

# Programmation Fonctionnelle en Haskell

Olivier Hermant

`olivier.hermant@mines-paristech.fr`

**MINES ParisTech, Centre de Recherche en Informatique**

26 septembre 2019

# 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,
  - ▶  $\lambda$ -calcul,
  - ▶ fonctions d'ordre supérieur,
  - ▶ types, constructeurs, filtrage.
- ▶ tout est dans tout :
  - ▶ Turing-complet
  - ▶ fonctionnel en Java (8, 9 et 10), en Python
  - ▶ boucles et tests en Haskell, objets en OCaml
- ▶ s'efforcer d'utiliser les **constructions idiomatiques**

- ▶ 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
- ▶ ne dépend d'aucun *état global*
- ▶ Haskell est un langage fonctionnel pur

## 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
- ▶ ne dépend d’aucun *état global*
- ▶ Haskell est un langage fonctionnel pur
- ▶ problème : demander une info à l’utilisateur (I/O)? **non pur**

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

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

- ▶ Ingrédients essentiels de tout langage fonctionnel
- ▶ type de `gcd`
- ▶ bonne pratique : écrire soi-même le type des fonctions
- ▶ Haskell **infère** les types (cf. semaine prochaine)

Dans l'invite de commande interactive (`ghci`) taper `:t gcd`

- ▶ 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...



- ▶ Ingrédients essentiels de tout langage fonctionnel
- ▶ type de `gcd`
- ▶ bonne pratique : écrire soi-même le type des fonctions
- ▶ Haskell **infère** les types (cf. semaine prochaine)

Dans l'invite de commande interactive (`ghci`) taper `:t gcd`

- ▶ 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,_) -> ...
```

- ▶ quel type pour `mod`? Pour `lookup`?
- ▶ polymorphe
- ▶ demande *certaines conditions* sur le type
- ▶ les typeclasses
  - ▶ Java  $\approx$  interfaces
- ▶ un peu de magie, lors de la définition de nouveaux types :
  - ▶ implémentation manuelle des fonctions demandées
  - ▶ implémentation automatique dans certains cas (`Eq`, `Show`)

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

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

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

```
tete Vide = ...           tete2 = case 1 of
tete Elem(a,queue) = ...   Vide -> ...
                           Elem(x,_) -> ...
```

- ▶ Solution 1 : lancer une Exception (avec error)
- ▶ Solution 2 : null

- ▶ 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          tete2 = case 1 of
tete Elem(x,queue) = Just x   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 une valeur dans la monade `m b`
- ▶ trois lois à respecter :

<code>m &gt;&gt;= return</code>	<code>= m</code>	(identité à droite)
<code>return x &gt;&gt;= f</code>	<code>= f x</code>	(identité à gauche)
<code>(m &gt;&gt;= f) &gt;&gt;= g</code>	<code>= m &gt;&gt;= \x -&gt; ((f x) &gt;&gt;= g)</code>	(associativité)

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

- ▶ les lois sont respectées (exercice)
- ▶ une fois à l'intérieur, plus moyen d'en sortir!
  - ▶ sauf localement ... pour retomber dedans, ou dans une autre monade, juste après

- ▶ effet de bord : afficher/demander des informations
- ▶ une fois dans une monade, on n'en sort plus : y cacher les "impuretés"
- ▶ la Monade IO :

```
putStrLn :: String -> IO ()  
getLine  :: IO String
```
- ▶ `putStrLn` a ses valeurs 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
- ▶ une fois dans une monade, on n'en sort plus : y cacher les "impuretés"
- ▶ la Monade IO :

```
putStrLn :: String -> IO ()  
getLine  :: IO String
```
- ▶ `putStrLn` a ses valeurs 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
- ▶ une fois dans une monade, on n'en sort plus : y cacher les "impuretés"
- ▶ la Monade IO :

```
putStrLn :: String -> IO ()
getLine  :: IO String
```
- ▶ `putStrLn` a ses valeurs 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 faire la composition (`putStrLn . getLine`)
  - ▶ **interdit** par typage : `getLine` ne retourne pas une chaîne

- ▶ effet de bord : afficher/demander des informations
- ▶ une fois dans une monade, on n'en sort plus : y cacher les "impuretés"
- ▶ la Monade IO :

```
putStrLn :: String -> IO ()
getLine  :: IO String
```
- ▶ `putStrLn` a ses valeurs 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 faire la composition (`putStrLn . getLine`)
  - ▶ **interdit** par typage : `getLine` ne retourne pas une chaîne
  - ▶ or `getLine` retourne dans la monade IO : utiliser `>>=`
    - ▶ argument de gauche : `IO a`
    - ▶ argument de droite : `a -> IO b`
    - ▶ type de retour : `IO b`

- ▶ effet de bord : afficher/demander des informations
- ▶ une fois dans une monade, on n'en sort plus : y cacher les "impuretés"
- ▶ la Monade IO :

```
putStrLn :: String -> IO ()
getLine  :: IO String
```
- ▶ `putStrLn` a ses valeurs 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 faire la composition (`putStrLn . getLine`)
  - ▶ **interdit** par typage : `getLine` ne retourne pas une chaîne
  - ▶ or `getLine` retourne dans la monade IO : 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 *purement pas* se débarrasser des Monades
- ▶ une fois apparue, on la transporte 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 *purement pas* se débarrasser des Monades
- ▶ une fois apparue, on la transporte 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 }
```



- ▶  $M$  est une monade ssi on a deux opérateurs :

$$\begin{aligned}\text{return} &:: a \rightarrow M\ a \\ \gg= &:: M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b\end{aligned}$$

- ▶  $F$  est un foncteur ssi on a un opérateurs :

$$\text{fmap} :: (a \rightarrow b) \rightarrow F\ a \rightarrow F\ b$$

- ▶  $F$  est un foncteur applicatif ssi on a deux opérateurs :

$$\begin{aligned}\text{pure} &:: a \rightarrow F\ a \\ \langle * \rangle &:: F\ a \rightarrow F\ (a \rightarrow b) \rightarrow F\ b\end{aligned}$$

- ▶ (ces opérateurs satisfont des lois, telles que  $\text{fmap}\ (f \ .\ g) = (\text{fmap}\ f) \ .\ (\text{fmap}\ g)$ )
- ▶ **Théorème** : toute monade est un foncteur applicatif, tout foncteur applicatif est un foncteur.
- ▶ **Exercice**. On peut “peler” une monade :  $M\ (M\ a) \rightarrow M\ a$

Afficher un `Maybe String` (`lookup`) ?

- ▶ Impossible de se débarrasser de `Maybe`,
- ▶ ou alors filtrer soi-même (même problème en Java)
- ▶ on retrouve une description textuelle de la monade
- ▶ et on tombe dans une autre monade ... les `String` (si)
- ▶ t-shirt Monades

La vision la plus générale d'une monade est qu'elle permet de faire des calculs / d'exécuter des actions, tout en **contenant** quelque chose.

- ▶ un cas particulier de monades : quel calcul? Non-déterminisme.
- ▶ et de foncteur : qu'est `fmap` dans ce cas?
- ▶ Compréhension, en Python et en Haskell aussi :
  - ▶ (il y a de la monade derrière ...)
    - ▶ `[(x,y) | x <- [1,2], y <- [1,5] ]`
  - ▶ ou, avec `do` :  
`do x <- [1,2]; y <- [1,5]; return (x,y)`

## Exercice

Ecrire la liste infinie  $[1,2,3,4,\dots]$

- ▶ avec une fonction qui la génère
- ▶ 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...
- ▶ quelle est la complexité de `append (++)` ?

- ▶ `composition` : `head . tail`
- ▶ `zipWith`, `take` : la librairie `Prelude`
- ▶ fonction identité anonyme : `\x -> x`
- ▶ `$` au lieu des parenthèses
- ▶ les parenthèses : `(+) ~ \x -> \y -> x + y`
  - ▶ transforme un symbole en fonction
  - ▶ inverse par les backquotes : ``mod`` (fonction en opérateur infixe)
- ▶ exemple : `(zipWith (+) [1..5]) . tail`

## Exercice

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

- ▶ composition : `head . tail`
- ▶ `zipWith`, `take` : la librairie `Prelude`
- ▶ fonction identité anonyme : `\x -> x`
- ▶ `$` au lieu des parenthèses
- ▶ les parenthèses : `(+) ~ \x -> \y -> x + y`
  - ▶ transforme un symbole en fonction
  - ▶ inverse par les backquotes : ``mod`` (fonction en opérateur infixe)
- ▶ exemple : `(zipWith (+) [1..5]) . tail`

## Exercice

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

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

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

## Un exemple de fonction bien connue

```
qs :: Ord a => List a -> List a
qs [] = []
qs (p:t1) = (qs $ filter (< p) t1) ++ [p] ++ (qs $ filter (>= p) t1)
```

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

## Un exemple de fonction bien connue

```
qs :: Ord a => List a -> List a
qs [] = []
qs (p:t1) = (qs $ filter (< p) t1) ++ [p] ++ (qs $ filter (>= p) t1)
```

- ▶ et que se passe-t-il si on met (<) et (>=) au lieu de < et >=?



`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/`