logo

background

Programmation Fonctionnelle en Haskell

Olivier Hermant
olivier.hermant@mines-paristech.fr

MINES ParisTech, Centre de Recherche en Informatique

October 04, 2017

Introduction : La Programmation Fonctionnelle

- programmation impérative :
  - procédurale,
  - comment résoudre
  - Machines de Turing, architecture de Von Neumann
  - état mémoire,
  - boucles, tests, ...
- programmation fonctionnelle :
  - spécification du problème,
  - λ-calcul,
  - fonctions d'ordre supérieur,
  - types, constructeurs, filtrage.
- tout est dans tout :
  - Turing-complet
  - fonctionnel en Java (8 et 9), en Python
  - boucles et tests en Haskell, objets en OCaml
- s'efforcer d'utiliser les constructions idiomatiques

Calcul du PGCD

- en Python :

```
def pgcd( x, y) :
    while y != 0 :
        r = x % y
        x = y
        y = r
    return x
```

- en Haskell :

```
gcd x 0 = x
gcd x y = gcd y (mod x y)
```

  - proche de la définition mathématique
  - plus récursif, aussi

Définition

Une fonction pure appelée avec les mêmes arguments a *toujours* le même résultat

- toujours le cas en maths
- permet optimisations haut-niveau, compile-time
  - "Exploiting Vector Instructions with Generalized Stream Fusion", *G. Mainland, R. Leshchinskiy, S. Peyton-Jones*, ICFP 2013
- en particulier : demander une info à l'utilisateur (I/O) : non pur
- ne dépend d'aucun *état global*
- Haskell est un langage fonctionnel pur

Types, Constructeurs, Filtrage

- Ingrédients essentiels de tout langage fonctionnel
- type de `gcd`

Types, Constructeurs, Filtrage

- Ingrédients essentiels de tout langage fonctionnel
- type de `gcd`
- bonne pratique : écrire soi-même le type

Types, Constructeurs, Filtrage

- Ingrédients essentiels de tout langage fonctionnel
- type de `gcd`
- bonne pratique : écrire soi-même le type
- Haskell infère les types

  Dans l'invite de commande interactive (`ghci`) taper `:t gcd`
- Type classes (préconditions)

Types, Constructeurs, Filtrage

- Ingrédients essentiels de tout langage fonctionnel
- type de `gcd`
- bonne pratique : écrire soi-même le type
- Haskell infère les types

  Dans l'invite de commande interactive (`ghci`) taper `:t gcd`
- Type classes (préconditions)
- définir un nouveau type : `data` Majuscule

```
data MaListe a = Vide | Elem (a, MaListe a)
```

  - `a` est un paramètre de type (polymorphisme)
  - Vide et Elem sont les constructeurs de type
  - définition inductive...

Types, Constructeurs, Filtrage

- Ingrédients essentiels de tout langage fonctionnel
- type de `gcd`
- bonne pratique : écrire soi-même le type
- Haskell infère les types

  Dans l'invite de commande interactive (`ghci`) taper `:t gcd`
- Type classes (préconditions)
- définir un nouveau type : `data` Majuscule

  `data MaListe a = Vide | Elem (a, MaListe a)`
  - `a` est un paramètre de type (polymorphisme)
  - Vide et Elem sont les constructeurs de type
  - définition inductive...
- motif de base en programmation fonctionnelle : filtrage

```
tete Vide = ...
tete Elem(a,queue) = ...
```

```
tete2 = case l of
          Vide -> ...
          Elem(x,_) -> ...
```

Effets de bord, états

- Haskell est pur
- comment faire des entrées/sorties ?

Effets de bord, états

- Haskell est pur
- comment faire des entrées/sorties ?
- les Monades (demo t-shirt)

Effets de bord, états

- Haskell est pur
- comment faire des entrées/sorties ?
- les Monades (demo t-shirt)
- question plus naïve : comment remplir les "`...`" ci-dessous ?

```
tete Vide = ...
tete Elem(a,queue) = ...
```

```
tete2 = case l of
          Vide -> ...
          Elem(x,_) -> ...
```

  - Solution 1 : lancer une Exception
  - Solition 2 : `null`

Introduction aux Monades - Maybe

- langage fortement typé : forcer le programmeur (par typage) à faire le travail
  - `null` n'est pas très bien typé ...
  - retourner `null` génère de potentielles NullPointerException
  - Exceptions = mécanisme fonctionnel (cf. CPS et opérateurs de contrôle)
  - non local !
- autre Exemple : `lookup` dans une liste d'associations

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

- Le code devient alors :

```
tete Vide = Nothing
tete Elem(x,queue) = Just x
```

```
tete2 = case l of
          Vide -> Nothing
          Elem(x,_) -> Just x
```

- et le type, `MaListe a -> Maybe a`
- deux constructeurs : `Nothing` et `Just`

Soit `m a` une monade (polymorphe en a)

- `return :: a -> m a`
- `>>= :: m a -> (a -> m b) -> m b`
- `>>=` est un opérateur (infixe) nommé bind
  - lui seul peut ouvrir la monade `m a`
  - accède au contenu (de type `a`)
  - le donne en argument à une fonction
  - la force à produire uen valeur dans la monade `m b`
  - les "impuretés" restent dans la monade (sous le tapis)
- trois lois à respecter :

`m >>= return`	=	m	(identité à droite)
`return x >>= f`	=	f x	(identité à gauche)
`(m >>= f) >>= g`	=	`m >>= \x -> ((f x) >>= g)`	(associativité)

- essentiellement : tout ce qui est censé *marcher par typage*, doit marcher.

- prenons le cas où la monade `m` est `Maybe`
- l'implémentation est la suivante :

```
return :: a -> Maybe a
return x = Just x
```

```
(>>=) :: Maybe a -> (a -> Maybe b)
(>>=) m g = case m of
  Nothing -> Nothing
  Just x -> g x
```

- les lois sont respectées (exercice)

- effet de bord : afficher/demander des informations
- la Monade `IO` :

```
putStrLn :: String -> IO ()
getLine :: IO String
```

- `putStrLn` retourne une valeur dans une Monade (effet de bord), pas d"'état"
- `getLine` retourne une chaîne (entrée par l'utilisateur = effet de bord), encapsulée dans la Monade IO.

- effet de bord : afficher/demander des informations
- la Monade `IO` :

```
putStrLn :: String -> IO ()
getLine :: IO String
```

- `putStrLn` retourne une valeur dans une Monade (effet de bord), pas d"'état"
- `getLine` retourne une chaîne (entrée par l'utilisateur = effet de bord), encapsulée dans la Monade IO.
- Quizz : comment faire `echo` en Haskell ?

- effet de bord : afficher/demander des informations
- la Monade `IO` :

```
putStrLn :: String -> IO ()
getLine :: IO String
```

- `putStrLn` retourne une valeur dans une Monade (effet de bord), pas d"'état"
- `getLine` retourne une chaîne (entrée par l'utilisateur = effet de bord), encapsulée dans la Monade IO.
- Quizz : comment faire `echo` en Haskell ?
  - on aimerait composer `getLine` et `putStrLn` (`putStrLn . getLine`)
  - interdit par le typage : `getLine` ne retourne pas une chaîne

- effet de bord : afficher/demander des informations
- la Monade `IO` :

```
putStrLn :: String -> IO ()
getLine :: IO String
```

- `putStrLn` retourne une valeur dans une Monade (effet de bord), pas d"'état"
- `getLine` retourne une chaîne (entrée par l'utilisateur = effet de bord), encapsulée dans la Monade IO.
- Quizz : comment faire `echo` en Haskell ?
  - on aimerait composer `getLine` et `putStrLn` (`putStrLn . getLine`)
  - interdit par le typage : `getLine` ne retourne pas une chaîne
  - or `getLine` retourne une Monade : utiliser »=
    - argument de gauche : `IO a`
    - argument de droite : `a -> IO b`
    - type de retour : `IO b`

- effet de bord : afficher/demander des informations
- la Monade `IO` :

```
putStrLn :: String -> IO ()
getLine :: IO String
```

- `putStrLn` retourne une valeur dans une Monade (effet de bord), pas d"'état"
- `getLine` retourne une chaîne (entrée par l'utilisateur = effet de bord), encapsulée dans la Monade IO.
- Quizz : comment faire `echo` en Haskell ?
  - on aimerait composer `getLine` et `putStrLn` (`putStrLn . getLine`)
  - interdit par le typage : `getLine` ne retourne pas une chaîne
  - or `getLine` retourne une Monade : utiliser »=
    - argument de gauche : `IO a`
    - argument de droite : `a -> IO b`
    - type de retour : `IO b`
  - dans notre cas,
    - `getLine :: IO String`
    - `putStrLn :: String -> IO ()`
    - type de retour : `IO ()`

Problématique :

- on ne peut se débarrasser (proprement, c.à.d. “purement”) des Monades
- une fois apparue, on la transporte donc en permanence
- exemple : `putStr “Bonjour, ” » putStr “MSI ” » putStr “!”`
- faire en sorte que le code reste lisible

```
do { putStr “A” ;
     putStr “B” ;
     putStr “C” }
```

Problématique :

- on ne peut se débarrasser (proprement, c.à.d. “purement”) des Monades
- une fois apparue, on la transporte donc en permanence
- exemple : `putStr “Bonjour, ” » putStr “MSI ” » putStr “!”`
- faire en sorte que le code reste lisible

```
do { putStr “A” ;
     putStr “B” ;
     putStr “C” }
```

- lorsque l'on a l'opérateur bind : `action1 >>= (\ x1 -> action2 >>= (\ x2 -> mk_action3 x1 x2 ))`

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

Encore quelques questions

Afficher un Maybe String (`lookup`) ?

- Impossible de se débarrasser de `Maybe`,
- ou alors filtrer soi-même (impur ! ! ! Même problème en Java)
- t-shirt Monades

- un cas particulier de Monades
- Compréhension, en Python et en Haskell aussi :
  - (il y a de la monade derrière ...)
    - `[(x,y) | x <- [1,2], y <- [1,5] ]`

Exercice

Ecrire la liste infinie [1,2,3,4,...]

- avec une fonction qui la génère de préférence
- style "impératif" : un compteur d'état ?
  - style "fonctionnel" : fonction auxiliaire
  - état ? Utiliser une monade (State Monad) ?
- tête et queues d'une liste infinie :
  - pb en OCaml
  - pas de pb en Haskell
  - il ne faut tout de même pas demander la lune...

Quelques Constructions de Base

- composition : `head . tail`
- `zipWith`, `take` : la librairie `Prelude`
- `$` au lieu des parenthèses
- les parenthèses : `(+)` $\sim$ `\x -> \y -> x + y`
  - transforme un symbole en fonction
- exemple : `(zipWith (+) [1..5]) . tail`

Exercice

Fonction qui prend une liste, et retourne la liste des `l[i] + l[i+1]` des l[i]

- composition : `head . tail`
- `zipWith`, `take` : la librairie `Prelude`
- `$` au lieu des parenthèses
- les parenthèses : `(+)` ∼ `\x -> \y -> x + y`
  - transforme un symbole en fonction
- exemple : `(zipWith (+) [1..5]) . tail`

Exercice

Fonction qui prend une liste, et retourne la liste des `l[i] + l[i+1]` des l[i]

- possibilité : `(uncurry $ zipWith (+)) . \l -> (l, tail l)`

Règles Générales

- Éviter constructions impératives et objet
- plus d'une ligne par programme ? Réfléchissez encore !
- écrire un ligne prend 5 minutes ? Normal.

Ressources en Ligne

`http://www.haskell.org`

- librairies built-in, Prelude : `http://zvon.org/other/haskell/Outputprelude/index.html`
- 99 problems in Haskell `https://wiki.haskell.org/H-99:_Ninety-Nine_Haskell_Problems`
- A gentle introduction to Haskell : `https://www.haskell.org/tutorial/`