Le langage Haskell

Une introduction au langage
Liens avec les catégories

exposé de : **Ph. Reitz**
le : **26 janvier 2001**

LIRMM
161, rue Ada
34392 Montpellier cedex 5

Tél : +33 467 41 86 63
Fax : +33 467 41 85 00
Mél : reitz@lirmm.fr
WWW : http://www.lirmm.fr/~reitz/

Préambule

◇ Origine

Conférence en 1987 sur les langages fonctionnels (FPLCA'87)

Constat : un nombre croissant de langages fonctionnels ayant tous plus ou moins les mêmes caractéristiques.

Le nom d'Haskell : en l'honneur de Haskell B. Curry.

◇ Principaux acteurs

- Yale Univ. (Etats Unis)
  → P. Hudak & al.
- Glasgow Univ. (Écosse)
  → P. Wadler, S. Peyton-Jones & al.
- Chalmers Univ. (Suède)
  → T. Johnsson, L. Augustsson & al.

◇ Principaux travaux de référence

Lisp/Scheme/FP	: λ-calcul
ML	: système de types
Miranda	: syntaxe concise
... et bien d'autres	

Les points forts du langage ▷

◇ Langage fonctionnel

- fonctions d'ordre supérieur (fonction = valeur manipulable par calcul)
- application partielle de fonction (currification)

◇ Langage typé

- typage **fort**, **statique** et **inféré**
- types **polymorphiques** (les types peuvent être variabilisés)
- notion de **filtrage** (ou appariement, déstructuration)
- types organisés en **classes** de type (sortes de types abstraits).

◇ Particularités du langage

- **évaluation paresseuse** : une (sous-)expression n'est évaluée que lorsque c'est nécessaire.
- **surcharge des noms** autorisée : le système de types a été enrichi pour l'intégrer (classes de types)
- gestion du concept d'état dans un cadre fonctionnel pur : les **monades** ; sont concernés : les entrées-sorties, les tableaux à composantes mutables, le compilateur `Haskell`(en particulier l'analyseur syntaxique)...

◁ Les points forts du langage

◇ Et tout ce qui caractérise un bon langage

- notion de module
  séparation interface / implantation
  possibilité de cacher une partie de l'implantation
- porté sur pratiquement toutes les plate-formes
  MacOS, MS-DOS, Windows, Unix/X11, ...
- mode de travail au choix compilé / interprété
- générateur `Haskell`→ `C`
- exploite l'indentation des programmes pour alléger l'écriture

`Haskell` $\overset{traducteur}{\longrightarrow}$ `Haskell` épuré

Programmer avec Haskell

◇ Qu'est-ce qu'un programme Haskell

programme = des définitions
→ de types
→ de variables (dont les fonctions)
→ de classes
→ de modules
\+ une expression à évaluer

◇ Au sein d'un même fichier...

Toutes les définitions sont mutuellement récursives → l'ordre des définitions n'a pas d'importance.

Valeurs et types

◇ Les valeurs

Une **valeur** est :
– manipulable par calcul (donnée ou fonction).
– dénotée par une ou plusieurs **expressions de calcul**.
Une expression de calcul ne dénote qu'une seule valeur.
De toutes les expressions dénotant une valeur, une seule est qualifiée de **forme normale**.

Calculer revient à rechercher, pour une expression e, la classe d'équivalence (caractérisée par la valeur dénotée par e) à laquelle e appartient (→ **réduction** à la forme normale).

◇ Les types

Un **type** est :
– non manipulable par calcul (seul le compilateur les manipule).
– dénoté par une **expression de type**. Deux expressions de type distinctes dénotent deux types distincts, sauf synonymie explicitement déclarée.
– une étiquette associée à toute valeur. Une valeur ne possède qu'un type.

Quelques conventions/notations

◇ Notation

`v::T` signifie que la valeur dénotée par `v` a pour type celui dénoté par `T`.

◇ Les identificateurs

– toute variable (normale, i.e. liée à une valeur, ou de type) possède un nom commençant par une **lettre minuscule**
Exemple :

```
a = 1          {* a est une variable normale *}
[] :: [a]      {* a est une variable de type *}
```

– tout constructeur (de données, de type ou de classe) possède un nom commençant par une **lettre majuscule**.
Exemple :

```
True           {* constructeur de données *}
Pile a         {* Pile est un constructeur de type *}
```


Les fonctions

◇ Les λ-abstractions

`\x -> e` est l'écriture `Haskell` de $\lambda x.e$ du λ-calcul.
`\`x_1 ... x_n `-> e` abrège `\`x_1 `-> (...(\`x_n `-> e)...)`

◇ L'application de fonction

Si `f` et `e` sont deux expressions, alors `f e` est l'expression de `f` *est appliquée à* `e`.

◇ Currification

L'application de fonction : associative à gauche, précédence maximale.

Exemple :
Soit l'expression `f a b c + g a b`.
Reconnue comme `(((f a) b) c) + ((g a) b)`.
Peut se récrire comme `x b c + g a b`, *avec* `x = f a`
→ `f` *appliquée partiellement (1 seul paramètre lié)*
→ `x` *est donc une fonction attendant au moins 2 paramètres.*

Les fonctions

◇ Les opérateurs (fonctions binaires infixes)

▷ Passage infixe ↔ préfixe

`a + b` peut aussi s'écrire `` `+` a b``
`f a b` peut aussi s'écrire `a `f` b`

▷ Les sections

`(x +)` est équivalent à `\y -> x+y`
`(+ y)` est équivalent à `\x -> x+y`
`(+)` est équivalent à `\x y -> x+y`

Le système de type de Haskell

◇ Le système des types prédéfinis

▷ Constructeurs de type de base

`Int`, `Integer`, `Bool`, `Char`, `Float`, `Double`, ...

▷ Constructeur de type produit, noté `( , )`

si $x_1 :: T_1, \ldots, x_n :: T_n$, alors $(x_1, \ldots, x_n) :: (T_1, \ldots, T_n)$, où $n \geq 2$.

▷ Constructeur de type somme, noté `|`

Pas de forme simple → nécessite l'usage de **constructeurs de données** (cf. page suivante).

▷ Constructeur de type fonction, noté `->`

si $x :: A$ et $y :: B$, alors `(\ x -> y) :: A -> B`.
L'application étant associative à gauche, `->` est associatif à droite.

```
          a ::  X
Si        b ::  Y  alors f ::  X -> (Y -> Z),
   (f a b) ::  Z      soit f ::  X -> Y -> Z
```


Le système de type de Haskell

◇ Les types synonymes

type $T\ v_1\ \ldots\ v_n\ =\ e$

Synonyme de type = expression déclarée comme synonyme d'une expression de type e existante.

Exemple :

```
type String = [Char]
type Liste a = [a]
```

Pas de création d'un nouveau type, seulement ajout d'un nouveau nom de type pour le compilateur.

◇ Le système des types utilisateurs

$$\begin{array}{lcl} \textbf{data}\ T\ v_1\ \ldots\ v_n & = & C_1\ t_{11}\ \ldots\ t_{1\alpha(1)} \\ & | & C_2\ t_{21}\ \ldots\ t_{2\alpha(2)} \\ & \ldots & \\ & | & C_m\ t_{m1}\ \ldots\ t_{m\alpha(m)} \end{array}$$

où :

- T = nom du constructeur de type
- v_i = i-ème variable de type
- C_j = nom du j-ème constructeur de données
- t_{jk} = type du k-ème constituant du constructeur C_j

Exemple :

```
data Bool = True | False
```


Le système de type de Haskell

◇ Exemple : le type des listes

```
data [a] = []  |  a : [a] {* pseudo-code ... *}
```

◇ Autres exemples

```
data Point a = Pt a a
p = Pt 3.5 (-2.3)            :: Point Float

data Arbre a = Vide
             | Noeud a (Arbre a) (Arbre a)

data IntBool = Ci Int | Cb Bool
x = (1, False)               :: (Int, Bool)
y = Ci 1 : Cb False : []     :: [IntBool]
```

◇ Sur les constructeurs de données...

Constructeur de données = constante ou fonction

Exemple :

```
True    :: Bool

Vide    :: Arbre a
Noeud   :: a -> Arbre a -> Arbre a -> Arbre a
```

À propos du polymorphisme

◇ Le polymorphisme de type

Une expression de type peut contenir des **variables de type**. Ces variables sont quantifiées universellement.

Exemple :

```
data Pile a = Vide | Empiler a (Pile a)
```

Le type `Pile a` *doit être lu comme* $\forall a Pile(a)$.

```
estVide p = (p == Vide)     :: Pile a -> Bool
```

Le type de la fonction `estVide` *est :* $\forall a(Pile(a) \rightarrow Bool)$
Une valeur de type `Pile a` *peut être de type polymorphique ou non.*

```
p = Vide           :: Pile a

q = Empiler 1.0 p  :: Pile Float
```

◇ Un polymorphisme ad hoc : la surcharge

Un même identificateur est lié à au moins deux valeurs de types différents :

```
+ :: Double -> Double -> Double
+ :: Int -> Int -> Int
```

Démonstration (extrait d'une session avec `Hugs`) :

```
? 1+2
3 :: Int
? 1.0+2.0
3.0 :: Double
```

Nous étudions plus loin comment `Haskell` gère cette surcharge...

À propos du polymorphisme

◇ Le polymorphisme de valeur (ou paramétrique)

```
empiler p n = Empiler n p
```

◇ La généricité

En `Ada` ou `C++`, généricité = polymorphisme de type + paramétrage classique :

```
template
        <class Elem,
        int taille>
class PileBornée {
        Elem tab[taille-1];
        ...};

PileBornée<int, 10> p;
```

En `Haskell`, pas de confusion des genres :

```
data PileBornée a = UnePile Int (Pile a)

creerPile n = UnePile n Vide

p = empiler (creerPile 10) 1   :: PileBornée Int
```


Le type prédéfini des listes

◇ Constructeurs de données

```
[]  :: [a]                {* la liste vide *}
(:) :: a -> [a] -> [a]    {* le CONS de Lisp *}
```

En pseudo-code `Haskell` :

```
data [a] = []
         | a : [a]
```

L'opérateur `:` est associatif à droite : `a:(b:c)` = `a:b:c`.

▷ Facilités syntaxiques

$[u_1, \ldots, u_n]$ est équivalent à $u_1 : \ldots : u_n : []$
`[a..b]` est équivalent à `[a, a+1, a+2, ..., b]`
`[a..]` est équivalent à `[a, a+1, a+2, ...`
`[a, b..c]` est équivalent à `[a, a+`r`, a+2`r`, ..., c]`, avec r =`b-a`
`[a, b..]` est équivalent à `[a, a+`r`, a+2`r`, ...`

◇ Les principales fonctions

```
(++) :: [a] -> [a] -> [a]   {* concaténation *}
head :: [a] -> a            {* tête *}
tail :: [a] -> [a]          {* queue *}
map  :: (a -> b) -> [a] -> [b]
...
```


Les listes

◇ Définition en compréhension

Spécifier comment les éléments doivent être calculés :
– générateurs : *var* `<-` *expression retournant une liste*
– gardes : *expression booléenne*

Exemple :

La liste des entiers pairs de 0 à 20 :

```
{* 1ère méthode *}
[ x | x <- [0..20], x mod 2 == 0 ]
{* 2ème méthode *}
[0, 2..20]
```

Exemple :

La liste des sommes de x et y, où x entier pair quelconque dans 2..50 et y entier quelconque dans 3..7 :

```
[ x+y | x <- [2..50], y <- [3..7], x mod 2 == 0 ]
```

Exemple :

Un classique : le tri rapide

```
tri []     = []
tri (x:r) = tri [y | y <- r, y<x]
          ++ [x]
          ++ tri [y | y <- r, y>x]
```

Les fonctions

◇ Conditionnelle

if *expr_1* **then** *expr_2* **else** *expr_3*

Les expressions *expr_2* et *expr_3* ont même type.

◇ Définition locale (1ère forme)

let *filtre* = *expr 1* **in** *expr 2*

Exemple :

```
let x = 2 in x*x
{* a pour valeur 4, de type Int *}

let (x, y) = ('a', 'b') in [x]++[y]
{* a pour valeur "ab", de type String *}
```

La première expression est fonctionnellement équivalente à :

```
(\ x -> x*x) 2
```

◇ Définition locale (2ème forme)

expression **where** *définitions*

Exemple :

```
x*y where x = 2 * z
          y = 3 * x
          z = -1
```

Avantages : autorise plusieurs définitions de variables, avec spécification éventuelle de leur type.

Les fonctions

◇ Le filtrage

case *expr* **of** {
clause 1 ;
...
clause_n }

où chaque *clause_i* est de la forme :

filtre_i | *garde_i1* -> *expr_i1*
where { *définitions locales* }
...
| *garde im* -> *expr im*
where { *définitions locales* }

Une **garde** est une expression booléenne.
Un **filtre** est soit :
– un nom de variable
– le caractère joker _
– un constructeur de données d'arité n suivi de n filtres
– un filtre nommé, noté *nom*@*filtre*
– un filtre gelé, noté ~*filtre*
Les parties **where** (définitions locales) sont optionnelles.

Les fonctions

Exemple :

```
case e of {
     l@(x:y:r) | x==y -> r
               | True -> l ;
     [x]       | True -> [x,x] ;
     _         | True -> [] }
```

L'expression est simplifiable (omission des gardes inutiles).

◇ Remarques

La conditionnelle **if** est implantée par un **case**.
La variable suivante est prédéfinie :

```
otherwise = True
```

Permet d'écrire alors :

```
case e of {
     l@(x:y:r) | x==y      -> r
               | otherwise -> l ;
     ...
```


Les fonctions

Définir une variable, c'est écrire une (ou plusieurs) équation :

nom = *expression*

Exemple :

Différentes définitions d'une même fonction add, *exploitant diverses facilités syntaxiques du langage :*

```
add = \ a ->(\ b ->
         if b==0 then a else 1 + add a (b-1))

add = \ a b -> if b==0 then a else 1 + add a (b-1)

add a b = if b==0 then a else 1 + add a (b-1)

add a b = case b of {
               0 -> a ;
               _ -> 1 + add a (b-1) }

add a b | b==0      = a
        | otherwise = 1 + add a (b-1)

add a 0 = a
add a b = 1 + add a (b-1)

add a 0     = a
add a (b+1) = 1 + add a b
```

Admirez la concision et l'élégance de l'écriture :

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib n + fib (n+1)
```

Les fonctions

Exemple :

Si vous n'êtes toujours pas convaincu, peut-être le serez-vous après ces quelques définitions :

```
fact 0       = 1
fact m@(n+1) = m * fact n

histogramme []    = []
histogramme (x:r) = maj (histogramme r)
  where
    maj :: [(a, Int)] -> [(a, Int)]
    maj []                      = [(x, 1)]
    maj (c@(y, n):r) | x==y     = (x, n+1):r
                     | otherwise = c : maj r

?  histogramme [1,2,1,3,2,1,2,3,2,3,1,2,3,2,3,2]
=  [(3,5), (2,7), (1,4)]
:: [(Int,Int)]

partitionner _ []    = []
partitionner f (x:r) = maj (partitionner f r)
  where
    maj []                      = [(v, [x])]
    maj (c@(k, l):r) | k==v     = (v, x:l):r
                     | otherwise = c : maj r
    v = f x

?  partitionner reste3 [0,1,2,3,4,5,6,7,8,9]
     where reste3 x = x `mod` 3
=  [(0,[0, 3, 6, 9]), (2,[2, 5, 8]), (1,[1, 4, 7])]
:: [(Int,[Int])]
```


Les fonctions

◇ Un dernier exemple

Le minimum sur des arbres binaires dont chaque noeud est associé à une valeur.

```
data Arbre a = Vide
             | Feuille a
             | Noeud a (Arbre a) (Arbre a)

infixe f (Noeud v g d) = infixe f g++[f v]++infixe f d
infixe f (Feuille v)   = [f v]
infixe _ Vide          = []

prof (Noeud _ g d) = 1+max (prof g) (prof d)
prof (Feuille _)   = 1
prof Vide          = 0

insere v Vide                          = Feuille v
insere v f@(Feuille u) | v==u          = f
                       | v>u           = Noeud u Vide n
                       | otherwise     = Noeud u n Vide
            where n = Feuille v
insere v n@(Noeud u g d) | v==u        = n
                         | v>u         = Noeud u g (ins d)
                         | otherwise   = Noeud u (ins g) d
            where ins = insere v
```


L'évaluation en Haskell

◇ L'évaluateur est paresseux

Évaluation des expressions à la demande (lazy)

Autorise la manipulation de structure de données *virtuellement* infinies.

Exemple :

Les nombres premiers, en utilisant le crible d'Eratostène.

```
premiers      = map head (iterate crible [2..])
crible (p:xs) = [ x | x<-xs, x `rem` p /= 0 ]
```

avec la fonction prédéfinie `iterate` *définie par :*

```
iterate f x   = x : iterate f (f x)
```

Autre exemple, la suite de Fibonacci :

```
fibs = 1:1:[ a+b | (a, b) <- zip fibs (tail fibs) ]
```

avec la fonction prédéfinie `zip` *définie par :*

```
zip (x:u) (y:v) = (x, y) : zip u v
zip _     _     = []
```

Dernier exemple : considérons la fonction suivante :

```
f t x y = if t then x else y
```

L'évaluation de `f (1==1) (10/2) (1/0)` *se termine sur 5.*

La gestion de la surcharge

◇ Énoncé

à un même nom sont associées au moins deux valeurs, de types distincts.

Pour une expression contenant ce nom, quelle valeur choisir (et donc quel type) ?

Exemple :

L'addition d'entiers ou de flottants

```
? 1 + 5
6 :: Int

? 3.0 + 5.0 ;
8.0 :: Float
```

L'opérateur `(+)` *est dit* ***surchargé****.*

Question : quel est le type de `(+)` *?*

La gestion de la surcharge

◇ Les solutions existantes

▷ Ada ou C++

En principe, pas de problème, puisque typage explicite.
Toutefois, les constantes posent parfois problème.
Outils offerts au programmeur en Ada :
– typage explicite, c-à-d coercion, voire conversion, de type (levée de la surchage par le type).
– indication du chemin permettant d'atteindre l'unité où est défini l'identificateur (levée de la surcharge par le nom).

▷ Caml-light ou Miranda

Surcharge interdite.
L'opérateur = est polymorphique (a -> a -> Bool) :
– Caml-light étiquette toute valeur par son type → le code de = est fonction de ces types.
– Miranda teste l'égalité du contenu mémoire.

La gestion de la surcharge

◇ Les solutions existantes

▷ Standard ML

Surcharge interdite à l'usager, autorisée dans le noyau.
Égalité définie seulement pour certains types.

▷ Haskell

– approche de ML : autoriser la surcharge dans des cas parfaitement identifiés, mais en offrant aussi cet outil au programmeur.
– prohiber l'approche de Miranda ou Caml : interdire la surcharge tout en l'autorisant de facto → report du problème dans le code.
La solution : les **classes de types**.

Les classes de types

◇ Définition

Classe de types = un ensemble de types (appelés **instances**) qui surchargent un même ensemble de noms typés (appelés **méthodes**).
Une classe peut hériter d'autres classes.
Un type peut être déclaré instance d'une (ou plusieurs) classe de type ; toutes les méthodes doivent alors être définies.
→ assez proche de la notion d'**interface** du langage Java.

◇ Exemple de classe / instance

La classe Eq :

```
class Eq a where
      (==) :: a -> a -> Bool {* égalité *}
      (/=) :: a -> a -> Bool {* inégalité *}
      x /= y  = not (x == y) {* définition par défaut *}
```

Interprétation : un type a est une instance de la classe Eq s'il surcharge les opérateurs == et /= avec les types indiqués.

```
instance Eq Int where
      x == y  = eqInt x y
      x /= y  = neqInt x y
```

Interprétation : Int est une telle instance, i.e. il est possible d'utiliser l'opérateur == pour tester l'égalité de 2 entiers.

Les classes de types

◇ Exemples d'héritage

La classe Ord des types possédant une relation d'ordre hérite de Eq :

```
class Eq a => Ord a where
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a

    x <  y            = x <= y && x /= y
    x >= y            = y <= x
    {* ... et ainsi de suite *}
```

La classe Text des types dont les valeurs sont affichables ou demandables :

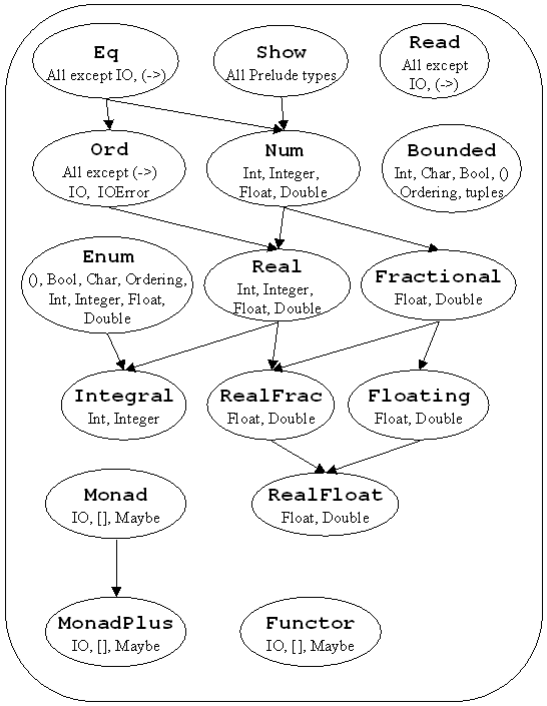
```
class Text a where
    show :: a -> String
    read :: String -> a
    {* ... et d'autres définitions *}
```

La classe Num des nombres hérite de Eq et Text :

```
class (Eq a, Text a) => Num a where
    (+), (-), (*), (/) :: a -> a -> a
    {* .... et ainsi de suite *}
```

Les classes de types

◇ La hiérarchie des classes prédéfinies



Les classes de types

◇ Incidence sur le système de type

Les variables de type peuvent être contraintes par des classes.

Exemple :

La fonction `\x y -> x == y` *a pour type* `(Eq a) => a -> a -> Bool`, *c'est à dire non pas* $\forall a,\ a \to a \to Bool$, *mais* $\forall a \in Eq,\ a \to a \to Bool$

Un classique :

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

member x (y:r) | x == y     = True
               | otherwise  = member x r
member _ []                 = False
```

`member` *n'a donc de sens que pour les listes dont les éléments sont comparables.*

◇ Remarque

Les littéraux numériques sont surchargés !

```
incr x = x+1
incr :: (Num a) => a -> a
```


Compilation des méthodes

Tout programme utilisant des classes est récrit en un programme sans classe.

◇ Idée de la réécriture sur un exemple

```
class Expl a where
        f :: a -> a -> a
        g :: a -> a

instance Expl Int where
        f x y = fInt x y
        g x   = gInt x
```

– à chaque classe est associé un dictionnaire

```
data ExplD a = ExplDic (a->a->a) (a->a)
cle_f (ExplDic x y) = x
cle_g (ExplDic x y) = y
```

– à chaque instance de chaque classe est associé un dictionnaire

```
ExplDInt :: ExplD Int
ExplDInt = ExplDic fInt gInt
```


Compilation des méthodes

– toute occurence de `f` ou `g` est remplacée par un accès au dictionnaire de la classe :

```
{* définition originale *}
fonc x y z | z         = f x y
           | otherwise = g x
{* après réécriture *}
fonc dic x y z | z         = cle_f dic x y
               | otherwise = cle_g dic x
```

– lors de l'application de `fonc`, le dictionnaire du type instance est passé en paramètre :

```
{* appel original *}
fonc 2 3 True
{* après réécriture *}
fonc ExplDInt 2 3 True
```

Les limitations des classes de type

– un nom surchargé n'appartient qu'à une seule classe → un nom surchargé ne peut avoir qu'un seul type.

Exemple :

Il n'est pas possible de définir `+` *afin de l'utiliser pour concaténer des listes (*`[1,2,3]+[4,5,6]`*) ou ajouter un élement au début ou à la fin d'une liste (*`1+[2,3]` *ou* `[1,2]+3`*).*

– la surcharge des littéraux pose un problème de choix d'une instance de type.

Exemple :

```
carre :: (Num a) => a -> a
carre x = x * x

? (carre 2, carre 3.0)
(4, 9.0) :: (Int, Double)
```

Ici l'interpréteur a choisi seul d'associer au nom `2` *l'entier de type* `Int` *ayant la valeur 2, idem pour le nom 3.0 lié à la valeur correspondante de type* `Double`*. Pourquoi pas* `(Integer, Float)` *?*

Ce choix est prédéfini dans le langage, et non modifiable.

◇ Par rapport aux LOO

Le concept de classe est introduit pour permettre le partage de noms signés (typés).

Ce n'est ni un partage de code, ni un partage de structure, ni une classification conceptuelle. Une classe n'est pas manipulable par calcul.

Intérêts des classes de type

◇ Classe de type = type abstrait de données

```
class Pile p where
{* p doit être un constructeur de type unaire *}
    vide    :: p a
    empiler :: a -> p a -> p a
    sommet  :: p a -> a
    depiler :: p a -> p a
    estVide :: p a -> Bool

test :: (Pile p) => a -> p a -> p a
test x p = empiler x (empiler x p)

data Pile1 a = Vide1 | Empiler1 a (Pile1 a)

instance Pile Pile1 where
    vide                  = Vide1
    empiler               = Empiler1
    sommet (Empiler1 x _)  = x
    depiler (Empiler1 _ p) = p
    estVide Vide1         = True
    estVide _             = False

t1 :: Pile1 Int
t1 = test 3 Vide1
```


Intérêts des classes de type

Il est possible d'implanter aussi une pile avec une liste :

```
instance Pile [] where
          {* ^ est le constructeur de type des listes ! *}
    vide       = []
    empiler    = (:)
    sommet     = head
    depiler    = tail
    estVide [] = True
    estVide _  = False

t2 :: [Int]
t2 = test 3 []
```


Les problèmes non résolus

◇ Quant l'inférence de type s'en mêle

```
class Acl a where
      one :: a
      two :: a
      tst :: a -> Bool

class Bcl a where
      fx :: (Acl b) => a -> b
      fy :: (Acl b) => a -> b

      fy u = if tst (fx u) then one else two
      {* Pb de typage pour fy *}
```

▷ Explication

Le type de `fy` est `(Bcl a, Acl b, Acl c) => a -> b`, sachant que le type de `fx` dans la définition est `(Bcl a, Acl c) => a -> c`.

Or aucun résultat (`one` ou `two`) ne vient contraindre `c` explicitement, alors que cette contrainte existe.

Le problème des entrées-sorties

◇ Les entrées-sorties

Les langages fonctionnels bannissent :
- les notions d'**état** et d'**effet de bord** (toute fonction appelée dans un même contexte retourne toujours le même résultat)
- toute influence de l'**ordre des calculs** (théorème de la confluence du λ-calcul).

Ces notions sont toutefois fondamentales dans le domaine des entrées-sorties.

→ un programme fonctionnel peut-il communiquer ?

◇ Les modèles d'entrée-sortie pour les LF

- modèle basé sur les canaux (stream)
- modèle basé sur les continuations
- modèle basé sur les systèmes

Tous ces modèles sont fonctionnellement équivalents, mais les styles de programmation diffèrent.

Le concept d'état peut être formalisé dans un cadre purement fonctionnel : les **monades** (théorie des catégories).

Foncteurs et monades ▷

◇ Quel est l'intérêt des catégories dans les LF

- le $\lambda-$calcul typé est une catégorie particulière.
- concept de **monade** : permet d'intégrer dans un cadre fonctionnel pur des idées qui, de prime abord, semblent en contradiction avec ses fondements
  → la notion d'**effet de bord**, fortement associée à celle d'**état**.
- d'où un engouement réciproque catégoriciens / informaticiens.
  Ils espèrent trouver une utilité à des concepts catégoriels très abstraits (égaliseurs, limites, adjoints, etc.) dans le monde de la programmation, en particulier la découverte de nouvelles structures de données ou de contrôle adaptées à la programmation des machines parallèles (PMP).
  → les langages fonctionnels ont en effet de très bonnes propriétés pour en faire des candidats idéaux dans le monde de la PMP.
  → `Haskell` est l'un des terrains où ces travaux sont très actifs.

◇ Concepts catégoriels dans `Haskell`

Deux concepts catégoriels sont concrétisés :

▷ les foncteurs

Concrétisés pour la forme.
Ne présentent pas d'intérêt véritable.

▷ les monades

Construction très utilisée, afin d'abstraire de nombreux types ayant des fonctionnalités similaires.

◁ Foncteurs et monades ▷

◇ Les foncteurs

Les foncteurs sont concrétisés par une simple classe de type :

```
class Functor f where
      fmap :: (a -> b) -> f a -> f b
```

sachant que la fonction `fmap` doit vérifier :

```
      fmap id = id
fmap (f . g) = fmap f . fmap g
```

avec
- `f . g` désigne la composition $f \circ g$ des fonctions f et g (opérateur `Haskell` prédéfini `(.)`)
- `=` est une relation d'équivalence (ce **n'est pas** le `=` d'une définition `Haskell`)
- `id` est la fonction identité ; en `Haskell` : `id x = x :: a -> a`

Noter qu'il n'y a aucun moyen de vérifier en `Haskell` les contraintes posées sur `fmap`.

▷ Le type des listes appartient à la classe des foncteurs

```
instance Functor [] where
    fmap = map
```

▷ Le type `Maybe` appartient aussi à cette classe

```
data Maybe a = Nothing | Just a

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


◁ Foncteurs et monades ▷

▷ `IO`, type des entrées-sorties, aussi un foncteur

```
data IO a = ... {* type prédéfini *}

instance Functor IO where
    fmap f x = x >>= (return . f)
```

avec `>>=` et `return` deux fonctions associées au caractère monadique du constructeur de type `IO`.

Foncteurs et monades

◇ Les monades - variante 1

Les monades furent, à une époque, concrétisées par une classe de type :

```
class Monad m where
    mmap   :: (a -> b) -> (m a -> m b)
    munit  :: a -> m a
    mjoin  :: m (m a) -> m a
```

sachant que ces fonctions devaient vérifier :

```
        mmap id = id
 mmap (f . g)  = (mmap f) . (mmap g)
mmap f . munit = munit . f
mmap f . mjoin = mjoin . mmap (mmap f)
```

La fonction `mmap` est celle des foncteurs.

Foncteurs et monades

◇ Les monades - variante 2

Les monades sont désormais concrétisées par la classe de type :

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    fail   :: String -> m a

    {* définitions par défaut *}
    p >> q  = p >>= \ _ -> q
    fail s  = error s
```

sachant que ces fonctions doivent vérifier :

```
          return a >>= k = k a
            m >>= return = m
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
              fmap f xs = xs >>= return . f
```

▷ intuitivement...

Dans une monade, `return` joue le rôle de l'élément unité d'un monoïde, alors que `>>=` joue le rôle de son opérateur.

Une monade sert à décrire un code par nature séquentiel.

En gros, le code séquentiel spécifie une séquence d'actions. Chaque action s'appuie sur une donnée de départ (état initial), la transforme (état suivant), et produit au passage une information utile.

Un exemple suit...

Foncteurs et monades

▷ Le type des listes appartient à la classe des monades

```
instance Monad [] where
    (x:xs) >>= f   = f x ++ (xs >>= f)
    []     >>= f   = []
    return x       = [x]
    fail s         = []
```

▷ Le type `Maybe` appartient aussi à cette classe

```
instance Monad Maybe where
    Just x  >>= k = k x
    Nothing >>= k = Nothing
    return        = Just
    fail s        = Nothing
```

◇ Notation do

Une notation spéciale est associée aux opérations monadiques, via le mot-clé **do** :

– l'expression *expr* `>>= (\`*filtre* `->` **do** *suite*) peut s'écrire :

do *filtre* `<-` *expr*
suite

– l'expression *expr* `>>` **do** *suite* peut s'écrire :

do *expr*
suite

– l'expression *expr* peut s'écrire :

do *expr*

Foncteurs et monades

◇ Monades avec une opération +

Il existe une extension à la classe des monades : la classe des monades ayant un opération + admettant un élément dit **zéro** (analogue des monoïdes en théorie des ensembles, avec le zéro jouant le rôle de l'élément neutre de l'opération du monoïde) :.

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

sachant que ces fonctions doivent vérifier :

```
mzero >>= k               = mzero
p >>= (\x -> mzero)       = mzero
mzero `mplus` p           = p
p `mplus` mzero           = p
```

Quelques types représentatifs :

```
instance MonadPlus Maybe where
    mzero             = Nothing
    Nothing `mplus` ys = ys
    xs      `mplus` ys = xs

instance MonadPlus [ ] where
    mzero = []
    mplus = (++)
```

Foncteurs et monades

▷ Intérêt des monades : la transformation d'états

Soit le code C suivant, calculant le PGCD de deux entiers :

```
while (x != y) if (x<y) y = y-x; else x = x-y;
```

Avec la notation **do**, ce code peut s'écrire en Haskell :

```
pgcd = do x <- getX
          y <- getY
          (if x == y
           then
             return x
           else if x < y
           then
             do setY (y-x)
                pgcd
           else
             do setX (x-y)
                pgcd
```

Toutes les fonctions d'accès aux états des variables s'appuient sur une monade.

Foncteurs et monades

▷ Principe de construction de la monade de l'exemple

étape 1 : définissons une notion d'**action** :

```
data Action e i = ACT ( e -> (e, i) )
```

Une action est une fonction qui, quand elle est activée à partir d'un état, transforme cet état et produit une information utile.

étape 2 : indiquons que le type des actions satisfait les propriétés d'une monade :

```
instance Monad Action where
    (ACT p) >>= k  =  ACT(\e0 -> let (e1, a) = p e0
                                     (ACT q) = k a
                                 in q e1 )
    return a       =  ACT( \e -> (e, a) )
```

étape 3 : précisons la forme des états gérés dans le problème :

```
type Etat = (Int, Int)
```

étape 4 : décrivons comment chaque action agit sur l'état courant et quelle information elle produit :

```
getX = ACT( \(x, y) -> ( (x, y), x) )
getY = ACT( \(x, y) -> ( (x, y), y) )

setX v = ACT( \(x, y) -> ( (v, y), () ) )
setY v = ACT( \(x, y) -> ( (x, v), () ) )
```

étape 5 : définissons la fonction calculant le PGCD :

```
calculePGCD x y = snd (apply pgcd (x, y) )
                  where apply (ACT p) e = p e
```


Foncteurs et monades

◇ Autres applications des monades

- mécanisme de gestion d'exceptions
- analyse syntaxique d'une grammaire ambigue
- gestion de mécanisme de type retour-arrière → un démonstrateur de type Prolog

Pratique du langage Haskell

◇ Les compilateurs

Trois compilateurs Haskell (Yale, Glasgow et Chalmers Univ.), tous du domaine public (FTP anonyme).

Hugs : un interpréteur / compilateur qui n'inclut pas toutes les spécifications du langage, mais largement suffisant pour programmer ; très peu gourmand en espace disque (typiquement quelques Mo sur un PC).

◇ En pratique

http://haskell.org

Bibliographie

◇ Sur les types

- L. Cardelli & P. Wegner : On understanding Types, Data Abstraction and Polymorphism, ACM Computer Surveys, Vol 17, No 4, p469-522, December 1985

◇ Sur les langages fonctionnels

- P. Wadler & al. : Introduction to Functional Programming, Prentice Hall, 1988
- P. Hudak : Conception, Evolution and Application of Functional Programming Languages, ACM Computer Surveys, Vol 21, No 3, p359-411, 1989
- S.L. Peyton-Jones : Mise en œuvre des langages fonctionnels, Masson, 1991 (édition originale : Prentice Hall, 1987)

◇ Sur le langage `Haskell`

- Special issue on the functional programming language Haskell, SIGPLAN Notices, Vol 27, No 5, May 1992
- plus tous les documents accessibles sur le site Web...