# Programmation Fonctionnelle en Haskell

Olivier Hermant
olivier.hermant@mines-paristech.fr

MINES ParisTech, Centre de recherche en informatique

20 septembre 2021

# 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+), Python
  - boucles et tests en Haskell, objets en OCaml
- s'efforcer d'utiliser les constructions idiomatiques

# Robin Milner (1934 – 2010)

*"Well typed programs cannot go wrong"*

1978, A Theory of Type Polymorphism in Programming

# Calcul du PGCD

- en Python :

```
def gcd( 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 (Fonction Pure)

Une fonction est pure ssi, quand elle est appelée avec les mêmes arguments, elle donne *toujours* le même résultat

- exemple : les maths
- optimisations haut-niveau (e.g. composition), 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 (Fonction Pure)

Une fonction est pure ssi, quand elle est appelée avec les mêmes arguments, elle donne *toujours* le même résultat

- exemple : les maths
- optimisations haut-niveau (e.g. composition), 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 : tout n'est pas pur
  - demander une info à l'utilisateur (I/O) ?
  - cf. cet après-midi

# 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 des fonctions

# Types, constructeurs, filtrage

- 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 : mot-clef `data`, puis 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 des fonctions
- Haskell infère les types (cf. semaine prochaine)

  Dans l'invite de commande interactive (`ghci`) taper `:t gcd`
- définir un nouveau type : mot-clef `data`, puis 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` ?
  - exercice : analyser le type de lookup, que fait cette fonction ? (indice : voir slide 3)
- polymorphe
- demande *certaines conditions* sur le type
- les typeclasses
  - Java ≈ interfaces
- un peu de magie, lors de la définition de nouveaux types :
  - implémentation manuelle des fonctions demandées
  - implémentation automatique dans certains cas (`Eq`, `Show`)
- dans le cas de `MaListe`, on peut directement écrire

```
data MaListe a = Vide | Elem (a, MaListe a) deriving
                                (Eq, Show)
```

## Exercice

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

- avec une fonction qui la génère

## Exercice

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

- avec une fonction qui la génère
- style "impératif" : compteur d'état ?
  - style "fonctionnel" : fonction auxiliaire
  - état ? Utiliser une monade (State Monad) ?

## Exercice

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

- avec une fonction qui la génère
- style "impératif" : 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 (++) ?

# Quelques constructions de base

- composition : `head . tail`
- sur les listes :
  - ajout en tête avec `:`, concaténation avec `++`
  - `zipWith`, `take` : la librairie `Prelude`
- fonction identité anonyme : `\x -> x`
- `$` au lieu des parenthèses
- les parenthèses : `(+)` $\sim$ `\x -> \y -> x + y`
  - transforme un opérateur infixe en fonction
  - inverse par les backquotes : `` `mod` `` (fonction $\rightsquigarrow$ opérateur infixe)
- exemple : `(zipWith (+) [1..5]) . tail`

## Exercice

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

# Quelques constructions de base

- composition : `head . tail`
- sur les listes :
  - ajout en tête avec `:`, concaténation avec `++`
  - `zipWith`, `take` : la librairie `Prelude`
- fonction identité anonyme : `\x -> x`
- `$` au lieu des parenthèses
- les parenthèses : `(+)` ∼ `\x -> \y -> x + y`
  - transforme un opérateur infixe en fonction
  - inverse par les backquotes : `` `mod` `` (fonction ⇝ 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:tl) = (qs $ filter (< p) tl) ++ [p] ++ (qs $ filter (>= p) tl)
```

- 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:tl) = (qs $ filter (< p) tl) ++ [p] ++ (qs $ filter (>= p) tl)
```

- que se passe-t-il si on ajoute des parenthèses : `(<)` et `(>=)` au lieu de < et >= ?

- question naïve : comment remplir les "..." ci-dessous ?

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

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

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

- Solution 2 :
  - `null` n'est pas très bien typé ...
  - retourner `null` génère de potentielles NullPointerException
- Solution 1 :
  - Exceptions = mécanisme fonctionnel (CPS, opérateurs de contrôle)
  - problème : non local
- langage fortement typé : forcer par typage à faire le travail (cf. slide 3)
- exemple : `lookup` dans une liste d'associations, ou `head`

# Le type de données Maybe

```
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`

# TP et Ressources en Ligne

1. se familiariser avec Haskell : reprendre le TP de 1A. Arbres binaires contenant des `Int`, calcul de hauteur, de nombre de nœuds, de feuilles. Parcours infixes, préfixes et postfixes.
2. se familiariser avec la librairie : 99 problems in Haskell.

`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/`

- Haskell est pur, le monde est impur
- comment faire des entrées/sorties ?

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

- Haskell est pur, le monde est impur
- comment faire des entrées/sorties ?
- les Monades (demo t-shirt)
- plus simple : comment rendre la fonction `head`, sur les listes, totale ?

$$\texttt{head []} \equiv ?$$

- Haskell est pur, le monde est impur
- comment faire des entrées/sorties ?
- les Monades (demo t-shirt)
- plus simple : comment rendre la fonction `head`, sur les listes, totale ?

$$\texttt{head []} \equiv ?$$

- la monade `Maybe`
  - partie "fonctionnelle pure" (sortir la valeur)
  - partie "impure" (cas d'échec)
  - nous oblige à retourner un **résultat empaqueté** dans un "calcul"

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
  - à la condition qu'elle sache produire une valeur dans la monade `m b`
- trois lois à respecter :

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

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

- prenons le cas où la monade `m` est `Maybe`

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

- l'implémentation (déjà faite) est la suivante :

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

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) v g = ?
```

- les lois sont respectées (exercice)
- une fois "impur" à l'intérieur de la monade, impossible d'en sortir !
  - sauf très localement ... pour retomber dedans juste après

- prenons le cas où la monade `m` est `Maybe`

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

- l'implémentation (déjà faite) est la suivante :

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

```
(>>=) :: Maybe a -> (a ->
Maybe b) -> Maybe b
(>>=) v g = ?
```

- les lois sont respectées (exercice)
- une fois "impur" à l'intérieur de la monade, impossible d'en sortir !
  - sauf très localement ... pour retomber dedans juste après

- prenons le cas où la monade `m` est `Maybe`

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

- l'implémentation (déjà faite) est la suivante :

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

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

- les lois sont respectées (exercice)
- une fois "impur" à l'intérieur de la monade, impossible d'en sortir !
  - sauf très localement ... pour retomber dedans juste après

- prenons le cas où la monade `m` est `Maybe`

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

- l'implémentation (déjà faite) est la suivante :

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

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

- les lois sont respectées (exercice)
- une fois "impur" à l'intérieur de la monade, impossible d'en sortir !
  - sauf très localement ... pour retomber dedans juste après

- effet de bord : afficher/demander des informations
- une fois dans une monade, on y reste : 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 y reste : 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 y reste : 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 y reste : 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 y reste : 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" }
```

- avec la version complète de bind :
  `action1 >>= (\x1 -> action2 >>= (\x2 -> mk_action3 x1 x2 ))`
  devient

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

- `M` est une monade ssi on a deux opérateurs :

```
return :: a -> M a
>>= :: M a -> (a -> M b) -> M b
```

- `F` est un foncteur ssi on a un opérateur :

```
fmap :: (a -> b) -> F a -> F b
```

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

```
pure :: a -> F a
<*> :: F a -> F (a -> b) -> F b
```

- (ces opérateurs satisfont des lois, telles que `fmap (f . g) = (fmap f) . (fmap g)`)

**Théorème**

Toute monade est un foncteur applicatif, tout foncteur applicatif est un foncteur.

- Exercice. "peler" une monade : `m (m a) -> m a`
  1. à la main, sur `Maybe`
  2. avec n'importe quelle monade

# Encore quelques questions

Afficher un Maybe String (`lookup`) ? Extraire un `Int` à partir d'un `Maybe Int` ?

- Impossible de se débarrasser de `Maybe`,
- ou alors filtrer soi-même
- soit on n'est pas injectif, soit on retrouve une description 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 une valeur.

- un cas particulier de monades : les listes
- c'est un foncteur : qu'est `fmap` dans ce cas ?

```
"fmap" :: (a -> b) -> [a] -> [b]
```

- 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)
```

- un cas particulier de monades : les listes
- c'est un foncteur : qu'est `fmap` dans ce cas ?

```
"fmap" :: (a -> b) -> [a] -> [b]
```

- vu en tant que monade :

```
"return" :: a -> [a]
">>=" :: [a] -> (a -> [b]) -> [b]
```

- 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)
```

- un cas particulier de monades : les listes
- c'est un foncteur : qu'est `fmap` dans ce cas ?

```
"fmap" :: (a -> b) -> [a] -> [b]
```

- vu en tant que monade :

```
"return" :: a -> [a]
">>=" :: [a] -> (a -> [b]) -> [b]
```

- quel "calcul" ? Non-déterminisme.
- 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)
```

- un cas particulier de monades : les listes
- c'est un foncteur : qu'est `fmap` dans ce cas ?

```
"fmap" :: (a -> b) -> [a] -> [b]
```

- vu en tant que monade :

```
"return" :: a -> [a]
">>=" :: [a] -> (a -> [b]) -> [b]
```

- quel "calcul" ? Non-déterminisme.
- 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 : écrire la variante avec >>=.