Principes de Programmation
Partiel 2017

18 juin 2018

Le partiel dure 2 heures. Aucun document n'est autorisé.

Les exercices sont indépendants. Il est autorisé de sauter des questions, mais il faut **indiquer à quelle question vous répondez** et **utiliser une nouvelle copie à chaque exercice**.

Certaines questions sont très difficiles, mais :

— à part les questions longues marquées (†), aucune ne nécessite plus d'une demie page,

— répondre partiellement à une question difficile peut être suffisant,

— il y a 50 points accessibles, la note final sera capée à 20, mais pas renormalisée.

Exercice 1 (Egalités et monades, 15pt)**.**

1. Rappelez les axiomes des monades.

   réponse.

   ```
   return =<< u      ~= x
   f =<< (return x)  ~= f x
   f =<< (g =<< u)   ~= (\z -> f =<< (g z)) =<< u
   ```

2. Montrer que le type suivant est bien une monade : [1]

   ```
   type Reader r a = r -> a
   instance Monad (Reader r) where
     return x = \_ -> x
     f =<< u  = \r -> f (u r) r
   ```

   réponse.

   On vérifie les 3 axiomes de monade à l'aide des instances de `=<<` et `return` si dessus, et à l'aide des équivalences β (beta) et η (eta) :

   ```
   return =<< u     ~= \r -> return (u r)  r          -- def (=<<)
                    ~= \r -> (\_ -> (u r)) r          -- def return
                    ~= \r -> u r                      -- beta
   ```

1. à wraper près.

```
                    ~= u                                    -- eta

  f =<< (return x) ~= \r -> f (return x  r) r               -- def (=<<)
                   ~= \r -> f ((\_ -> x) r) r               -- def return
                   ~= \r -> f x r                           -- beta
                   ~= f x                                   -- eta

  f =<< (g =<< u)  ~= \r -> f ((g =<< u) r) r               -- def (=<<)
                   ~= \r -> f ((\s -> g (u s) s) r) r       -- def (=<<)
                   ~= \r -> f (g (u r) r) r                 -- beta
                   ~= \r -> (\t -> f ((g (u r)) t) t) r     -- beta
                   ~= \r -> f =<< (g (u r)) r               -- def (=<<)
                   ~= \r -> (\z -> f =<< (g z)) (u r) r     -- beta
                   ~= (\z -> f =<< (g z)) =<< u             -- def (=<<)
```

3. (†) Montrer que le datatype suivant est bien un foncteur : [2]

   ```
   data PTree a = Leaf a | Node (Char -> PTree)
   ```

   réponse.

   On commence par définir l'instanciation :

   ```
   instance Functor PTree where
     fmap f (Leaf x)  = Leaf (f x)
     fmap f (Node tr) = Node (\c -> fmap f (tr c))
   ```

   Puis on vérifie les deux axiomes de foncteurs :

   ```
   fmap id u          ~=  u
   fmap f (fmap g u)  ~=  fmap (f.g) u
   ```

   Pour les 2 axiomes, on raisonne par récurence sur la taille de l'abre. Le cas de base est pour `u = (Leaf x)` puis on étudie le cas `u = (Node tr)` en supposant les équations pour `u = (tr c)` pour n'importe quel charactère `c`.

   ```
   fmap id (Leaf x)          ~= Leaf (id x)                        -- def fmap
                             ~= Leaf x                             -- def id
   fmap id (Node tr)         ~= Node (\c -> fmap id (tr c))        -- def fmap
                             ~= Node (\c -> fmap (tr c))           -- hyp. recurtion

   fmap f (fmap g (Leaf x))  ~= fmap f (Leaf (g x))                -- def fmap
                             ~= Leaf (f (g x))                     -- def fmap
                             ~= Leaf ((f.g) x))                    -- def (.)
                             ~= fmap (f.g) (Leaf x)                -- def (.)
   fmap f (fmap g (Node tr)) ~= fmap f (Node (\c -> fmap g (tr c)))  -- def fmap
   ```

2. Dans un TP, nous avons vu le même datatype avec en plus le constructeur vide, la notion de PTree est floue, et les deux peuvent être utilisés, aisin que beaucoup d'autres (par exemple, on utilisera souvent {0,1} au lieu des charactères).

```
                    ~= Node (k -> fmap f
                                    ((\c -> fmap g (tr c)) k) -- def fmap
                    ~= Node (k -> fmap f (fmap g (tr k))    -- beta
                    ~= Node (k -> fmap (f.g) (tr k))        -- hyp. recurtion
                    ~= fmap (f.g) (Node tr)                 -- def fmap
```

4. (†) Montrer qu'il s'agit même d'une monade.

   réponse.

   On commence par définir l'instanciation :

   ```
   instance Functor PTree where
     return x        = Leaf x
     f =<< (Leaf x)  = f x
     f =<< (Node tr) = Node (\c -> f =<< (tr c))
   ```

   Puis on vérifie les trois axiomes de monades :
   Pour le premier et le troisième axiome, on raisonne par récurence sur la taille de l'abre. Le cas de base est pour `u = (Leaf x)` puis on étudie le cas `u = (Node tr)` en supposant les équations pour `u = (tr c)` pour n'importe quel charactère `c`. (Le second axiome est directe, pas besoin de récurence.)

   ```
   return =<< (Leaf x)     ~= return x                          -- def (=<<)
                           ~= Leaf x                            -- def return
   return =<< (Node tr)    ~= Node (\c -> fmap return (tr c))   -- def (=<<)
                           ~= Node (\c -> tr c)                 -- hyp. recurtion
                           ~= Node tr                           -- eta

   f =<< (return x)        ~= f =<< (Leaf x)                    -- def return
                           ~= f x                               -- def (=<<)

   f =<< (g =<< (Leaf x))  ~= f =<< (g x)                          -- def (=<<)
                           ~= (\z -> f =<< (g z)) x                -- beta
                           ~= (\z -> f =<< (g z)) =<< (Leaf x)     -- def (=<<)


   f =<< (g =<< (Node tr)) ~= f =<< (Node (\c -> g =<< (tr c)))    -- def (=<<)
                           ~= Node (\k -> f =<<
                                          ((\c -> g =<< (tr c)) k)) -- def (=<<)
                           ~= Node (\k -> f =<< (g =<< (tr k)))    -- def (=<<)
                           ~= Node (\k -> (\z -> f =<< (g z))
                                          =<< (tr k) )             -- hyp. recurtion
                           ~= (\z -> f =<< (g z)) =<< (Node tr)    -- def (=<<)
   ```

La typeclass des monades gradées n'existe pas en Haskell, mais pourrait être ajoutée. Elle est définie comme suit :

```
class GradMonad mon where
  returnG :: a -> mon () a
```

```
+<<     :: (Ord cle) => (a -> mon cle1 b) -> mon cle2 a -> mon (cle1,cle2) b
proj1   :: (Ord cle) => mon (cle,()) b -> mon cle b
proj2   :: (Ord cle) => mon ((),cle) b -> mon cle b
assos   :: (Ord cle) => mon ((cle1,cle2),cle3) b -> mon (cle1,(cle2,cle2)) b
```

comme une monade, il doit vérifier les axiomes suivants :

```
proj1 (returnG +<< u    )  ~=   u
proj2 (f +<< (returnG x))  ~=   f x
assos (f +<< (g +<< u)  )  ~=   (\y -> f +<< (g y) ) +<< u
```

5. (†) Montrez que `Dictionaire` est une monade gradée où :

   ```
   type Dictionaire cle cont = [(cle,cont)]
   ```

   est tel que une clé est présente au plus une fois et elles sont rangés en ordre croissant.

   réponse.

   Cette question était à la base ouverte : avoir une idée sufisait à avoir les points. Du fait des nombreuses typos dans l'énonce, cet exercice est même devenu bonus.

   On commence par définir l'instanciation, attention, on utilise ici le return, le bind et le fmap des listes :

   ```
   -- we use the notation
   bnd     :: (a -> Dictionaire cle1 b) -> (cle2,a) -> Dictionaire (cle2,cle1) b
   bnd f (c,x)  = map (coupler c) (f x)
   coupler :: cle1 -> (cle2,a) -> ((cle1,cle2),a)
   coupler c (k,y) = ((c,k),y)
   pj1 :: ((cle,())),a) -> (cle,a)
   pj1 ((c,())),x) = (c,x)
   pj2 :: (((),cle)),a) -> (cle,a)
   pj2 (((),c)),x) = (c,x)
   coupler :: (((cle1,cle2),cle3),a) -> ((cle1,(cle2,cle3)),a)
   ass (((c1,c2)c3),x)) = ((c1,(c2,c3)),x)

   instance GradMonad Dictionaire where
     returnG x  = [(),x]
     f +<< l    = (bnd f) =<< l
     proj1 l    = map pj1 l
     proj2 l    = map pj2 l
     assos l    = map ass l
   ```

   Puis on vérifie les trois axiomes de monades gradées :

   ```
     f +<< []        = []
     f +<< ((c,x):l) = (map (\(k,y)->((k,c),y)) (f x)) ++ (f +<< l)

     proj1 (returnG +<< l)      ~= proj1 ((bnd returnG) =<< l)
   ```

```
                                ~= map pj1 ((bnd returnG) =<< l)
                                ~= (\(c,x)-> map pj1 (bnd returnG (c,x)) ) =<< l
                                ~= (\(c,x)-> map pj1 (map (coupler c) (returnG x)) )
                                   =<< l
                                ~= (\(c,x)-> map pj1 (map (coupler c) (return ((),x))) )
                                   =<< l
                                ~= (\(c,x)-> map pj1 (return (coupler c ((),x)))) =<< l
                                ~= (\(c,x)-> map pj1 (return ((c,()),x))) =<< l
                                ~= (\(c,x)-> return (pj1 ((c,()),x))) =<< l
                                ~= (\(c,x)-> return (c,x)) =<< l
                                ~= return =<< l
                                ~= l

proj2 (f +<< (returnG x))      ~= proj2 ((bnd f) =<< (returnG x))
                                ~= proj2 ((bnd f) =<< (return ((),x)))
                                ~= proj2 (bnd f ((),x))
                                ~= map pj2 (bnd f ((),x))
                                ~= map pj2 (map (coupler ()) (f x))
                                ~= map (pj2.(coupler ())) (f x)
                                ~= map id (f x)
                                ~= f x

assos (f +<< (g +<< l))
   ~= assos (f +<< ((bnd g) =<< l))
   ~= assos ((bnd f) =<< ((bnd g) =<< l))
   ~= assos ((bnd f) =<< ((bnd g) =<< l))
   ~= assos ((\ (c,x) -> (bnd f) =<< (bnd g (c,x)) ) =<< l)
   ~= map ass ((\(c,x) -> (bnd f) =<< (bnd g (c,x)) ) =<< l)
   ~= (\(k,y) -> map ass ((\(c,x) -> (bnd f) =<< (bnd g (c,x))) (k,y)) )
       =<< l
   ~= (\(k,y) -> map ass ((bnd f) =<< (bnd g (k,y))) ) =<< l
   ~= (\(k,y) -> map ass ((bnd f) =<< (map (coupler k) (g y))) ) =<< l
   ~= (\(k,y) -> map ass (((bnd f).(coupler k)) =<< (g y)) ) =<< l
   ~= (\(k,y) -> (\(c,x) -> map ass (((bnd f).(coupler k)) (c,x)))
                 =<< (g y) ) =<< l
   ~= (\(k,y) -> (\(c,x) -> map ass (bnd f (coupler k (c,x))))
                 =<< (g y) ) =<< l
   ~= (\(k,y) -> (\(c,x) -> map ass (bnd f ((k,c),x)))
                 =<< (g y) ) =<< l
   ~= (\(k,y) -> (\(c,x) -> map ass (map (coupler (k,c)) (f x)) )
                 =<< (g y) ) =<< l
   ~= (\(k,y) -> (\(c,x) -> map (ass.(coupler (k,c))) (f x))
                 =<< (g y) ) =<< l
   ~= (\(k,y) -> (\(c,x) -> map ((coupler k).(coupler c)) (f x))
                 =<< (g y) ) =<< l
   ~= (\(k,y) -> (\(c,x) -> map (coupler k) (map (coupler c) (f x)))
```

```
                =<< (g y) ) =<< l
~= (\(k,y) -> (\(c,x) -> map (coupler k) (bnd f (c,x)))
                =<< (g y) ) =<< l
~= (\(k,y) -> map (coupler k) ((bnd f) =<< (g y)) ) =<< l
~= (\(k,y) -> map (coupler k) ((\z -> (bnd f) =<< (g z)) ) y) =<< l
~= (bnd (\z -> (bnd f) =<< (g z) )) =<< l
~= (bnd (\z -> f +<< (g z) )) =<< l
~= (\z -> f +<< (g z) ) +<< l
```

Exercice 2 (Programmation en situation, 15pt)**.**
Correction dans un fichier séparé compilable disponible ici.

Exercice 3 (Programmation découverte, 15pt)**.**
Avec une extension de syntaxe, on peut redéfinir les arbre rouge-noir ainsi :

```
data Z   = Z
data S a = S
data R   = R
data N   = N

data ARN c n a where
  Feuille :: ARN N Z a
  NoeudN  :: ARN c1 n a -> a -> ARN c2 n a -> ARN N (S n) a
  NoeudR  :: ARN N  n a -> a -> ARN N  n a -> ARN R n a
```

1. Décrivez ce datatype. En quoi il est différent des datatypes habituels ? En quoi représente-t-il mieux les ARN que ceux utilisés en projet ?

   réponse.

   Les types `Z, S, R` et `N` sont des types phantome : on leur donne un constructeur par principe, mais ils servent surtout à quantifier le type `ARN`.

   Le datatype `ARN` est paramétrique, il dispose de 3 paramètres :
   — `c`, qui, en pratique sera le type `R` ou le type `N`, il dénote la couleur de la racine,
   — `n`, qui, en pratique sera un type de la forme `S(S(...(S Z)..)`, il dénote (en unaire) la hauteur de noirs,
   — `a`, qui est le type du contenu de l'arbre.

   Les types `c` et `n` sont phantome : on ne les considère jamais seuls.

   Cette syntaxe pour le type algébrique `ARN` est la syntaxe des types algébriques généralisés "GADT". Syntaxiquement, elle permet d'avoir des constructeurs dont les paramètres en entré et en sortie sont plus spécifique. Ainsi, `NoeudN  :: ARN c1 n a -> a -> ARN c2 n a -> ARN N (S n) a`, se lit : `NoeudN` est un constructeur, si on lui donne des paramètres de

types `g::ARN c1 n a` `x::a` et `d::ARN c2 n a` pour n'importe quels types `c1, c2, n` et `a`, celà forme un terme `Feuille g x d :: ARN N (S n) a`.

Le datatype `ARN` a trois constructeurs, que l'on indique avec une syntaxe inhabituelle (on y revien après) :

— `Feuille` n'ai pas d'argument, c'est une constante pour un `ARN` noir, de hauteur noire 0, et de contenant quelqonque (puisque il est vide),

— `NoeudN` demande 3 arguments, son fils gauche $g :: ARNc1na$, son contenu `x :: a`, sont fils droit $d :: ARNc2na$; on crée un arbre `NoeudN g x d :: ARN N (S n) a` de racine noir, et on n'a aucune restriction sur les couleurs des fils (`c1` et `c2` quelconque) ; la hauteur noire des deux fils doit par contre être la même `n`, et la hauteur noir de l'arbre ainsi créé est alors `S n` (càd n+1) ; enfin, les fils ont le même type de contenu `a` que l'arbre créé,

— `NoeudR` demande 3 arguments, son fils gauche $g :: ARNNna$, son contenu `x :: a`, sont fils droit $d :: ARNNna$; on crée un arbre `NoeudR g x d :: ARN N n a` de racine rouge, dont les deux fils ont une racine noire ; la hauteur noire des deux fils doit être la même `n`, et la hauteur noir de l'arbre ainsi créé `n` n'est cette fois-ci pas implémentée ; enfin, les fils ont le même type de contenu `a` que l'arbre créé.

Ce type permet de forcer les deux invariants des ARNs à être respecter :

— il ne peux pas y avoir de noeud rouge avec un fils rouge,

— la hauteur noir est la même pour tout chemain entre la racine et une feuille.

Ce type garanti donc que les opérations implémentées (insertions, supression...) sont correctes. De façon général, les GADTs permettent de garentir ce genre d'invariants pour l'utilisateur, mais aussi pour optimiser le code (on fait moins de vérifications runtime si on connait les invariants). Celà permet aussi d'écrire des types existanciel, ainsi le type suivant (utilisé par la suite) désigne un ARN qui est correcte mais dont la couleur de la racine et la hauteur nors sont "oubliés", ou plus exactement abstrait, mais elles existent :

```
data ARNAbst a where Abstraire :: ARN c n a -> ARNAbst a
```

2. Pendant l'insertion dans un arbre rouge noir, il y a des moment où un invariant n'est pas respecté. Lequel ? Écrire un datatype dans le même style décrivant cet situation intermédiaire.

   réponse.

   Pendant l'insertion, il peut y avoir (au plus) un noeud rouge dont un fils est rouge (l'autre est bien noir).

   Il faut donc un datatype pouvant caractériser ce type intermédiaire, par exemple :

```
data ARNinter c n a where
  Correcte :: ARN c n a -> ARNinter c n a
  NoeudNG  :: ARNinter c1 n a -> a -> ARN c2 n a -> ARNinter N (S n) a
```

```
  NoeudND  :: ARN c1 n a -> a -> ARNinter c2 n a -> ARN Ninter (S n) a
  NoeudRG  :: ARNinter N  n a -> a -> ARN N  n a -> ARNinter R n a
  NoeudRD  :: ARN N  n a -> a -> ARNinter N  n a -> ARNinter R n a
  DoubleRG :: ARN R n a -> a -> ARN N  n a -> ARNinter R n a
  DoubleRD :: ARN N  n a -> a -> ARN R  n a -> ARNinter R n a
```

Celà veut dire que un tel arbre est soit un ARN correcte (il n'y a pas de doublon rouge), soit un noeud noir dont le fils gauche contien un doublon, soit un noeud noir dont le fils droit contien un doublon, soit un noeud rouge dont le fils gauche contien un doublon, soit un noeud rouge dont le fils droit contien un doublon, soit un noeud rouge dont le fils gauche est rouge, soit un noeud rouge dont le fils droit est rouge...

Ce genre de datatype à trop de cas, ce serait très long à manipuler, mais on peut le rendre plus simple à l'aide d'une observation : Dans l'insertion, on ne considère en fait que le cas où le doublon est à la racine du sous-arbre traité, les constructeurs `NoeudNG`, `NoeudND`, `NoeudRG` et `NoeudRD` ne sont donc pas utiles. De plus, la couleur de noeud à la racine n'est alors pas important. Ainsi il nous reste :

```
data ARNinter n a where
  Simple  :: ARN c n a -> ARNinter n a
  DoubleG :: ARN R n a -> a -> ARN N  n a -> ARNinter n a
  DoubleD :: ARN N  n a -> a -> ARN R  n a -> ARNinter n a
```

Suite de la correction dans un fichier séparé compilable disponible ici.