

“Bon converti sera meilleur prêcheur.”

Aliaume Lopez

12 Décembre 2018

Introduction

L'objectif qui nous guide

« L'expérience, ce n'est pas ce qui arrive à quelqu'un, c'est ce que quelqu'un fait avec ce qui lui arrive. »

1. Maintenir un code *lisible*
2. Produire un code *testable*
3. Conserver un code *efficace*

Quelques méthodes

1. Factoriser au maximum les fonctions
2. Reconnaître les motifs récurrents
3. Développer des méthodes de conception

*Du fait de complexité **contextuelle**, il faut comprendre l'ensemble des **actions** s'offrant à nous, à court terme.*

**« Le style est l'expression de la
pensée. »**

Motifs à la définition

```
mysorting :: [Int] -> [Int]
mysorting l = sort q
  where
    q = take (n `div` 2) l
    n = length l
```

Construction de fonction

« Ce qui vaut la peine d'être fait vaut la peine d'être bien fait. »

```
myLast :: [a] -> a
myLast []      = error "No end for empty lists!"
myLast [x]     = x
myLast (_:xs) = myLast xs
```

```
myLastByComposition = head . reverse
```

```
myLastByFolding l = foldr _ _ l
```

Construction de fonction

« Mieux vaut plier que rompre. »

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Ce n'est pas sans rappeler l'encodage de Church...

Polymorphisme paramétrique

Pour quoi faire ?

Theorems for free — Wadler

*Write down the definition of a polymorphic function on a piece of paper. Tell me its **type**, but be careful not to let me see the function's definition. I will tell you a **theorem** that the function satisfies.*

« Un uniforme ? C'est un avant-projet de cercueil. »

Ça ne peut pas nous aider à factoriser des raisonnements *méta*...

Polymorphisme non paramétrique

« Être raisonnable en toutes circonstances ? Il faudrait être fou... »

Une fonction palindrome en une ligne

```
isPalindrome :: (Eq a) => [a] -> Bool
```

```
isPalindrome = Control.Monad.liftM2 (==) id reverse
```

Exercice sur le quicksort

Construction d'un code polymorphe

```
qsort :: [Int] -> [Int]
qsort []      = []
qsort [x]     = [x]
qsort (x:xs) = qsort l1 ++ [x] ++ qsort l2
  where
    l1 = [ y | y <- xs, y <= x ]
    l2 = [ y | y <- xs, y >  x ]
```


Construction d'un code polymorphe

```
qsort :: (Int -> Int -> Bool) -> [Int] -> [Int]
qsort (<=) []      = []
qsort (<=) [x]     = [x]
qsort (<=) (x:xs) = qsort l1 (<=) ++ [x] ++ qsort (<=) l2
  where
    l1 = [ y | y <- xs,      y <= x ]
    l2 = [ y | y <- xs, not (y <= x) ]
```

Construction d'un code polymorphe

```
qsort :: (a -> a -> Bool) -> [a] -> [a]
qsort (<=) []      = []
qsort (<=) [x]     = [x]
qsort (<=) (x:xs) = qsort (<=) l1 ++ [x] ++ qsort (<=) l2
  where
    l1 = [ y | y <- xs,      y <= x ]
    l2 = [ y | y <- xs, not (y <= x) ]
```

Construction d'un code polymorphe

« La vie, c'est très drôle, si on prend le temps de regarder »

```
data Comparable a = Comparable (a -> a -> Bool)
```

```
qsort :: Comparable a -> [a] -> [a]
```

```
cmpInt :: Comparable Int
```

```
cmpInt = Comparable (<=)
```

Exercice sur le quicksort (bis)

Typeclass to the rescue

« La vie, c'est très drôle, si on prend le temps de regarder »

```
data Comparable a = Comparable (a -> a -> Bool)
```

```
qsort :: Comparable a -> [a] -> [a]
```

```
cmpInt :: Comparable Int
```

```
cmpInt = Comparable (<=)
```

Typeclass to the rescue

« La vie, c'est très drôle, si on prend le temps de regarder »

```
class Comparable a where cmp :: a -> a -> Bool  
qsort :: Comparable a => [a] -> [a]
```

```
instance Comparable Int where  
    cmp = (<=)
```

Remarque

La donnée `Comparable` n'est plus passée en argument de la fonction, mais est *inférée* par le système de type.

« L'autorité contraint à l'obéissance, mais la raison y persuade. »

Pour chaque type de donnée, il ne peut y avoir au plus qu'une seule instance. Et on ne peut pas avoir des instances qui se recoupent.

Remarque

La donnée `Comparable` n'est plus passée en argument de la fonction, mais est *inférée* par le système de type.

« L'autorité contraint à l'obéissance, mais la raison y persuade. »

Pour chaque type de donnée, il ne peut y avoir au plus qu'une seule instance. Et on ne peut pas avoir des instances qui se recoupent.

Overlapping instances

```
instance Comparable [a] ...
```

```
instance Comparable [Int] ...
```


Factorisons par l'héritage !

Héritage de classes

```
class Monoid a => Group a where  
  inv :: a -> a
```

« Qui ne sait rien, de rien ne doute »

On peut définir plusieurs instances de Monoid pour Int...

Comment choisir ?

« Qui ne sait rien, de rien ne doute »

On peut définir plusieurs instances de `Monoid` pour `Int...`

Comment choisir ?

On construit un alias newtype `Sum ..`

Sucreries

1. Dérivation automatique de Eq, Ord, Enum, Show, Read, Compare
2. Dérivation d'instance semi-manuelle
3. Extensions du compilateur (Generalised Newtype Deriving, Deviring Via, Generics)

Exercice: Sérialisation

Typeclassopedia

MONOÏDES

**« La pierre n'a point d'espoir d'être autre chose que pierre.
Mais de collaborer, elle s'assemble et devient temple »**

- Entiers
- Listes
- HaTeX
- Configuration

HaTeX

```
instance Monoid LaTeX where
  mempty = TeXEmpty
  mappend TeXEmpty x = x
  mappend x TeXEmpty = x
  -- This equation is to make 'mappend' associative.
  mappend (TeXSeq x y) z = TeXSeq x $ mappend y z
  --
  mappend x y = TeXSeq x y
```

HaTeX

```
superModifier :: (a -> [LaTeX]) -> [a] -> LaTeX  
superModifier f = mconcat . mconcat . map f
```

Configuration

1. Non commutatif mais associatif
2. Agrégation depuis plusieurs sources

« Simplifier ce n'est pas faire simple »

```
class Foldable t where  
    foldMap :: Monoid m => (a -> m) -> t a -> m
```

Applications.

1. find, all, any ...
2. minimum, maximum ...

FONCTEURS

« Passer en revue les options »

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

« On peut traverser ? Négatif C'est un fleuve a crocodiles »

```
class Functor f where
```

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

« **Agent Starling**, que faites-vous quand vous n'enquêtez pas ? »

On peut comprendre `Maybe` comme un endofoncteur défini par

$$MX = 1 + X$$

« Non mais tu te prends pour un Jedi, ou quoi, à faire des passes avec les mains comme ça ? »

Question 1

Mais dans quelle catégorie ?!

Question 2

Quid des morphismes ?

Question 3

Comment vérifier les équations ?


```
instance Functor Maybe where
    fmap f Nothing  = Nothing
    fmap f (Just x) = Just  (f x)
```

```
instance Functor [] where
    fmap f []      = []
    fmap f (x:xs) = f x : fmap f xs
```

```
instance Functor ((->) a) where
    fmap f g = ...
```

```
instance Functor Maybe where
    fmap f Nothing  = Nothing
    fmap f (Just x) = Just (f x)
```

```
instance Functor [] where
    fmap f []      = []
    fmap f (x:xs) = f x : fmap f xs
```

```
instance Functor ((->) a) where
    fmap f g = f . g
```

Exercices

« Vous voulez peut-être que je sorte pour pousser ? »

$$D_A : LMA \rightarrow MLA$$

APPLICATIVE

Faire sortir des fonctions

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Quel est l'équivalent catégorique ?

« Mauvaise question ? Mauvaise réponse ! »

Un foncteur F auquel on ajoute un morphisme $unit : 1 \rightarrow F1$, une structure de *lax* naturelle $l_{A,B} : FA \times FB \rightarrow F(A \times B)$ et une *strength* $s_{A,B} : A \times FB \rightarrow F(A \times B)$.

Dans un sens ...

$$pure = \rho; A \times unit; s_{A,1}; F_{\rho^{-1}}$$

$$fapply = l_A \Rightarrow_{B,A}; T_{eval}$$

Exercices

- Option
- Listes
- Fonctions $TB = (a \implies B)$

Un exemple d'application

Une version inefficace du palindrome

```
isPalindrome = pure (==) <*> id <*> reverse
```

ALTERNATIVE

Alternative

```
class Applicative f => Alternative f where
    empty :: f a
    (<|>) :: f a -> f a -> f a
```

- Option
- Listes

MONADE

```
class Functor a => Monad f where
    return :: a -> f a
    (>>=)   :: f a -> (a -> f b) -> f b
```

Do notation (Kleisli arrows)

do

 x <- expr1

 expr2

==

expr1 >>= (\x -> expr2)

Exercices

- Option
- Listes (compréhension comme monadique)
- State/Read/Write
- Probabilités

Listes (compréhension comme monadique)

```
[ x | x <- l1, y <- l2, x + y == 5 ]
```

```
===
```

```
do
```

```
  x <- l1
```

```
  y <- l2
```

```
  if x + y == 5 then
```

```
    return x
```

```
  else
```

```
    empty
```


Monad

Équivalence avec la définition usuelle ?

Dans un compilateur

```
data Compiler a = ...
```

```
createFreshVariable :: Compiler VarName
```

```
getOrCreateVariable :: VarName -> Compiler Asm
```

```
produceAsm           :: Asm      -> Compiler ()
```

```
registerString      :: String   -> Compiler VarName
```

Hakyll

```
match "posts/*" $ do
    route    $ setExtension "html"
    compile  $ pandocCompiler
        >>= loadAndApplyTemplate
            "templates/post.html"
            postCtx
        >>= loadAndApplyTemplate
            "templates/default.html"
            postCtx
        >>= relativizeUrls
```

Hakyll

- Séparation de la sémantique et de l'API
- Application déclarative
- Architecture flexible (checks, autoupdate. . .)

Applications

Optparse-Applicative

```
sample :: Parser Sample
sample = Sample
    <$> strOption
        ( long "hello"
        <> metavar "TARGET"
        <> help "Target for the greeting" )
    <*> switch
        ( long "quiet"
        <> short 'q'
        <> help "Whether to be quiet" )
    <*> option auto
        ( long "enthusiasm"
        <> help "How enthusiastically to greet"
        ...
```

Optparse-Applicative

hello - a test for optparse-applicative

Usage: hello --hello TARGET [-q|--quiet]
 [--enthusiasm INT]

Print a greeting for TARGET

Available options:

--hello TARGET	Target for the greeting
-q,--quiet	Whether to be quiet
--enthusiasm INT	How enthusiastically to greet (default: 1)
-h,--help	Show this help text

- Une alternative aux `tagless` interpreters
- Qui permet d'utiliser un peu notre bagage en catégories
- Et qui a le bon goût d'être très facile à implémenter !