

# Une introduction à Haskell

Emmanuel Beffara

29 septembre 2003

# Haskell ?

“**Haskell** is a general-purpose purely functional programming language”

- ▶ du nom de Haskell B. Curry ...

# Haskell ?

“Haskell is a general-purpose purely **functional** programming language”

- ▶ du nom de Haskell B. Curry ...
- ▶ on sait ce que veut dire fonctionnel

# Haskell ?

“Haskell is a general-purpose **purely functional** programming language”

- ▶ du nom de Haskell B. Curry ...
- ▶ on sait ce que veut dire fonctionnel
- ▶ “pur” = sans effets de bord (je reviendrai là-dessus), condition nécessaire au raisonnement équationnel sur un programme

# Haskell ?

“Haskell is a **general-purpose** purely functional  
**programming language**”

- ▶ du nom de Haskell B. Curry ...
- ▶ on sait ce que veut dire fonctionnel
- ▶ “pur” = sans effets de bord (je reviendrai là-dessus), condition nécessaire au raisonnement équationnel sur un programme
- ▶ (de plus en plus) adapté à la programmation réelle

# Caractéristiques générales

Aspects généraux:

- ▶ Sémantique non stricte (évaluation paresseuse)

# Caractéristiques générales

Aspects généraux:

- ▶ Sémantique non stricte (évaluation paresseuse)
- ▶ Typage expressif (classes de types, etc.)

# Caractéristiques générales

Aspects généraux:

- ▶ Sémantique non stricte (évaluation paresseuse)
- ▶ Typage expressif (classes de types, etc.)

Quelques principes fondateurs:

- ▶ Il y en a ...

# Caractéristiques générales

Aspects généraux:

- ▶ Sémantique non stricte (évaluation paresseuse)
- ▶ Typage expressif (classes de types, etc.)

Quelques principes fondateurs:

- ▶ Ne pas sacrifier un concept au profit de la performance
- ▶ Inventer des notions compliquées plutôt que renoncer à la pureté

# Un peu de syntaxe

Quelques conventions lexicales:

- ▶ Une majuscule initiale marque un type ou un constructeur:  
*Int*, *Just* 3 ...
- ▶ Une minuscule initiale marque une variable: *x*, *putChar* ...

# Un peu de syntaxe

Quelques conventions lexicales:

- ▶ Une majuscule initiale marque un type ou un constructeur:  
*Int*, *Just* 3 ...
- ▶ Une minuscule initiale marque une variable: *x*, *putChar* ...
- ▶ Chaque nom peut être précédé d'un préfixe désignant un nom de module: *Array.IArray*, *List.length* ...

# Un peu de syntaxe

Quelques conventions lexicales:

- ▶ Une majuscule initiale marque un type ou un constructeur:  
*Int*, *Just* 3 ...
- ▶ Une minuscule initiale marque une variable: *x*, *putChar* ...
- ▶ Chaque nom peut être précédé d'un préfixe désignant un nom de module: *Array.IArray*, *List.length* ...
- ▶ Les opérateurs infixes sont des suites de symboles non-alphanumériques: *\**, *++*, *>>=* ...
- ▶ Les commentaires ont deux formes possibles:

```
foo x = bar x + 2      -- commentaire en fin de ligne  
iter x = iter (x - 1) {- délimité -} + 32
```

# Déclaration de valeurs

- Les définitions se font de façon très déclarative:

*answer* = 42

*twice* *n* = 2 \* *n*

*something* *x y* = *x* + *y* \* (*x* - 1)

- Toutes les définitions d'un module sont implicitement récursives:

*length* [] = 0

*length* (\_ : *tail*) = 1 + *length* *tail*

- Tout cela reste vrai pour des valeurs non fonctionnelles:

*evens* = *map twice ints*

*ints* = 0 : *map succ ints*

*bottom* = *bottom*

► Définitions locales avec **where**:

```
evens = filter even ints  
  where even 0 = True  
        even 1 = False  
        even n = even (n - 2)
```

► Ou avec **let**:

```
odds = let odd 0 = False  
        odd 1 = True  
        odd n = odd (n - 2)  
  in filter odd ints
```

# Pattern-matching

- Pattern-matching avec gardes:

$$\begin{array}{l} \text{fibonacci } n \mid n \leq 1 = 1 \\ \quad \mid \text{True} = \text{fibonacci } (n - 1) + \text{fibonacci } (n - 2) \end{array}$$

# Pattern-matching

- Pattern-matching avec gardes:

$$\begin{aligned} \text{fibonacci } n \mid n \leq 1 &= 1 \\ \mid \text{True} &= \text{fibonacci } (n - 1) + \text{fibonacci } (n - 2) \end{aligned}$$

- Sur plusieurs arguments:

$$\text{somme } [] \_ = []$$

$$\text{somme } \_ [] = []$$

$$\text{somme } (h1 : t1) (h2 : t2) = (h1 + h2) : \text{somme } t1 \ t2$$

# Pattern-matching

- Pattern-matching avec gardes:

$$\begin{aligned} \text{fibonacci } n \mid n \leq 1 &= 1 \\ \mid \text{True} &= \text{fibonacci } (n - 1) + \text{fibonacci } (n - 2) \end{aligned}$$

- Sur plusieurs arguments:

$$\begin{aligned} \text{somme } [] \_ &= [] \\ \text{somme } \_ [] &= [] \\ \text{somme } (h1 : t1) (h2 : t2) &= (h1 + h2) : \text{somme } t1 \ t2 \end{aligned}$$

- Avec définitions locales:

$$\begin{aligned} \text{machin } x \ y \mid x > z &= y + 1 \\ \mid y \leq z &= x * y \\ \mid \text{otherwise} &= z / (x - y) \\ \text{where } z &= (x + y) / 2 \end{aligned}$$

# Types prédéfinis

Types de base:

- ▶ Bornés:  $()$ , *Bool*, *Int*, *Char*
- ▶ Non bornés: *Integer* ...

Constructeurs:

- ▶ listes:  $[a]$
- ▶ tuples:  $(a, b, c)$
- ▶ fonctions:  $a \rightarrow b$
- ▶ et aussi *Maybe a*, *Either a b* ...

# Déclaration de types

- Synonymes:

```
type String = [Char]
```

- Sommes simples:

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

- Types construits:

```
data Tree a b = Leaf b  
               | Node a (Tree a b) (Tree a b)
```

définit un type avec deux paramètres et deux constructeurs.

L'application partielle est autorisée:

```
type SimpleTree = Tree ()
```

définit un type *SimpleTree* à un paramètre.

# Prototypage

- Il est possible de déclarer explicitement le type d'une expression:

$$succ :: Integer \rightarrow Integer$$

$$succ\ n = n + 1$$

- Un prototype est une déclaration comme une autre:

$$rev = do\_rev\ []$$

$$\mathbf{where}\ do\_rev :: [a] \rightarrow [a] \rightarrow [a]$$

$$do\_rev\ acc\ [] = acc$$

$$do\_rev\ acc\ (head : tail) = do\_rev\ (head : acc)\ tail$$

Le typage explicite n'est pas obligatoire, mais il permet de repérer les erreurs et de spécifier des types plus précis que ceux inférés.

## Layout

L'espacement en début de ligne sert à marquer des listes de déclarations, par exemple

$$\begin{aligned} z &= f\ a + f\ b \textbf{ where } a = 5 \\ &\quad b = 4 \\ &\quad f\ x = x + 1 \end{aligned}$$

est équivalent à

$$z = f\ a + f\ b \textbf{ where } \{ a = 5; b = 4; f\ x = x + 1 \}$$

et à

$$\begin{aligned} z &= f\ a + f\ b \textbf{ where } a = 5; b = 4 \\ &\quad f\ x = x + 1 \end{aligned}$$

# Opérateurs binaires

- ▶ Chaque opérateur binaire peut avoir une priorité donnée:

**`infixr 6 +`** -- déclare `+` associatif à droite de priorité 6

- ▶ Chaque opérateur peut être appliqué partiellement:

- ▶ `(/)` est la fonction de division
- ▶ `(3/)` est équivalent à  $\lambda x \rightarrow 3 / x$
- ▶ `(/3)` est équivalent à  $\lambda x \rightarrow x / 3$

- ▶ Les fonctions peuvent être utilisées de façon infix:

`lst 'union' [1, 2, 3]` est équivalent à `union lst [1, 2, 3]`

# Fonctions et applications

- Les fonctions anonymes s'écrivent

$\lambda x y \rightarrow \text{truc}$       -- ça s'écrit `\x y -> truc`

- L'opérateur de composition se note avec un point:

$f \circ g \circ h$       -- ça s'écrit `f . g . h`

- \$ est l'opérateur d'application:

$f1 \$ f2 \$ f3 \$ f4 \$ f5 x = f1 (f2 (f3 (f4 (f5 x))))$

l'intérêt: \$ a une très basse priorité.

# Listes et compréhensions

- ▶ Syntaxe des listes:
  - ▶ liste vide: `[]`
  - ▶ construction: *head* : *tail*
  - ▶ ou alors: `[1, 2, 3, 4]`
- ▶ Concaténation: *lst1* ++ *lst2*
- ▶ Les filtrages et transformations bénéficient de sucre syntaxique:

$$\begin{aligned} [x + 3 \mid x \leftarrow lst1 ++ lst2] \\ = map (\lambda x \rightarrow x + 3) (lst1 ++ lst2) \end{aligned}$$

$$\begin{aligned} [x / 2 \mid x \leftarrow ints, x > 5] \\ = map (/2) \$ filter (>5) ints \end{aligned}$$

# Combinateurs

Dans le folklore Haskell, on aime programmer en combinant des opérateurs très génériques.

$$\textit{sum\_list} = \textit{foldl} (+) 0$$
$$\textit{combinedLength} = \textit{foldr} ((+) \circ \textit{length}) 0$$

On aime bien aussi le code très compact:

```
qsort [] = []  
qsort (x : xs) = qsort lower ++ [x] ++ qsort upper  
  where (lower, upper) = partition (<x) xs
```

# Types et sortes

- Formellement, l'algèbre des types Haskell se définit par

|                        |                |                                                 |
|------------------------|----------------|-------------------------------------------------|
| $\tau ::= C$           | constante      | $Bool, Int, Maybe, Either, (\rightarrow) \dots$ |
| $\alpha$               | variable       | $a, b \dots$                                    |
| $\forall \alpha. \tau$ | quantification | $forall\ a. \tau$                               |
| $\tau \tau$            | application    |                                                 |

En particulier  $a \rightarrow b$  est synonyme de  $(\rightarrow) a b$ .

- Chaque constante ou variable de type a une sorte:

$Int :: *$

$Maybe :: * \rightarrow *$

$[] :: * \rightarrow *$       -- les listes

$(\rightarrow) :: * \rightarrow * \rightarrow *$       -- le type des fonctions

$(,,,) :: * \rightarrow * \rightarrow * \rightarrow * \rightarrow *$       -- les quadruplets ...

- ▶ Les types sont des termes bien sortés sur ce langage.  
Définissons par exemple un type d'arbres générique:

$$\begin{aligned} \text{data } \textit{GenericTree} \ b \ v = & \textit{Leaf} \ v \\ & | \textit{Node} \ (b \ (\textit{GenericTree} \ b \ v)) \end{aligned}$$

définit *GenericTree* de sorte  $(* \rightarrow *) \rightarrow * \rightarrow *$

- ▶ On en dérive un type d'arbres d'arité quelconque:

$$\textit{RoseTree} :: * \rightarrow *$$
$$\text{type } \textit{RoseTree} = \textit{GenericTree} \ []$$

- ▶ Et aussi des arbres binaires:

$$\text{data } \textit{Pair} \ a = \textit{Pair} \ a \ a$$
$$\textit{bin} :: \textit{GenericTree} \ \textit{Pair} \ \textit{Int}$$
$$\textit{bin} = \textit{Node} \ (\textit{Pair} \ (\textit{Leaf} \ 1) \ (\textit{Node} \ (\textit{Pair} \ \textit{bin} \ (\textit{Leaf} \ 2))))$$

## Le problème de la surcharge

La surcharge d'opérateurs est un problème bien connu. Si on ne veut pas toucher au typage, il y a deux approches possibles:

- ▶ Définir un opérateur différent pour chaque type
  - ▶ Pour le calcul: `+` pour *int*, `+.`  pour *float* ...
  - ▶ Pour le formatage: *string\_of\_int*, *string\_of\_float* ...

Problème: le code devient verbeux et moins réutilisable.

## Le problème de la surcharge

La surcharge d'opérateurs est un problème bien connu. Si on ne veut pas toucher au typage, il y a deux approches possibles:

- ▶ Définir un opérateur différent pour chaque type
  - ▶ Pour le calcul: `+` pour *int*, `+.`  pour *float* ...
  - ▶ Pour le formatage: *string\_of\_int*, *string\_of\_float* ...

Problème: le code devient verbeux et moins réutilisable.

- ▶ Définir des opérateurs valables pour **tous** les types:
  - ▶ `(=)` : `'a -> 'a -> bool`, égalité structurelle sur les objets
  - ▶ `compare` : `'a -> 'a -> int`, comparaison générique, avec des conventions imposées par le langage.

Problème: c'est limité à une poignée de fonctions built-in, l'approche n'est pas généralisable.

# Classes de types

- Une classe de types est un prédicat sur les types, qui identifie une propriété particulière. Par exemple, les types totalement ordonnés:

**data** *Ordering* = *LT* | *EQ* | *GT*

**class** *Ord* *a* **where**

*compare* :: *a* → *a* → *Ordering*

signifie que les types de la classe *Ord* sont munis d'un opérateur binaire *compare*.

# Classes de types

- ▶ Une classe de types est un prédicat sur les types, qui identifie une propriété particulière. Par exemple, les types totalement ordonnés:

```
data Ordering = LT | EQ | GT
```

```
class Ord a where
```

```
    compare :: a → a → Ordering
```

signifie que les types de la classe *Ord* sont munis d'un opérateur binaire *compare*.

- ▶ Une déclaration d'instance définit l'implémentation d'une classe pour un type particulier:

```
instance Ord Bool where
```

```
    compare x y =
```

```
        if x == y then EQ else if y then GT else LT
```

## Contraintes de typage

- Après une telle définition, le type des fonctions utilisant *compare* exprime que les arguments doivent être d'un type comparable:

$$\begin{aligned} (<) &:: \text{Ord } a \Rightarrow a \rightarrow a \rightarrow \text{Bool} \\ x < y &= \text{compare } x \ y == \text{LT} \end{aligned}$$

## Contraintes de typage

- ▶ Après une telle définition, le type des fonctions utilisant *compare* exprime que les arguments doivent être d'un type comparable:

$$(<) :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Bool$$
$$x < y = compare\ x\ y == LT$$

- ▶ Toute conjonction de contraintes est autorisée:

$$sort\_show :: (Ord\ a, Show\ a) \Rightarrow [a] \rightarrow String$$
$$sort\_show = foldl\ write\ "" \circ sort$$
$$\textbf{where}\ write\ ""\ obj = show\ obj$$
$$write\ str\ obj = str \mathrel{++} ", " \mathrel{++} show\ obj$$

## Contraintes sur les instances

Une déclaration d'instance peut être polymorphe, par exemple on déclare que les listes se comparent selon l'ordre lexicographique:

```
instance Ord a  $\Rightarrow$  Ord [a] where  
  compare [] [] = EQ  
  compare _ [] = GT  
  compare [] _ = LT  
  compare (x : xs) (y : ys)  
    | comp == EQ = compare xs ys  
    | otherwise = comp  
  where comp = compare x y
```

## Contraintes de classe

On peut aussi déclarer qu'être instance d'une classe nécessite d'être instance d'une autre. Par exemple, voici une classe des types bornés:

```
class Ord a  $\Rightarrow$  Bounded a where  
    maxBound :: a  
    minBound :: a
```

C'est une sorte d'héritage: les objets de la classe *Bounded* fournissent toutes les méthodes de la classe *Ord*.

## Quelques classes standard

► Égalité et ordre:

*Eq a*      -- définit  $x == y$

*Ord a*      -- ajoute *compare*, ( $<$ ), ( $\leq$ ) ...

*Bounded a*      -- ajoute *minBound* et *maxBound*

► Types numériques:

*Num a*      -- donne un structure d'anneau ( $+$ ,  $-$ ,  $*$  ...)

*Fractional a*      -- ajoute quotient et inverse

*Floating a*      -- ajoute les fonctions usuelles (*exp*, *cos*, ...)

Un effet amusant:  $42$  est de type  $\text{Num } a \Rightarrow a$ .

► D'autres classes intéressantes:

*Show a*      -- fournit  $\text{show} :: a \rightarrow \text{String}$

*Read a*      -- fournit de quoi décoder une chaîne

*Enum a*      -- identifie les types indexables par les entiers

## Dérivation d'instances

Pour certaines classes standard, une implémentation peut être dérivée automatiquement de la définition du type:

```
data Color = Red | Green | Blue deriving Enum
```

définit un encodage et un décodage naturel de *Color* dans *Int*.

```
data Term a = Const a  
           | Var String  
           | App (Term a) (Term a)  
deriving (Eq, Show)
```

produit automatiquement des instances de la forme

```
instance Eq a  $\Rightarrow$  Eq (Term a) where...  
instance Show a  $\Rightarrow$  Show (Term a) where...
```

## Au-delà des types de base

- Les classes considérées jusque là concernent des types de sorte  $*$ , mais le formalisme s'étend à toutes les sortes.

## Au-delà des types de base

- ▶ Les classes considérées jusque là concernent des types de sorte  $*$ , mais le formalisme s'étend à toutes les sortes.
- ▶ Par exemple, on peut définir la classe des conteneurs:

**class** *Set* *s* **where**

*empty* :: *s* *a*

*singleton* :: *a*  $\rightarrow$  *s* *a*

*union* :: *s* *a*  $\rightarrow$  *s* *a*  $\rightarrow$  *s* *a*

*elem* :: *a*  $\rightarrow$  *s* *a*  $\rightarrow$  *Bool*

*map* :: (*a*  $\rightarrow$  *b*)  $\rightarrow$  *s* *a*  $\rightarrow$  *s* *b*

*size* :: *s* *a*  $\rightarrow$  *Int*

# Foncteurs

Un foncteur  $f$  envoie chaque objet  $a$  sur un objet  $f\ a$  et les morphismes de  $a \rightarrow b$  vers des morphismes de  $f\ a \rightarrow f\ b$ :

```
class Functor f where  
  fmap :: (a → b) → f a → f b
```

Pour définir vraiment un foncteur, il faudrait en plus imposer les équations

$$\begin{aligned} \text{fmap } id &== id \\ \text{fmap } (f \circ g) &== \text{fmap } f \circ \text{fmap } g \end{aligned}$$

En fait il n'y a pas moyen de vérifier ce genre de chose par typage, donc on suppose simplement que c'est le cas pour chaque instance.

Le type des listes est un foncteur:

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

C'est aussi le cas des fonctions d'un ensemble donné:

```
instance Functor (( $\rightarrow$ ) a) where  
    fmap f g = f  $\circ$  g
```

## D'autres classes ...

De même, on peut définir un produit cartésien:

```
class Product p where  
    times :: a → b → p a b  
    fst :: p a b → a  
    snd :: p a b → b
```

Et aussi un coproduit:

```
class Coproduct p where  
    left :: a → p a b  
    right :: b → p a b  
    switch :: (a → c) → (b → c) → p a b → c
```

## Lecture d'une chaîne

On suppose qu'on a des fonctions

$$\begin{aligned} readInt &:: String \rightarrow (Int, String) \\ skipWhite &:: String \rightarrow ((), String) \end{aligned}$$

et on veut lire trois entiers et les additionner:

$$\begin{aligned} read3\ s = & \mathbf{let} \ (x, s1) = readInt\ s \\ & \quad (-, s2) = skipWhite\ s1 \\ & \quad \dots \\ & \quad (z, s5) = readInt\ s4 \\ & \mathbf{in} \ (x + y + z, s5) \end{aligned}$$

C'est laid.

## Lecture d'une chaîne

Chaque fonction de lecture a la même forme:

**type** *Reader a = String*  $\rightarrow$  (*a*, *String*)

On introduit l'opérateur suivant:

*bind* :: *Reader a*  $\rightarrow$  (*a*  $\rightarrow$  *Reader b*)  $\rightarrow$  *Reader b*

*bind* *f g s* = **let** (*x*, *s'*) = *f s* **in** *g x s'*

*return* :: *a*  $\rightarrow$  *Reader a*

*return* *x s* = (*x*, *s*)

## Lecture d'une chaîne

On suppose qu'on a des fonctions

$$\begin{aligned} readInt &:: String \rightarrow (Int, String) \\ skipWhite &:: String \rightarrow ((), String) \end{aligned}$$

et on veut lire trois entiers et les additionner:

$$\begin{aligned} read3\ s = & \mathbf{let} \ (x, s1) = readInt\ s \\ & \quad (-, s2) = skipWhite\ s1 \\ & \quad \dots \\ & \quad (z, s5) = readInt\ s4 \\ & \mathbf{in} \ (x + y + z, s5) \end{aligned}$$

## Lecture d'une chaîne

On suppose qu'on a des fonctions

$$\text{readInt} :: \text{Reader Int}$$
$$\text{skipWhite} :: \text{Reader ()}$$

et on veut lire trois entiers et les additionner:

$$\begin{aligned} \text{read3 } s = & \text{let } (x, s1) = \text{readInt } s \\ & (-, s2) = \text{skipWhite } s1 \\ & \dots \\ & (z, s5) = \text{readInt } s4 \\ & \text{in } (x + y + z, s5) \end{aligned}$$

## Lecture d'une chaîne

On suppose qu'on a des fonctions

$$\begin{aligned} readInt &:: Reader\ Int \\ skipWhite &:: Reader\ () \end{aligned}$$

et on veut lire trois entiers et les additionner:

$$\begin{aligned} read3\ s = & bind\ readInt\ (\lambda x \rightarrow \\ & bind\ skipWhite\ (\lambda\_ \rightarrow \\ & \dots \\ & bind\ readInt\ (\lambda z \rightarrow \\ & return\ (x + y + z)))) \end{aligned}$$

On utilise *bind* pour le passage de chaîne.

## Lecture d'une chaîne

On suppose qu'on a des fonctions

$$\begin{aligned} readInt &:: Reader\ Int \\ skipWhite &:: Reader\ () \end{aligned}$$

et on veut lire trois entiers et les additionner:

$$\begin{aligned} read3\ s &= readInt\ 'bind'\ \lambda x \rightarrow \\ &\quad skipWhite\ 'bind'\ \lambda\_ \rightarrow \\ &\quad \dots \\ &\quad readInt\ 'bind'\ \lambda z \rightarrow \\ &\quad return\ (x + y + z) \end{aligned}$$

On écrit *bind* comme un opérateur infixe.

## Lecture d'une chaîne

On suppose qu'on a des fonctions

$$\begin{aligned} readInt &:: Reader\ Int \\ skipWhite &:: Reader\ () \end{aligned}$$

et on veut lire trois entiers et les additionner:

$$\begin{aligned} read3 = & \mathbf{do}\ x \leftarrow readInt \\ & \_ \leftarrow skipWhite \\ & \dots \\ & z \leftarrow readInt \\ & \mathbf{return}\ (x + y + z) \end{aligned}$$

On ajoute du sucre syntaxique sur *bind*.

## Lecture d'une chaîne

On suppose qu'on a des fonctions

$$\begin{aligned} readInt &:: Reader\ Int \\ skipWhite &:: Reader\ () \end{aligned}$$

et on veut lire trois entiers et les additionner:

$$\begin{aligned} read3 = &\mathbf{do}\ x \leftarrow readInt \\ &skipWhite \\ &\dots \\ &z \leftarrow readInt \\ &return\ (x + y + z) \end{aligned}$$

On ajoute du sucre syntaxique sur *bind*.

## C'était quoi ?

- Regardons de près le type de ces opérateurs:

$$\text{bind} :: \text{Reader } a \rightarrow (a \rightarrow \text{Reader } b) \rightarrow \text{Reader } b$$
$$\text{return} :: a \rightarrow \text{Reader } a$$

- Que se passe-t-il si on applique *bind* à l'identité ?

## C'était quoi ?

- Regardons de près le type de ces opérateurs:

$$\text{bind} :: \text{Reader } a \rightarrow (a \rightarrow \text{Reader } b) \rightarrow \text{Reader } b$$
$$\text{return} :: a \rightarrow \text{Reader } a$$

- Que se passe-t-il si on applique *bind* à l'identité ?

$$\text{join} :: \text{Reader } (\text{Reader } a) \rightarrow \text{Reader } a$$
$$\text{join } x = x \text{ 'bind' } \text{id}$$

## C'était quoi ?

- Regardons de près le type de ces opérateurs:

$$\begin{aligned} bind &:: Reader\ a \rightarrow (a \rightarrow Reader\ b) \rightarrow Reader\ b \\ return &:: a \rightarrow Reader\ a \end{aligned}$$

- Que se passe-t-il si on applique *bind* à l'identité ?

$$\begin{aligned} join &:: Reader\ (Reader\ a) \rightarrow Reader\ a \\ join\ x &= x\ 'bind'\ id \end{aligned}$$

- Remarquons que *Reader* est un foncteur:

$$\begin{aligned} rmap &:: (a \rightarrow b) \rightarrow Reader\ a \rightarrow Reader\ b \\ rmap\ f\ read &= read\ 'bind'\ (return \circ f) \end{aligned}$$

## C'était quoi ?

- ▶ Regardons de près le type de ces opérateurs:

$$\begin{aligned} bind &:: Reader\ a \rightarrow (a \rightarrow Reader\ b) \rightarrow Reader\ b \\ return &:: a \rightarrow Reader\ a \end{aligned}$$

- ▶ Que se passe-t-il si on applique *bind* à l'identité ?

$$\begin{aligned} join &:: Reader\ (Reader\ a) \rightarrow Reader\ a \\ join\ x &= x\ 'bind'\ id \end{aligned}$$

- ▶ Remarquons que *Reader* est un foncteur:

$$\begin{aligned} rmap &:: (a \rightarrow b) \rightarrow Reader\ a \rightarrow Reader\ b \\ rmap\ f\ read &= read\ 'bind'\ (return \circ f) \end{aligned}$$

$(Reader, join, return)$  est une monade !

# Monades

Pour un catégoricien, une monade est un triplet  $(M, \mu, \eta)$  où  $M$  est un foncteur et où  $\mu$  et  $\eta$  sont des transformations naturelles telles que pour tout objet  $A$ ,

$$\mu_A : M(M(A)) \rightarrow M(A) \quad \text{et} \quad \eta_A : A \rightarrow M(A)$$

sont des morphismes tels que

$$\begin{aligned}\mu_A \circ \eta_{M(A)} &= \mu_A \circ M(\eta_A) = \text{id}_{M(A)} \\ \mu_A \circ M(\mu_A) &= \mu_A \circ \mu_{M(A)}\end{aligned}$$

On en fait une classe de types en posant

**class** *Monad* *m* **where**

$(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$       -- map + contraction  
 $return :: a \rightarrow m\ a$       -- injection canonique

On suppose les identités suivantes:

$(f \gg= return) = f$       neutralité à droite

$(return\ x \gg= f) = f\ x$       neutralité à gauche

$((f \gg= g) \gg= h) = (f \gg= \lambda x \rightarrow (g\ x \gg= h))$       associativité

On retrouve  $\mu$  et le fait que  $m$  est un foncteur en posant

$join = (\gg=id)$

$fmap\ f = (\gg=return \circ f)$

## Opérateurs généraux

Un certain nombre d'opérateurs se définissent pour toute monade:

$$\text{liftM} :: \text{Monad } m \Rightarrow (a \rightarrow b) \rightarrow (m\ a \rightarrow m\ b)$$
$$\text{ap} :: \text{Monad } m \Rightarrow m\ (a \rightarrow b) \rightarrow (m\ a \rightarrow m\ b)$$

Et aussi toutes les opérations sur les listes:

$$\text{sequence} :: \text{Monad } m \Rightarrow [m\ a] \rightarrow m\ [a]$$
$$\text{mapM} :: \text{Monad } m \Rightarrow (a \rightarrow m\ b) \rightarrow [a] \rightarrow m\ [b]$$

et ainsi de suite.

## Monades d'état

Le principe se généralise:

```
data State s a = ST (s → (a, s))

instance Monad (State s) where
  (ST f) >>= g = ST fg
  where fg s = g' s' where (x, s') = f s; ST g' = g x
  return x = ST (λs → (x, s))
```

On obtient une monade pour représenter un état de type  $a$ . Pour lire et modifier l'état, on peut définir deux opérateurs primitifs:

```
getState :: State a a
getState = ST (λs → (s, s))

setState :: a → State a ()
setState s = ST (λs → ((), s))
```

## Effets de bord

Un effet de bord est une modification de l'état du monde, donc une action à effet de bord peut se voir comme un fonction de

$$RealWorld \rightarrow (a, RealWorld)$$

C'est-à-dire qu'on peut encapsuler tout effet de bord dans une monade:

`type IO = State RealWorld`

Pas de problème de pureté: l'évaluation d'un objet de type `IO a` se fait sans effet de bord. Seul l'environnement d'exécution interprète un tel objet.

## Exemple: entrées et sorties

Le plus simple est la lecture d'un fichier:

$$readFile :: FilePath \rightarrow IO String$$

Donc `readFile "toto"` est une action qui renvoie tout le contenu du fichier `toto`, lu de façon paresseuse.

Pour lire et écrire sur le terminal, rien de plus simple:

$$putChar :: Char \rightarrow IO ()$$
$$getChar :: IO Char$$

Par besoin d'un faux argument `()`,  
et ce n'est pas une question paresse.

## La monade des listes

Sur le type des listes, on a deux opérateurs intéressants:

$$\begin{aligned} ([]) &:: a \rightarrow [a] && \text{-- singleton} \\ \text{concat} &:: [[a]] \rightarrow [a] && \text{-- concaténation} \end{aligned}$$

Ces opérateurs définissent `[]` comme une monade.

À quoi ressemble la composition ?

$$\begin{aligned} \text{bind} &:: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b] \\ \text{bind } lst \ f &= \text{concat } (\text{map } f \ lst) \end{aligned}$$

En particulier:

$$\begin{aligned} \text{map } f \ lst &= \text{bind } lst \ (\lambda x \rightarrow [f \ x]) \\ \text{filter } p \ lst &= \text{bind } lst \ (\lambda x \rightarrow \text{if } p \ x \ \text{then } [x] \ \text{else } []) \end{aligned}$$

# Comprendre la compréhension

Posons

$$\begin{aligned} \text{assert} &:: \text{Bool} \rightarrow [()] \\ \text{assert } \text{True} &= [()] \\ \text{assert } \text{False} &= [] \end{aligned}$$

Alors on peut écrire

$$\begin{aligned} \text{toto} &= \mathbf{do} \ x \leftarrow \text{lst} \\ &\quad \text{assert } (x > 6) \\ &\quad \text{return } (x / 3) \end{aligned}$$

C'est équivalent à

$$\text{toto} = [x / 3 \mid x \leftarrow \text{lst}, x > 6]$$

## Non-déterminisme et backtracking

Dans les listes, deux opérateurs sont fondamentaux:  $\text{++}$  et  $[]$ , composition et élément neutre du monoïde des listes.

Ces opérateurs se traduisent en combinateurs sur les monades:

```
class Monad m  $\Rightarrow$  MonadPlus m where  
    mzero :: m a  
    mplus :: m a  $\rightarrow$  m a  $\rightarrow$  m a      -- on va le noter  $\oplus$ 
```

Sur des ensembles, on interprète cela comme du non-déterminisme:

```
mzero = []      -- aucun résultat possible  
x 'mplus' y = x ++ y    -- les résultats de x et ceux de y
```

## Un exemple ?

Un exemple sans intérêt cher aux pratiquants de Prolog:

```
type Personne
pere :: Personne → Personne
mere :: Personne → Personne
```

Les parents:

```
parents :: Personne → [Personne]
parents x = return (pere x) ⊕ return (mere x)
```

Les aïeux:

```
aieux :: Personne → [Personne]
aieux x = return x ⊕ ascendents
  where ascendents = do y ← parents x
           aieux y
```

## Composition de monades

Si on a de l'état et du non-déterminisme, on peut déduire une monade qui implémente les deux ...

```
data NonDet m a = ND (m [a])

instance Monad m  $\Rightarrow$  Monad (NonDet m) where
    return = ND  $\circ$  return  $\circ$  return
    (ND x)  $\gg=$  g = ND $ do lst  $\leftarrow$  x
                        lst'  $\leftarrow$  mapM (unND  $\circ$  g) lst
                        return (concat lst')
    where unND (ND x) = x
```

## Pour aller plus loin

Les amateurs de sensations fortes aimeront les sujets suivants:

- ▶ Classes de types multi-paramètres ...
  - ▶ Pour définir des relations entre types
  - ▶ Dépendances fonctionnelles ...
- ▶ Plus fort que les monades: les *arrows*
  - ▶ Functional Reactive Programmming
  - ▶ Circuits ...
- ▶ Questions d'implémentation
  - ▶ Classes de types et programmation objet
  - ▶ Évaluation optimiste ...
- ▶ Template Haskell