Présentation des States

Jean-Luc JOULIN

Version 1
18 septembre 2022

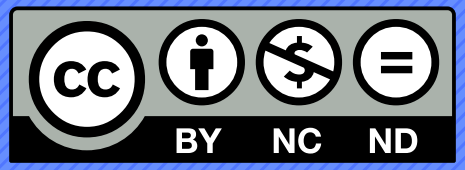
License

Cette présentation est diffusée suivant les termes de la license Creative Common :

BY **Attribution**. Cette présentation peut être librement utilisée, à condition de l'attribuer à l'auteur en citant son nom.

NC **Pas d'utilisation Commerciale**. Aucune utilisation commerciale n'est permise.

ND **Pas de Modification**. Aucune œuvre dérivée basée sur cette présentation n'est autorisée.



Changements d'états

Changements d'états

AVANTAGES :

- Nécessaires dans certains cas.
- Permet de conserver des valeurs entre différentes étapes de calcul.
- Plus d'efficacité pour certaines tâches.

INCONVÉNIENTS :

- Risque d'erreurs (changement d'états non anticipés).
- Opposé au fonctionnement normal de Haskell.

Solution

- Utilisation d'un module dédié à cet usage.
- Utilisation d'une monade spécifique.
- Cloisonnement entre fonctions pures et impures.
- Cloisonnement entre approche fonctionnelle et approche séquentielle par états.

Présentation générale

Modules States

- Modules indépendants.
- Plusieurs implémentations (très ressemblantes).
  - Module transformers (Hackage).
  - Module mtl (Hackage).
- Plusieurs variantes.
  - Évaluation stricte.
  - Évaluation paresseuse.

Modules States (Imports)

- Module transformers (Hackage).
  - Évaluation stricte.

```
import Control.Monad.Trans.State.Strict
```

  - Évaluation paresseuse.

```
import Control.Monad.Trans.State.Lazy
```

- Module mtl (Hackage).
  - Évaluation stricte.

```
import Control.Monad.State.Strict
```

  - Évaluation paresseuse.

```
import Control.Monad.State.Lazy
```

L'utilisation du module transformers est recommandée.

Règle de base

- Monade de type State Etat Retour
- Le type Retour est retourné par la fonction return.
- Le type Etat est modifié par des fonctions dédiées.

Transformateur (transformer)

- Possibilité de lancer d'autres monade à l'intérieur d'une monade States.
- Monade de type StateT Etat MonadeIn Retour.
- Le type Retour est retourné par la fonction return.
- La monade MonadeIn peut être lancée par la fonction lift et son résultat utilisé dans la monade States.
- Le type Etat est modifié par des fonctions dédiées.

State s est un synonyme vers StateT s Identity

Présentation détaillée

Modification de l'état dans un monade

get :: Monad m ⇒ StateT s m s

Retourne l'état actuel d'une monade State.

```
do
  ...
  stat <- get
  ...
```

put :: Monad m ⇒ s → StateT s m ()

Définie l'état actuel d'une monade State.

```
do
  ...
  put stat
  ...
```

modify :: Monad m ⇒ (s → s) → StateT s m ()

Modifie l'état d'une monade State en appliquant une fonction sur l'état actuel.

```
do
  ...
  modify myFunction
  ...
  modify (\s -> s{count = 3})
  ...
```

Lancement d'une monade State

runState :: State s a → s → (a, s)

Lance un monade State avec un état initial et retourne la valeur finale et l'état final.

```
Prelude> runState (do mapM ajout [1 .. 5]; invZero; ajout2 8; invZero2 ) initS
("Inversion de la valeur de zero",Etat {count = 23, zero = False})
```

evalState :: State s a → s → a

Lance un monade State avec un état initial et retourne la valeur finale uniquement.

```
Prelude> evalState (do mapM ajout [1 .. 5]; invZero; ajout2 8; invZero2 ) initS
"Inversion de la valeur de zero"
```

execState :: State s a → s → s

Lance un monade State avec un état initial et retourne l'état final uniquement.

```
Prelude> execState (do mapM ajout [1 .. 5]; invZero; ajout2 8; invZero2 ) initS
Etat {count = 23, zero = False}
```

Lancement d'une monade StateT

runStateT :: StateT s m a → s → m (a, s)

Lance un monade State avec un état initial et retourne la valeur finale et l'état final.

```
Prelude> runStateT (do mapM ajout [1 .. 5]; invZero; ajout2 8; invZero2 ) initS
("Inversion de la valeur de zero",Etat {count = 23, zero = False})
```

evalStateT :: Monad m ⇒ StateT s m a → s → m a

Lance un monade State avec un état initial et retourne la valeur finale uniquement.

```
Prelude> evalStateT (do mapM ajout [1 .. 5]; invZero; ajout2 8; invZero2 ) initS
"Inversion de la valeur de zero"
```

execStateT :: Monad m ⇒ StateT s m a → s → m s

Lance un monade State avec un état initial et retourne l'état final uniquement.

```
Prelude> execStateT (do mapM ajout [1 .. 5]; invZero; ajout2 8; invZero2 ) initS
Etat {count = 23, zero = False}
```

Exemple simple

```haskell
import           Control.Monad.Trans.State.Lazy

data Etat = Etat
  { count :: Int
  , zero  :: Bool
  }
  deriving Show

initS = Etat 0 False

ajout :: Int -> State Etat String
ajout v = do
  stat <- get
  put (stat { count = count stat + v })
  return $ "Ajout de la valeur " ++ show v ++ " a count. count = " ++ show (count stat + v)

ajout2 :: Int -> State Etat String
ajout2 v = do
  modify (\st -> st { count = count st + v })
  return $ "Ajout de la valeur " ++ show v ++ " a count"

invZero :: State Etat String
invZero = do
  stat <- get
  let newval = not (zero stat)
  put (stat { zero = newval })
  return $ "Inversion de zero de " ++ show (zero stat) ++ " vers " ++ show newval

invZero2 :: State Etat String
invZero2 = do
  modify (\st -> st { zero = not (zero st) })
  return "Inversion de la valeur de zero"
```

Exemple simple (évaluation)

```
ghci> runState (return ()) initS
((),Etat {count = 0, zero = False})
```

```
ghci> runState (do ajout 2) initS
("Ajout de la valeur 2 a count. count = 2",Etat {count = 2, zero = False
    })
```

```
ghci> runState (do ajout 2;ajout 3; invZero) initS
("Inversion de zero de False vers True",Etat {count = 5, zero = True})
```

```
ghci> evalState (do ajout 2;ajout 3; invZero) initS
"Inversion de zero de False vers True"
```

```
ghci> execState (do ajout 2;ajout 3; invZero) initS
Etat {count = 5, zero = True}
```

```
ghci> execState (do mapM ajout [1 .. 5]; invZero; ajout2 8; invZero2 )
    initS
Etat {count = 23, zero = False}
```

Exemple de transformateur de monade

```haskell
import           Control.Monad.Trans.State.Lazy
import           Control.Monad.Trans.Class
import           System.Directory

data Etat = Etat
    { count :: Int
    , files :: [String]
    }
    deriving Show

initS = Etat 0 []

ajoutRep :: String -> StateT Etat IO [String]
ajoutRep path = do
    fils <- lift $ do
        tst <- doesDirectoryExist path
        if tst then listDirectory path else return []
    modify (\st -> st { files = fils ++ files st })
    return fils

ajoutFic :: String -> StateT Etat IO (Maybe String)
ajoutFic path = do
    mbfil <- lift $ do
        tst <- doesFileExist path
        if tst then return (Just path) else return Nothing
    case mbfil of
        Nothing -> return ()
        Just f  -> modify (\st -> st { files = f : files st })
    return mbfil

ajoutVal :: Int -> StateT Etat IO String
ajoutVal v = do
    stat <- get
    put (stat { count = count stat + v })
    return $ "Ajout de la valeur " ++ show v ++ " a count. count = " ++ show (count stat + v)
```

Exemple de transformateur (évaluation)

```
ghci> runStateT (ajoutFic "/home/jean-luc/azerty") initS
(Nothing,Etat {count = 0, files = []})
```

```
ghci> runStateT (ajoutFic "/home/jean-luc/.profile") initS
(Just "/home/jean-luc/.profile",Etat {count = 0, files = ["
    /home/jean-luc/.profile"]})
```

```
ghci> runStateT (ajoutRep "/home/jean-luc/LaTeX/Présentations
    Haskell/Exemples") initS
(["state.hs","stateT.hs",".back"],Etat {count = 0, files = ["state.hs","
    stateT.hs",".back"]})
```

```
ghci> runStateT (do ajoutRep "/home/jean-luc/LaTeX/Présentations
    Haskell/Exemples";ajoutVal 5;ajoutFic "/home/jean-luc/.profile")
    initS
(Just "/home/jean-luc/.profile",Etat {count = 5, files = ["
    /home/jean-luc/.profile","state.hs","stateT.hs",".back"]})
```