Principes de Programmation
TP7: Monades et concurence

10 mai 2019

Exercice 1 (La monade IO). La monade IO permet de sortir de toutes les restrictions normalement imposé par la pureté de Haskell et de faire tous les effets de bords que l'on veut. Par contre, une fois dans la monade IO, on ne peux pas en sortir, au sens où un programme de type `Int->Int` ne peut pas utiliser les effets de bord de IO, même s'il fait des trucs très compliqué.

En particulier, la monade IO permet de faire des *threads* et des références, ce qui, on s'en doute, permet de recréer les problèmes usuels de concurrence. On va donc voir ce qu'elle permet

1. Importez les bibliothèques `Data.IORef` et `Control.Concurrent`.
2. La première permet de créer des références mutables (càd. des pointeurs) de type `IORef` grâce aux opérateurs : [1]

   ```
   newIORef :: a -> IO (IORef a)
   readIORef :: IORef a -> IO a
   writeIORef :: IORef a -> a -> IO ()
   modifyIORef' :: IORef a -> (a->a) -> IO ()
   ```

   À l'aide de ces opérateurs, écrire la fonction :

   ```
   decrease :: IORef Int -> Int -> IO ()
   ```

   Tel que `decrease r 3` va élever 3 à `r` si celui-ci est plus grand au égal à `3` (on n'a pas le droit de passer dans les négatif.
3. Testez votre fonction dans un main.
4. La seconde bibliothèque permet de lancer des threads en parallèle grâce aux fonctions :

   ```
   forkIO :: IO () -> IO ThreadId
   threadDelay :: Int -> IO ()
   ```

   le premier lance un thread et renvoi l'identité du thread, le second est un sleep (en millisecondes).
   Écrivez un main qui met 10 dans une référence puis la décrémente de

1. le ' dans `modifyIORef'` signifie que la fonction est évaluée strictement, c'est à dire que l'on force le programme à ne pas être paresseux.

deux grâce à `decrease`, mais ce plusieurs fois (6-7 fois) en parallèle, puis attendez le résultat à l'aide d'un `threadDelay 1000000`, affichez le résultat et bouclez.

5. Testez ce résultat en compilant et en lançant cette boucle infinie. Que se passe-t-il ?
6. Pour pallier à ce soucis, il nous faut des opérations atomiques, qui lock la variable entre lecture et écriture. Pour ça on utilise la fonction :

   ```
   atomicModifyIORef' :: IORef a -> (a -> (a,b)) -> IO b
   ```

   Qui, en plus d'être atomique peut rendre un résultat de type `b`. Utilisez là pour faire une version correcte du programme précédent.
7. Essayez d'écrire une fonction :

   ```
   transfert :: IORef Int -> IORef Int -> Int -> IO ()
   ```

   qui transfert un montant de la première référence à la seconde à condition que la premiere soit positive.
8. Essayez d'écrire une fonction :

   ```
   printIfEqual :: IORef Int -> IORef Int -> IO ()
   ```

   qui décrémente les deux montant à condition que tous deux soient égaux.
9. Faites un petit exemple avec deux références à 10 et à 0 et trois threads qui tournent en permanence : `printIfEqual ref1 ref2`, `transfert ref1 ref2 10` et `transfert ref2 ref1 10`. Que remarquez-vous ?

Indice : ce n'est pas possible de faire la dernière question en restant atomique car la fonction `atomicModifyIORef'` n'est valable que sur une seule référence. On pourrait mettre tous les "comptes" dans une seule référence, mais on serait alors obligé de faire une action à la fois, ce qui n'est pas pratique...

Exercice 2 (La monade STM). On utilisera une autre monade que IO pour faire de la concurrence avec partage de mémoire. Celle-ci s'appelle `STM` pour "Software Transactional Memory".

1. Importez la bibliothèque `Control.Concurrent.STM`, celle-ci utilise le type `TVar` avec les fonctions

   ```
   newTVar   :: a -> STM (TVar a)
   readTVar  :: TVar a -> STM a
   writeTVar :: TVar a -> a -> STM ()
   ```

   La monade STM est une version restreinte de la monade IO, elle correspond à une "opération atomique" au sens où une opération retournant le type `STM a` a la garantie d'être exécutée de manière atomique, sans conflit avec un autre thread. Pour pouvoir ensuite l'exécuter, on utilise la fonction :

   ```
   atomically :: STM a -> IO a
   ```

qui exécute l'opération atomiquement. Ainsi, à chaque invocation de atomique en parallèle, on peut exécuter une opération différente. Écrivez une version correcte de :

```
type Compte = TVar Int
transfert :: Compte -> Compte -> Int -> STM ()
```

2. Testez cette fonction dans un main qui va faire plusieurs opérations atomiques en parallèle.

Le fonctionnement de cette monade est subtile : lorsque l'on est dedans, on ne travail pas sur la référence directement, mais sur une copie, que l'on peut manipuler à volonté, lorsque le calcul est fini, la fonction `atomic` vérifie que les `TVar` utilisées n'ont pas été modifiées par un autre thread pendant le calcule et soumet les changements (les deux dernières étapes posent un lock en écriture), s'il y a eu un changement, le calcul est jeté et tout le processus atomique est relancé (proprement, sans monopoliser le processeur). Ainsi, le lock n'est posé que sur la vérification que l'on peut faire l'écriture à la fin, ce qui est très efficace.

3. On peut même utiliser ce processus de backtrack pour nos propre programme grâce à la la commande d'erreur :

   ```
   retry :: STM a
   ```

   qui interrompt l'opération atomique en cours et la réinitialise. Utilisez là pour améliorer la fonction transfert de sorte à réessayer indéfiniment le transfert jusqu'à ce qu'on puisse le faire (càd. jusqu'à ce qu'il y ai suffisamment d'argent sur le compte débité).

4. On peut même utiliser des `TVar` particulières, appelées `TMVar` qui servent ce canal de communication : Elles sont soit vide soit contenant une valeur (comme le `Maybe`), on peut les lire et écrire dessus avec les commandes :

   ```
   takeTMVar :: TMVar a -> STM a
   putTMVar  :: TMVar a -> a -> STM ()
   ```

   Mais si on lit (ou on prend) une valeur qui n'existe pas, on recommence l'action atomique, et si on écrit dans une `TVar` non vide, pareil. Écrivez une fonction

   ```
   type Transfert = (Compte, Compte, Int)
   applyTransfert :: TMVar Transfert -> STM ()
   ```

   qui prend un canal dans lesquels il récupère des données pour un transfert et fait le transfert associé.

5. Mieux, encore, on peut utiliser un *buffer* sur le canal et l'utiliser de la même manière avec les fonctions :

   ```
   readTChan  :: TChan a -> STM a
   writeTChan :: TChan a -> a -> STM ()
   ```

   Améliorez `applyTransfert` pour qu'on puisse lui demander plusieurs transferts qu'il mettra en buffer.

6. On veut maintenant faire tourner `applyTransfert` en boucle dans deux *treads* différents. Faites un main qui permet ça et qui demande en même temps l'application de quelques transferts (via d'autres threads).
7. Demandez un transfert impossible (de 100000 sur un compte avec 100) au milieux de tous vos transferts. Que remarquez-vous ?
8. Pour éviter ça, on voudrait remettre le transfert en question en queue, pour ça, on utilise :

   ```
   orElse :: STM a -> STM a -> STM a
   check  :: Bool -> STM ()
   ```

   `orElse` essaie de faire le premier calcul et s'il n'y arrive pas, backtrack et fait le second ; tandit que `check` fait échouer le calcule en cours si le booléen donné est faux. Utilisez-les pour donner aune autre version de `transfert` et `applyTransfert` qui remet les informations de transfert dans le buffer pour réessayer plus tard lorsqu'il y a un soucis (fonds insufisant ou compte indisponible).

Exercice 3 (La monade de continuation).

1. Importez la librairie `Control.Monad.Cont`.
2. Créez la structure de donnée suivante :

```
data Couleur    = Rouge | Verte | Bleue | Orange | Violette
type Clef       = Couleur
type Porte      = Couleur
data Labyrinthe =   Tresor  Clef
                  | Branche Labyrinthe Labyrinthe
                  | Porte   Porte      Labyrinthe
                  | Sortie | Impasse
```

Il s'agit d'un arbre d'où l'on voudrait sortir. Chaque clef peut être utilisée qu'une seule fois pour ouvrir une porte de même couleur. Pour faciliter les choses, il est possible de se téléporer vers tout endroit déjà visités. On va utiliser le `Callcc` de la monade `Cont` pour faire ces téléportations.

3. Dérivez tout ce que vous pouvez sur cette structure.

Dans un premier temps, on appliquera un algorithme glouton :

— Lorsque l'on voit une porte, si l'on connaît une clé de la bonne couleur, on détruit la clé et on ouvre la porte.
— Lorsque l'on voit une porte, et que l'on ne connaît pas de clé de la bonne couleur, on retient l'emplacement et on revient au dernier embranchement,
— Lorsque l'on voit un embranchement, on prend à gauche et on retient l'embranchement.
— Lorsque l'on tombe sur une impasse ou une porte sans clé et que l'on a plus d'embranchement en amont, on échoue,
— Lorsque l'on voit une clé, si l'on connaît une porte de la bonne couleur, on se téléporte à la dernière pour l'ouvrir, sinon on continue.
— Lorsque l'on voit une clé, et que l'on ne connaît pas de porte de la bonne couleur, on la récupère et on revient au dernier embranchement.

En pratique :

4. Créez une structure de donnée `Inventaire` avec trois sections : les clés, les portes et les embranchements non empruntés.
   — Pour les clés, il s'agit de stoker toutes les clés trouvés.
   — Pour les portes, il s'agit de stoker des couples couleur-téléport.
   — Pour les embranchement, il s'agit de stoker des téléports.
5. Remarquez qu'il n'y a pas de structure pour les emplacements où on poura se téléporter (portes et embranchement). Il faut donc créer ce type, on choisira le type

   ```
   type Teleport = Inventaire -> Cont Bool a
   ```

   L'idée est que l'on peut se téléporter à n'importe quel moment, quel que soit le calcule que l'on est en train de faire, et donc quelque soit le type

`a`, ce qui est important c'est que l'on donne l'inventaire en même temps. Il s'agit en fait du type [2] du break de `callCC`, que l'on vera juste après.

6. Écrivez des fonctions d'ajout de clefs, portes et branchements dans l'inventaire.

```
ajouterPorte   :: (Porte,Emplacement) -> Inventaire -> Inventaire
ajouterCle     :: Cle -> Inventaire -> Inventaire
ajouterBranche :: Emplacement -> Inventaire -> Inventaire
```

7. Écrivez une fonction

```
rechercheBranche :: Inventaire -> Maybe (Teleport, Inventaire)
```

qui rend `Nothing` s'il n'y a plus de branchement ou fait un pop sur la liste.

8. Écrivez une fonction

```
recherchePorte :: Porte -> Inventaire -> Maybe (Teleport, Inventaire)
```

qui recherche une porte dans un inventaire, et retourne le téléporteur correspondant et l'environnement mis à jours.

9. Écrivez une fonction

```
rechercheClef :: Clef -> Inventaire -> Maybe (Inventaire)
```

10. Créez deux continuations bien particulière pour sortir du labyrinthe :

```
sorti :: Emplacement
sorti = cont (\_ -> True)
perdu :: Emplacement
perdu = cont (\_ -> False)
```

Il s'agit de deux breaks, rendant vrais ou faux et ignorant tout autre calcule qui viendra par la suite. On peut les voire comme des exceptions ratrapées par `reuCont` (voire plus loin)

11. Écrivez une version simplifée de la fonction :

```
parcours :: Labyrinthe -> Inventaire -> Continuation Bool Inventaire
```

Qui échoue lorsqu'elle arrive sur une porte ou sur une clée, et qui pred toujours à gauche sur les branchements.

12. Testez la dans un main en utilisant la commande

```
runParcours :: Labyrinthe -> Inventaire -> Bool
runParcours l i = runCont (parcours l i) (\_ ->False)
```

Il s'agit d'éxécuter parcours, mais en sortant de la monade. Si un exception `sortie` ou `perdu` est lancée, elle renvoit le booléen correspondant, sinon elle renvoit `False`.

2. On utilisera des booléens pour le type final de retour, et les inventaires comme type appelé par les callCC.

13. Écrivez la fonction :

    ```
    impasse :: Inventaire -> Emplacement
    ```

    que l'on lance lorsque l'on tombe sur une impasse. Celle-ci récupère le dernier branchement et s'y téléporte, à condition de le trouver, s'il n'y en a pas, on est perdu...

14. Écrivez la fonction :

    ```
    branchement1 :: Inventaire -> Labyrinthe ->  Labyrinthe -> Cont Bool Inventaire
    branchement1 i1 g c d =
        i2 <- parcours g (addBranche retour i1)
        parcours d i2
    ```

    Que fait cette fonction ?

15. Écrivez la fonction :

    ```
    branchement2 :: Inventaire -> Labyrinthe ->  Labyrinthe -> Cont Bool Inventaire
    branchement2 i1 g c d =
        parcours g (addBranche retour i1)
        where
        retour i2 = parcours d i2
    ```

    Que fait cette fonction ?

16. Avec callCC, on peut réécrire cette fonction ainsi :

    ```
    branchement :: Inventaire -> Labyrinthe ->  Labyrinthe -> Cont Bool Inventaire
    branchement i1 g c d = do
        i2 <- callCC $ \retour ->
                       parcours g (addBranche retour i1)
        parcours d i2
    ```

    Cette fonction est un mix des deux précédentes : on peut exécuter `parcours d i2` ou bien après un `return i2` dans l'execution de `parcours g (addBranche retour i1)`, ou bien après un `retour i2` où `retour` a été trouvé dans l'inventaire.

17. Écrivez l'algorithme avec backtrack décrit au dessus. L'idée est de rentrer dans un callCC chaque fois que l'on doit retenir un emplacement.

18. Testez sur plusieurs labyrinthes.

19. Testez sur `Branche (Tresor Vert) (Branche (Porte Verte Impasse) (Porte Verte Sortie))`. Que se passe-t-il ?

20. (Difficile) Améliorer l'algo pour ne pas manquer de chemins.

21. (Difficile) Regardez la doc de `ContT` et utilisez le pour créer une monade qui combine `Cont` et `State`. Utilisez la pour réécrire l'exercice avec l'inventaire inscrit dans un état.

Exercice 4 (Une petite BdD relationelle). Nous allons créer une base de donnée relationnelle miniature, avec une seule table prédéfinie (on pourra la généraliser par la suite). Pour garantir l'atomicité et l'isolation des processus, on utilisera la monade STM.

De plus, on utilisera des tableaux de référence `TVar a` appelés `TArray Int a` et avec les opérations :

```
readArray  :: TArray Int a -> Int -> SMT a
writeArray :: TArray Int a -> Int -> a -> SMT ()
newArray   :: (Int, Int) -> a -> SMT (TArray Int a)
getAssocs  :: TArray Int a -> SMT [(Int,a)]
```

qui permetre de récupérer un élément d'un tableau, de déposer un élément dans le tableau, de créer un tableau et de fournir la liste d'association des éléments du tableau (à utiliser pour l'insertion).

1. Importez la bibliothèque `Control.Concurrent.STM` ainsi que `Data.Map.Strict` et `Data.Set.Strict`, et créez les types de lignes et colonnes :

   ```
   data Ligne = Ligne { id :: Int
                      , columns :: TArray Int Valeur
                      }
   data Valeur = Int Int | Float Float | String String
   ```

   Il faut le lire ainsi :
   — une ligne est composé d'un identifiant et d'un tableaux contenant des valeurs qui peuve être des entiers, des flotants ou des strings,
   — ces tableau est un tableau de `TVar`, ses cases peuvent donc être lues et modifiées indépendamment sans bloquer d'autres lectures/écritures,
   — une colonne est désignée par un entier, qui s'il est 0 représente l'identifiant sinon représente l'indexe plus un dans le tableau.

2. Écrivez la fonction

   ```
   getColumn    :: Int -> Ligne -> STM Valeur
   ```

   qui récupère la bonne valeur en fonction du nom de colonne donné (un entier).

3. Supposons que l'on ai restrein notre ensemble de lignes avec un `where`, essayons de rendre les valeurs d'une colonne choisie :

   ```
   selectColumn    :: Int -> [Ligne] -> STM [Valeur]
   ```

4. Créez les type de `ColumnMap` et `Table`

   ```
   type ColumnMap = Map Int [Ligne]
   data Table  = Table { ids :: TVar (Map Int Ligne)
                       , columnMaps :: TArray Int ColumnMap
                       }
   ```

   Il faut le lire ainsi :

— une `ColumnMap` est une table d'association associée à une colonne, elle associe à chaque (hash de) valeur la liste des lignes qui ont cette valeur dans cette colonne.
— une table est alors l'ensemble des lignes et le tableaux de `TVar` contenant chacune des tables d'associations de chaque table ; ils sont tous dans des `TVar`, car si une valeur est modifiée, la table d'association doit être changée.

5. Pour pouvoir rechercher une valeur, il faut utiliser son Hash, il va donc faloir instancier `Hashable Valeur`.
6. Si on veut combiner des requètes, on a besoin des opérateurs :

```
and :: [Ligne] -> [Ligne] -> STM [Ligne]
or  :: [Ligne] -> [Ligne] -> STM [Ligne]
```

   Qui récupère l'union et l'intersection de résultats.
7. Écrivez la fonction

```
whereEqual :: Table -> Int -> Valeur -> STM [Ligne]
```

   qui récupère une table, un nom de colonne, une valeur (`Left n` sur une valeur de type `Int` et `Right st` sur une valeur de type `String`), et rend toutes les lignes associées à cette valeur.
   Vous avez le droit d'utiliser la fonction

```
maybeToList :: Maybe a -> [a]
```

   du module Data.Maybe.
8. Écrivez, de préférence en style monadique, les fonctions

```
updateList :: (a -> Maybe b) -> [a] -> [b]
updateList2 :: (a -> Maybe [b]) -> [a] -> [b]
```

   qui font un map et un bind, mais en enlevant des éléments à la volée (ceux pour lesquels la fonction donnée renvoit Nothing).
9. Écrivez une fonction

```
whereIn :: Table -> Int -> [Valeur] -> STM [Ligne]
```

10. Pour l'update et le delete, il faut pouvoir récupérer une ligne où qu'elle soit. Pour ca, on va iddentifier les ligne par leur iddentifiant. Instanciez `Eq Ligne`.
11. On va aussi avoir besoin des fonctions :

```
retirer :: (Eq a) => a -> [a] -> Maybe [a]
ajouter :: a -> Maybe [a] -> Maybe [a]
```

    Qui ajoutent ou retirent un élément à une liste, mais en iddentifiant la liste vide à Nothing, comme suit :

```
retirer 2 (Just [4,2,7])  ------------->  Just [4,7]
retirer 2 (Just [2,2])    ------------->  Nothing
retirer 2 Nothing         ------------->  Nothing
ajouter 2 (Just [5,7])    ------------->  Just [2,5,7]
ajouter 2 Nothing         ------------->  [2]
```

12. En utilisant les fonctions suivantes du module Data.Map.Strict :

    ```
    alter  :: Ord cle => (Maybe a -> Maybe a) -> cle -> Map cle a -> Map cle a
    ```

    qui va metre à jour la valeur associée à la clé entrée avec la fonction donnée, et qui est capable de créer ou de supprimer la valeur si elle n'existait pas, écrivez

    ```
    update :: Table -> [Ligne] ->  Int -> Valeur -> STM ()
    ```

    Attention, l'`update` change aussi la table d'association correspondante.

13. Écrivez

    ```
    updateWith :: Table -> [Ligne] -> Int -> (Ligne -> STM Valeur) -> STM ()
    ```

14. Écrivez

    ```
    insert :: Table -> Ligne ->  STM ()
    ```

    Attention, il va falloir mettre à jours toutes les tables d'association !

15. Écrivez une requète corresponant à la requète
    `UPDATE SET final = note1+note2 WHERE annee = 2019 AND promo = "info"`
    où note, note1, note2, promo et annee sont les colonnes 1, 2, 3, 4 et 5

16. Écrivez une autre requètes et lancez-les dans des threads différents (il vous faudra aussi créer la bdd avant).