Monades et Promise.js

Vincent Archambault-B
IFT 2035 - Université de Montréal

Pour obtenir le code source de ce document

- https://github.com/archambaultv/IFT2035-UdeM
- vincent.archambault-bouffard@umontreal.ca

Qu'est-ce qu'un type ?

- Un type qualifie les propriétés d'une donnée
- Une donnée est le résultat d'un calcul
- *Un type qualifie le résultat d'un calcul*

Type qualifie le résultat d'un calcul

- Un calcul n'est pas uniquement représenté par une fonction
- Une fonction est un calcul paramètré
- Chaque expression en Haskell représente un calcul

```haskell
x :: Int
x = 4

y :: Int
y = 2 + 2 + 4 * 4

z :: Int
z = let a = 2
        b = 4
    in a * a + b

a :: Int
a = foldl (+) 0 [1, 2, 3, 4]
```

Calcul dont le résultat est un Int (Haskell)

Quel type représente un calcul qui peut échouer ?

- Maybe

```haskell
data Maybe a = Nothing
             | Just a

lookup :: key -> [(key, v)] -> Maybe v
lookup = …
```

Haskell

Quel type représente un calcul qui peut retourner une exception ?

- Either

```haskell
data Either b a = Left b
                | Right a

-- Exception sous forme de String
lookup :: key -> [(key, v)] -> Either String v
lookup = …

-- Exception sous forme d'un autre type
lookup2 :: key -> [(key, v)] -> Either Error v
lookup2 = ...
```

Haskell

Quel type représente un calcul qui peut générer plusieurs résultats ?

- List

```haskell
data List a = Nil
            | Cons a (List a)

-- Retourne l'ensemble des sudokus valides
-- en ajoutant un seul nombre à celui passé
-- en entrée
remplirUneCase :: Soduku -> [Sudoku]
remplirUneCase = ...
```

Haskell

Quel type représente un calcul qui met à jour un état global ?

- state -> (a, state)

```haskell
data State s a = State (s -> (a, s))

runState :: State s a -> s -> (a, s)
runState (State f) s = f s

eval :: Exp -> Env -> (Value, Env)
eval = ...

evalS :: Exp -> State Env Value
evalS = ...
```

Haskell

Comment séquencer un calcul qui peut échouer ?

```
xs :: [(Int, Value)]
xs = ...

lookup :: key -> [(key, v)] -> Maybe v
lookup = ...

-- On cherche k1, k2 et k3 dans xs
foo :: [(k, v)] -> (k, k, k) -> Maybe (v, v, v)
foo xs (k1, k2, k3) =
  case lookup k1 xs of
    Nothing -> Nothing
    Just v1 -> case lookup k2 xs
                 Nothing -> Nothing
                 Just v2 -> case lookup k3 xs
                              Nothing -> Nothing
                              Just v3 -> Just (v1, v2, v3)
```

Approche directe (Haskell)

Comment séquencer un calcul qui peut retourner une exception ?

```
xs :: [(Int, Value)]
xs = ...

lookup :: key -> [(key, v)] -> Either String v
lookup = ...

-- On cherche k1, k2 et k3 dans xs
foo :: [(k, v)] -> (k, k, k) -> Either String (v, v, v)
foo xs (k1, k2, k3) =
  case lookup k1 xs of
    Left err -> Left err
    Right v1 -> case lookup k2 xs
                  Left err -> Left err
                  Right v2 -> case lookup k3 xs
                                Left err -> Left err
                                Right v3 -> Right (v1, v2, v3)
```

Approche directe (Haskell)

Comment séquencer un calcul qui peut générer plusieurs résultats ?

```
-- On veut remplir 3 cases et obtenir l'ensemble des sudoku valides
remplir3Cases :: Sudoku -> [Soduku]
remplir3Cases x =
  let xs1 = remplirUneCase x
  let xs2 = concat (map remplirUneCase xs1)
  let xs3 = concat (map remplirUneCase xs2)
  in xs3
```

Approche directe (Haskell)

Comment séquencer un calcul qui met à jour un état global ?

```
-- On doit faire d'abord l'évaluation de l'opérateur et ensuite de l'argument
eval :: Exp -> Env -> (Value, Env)
eval (App f arg) env =
  let (fValue, fEnv) = eval f env
  let (argValue, argEnv) = eval arg fEnv
  in case fValue of
        ... -> ... (result, argEnv)

evalS :: Exp -> State Env Value
evalS (App f arg) = State $ \env ->
  let (fValue, fEnv) = runState (evalS f) env
  let (argValue, argEnv) = runState (evalS arg) fEnv
  in case fValue of
        ... -> ... (result, argEnv)
```

Approche directe (Haskell)

Peut-on faire mieux ???

- Oui, il suffit d'utiliser une fonction d'ordre supérieur pour séquence deux calculs

Comment séquencer un calcul qui peut échouer ?

```
xs :: [(Int, Value)]
xs = ...

lookup :: key -> [(key, v)] -> Maybe v
lookup = ...

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) Nothing _ = Nothing
(>>=) (Just a) f = f a

-- On cherche k1, k2 et k3 dans xs
foo :: [(k, v)] -> (k, k, k) -> Maybe (v, v, v)
foo xs (k1, k2, k3) = lookup k1 xs >>=
                      (\v1 -> lookup k2 xs >>=
                         (\v2 -> lookup k3 xs >>=
                             (\v3 -> Just (v1, v2, v3))))
```

Approche avec fonction d'ordre supérieur (Haskell)

Comment séquencer un calcul qui peut retourner une exception ?

```
xs :: [(Int, Value)]
xs = ...

lookup :: key -> [(key, v)] -> Either String v
lookup = ...

(>>=) :: Either b a -> (a -> Either b c) -> Either b c
(>>=) (Left err) _ = Left err
(>>=) (Right a) f = f a

-- On cherche k1, k2 et k3 dans xs
foo :: [(k, v)] -> (k, k, k) -> Either String (v, v, v)
foo xs (k1, k2, k3) = lookup k1 xs >>=
                      (\v1 -> lookup k2 xs >>=
                         (\v2 -> lookup k3 xs >>=
                             (\v3 -> Right (v1, v2, v3))))
```

Approche avec fonction d'ordre supérieur (Haskell)

Comment séquencer un calcul qui peut générer plusieurs résultats ?

```
(>>=) :: [a] -> (a -> [b]) -> [b]
(>>=) a f = concat (map f a)

-- On veut remplir 3 cases et obtenir l'ensemble des sudoku valides
remplir3Cases :: Sudoku -> [Soduku]
remplir3Cases x = remplirUneCase x >>=
                    (\x1 -> remplirUneCase x1 >>=
                       (\x2 -> remplirUneCase x2))
```

Approche avec fonction d'ordre supérieur (Haskell)

Comment séquencer un calcul qui met à jour un état global ?

```
-- On doit faire d'abord l'évaluation de l'opérateur et ensuite de l'argument
(>>=) :: State b a -> (a -> State b c) -> State b c
(>>=) (State foo) f =
  State (\s0 -> let (x, s1) = foo s0 -- Exécute le premier calcul
                in runState (f x) s1) -- Exécute le deuxième calcul

evalS :: Exp -> State Env Value
evalS (App f arg) = evalS f >>=
                      (\fValue -> evalS arg >>=
                        (\argValue -> case fValue of ... ))
```

Approche avec fonction d'ordre supérieur (Haskell)

Peut-on faire mieux ???

- Oui, il s'agit de combiner fonction d'ordre supérieur et une syntaxe spéciale (programmation structurée)

La notation Do

- La notation Do permet de séquence plusieurs calculs en générant le code utilisant la fonction d'ordre supérieur (>>=)

La notation Do

```
foo = do
  x1 <- calcul1
  x2 <- calcul2
  calcul3
  calcul4
```

Notation Do

```
foo = calcul1 >>=
        (\x1 -> calcul2 >>=
          (\x2 -> calcul3 >>=
            (\_ -> calcul4)))
```

Traduction

Comment séquencer un calcul qui peut échouer ?

```
xs :: [(Int, Value)]
xs = ...

lookup :: key -> [(key, v)] -> Maybe v
lookup = ...

-- On cherche k1, k2 et k3 dans xs
foo :: [(k, v)] -> (k, k, k) -> Maybe (v, v, v)
foo xs (k1, k2, k3) = do
  v1 <- lookup k1 xs
  v2 <- lookup k2 xs
  v3 <- lookup k3 xs
  Just (v1, v2, v3)
```

Approche avec la notation Do (Haskell)

Comment séquencer un calcul qui peut retourner une exception ?

```
xs :: [(Int, Value)]
xs = ...

lookup :: key -> [(key, v)] -> Either String v
lookup = ...

-- On cherche k1, k2 et k3 dans xs
foo :: [(k, v)] -> (k, k, k) -> Either String (v, v, v)
foo xs (k1, k2, k3) = do
  v1 <- lookup k1 xs
  v2 <- lookup k2 xs
  v3 <- lookup k3 xs
  Right (v1, v2, v3)
```

Approche avec la notation Do (Haskell)

Comment séquencer un calcul qui peut générer plusieurs résultats ?

```
(>>=) :: [a] -> (a -> [b]) -> [b]
(>>=) a f = concat (map f a)

-- On veut remplir 3 cases et obtenir l'ensemble des sudoku valides
remplir3Cases :: Sudoku -> [Soduku]
remplir3Cases x = do
  x1 <- remplirUneCase x
  x2 <- remplirUneCase x1
  remplirUneCase x2
```

Approche avec la notation Do (Haskell)

Comment séquencer un calcul qui met à jour un état global ?

```
-- On doit faire d'abord l'évaluation de l'opérateur et ensuite de l'argument
evalS :: Exp -> State Env Value
evalS (App f arg) = do
  fValue <- evalS f
  argValue <- evalS arg
  case fValue of ...
```

Approche avec la notation Do (Haskell)

Qu'on en commun Maybe, Either, List et State ?

- Chaque type représente un calcul sur une valeur quelconque a. (Type paramétrique)
- Chaque type possède sa version de >>=
- Chaque type possède un constructeur représentant le déroulement normal (Just, Right, Cons, State)

Qu'est-ce qu'un monade ?

- Un type paramétrique
- Une définition de **>>=**
- Une définition de **return** qui indique le constructeur du déroulement normal

Qu'est-ce qu'un monade ?

- En Haskell, ces trois critères sont regroupés dans la typeclass Monad

Monade Maybe

```
instance Monad Maybe where

  return a = Just a

  (>>=) Nothing _ = Nothing
  (>>=) (Just a) f = f a
```

Définition du Monade Maybe

Monade (Either b)

```
instance Monad (Either b) where

  return a = Right a

  (>>=) (Left err) _ = Left err
  (>>=) (Right a) f = f a
```

Approche avec fonction d'ordre supérieur (Haskell)

Comment séquencer un calcul qui peut générer plusieurs résultats ?

```
instance Monad List where

  return a = [a]

  (>>=) a f = concat (map f a)
```

Approche avec fonction d'ordre supérieur (Haskell)

Comment séquencer un calcul qui met à jour un état global ?

```
instance Monad (State s) where
   return a = State (\s -> (a, s))

   (>>=) (State foo) f =
      State (\s0 -> let (x, s1) = foo s0 -- Exécute le premier calcul
                    in runState (f x) s1) -- Exécute le deuxième calcul
```

Approche avec fonction d'ordre supérieur (Haskell)

Monade <=> code impératif

- Les monades permettent de représenter des calculs qui produisent des effets de bords (échec, exception, backtracking, changement d'état, log, Input / Ouput, etc …)

Monade <=> code impératif

- La notation Do permet à Haskell d'exprimer du code impératif

```
foo :: M T1
foo = do {
  x1 :: T2 <- calcul1;
  x2 :: T3 <- calcul2;
  calcul3;
  calcul4;
  ...
}
```

Code Haskell

```
T1 foo () {
  T2 x1 = calcul1;
  T3 x2 = calcul2;
  calcul3;
  calcul4;
  ...
}
```

Code C

Monade <=> code impératif

- Les monades permettent de représenter du code impératif dans les langages fonctionnels
- Quel serait leur utilité dans un langage impératif ?
  - *Permettre de manipuler le résultat d'un calcul pour lequel il n'existe pas de syntaxe de programmation structurée*

Monade et Promise.js

- Javascript (notamment Node.js), permet d'appeler des fonctions de façon asynchrone
- Cela se fait en général avec des callbacks (Continuation !!!)

```
var callback = function(error, val) {
  if (error) {
     console.log("Error " + result);
     return;
  }
  console.log("Success " + result);
}

connectToWebsite(url, callback)
```

Javascript

Monade et Promise.js

- Comment chaîner plusieurs callback ?

```
f1(data, function (err, r1){
  if (err) {
    console.log("Error " + err)
  } else {
    data2 = ...
    f2 (data2, function (err, r2){
      if (err) {
        console.log("Error " + err)
      } else {
        data3 = ...
        f3 (data3, function (err, r3){
        ...
        })
    })
})
```

Javascript

Ça ressemble pas mal à ceci (mais asynchrone) :

```
foo data =
  case f1 data of
    Left err -> Left err
    Right v1 -> case f2 data2 xs
                  Left err -> Left err
                  Right v2 -> case f2 data2 xs
                                Left err -> Left err
                                Right v3 -> ...
```

Comparaison avec l'approche directe du monade Either en Haskell

Existe-t-il un monade pour les opérations asynchrones qui peuvent échouer ?

- OUI !
- Le monade Promise

Monade et Promise.js

- Si f1 retourne une Promise

```
f1(data)
  .then(r1 => data2 = ... ; f2(data2))
  .then(r2 => data3 = ... ; f2(data3))
  .catch(failureCallback)
```

Javascript

Monade et Promise.js

- .then est l'équivalent de >>=
- .catch gère le cas où une erreur se produit
- Le calcul arrête dès qu'une erreur se produit
- Contrairement au callback, il est possible de construire un tableau d'objet Promise représentant les résultats de calcul asynchrone