PAF TD 9 : Composition contextuelle et transformers

Dans ce TD nous étudions la composition de contextes fonctoriels, applicatifs et monadiques, ainsi que les *(monad) transformers*.

Exercice 1 : le transformer IdentityT

Le *transformer* le plus simple est celui qui emboîte une monade quelconque dans l'identité.

On donne ci-dessous le code de la monade `Identity` :

```
newtype Identity a = Identity { runIdentity :: a }
  deriving (Show, Eq)

instance Functor Identity where
  fmap g (Identity x) = Identity (g x)

instance Applicative Identity where
  pure = Identity
  (Identity g) <*> (Identity x) = Identity (g x)

instance Monad Identity where
  (Identity x) >>= f = f x
```

Question 1.1.

Définir la représentation du *transformer* `IdentityT` basé sur `Identity`.

On utilisera le nom d'accesseur `runIdentityT`.

Question 1.2.

Définir les instances de `Functor`, `Applicative` et `Monad` pour `(IdentityT m)` en s'inspirant des instances de `Identity`.

Question 1.3.

Soit le type suivante :

```
type MyIdentity a = IdentityT Identity a
```

Décrire en Haskell l'isomorphisme entre `MyIdentity a` et `Identity a` (pour tout type `a`).

Exercice 2 : lifting automatique

Dans cet exercice nous nous basons sur le type suivant :

```
type Deep a = IdentityT (IdentityT (IdentityT IO)) a
```

Question 2.1.

Donner une valeur possible du type `Deep Integer`, et qui affiche `"Hello"` (si on l'exécute).

Question 2.2.

Soit le programme suivant (dans `IO ()`) :

```
progIO :: IO Integer
progIO = do
  putStrLn "Hello"
  str <- getLine
  return $ read str
```

Transformez ce programme pour pour qu'il soit de la signature suivante :

```
progIODeep :: Deep Integer
progIODeep = do
   ???
```

Question 2.3.

Pour pouvoir "creuser" un peu plus simplement dans les niveaux de monade, on définit des instances des *transformers* pour la *typeclasse* suivante :

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

Proposer une instance de `MonadTrans` pour `IdentityT`. En déduire une nouvelle version du programme qui utilise `lift`.

Question 2.4.

Les *transformers* qui réalisent des effets de bord utilisent toujours `IO` et bien sûr.

Il n'existe pas de *transformer* `IOT`, pouvez-vous expliquer pourquoi ?

Comme `IO` est donc le plus souvent le contexte le plus englobant (lorsque l'on en a besoin), on peut exploiter ce fait pour simplifier le *lifting* spécifiquement pour `IO` avec la *typeclasse* suivante :

```haskell
class Monad m => MonadIO m where
  liftIO :: IO a -> m a
```

Donner des instances de cette *typeclasse* pour `IO` (qui est bien une monade) et `IdentityT m` (et un `m` satisfaisant `MonadIO`).

En déduire une dernière version du programme `progIODeep`.

Exerecice 3 : Composition contextuelle

Nous étudions dans cet exercice un opérateur générique de composition de contexte, défini ci-dessous :

```haskell
newtype Compose t m a = Compose { getCompose :: t (m a) }
  deriving (Show, Eq)
```

Remarque : on a utilisé les mêmes noms de contexte que pour les transformers car bien sûr l'opérateur `Compose` peut être vu comme un opérateur de construction "automatique" de *transformers*.

Question 3.1.

Les foncteurs (simples) sont composables, ce qui signifie qu'il est possible de définir l'instance suivante :

```haskell
instance (Functor t, Functor m) => Functor (Compose t m) where
  fmap = fmapCompose
```

Par exemple :

```
>>> :t Compose $ Just (Identity (42 :: Integer))
...  :: Compose Maybe Identity Integer

>>> Compose $ Just (Identity (42 :: Integer))
Compose {getCompose = Just (Identity {runIdentity = 42})}

>>> fmap (+1) $ Compose $ Just (Identity (42 :: Integer))
Compose {getCompose = Just (Identity {runIdentity = 43})}
```

Proposer une implémentation de :

```haskell
fmapCompose :: (Functor t, Functor m) => (a -> b) -> Compose t m a -> Compose t m b
```

Question 3.2. (difficile)

Les foncteurs applicatifs sont également composables, ce qui signifie qu'il est possible de définir l'instance suivante :

```haskell
instance (Applicative t, Applicative m) => Applicative (Compose t m) where
  pure = pureCompose
  (<*>) = applyCompose
```

Par exemple :

```
>>> :t Compose $ Just (Identity (42 :: Integer))
...  :: Compose Maybe Identity Integer

>>> Compose $ Just (Identity (42 :: Integer))
Compose {getCompose = Just (Identity {runIdentity = 42})}

>>> fmap (+1) $ Compose $ Just (Identity (42 :: Integer))
Compose {getCompose = Just (Identity {runIdentity = 43})}
```

Proposer une implémentation de :

```haskell
pureCompose :: (Applicative t, Applicative m) => a -> Compose t m a

applyCompose :: (Applicative t, Applicative m) =>
                Compose t m (a -> b) -> Compose t m a -> Compose t m b
```

Question 3.3 (difficile)

Malheureusement, deux monades composées avec `Compose` ne forment pas systématiquement une monade. Ce qui veut dire qu'il n'est pas possible d'implémenter, en toute généralité, le *bind* suivant :

```haskell
bindCompose :: (Monad t, Monad m)
                => Compose t m a -> (a -> (Compose t m b)) -> (Compose t m b)
```

Voici une définition "presque" complète :

```haskell
bindCompose (Compose x) f =
  let f' v = getCompose (f v)
  in Compose $
     x >>= (\y -> let z = swap (fmap f' y)
                  in fmap join z)
```

La fonction `join` existe dans `Control.Monad` et possède le type suivant :

```haskell
join :: Monad m => m (m a) -> m a
```

Pouvez-vous redérinir cette fonction ?

Pouvez-vous en déduire le type de la fonction `swap` ?

Cette fonction doit être générique, complétez ainsi la définition de la *typeclasse* suivante :

```haskell
class Swap t m where
  swap :: ????  ->  ????
```

Proposer une instance de `Swap Identity Maybe`.

Dans un rapport scientifique *"Composing Monads"* (par Mark P. Jones et Luc Duponcheel) il est montré que deux monades `t` et `m` sont composables si on peut instancier `Swap m t` en respectant des lois décrites dans l'article (cf. http://web.cecs.pdx.edu/~mpj/pubs/RR-1004.pdf)

Proposer donc une instance implémentable de `Monad` pour `(Compose t m)`