (l <> ll)
```

   On doit prouver que c'est correcte :
   — On vérifie le type.
   — On vérifie la première équation :

```haskell
    mempty <> l   ~=  [] <> l                  -- def mempty
                  ~=  l                        -- def <>
```

   — On vérifie la seconde par induction sur la taille de la première liste :
   — si elle est vide :

```haskell
[]    <> mempty   ~=  mempty                   -- def <>
                  ~=  <>                       -- def mempty
```

   — si elle est non-vide :

```haskell
(x:l) <> mempty   ~=  x : (l <> mempty)        -- def <>
                  ~=  x : l                    -- hypoth. d'induction
```

   — On vérifie la troisième par induction sur la taille de la première liste :

```haskell
([]     <> l2) <> l3  ~=  l2 <> l3                  -- def <>
                      ~=  [] <> (l2 <> l3)          -- def <>
((x:l1) <> l2) <> l3  ~=  (x : (l1 <> l2)) <> l3    -- def <>
                      ~=  x : ((l1 <> l2) <> l3)    -- def <>
                      ~=  x : (l1 <> (l2 <> l3))    -- hypoth. d'induction
                      ~=  (x:l1) <> (l2 <> l3)      -- def <>
```

3. (bonus) Montrez que `(a -> a)` est un monoid. [3]

   ⇝

```haskell
instance Monoid (a->a) where
  mempty        = \x -> x
  f <> g        = \x -> f (g x)
```

   On doit prouver que c'est correcte :
   — On vérifie le type.
   — On vérifie la première équation :

2. Uniquement valable sur les listes finies
3. Malheureusement, celui-ci n'est pas implémenté dans Haskell. La raison est qu'il a fallu choisir avec celui de la question suivante dont le type est en conflit.

```
mempty <> g     ~=  (\y -> y) <> g                  -- def mempty
                ~=  \x -> (\y -> y) (g x)          -- def <>
                ~=  \x -> g x                      -- beta
                ~=  g                              -- eta
```

— On vérifie la seconde équation :

```
f <> mempty     ~=  f <> (\y -> y)                 -- def mempty
                ~=  \x -> f ((\y -> y) x)          -- def <>
                ~=  \x -> f x                      -- beta
                ~=  f                              -- eta
```

— On vérifie la troisième équation :

```
(f <> g) <> h   ~=  \x -> (f <> g) (h x)           -- def <>
                ~=  \x -> (\y -> f (g y)) (h x)    -- def <>
                ~=  \x -> f (g (h x))              -- beta
                ~=  \x -> f ((\y -> g (h y)) x)    -- beta
                ~=  \x -> f ((g <> h) x)           -- def <>
                ~=  f <> (g <> h)                  -- def <>
```

(hors programme) Attention, pour des raisons qui ne sont pas au cours, vous ne pourrez pas écrire l'instance ci dessus en Haskell, il faut introduire un wraper :

```
newtype Endo a = Endo (a->a)
instance Monoid (Endo a) where
  mempty               = Endo (\x -> x)
  (Endo f) <> (Endo g) = Endo (\x -> f (g x))
```

4. Montrez que si `m` est un monoid, alors `(a -> m)` est un monoid.
   ⇝

```
instance Monoid m => Monoid (a->m) where
  mempty        = \_ -> mempty
  f <> g        = \x -> (f x) <> (g x)
```

On doit prouver que c'est correcte :

— On vérifie le type.

— On vérifie la première équation :

```
mempty <> g     ~=  (\_ -> mempty) <> g                -- def mempty
                ~=  \x -> ((\_ -> mempty) x) <> (g x)  -- def <>
                ~=  \x -> mempty <> (g x)              -- beta
                ~=  \x -> g x                          -- regle 1 (sur a)
                ~=  g                                  -- eta
```

— On vérifie la seconde équation :

```
f <> mempty     ~=  f <> (\_ -> mempty)                -- def mempty
                ~=  \x -> (f x) <> ((\_ -> mempty) x)  -- def <>
                ~=  \x -> (f x) <> mempty              -- beta
                ~=  \x -> f x                          -- regle 2 (sur a)
                ~=  f                                  -- eta
```

— On vérifie la troisième équation :

```
(f <> g) <> h  ~=  (\y -> (f y) <> (g y) ) <> h              -- def <>
               ~=  \x -> ((\y -> (f y) <> (g y)) x) <> (h x) -- def <>
               ~=  \x -> ((f x) <> (g x)) <> (h x)           -- beta
               ~=  \x -> (f x) <> ((g x) <> (h x))           -- regle 3 (sur a)
               ~=  \x -> (f x) <> ((\y -> (g y) <> (h y)) x) -- beta
               ~=  f <> (\y -> (g y) <> (h y) )              -- def <>
               ~=  f <> (g <> h)                             -- def <>
```

(hors programme) Attention, pour des raisons qui ne sont pas au cours, vous ne pourrez pas écrire l'instance ci dessus en Haskell, il faut introduire un wraper :

```
newtype Monoid m => Writer a m = Writer (a->m)
instance Monoid m => Monoid (Writer a m) where
  mempty                  = Writer (\_ -> mempty)
  (Writer f) <> (Writer g) = Writer (\x -> (f x) <> (g x))
```

5. (bonus) On définit :

```
newtype Dual a = Dual a

instance Monoid a => Monoid (Dual a) where
  mempty              = Dual mempty
  (Dual x) <> (Dual y) = Dual (y <> x)
```

Que fait `Dual` ?

⇝ Lorsque le type `a` définit un monoid (`mempty`,`<>`), rien ne dit que `x<>y = y<>x`; dans le cas de `(a->a)`, par exemple on a vu que c'était faux. Du fait de cette non-commutativité, on peut vouloir utiliser le monoid avec l'opération inverse, c'est à dire le monoid dual.
On doit prouver encore prouver que c'est correcte :

— On vérifie le type.

— On vérifie la première équation :

```
mempty <> (Dual x)  ~=  (Dual mempty) <> (Dual x)   -- def mempty
                    ~=  Dual (x <> mempty)          -- def <>
                    ~=  Dual x                      -- regle 2 (sur a)
```

— On vérifie la seconde équation :

```
(Dual x) <> mempty  ~=  (Dual x) <> (Dual mempty)   -- def mempty
                    ~=  Dual (mempty <> x)          -- def <>
                    ~=  Dual x                      -- regle 1 (sur a)
```

— On vérifie la troisième équation :

```
((Dual x) <> (Dual y)) <> (Dual z)  ~=  (Dual (y <> x)) <> (Dual z)        -- def <>
                                    ~=  Dual (z <> (y <> x))              -- def <>
                                    ~=  Dual ((z <> y) <> x)              -- regle 3 (sur a)
                                    ~=  (Dual x) <> (Dual (z <> y))       -- def <>
                                    ~=  (Dual x) <> ((Dual y) <> (Dual z)) -- def <>
```

Exercice 2 (Foldables (Bonus)).
La classe `foldable` est définie par :

```
class Foldable struct where
foldr   :: (a -> b -> b) -> b -> struct a -> b
foldl   :: (b -> a -> b) -> b -> struct a -> b
foldMap :: Monoid m => (a -> m) -> struct a -> m
```

Il n'y a pas d'équations explicite entre ces définitions, mais il n'est demandé qu'une fonctions, les autres étant traduites.

1. Écrire `foldl`, `foldr` et `foldMap` pour les listes.
   ⇝
   ```
   instance Foldable [a] where
     foldr _ x []     = x
     foldr f x (y:l) = f y (foldr f x l)

     foldl _ x []     = x
     foldl f x (y:l) = foldl f (f x y) l

     foldMap f []     = mempty
     foldMap f (y:l) = (f y) <> (foldMap f l)
   ```
2. Comment écrire `foldMap` à partir de `foldr` ?
   ⇝
   ```
   foldMapr :: Monoid m, Foldable struct  => (a -> m) -> struct a -> m
   foldMapr f s = foldr (\x u -> (f x) <> u) mempty s
   ```
   ou alors (mais illisible) :
   ```
   foldMapr f = foldr ((<>).f) mempty
   ```
3. Vérifiez que la traduction est correcte sur les listes.
   ⇝ Par induction sur la taille de la liste :
   ```
   foldMapr f []     ~= foldr (\x u -> (f x) <> u) mempty []                          -- def foldMapr
                     ~= mempty                                                        -- def foldr
                     ~= foldMap f []                                                  -- def foldMap
   foldMapr f (y:l)  ~= foldr (\x u -> (f x) <> u) mempty (y:l)                       -- def foldMapr
                     ~= (\x u -> (f x) <> u) y (foldr (\x u -> (f x) <> u) mempty l)) -- def foldr
                     ~= (\u -> (f y) <> u) (foldr (\x u -> (f x) <> u) mempty l))     -- beta
                     ~= (f y) <> (foldr (\x u -> (f x) <> u) mempty l))               -- beta
                     ~= (f y) <> (foldMapr f l)                                       -- foldMapr
                     ~= (f y) <> (foldMap f l)                                        -- induction
                     ~= foldMap f (y:l)                                               -- foldMap
   ```
4. Comment écrire `foldMap` à partir de `foldl` ?
   ⇝
   ```
   foldMapl :: Monoid m, Foldable struct  => (a -> m) -> struct a -> m
   foldMapl f s = foldl (\u x -> (f x) <> u) mempty s
   ```

5. Vérifiez que la traduction est correcte sur les listes.
   ⇝ Soit :

```
foldMapl' :: Monoid m, Foldable struct  => (a -> m) -> struct a -> m -> m
foldMapl' f  v s= foldl (\u x -> (f x) <> u) v s
```

   On prouve que

```
foldMapl' f v s ~= (foldMapl f s) <> v
```

   par induction sur la taille de la liste

```
foldMapl' f v []     ~= foldl (\u x -> (f x) <> u) v []                          -- def foldMapl'
                     ~= v                                                       -- def foldr
                     ~= mempty  <> v                                            -- regle 1
                     ~= foldMap f [] <> v                                       -- def foldMap
foldMapl' f v (y:l)  ~= foldr (\u x -> (f x) <> u) v (y:l)                       -- def foldMapl'
                     ~= foldl (\u x -> (f x) <> u) ((\u x -> (f x) <> u) v y) l  -- def foldl
                     ~= foldl (\u x -> (f x) <> u) ((\x -> (f x) <> v) y) l      -- beta
                     ~= foldl (\v x -> (f x) <> u) ( (f y) <> v ) l              -- beta
                     ~= foldMapl' f  ( (f y) <> v ) l                           -- foldMapl'
                     ~= (foldMapl f l) <> ((f y) <> v)                          -- induction
                     ~= ((foldMapl f l) <> (f y)) <> v                          -- regle 3
                     ~= (foldMap f (y:l)) <> v                                  -- foldMap
```

   En particulier, on a

```
foldMapl f l ~= foldMapl' f mempty l
             ~= (foldMap f (y:l)) <> mempty
             ~= foldMap f l
```

6. Comment écrire `foldr` à partir de `foldMap` ? [4]
   ⇝

```
foldr' :: Foldable struct  => (a -> b -> b) -> b -> struct a -> b
foldr' f x s = foldMap f s x
```

   On utilise le fait que `(b->b)` est un monoid et donc que `foldMap :: Foldable struct  => (a -> (b -> b)) -> struct a -> (b -> b)`.
   Attention, pour des raisons qui ne sont pas au cours, vous ne pourrez pas écrire l'instance ci dessus en Haskell, il faut utiliser le wraper `Endo` :

```
foldr' :: Foldable struct  => (a -> b -> b) -> b -> struct a -> b
foldr' f x s = getEndo (foldMap (Endo . f) s) x


getEndo (Endo g) = g
```

7. Vérifiez que la traduction est correcte sur les listes.
   ⇝ Par induction sur la taille de la liste :

4. On supposera que $(a \to a)$ est bien un monoid. Dans Haskell, il est utilisé un wraper qui complexifie les choses.

```
foldr' f x []      ~= foldMap f [] x                -- def foldr'
                   ~= mempty x                      -- def foldMap
                   ~= (\y->y) x                     -- def mempty pour (a->a)
                   ~= x                             -- beta
                   ~= foldr f x []                  -- def foldr
foldr' f x (y:l)   ~= foldMap f (y:l) x             -- def foldr'
                   ~= (f y) <> (foldMap f l) x      -- def foldMap
                   ~= (\z -> f y (foldMap f l z)) x -- def <> pour (a->a)
                   ~= f y (foldMap f l x)           -- beta
                   ~= f y (foldr' f l x)            -- def foldr'
                   ~= f y (foldr f x l)             -- induction
                   ~= foldr f x (y:l)               -- def foldr
```

8. Comment écrire `foldl` à partir de `foldMap` ?

   ⇝

   ```
   foldl' :: Foldable struct  => (b -> a -> b) -> b -> struct a -> b
   foldl' f x s = getDual (foldMap (Dual . (flip f)) s) x

   getDual (Dual g) = g
   flip f = \x y -> f y x
   ```

   On utilise le fait que `Dual (b->b)` est un monoid.
   Attention, pour des raisons qui ne sont pas au cours, vous ne pourrez pas écrire l'instance ci dessus en Haskell, il faut utiliser le wraper `Endo` :

   ```
   foldl' :: Foldable struct  => (b -> a -> b) -> b -> struct a -> b
   foldl' f x s = (getEndo . getDual) foldMap (Dual . Endo . (flip f)) s x
   ```

9. Vérifiez que la traduction est correcte sur les listes.

   ⇝ Par induction sur la taille de la liste :

```
foldl' f x []  ~= getDual (foldMap (Dual . (flip f)) []) x   -- def foldl'
               ~= getDual mempty x                           -- def foldMap
               ~= getDual (Dual (\y->y)) x                   -- def mempty pour Dual (a->a)
               ~= (\y->y) x                                  -- def getDual
               ~= x                                          -- beta
```

```
foldl' f x (y:l)  ~= getDual (foldMap (Dual . (flip f)) (y:l)) x                    -- def foldr'
                  ~= getDual ((Dual . (flip f)) y) <>
                        (foldMap (Dual . (flip f)) l)) x                            -- def foldMap
                  ~= getDual ((Dual . (flip f)) y) <>
                        (Dual (getDual (foldMap (Dual . (flip f)) l))) x            -- Wraper
                  ~= getDual ((Dual ((flip f) y)) <>
                        (Dual (\z -> getDual (foldMap (Dual . (flip f)) l) z ))) x  -- eta
                  ~= getDual ((Dual ((flip f) y)) <>
                        (Dual (\z -> foldl' f z l))) x                              -- induction
                  ~= getDual ((Dual ((flip f) y)) <>
                        (Dual (\z -> foldl' f z l))) x                              -- def .
                  ~= getDual (Dual( (\z -> foldl' f z l)  <> ((flip f) y) )) x      -- def <> sur Dual b
                  ~= (\z -> foldl' f z l)  <> ((flip f) y) x                        -- def <> sur getDual
                  ~= (\z -> foldl' f z l) ((flip f) y x)                            -- def <> sur (a->a)
                  ~= foldl' f ((flip f) y x) l                                      -- beta
                  ~= foldl' f (f x y) l                                             -- def flip
                  ~= foldl f (f x y) l                                              -- induction
                  ~= foldl f x (y:l)                                                -- def foldl
```

10. Montrez que `Tree` est foldable.
    ⇝ Il suffit d'implémenter `foldMap` :

    ```
    instance Foldable Tree a where
      foldMap f Leaf         = mempty
      foldMap f (Node x l r) = (foldMap f l) <> (f x) <> (foldMap f r)
    ```

11. Montrez que `Maybe` est foldable.
    ⇝ Il suffit d'implémenter `foldMap` :

    ```
    instance Foldable Maybe a where
      foldMap f Nothing  = mempty
      foldMap f (Just x) = f x
    ```