Le langage Haskell

Un langage fonctionnel, pur et paresseux...
et qui a de la classe.

Benoît Fraikin

Département d'informatique

6 juillet 2011

UNIVERSITÉ DE
SHERBROOKE

Plan de la séance

1. La petite histoire d'Haskell
   - Les origines
   - Le développement
2. Présentation d'Haskell
   - Les caractéristiques d'Haskell
   - Pureté et évaluation paresseuse
   - Conséquence de ces choix
3. Les classes de types et les monades
   - Le concept de classe de types
   - Le concept de Monade
   - Un exemple
4. Conclusion
   - Haskell
   - Développements à venir
   - Petite bibliographie

UNIVERSITÉ DE SHERBROOKE

Un programme, une personne

Nos ancêtres (1950–1960)

- Fortran (John W. Backus, 1953)
- Lisp (John McCarthy, 1958) $\Longleftarrow$ λ-calcul, GC
- Algol (ETH Zurich, 1958)
- Cobol (Grace Hopper *et al.*, 1959)

Une première étape

vers des langages plus complexes (1960–1980)

- APL (Kenneth E. Iverson, à partir de 1958, 1960)
- PL/I (début des années 1960)
- Simula (Ole-Johan Dahl et K. Nygaard, 1960's) $\Longleftarrow$ Classe
- Smalltalk (Alan Kay *et al.*, dans les années 1970)
- C (Dennie Ritchie et Ken Thompson, 1969–1973)
- Prolog (A. Colmerauer et P. Roussel, 1972) $\Longleftarrow$ Style déclaratif
- ML (Robert Milner, 1978) $\Longleftarrow$ Hindler-Milner theorème

Historique

de 1987 à 1998 (Paul Hudak et al., 2007)

- Hudak et Peyton Jones en 87
- inspiré par Miranda (propriétaire)
- Haskell v1.0 en 1990 (v1.1 en 91 et v1.2 en 92)
- Haskell v1.3 en 1996 (v1.4 en 97)
- Haskell 98 (99 et 02)

Quelques acteurs principaux

- Paul Hudak (Yale U.)
- Simon Peyton Jones (Microsoft Research, Glasgow U.)
- Richard Bird (Oxford U.)
- Simon Marlow (Microsoft Research)
- Philip Wadler (Edinburgh U.)
- Graham Hutton (Notthingham U.)
- Eric Meijer (Microsoft Research)
- John Hugues (Chalmers U.)
- Ralph Lämmer (U. Koblenz-Landau, ex. Microsoft Research)
- Don Stewart (Galois Inc., Standard Chartered Bank)

Citations

Simon Peyton Jones

Avoid success at all costs

Tony Hoare

I fear that Haskell is doomed to succeed.

Survol

En bref et de manière incomplète

- Langage fonctionnel, pure, non-strict (John Hugues, 1989)
- Types algébriques abstraits et classes de types (*type classes*)
- Curryfication, notation infixe et préfixée, sections
- Filtrage d'expression (*pattern matching*)
- Création de liste par compréhension
- Exploite la mise en page du code (*layout*)
- Encourage un style « équationnel » et déclaratif
- Permet la programmation littérale

Système de typage

A common mistake people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools. — Douglas Adams

Typage fort

- Haskell est typé statiquement
- Les types sont inférés
- Reste flexible
- Addition des genres (*kind*) ⇒ types d'ordre supérieur
- Une des plus grande force d'Haskell
- Mais pas de module !

Les types

Types internes

- **Bool**, **Char**, **Integer**
- **Int**, **Double**
- **String** (liste de **Char**)

Créateurs primitifs

- les couples
- les listes

Exemple

```
indent = 4          :: Integer
space = 1.5         :: Double

b = ('a',1)         :: (Char,Integer)
key = fst b         :: Char

liste = [1,2,3,4]   :: [Integer]
un = head liste     :: Integer

name = "Benoit"     :: String
```

Construction de liste par compréhension

Liste par compréhension

- $[\ e \mid \phi(e)\ ]$
- Ressemble à la définition d'un ensemble par compréhension
- Utilise $\leftarrow$ dans un sens proche de $\in$

Exemple

```
fonctions = [(λ x → x + n) | n ← [1,3..9], 15 `mod` n > 1 ]
couples = [(y, ord y) | y ← "abcdefghij"]
valeurs = [ f o | f ← fonctions, (_,o) ← couples ]
```

UNIVERSITÉ DE SHERBROOKE

Les types

Création de type

- les renommages
- les énumérations
- les unions
- les enregistrements

Exemple

```
type Coord = (Integer,Integer)
data Couleur = Vert | Rouge | Bleu
data Pixel = P Coord Couleur
data Forme = Rectangle Coord Coord
           | Cercle Coord Coord
           | Triangle Coord Coord Coord
```

UNIVERSITÉ DE SHERBROOKE

Paradigme fonctionnel

Grandes lignes

- Fonctions : données du premier ordre
- Récursivité omniprésente
- Curryfication automatique
- Définition dans un style « équationnel » recommandé
- Composition et section

Exemples de définition de fonctions

fonction **length**

```
length :: [a] → Int
length [] = 0
length (x:xs) = 1 + length xs
```

fonction **map**

```
map :: (a → b) → [a] → [b]
map g [] = []
map g (x:xs) = g x : map g xs
```

UNIVERSITÉ DE SHERBROOKE

Exemples de garde

fonction filter

```haskell
filter :: (a → Bool) → [a] → [a]
filter test []    = []
filter test (x:xs) | test x     = x : filter test xs
filter test (x:xs) | otherwise = filter test xs
```

fonction *fibonacci*

```haskell
fibonacci :: Integer → Integer
fibonacci n | n < 2 = n
fibonacci n | otherwise = fibonacci (n-2) + fibonacci (n-1)
```

UNIVERSITÉ DE SHERBROOKE

Exemples d'opérateurs et de section

Sections

```haskell
double :: Integer → Integer
double = (×2)
```

Opérateur

```haskell
(++) :: String → String → String
--
s1 = "abc" ++ "def"        -- s1 = "abcdef"
s2 = (++) "abc" "def"      -- s2 = "abcdef"
l = double `map` [1,2,3]   -- l = [2,4,6]
```

UNIVERSITÉ DE SHERBROOKE

Exemples d'application et de composition

Application

```haskell
main :: (Tree String) → IO ()
main t = putStrLn $ drawTree t
```

Composition

```haskell
concatMap :: (a → [b]) → [a] → [b]
concatMap = concat ∘ map
-- ou plutôt
concatMap f =  foldr ((++) ∘ f) []
```

Pureté

Absence d'effets de bord

- Choix conceptuel initial
- Implique la transparence référentielle
- Raison du choix de l'évaluation non-stricte

Transparence référentielle

Définition

propriété d'une expression d'être remplaçable par sa valeur sans changer le programme.

. . . comme en mathématique

- Propriété des expressions mathématiques
- Un effet de bord apparaît dans les types en Haskell
- Permet la détection des expressions transparentes
- Permet l'optimisation possible sans intervention
- Optimisation parfois désirée : inline en C

Principe de l'évaluation paresseuse

lazy evaluation

- Méthode d'évaluation non-strict
- Stratégie d'implémentation ...
- ...de la sémantique d'appel par nécessité (*call by need*)
- $\simeq$ appel par nom (*call by name*) avec « mémoization »
- Allègent certaines contraintes (`foldr`)
- Permet les listes infinies et l'auto-référence
- ...et donc les définitions par des systèmes d'équations

Les listes infinies

Exemples

```haskell
-- liste d'un caractère
queDesAs = 'a' : queDesAs

-- Entiers naturels
naturels = [0..]           -- ne pas afficher
pairs = map (×2) naturels
impairs = map (+1) pairs

fibonacci = (map fib [0..]) !! -- version efficace
  where fib 0 = 0
        fib 1 = 1
        fib n = fibonacci (n-2) + fibonacci (n-1)
```

UNIVERSITÉ DE SHERBROOKE

Fonctions auto-référencées

Exemples

```
-- l'opérateur de point fixe
fix :: ( a → a ) → a
fix f = f (fix f)

const :: a → b → a
const x y = x

fix $ const "hello" -- retourne "hello"

-- la génération des nombres premiers
premiers = 2 : crible [3,5..]
  where crible (n:ns) = n : crible [ m | m ← ns, mod m n /= 0 ]

premiers !! 1000
```

UNIVERSITÉ DE SHERBROOKE

Impact sur . . .

le temps d'exécution

- Difficile à profiler (Samson et Peyton Jones)
- Coût réduit par la mémorisation de la paresse

la mémoire utilisée

- Encore plus difficile à profiler (Runciman and Wakeling)
- La paresse nécessite des *thunks*
- L'ordre des évaluations dépend du contexte
- Fuites de mémoire possible !
- Introduction de constructions strictes (Haskell 1.3)

Impact sur . . .

l'écriture d'un programme

- Style plus proche des mathématiques
- L'ordre des définitions n'a pas d'importance
- Une forme non récursive terminale peut être voulue
- $\Rightarrow$ utilisation de `foldr` et `foldl'` plutôt que `foldl`

le déverminage

- Complique terriblement le traçage
- Tracer $\Rightarrow$ des effets de bord
- GHC tente de le résoudre (dévermineur intégré)

Conséquence importante de la pureté

Le problème des entrées/sorties

Comment conserver la pureté et gérer les entrées/sorties d'un programme ?

Trois solutions

- Les flots (*Stream*)

```
type Behaviour = [Response] → [Request]
```

- Les continuations

```
type FailCont = IOError → Behaviour
type StrCont = StrCont → Behaviour
```

- Les monades

UNIVERSITÉ DE SHERBROOKE

Pourquoi des classes de type ?

Le polymorphisme

- Utiliser le même nom pour le même comportement
- Trois grandes catégories
  - polymorphisme d'inclusion (*OO*)
  - polymorphisme *ad-hoc* (surcharge)
  - polymorphisme paramétrique (« vrai » polymorphisme)
- Les classes de type permettent le polymorphisme *ad-hoc*

UNIVERSITÉ DE SHERBROOKE

Polymorphismes

Le polymorphisme paramétrique

```
head :: [a] → a
head [1,2,3] -- 1
head "abc"   -- 'a'

map :: (a→b) → [a] → [b]
map (+1) [1,2,3]   -- [2,3,4]
map toUpper "abc" -- "ABC"
```

Le polymorphisme *ad-hoc*

```
1 > 2       -- ok
'b' > 'a'   -- ok
h1 = Map.fromList [('a',2)('b',1)]
h2 = Map.fromList [('a',1)('b',2)]
h1 > h2     -- ???

(>) :: (Ord a) ⇒ a → a → Bool
```

- classe de type = interface
- doit être instanciée
- ...en partie seulement

UNIVERSITÉ DE SHERBROOKE

Exemple `Ord`

La classe des types ordonnés

```
class  (Eq a) ⇒ Ord a  where
    compare              :: a → a → Ordering
    (<), (≤), (>), (≥)   :: a → a → Bool
    max, min             :: a → a → a
```

Instance pour `Map`

```
instance (Ord k, Ord v) ⇒ Ord (Map k v) where
        compare m1 m2 = compare (toAscList m1) (toAscList m2)
```

UNIVERSITÉ DE SHERBROOKE

Les classes de types

La base

- `Show`
- `Read`
- `Eq`
- `Ord`
- `Enum`
- `Bounded`

et l'ajout des catégories

- `Functor`
- `Applicative`
- `Traversable`
- `Foldable`
- `Monoid`
- `Monad`
- `Arrow`
- `Category`

UNIVERSITÉ DE SHERBROOKE

Pourquoi des monades ?

> *The "pure way" is like a honeymoon : it's all beautiful, really. [...] The "monadic way" is more like 5 years after the honeymoon. [...] It's just more an age of pragmatism.*

Un programme n'est pas une boîte close

- Comment gérer les entrées/sorties en Haskell ?
- La pureté limite les possibilités ⇒ bris nécessaire
- Le typage doit refléter l'effet de bord
- Initialement : deux solutions
- . . . abandonnées au profit des monades

Les monades

Histoire

- Utilisées pour décrire un programme par Moggi (1989)
- Introduites pour exprimer un comportement par Wadler (1995)
- Issues de la théorie des catégories
- Rapidement adoptées
- Le concept s'est déclinée en de multiples saveurs
- Développement « à l'impérative » sans perdre la pureté
- Ont suivis d'autres emprunts à la théorie des catégories

Qu'est-ce qu'une monade ?

Définition (Saunders Mac Lane, 1969)

Une monade est juste un monoïde dans la catégorie des endofoncteurs.

Une représentation d'un calcul

- Capture l'idée de « calcul à faire »
- . . . avec un résultat pur
- . . . avec une mise en chaîne des opérations

En Haskell

Définition de la classe Monade

```
class Monad m where
    return :: a → m a
    (>>=)  :: m a → ( a → m b ) → m b
```

Une représentation d'un calcul

- La fonction `return`
    - crée une monade *pure*
- La fonction `>>=`
    - extrait la valeur d'une monade
    - la transmet à une fonction qui en crée une autre

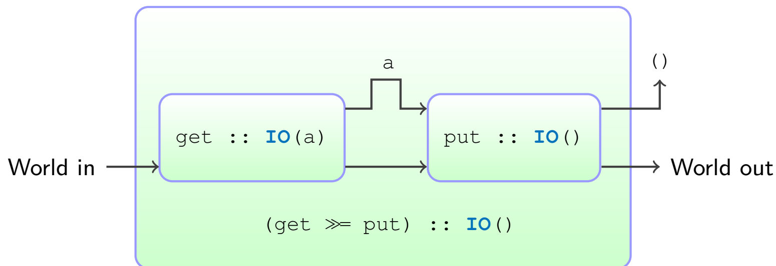
Gérer les entrées et les sorties

La monade `IO`

- Un résultat pur est une lecture : monade `IO` avec contenu
- Une mise en chaîne est la transmission du contenu
- ...et son utilisation dans la suite
- Un affichage est une monade `IO` sans contenu

UNIVERSITÉ DE SHERBROOKE

Illustration



UNIVERSITÉ DE SHERBROOKE

Exemple I

Lecture, modification et écriture

```
getChar :: IO (Char)
ord :: Char → Int
show :: Show a ⇒ a → String
putStrLn :: String → IO ()

main :: IO ()
main = getChar >>= printOrd
  where printOrd :: Char → IO()
        printOrd = putStrLn ∘ show ∘ ord
```

UNIVERSITÉ DE SHERBROOKE

Exemple II

Lecture, test, modification et écriture

```
getChar :: IO (Char)
isLetter :: Char → Bool
ord :: Char → Int
show :: Show a ⇒ a → String
putStrLn :: String → IO ()

main :: IO ()
main = getChar >>= (λ c → if isLetter c
                           then printOrd c
                           else putStrLn "erreur")
  where printOrd :: Char → IO()
        printOrd = putStrLn ∘ show ∘ ord
```

Exemple III

Lorsque la notation devient lourde

```
main :: IO ()
main = getChar >>= (λ c →
       if isDigit c
       then getChar >>= (λ c′ → if isDigit c′
                                 then printInt (charsToInt c c′)
                                 else putStrLn "erreur")
       else putStrLn "erreur")
  where charsToInt :: Char → Char → Int
        charsToInt c c′ = let i = ord c
                              i′ = ord c′
                          in i + i′
        printInt :: Int → IO ()
        printInt = putStrLn ∘ show
```

UNIVERSITÉ DE SHERBROOKE

La do-notation I

Une écriture à l'impérative

```
main :: IO ()
main = do c ← getChar
          if isDigit c
          then do c' ← getChar
                  if isDigit c'
                  then printInt (charsToInt c c')
                  else putStrLn "erreur"
          else putStrLn "erreur"
  where charsToInt :: Char → Char → Int
        charsToInt c c' = let (i,i') = (ord c, ord c')
                          in i + i'

        printInt :: Int → IO ()
        printInt = putStrLn ∘ show
```

UNIVERSITÉ DE SHERBROOKE

La do-notation II

Amélioration

```
main :: IO ()
main = do c ← getChar
          unless (isDigit c) (fail "erreur")
          c' ← getChar
          unless (isDigit c') (fail "erreur")
          printInt (charsToInt c c')
  where
    charsToInt :: Char → Char → Int
    charsToInt c c' = let (i,i') = (ord c, ord c')
                      in i + i'

    printInt :: Int → IO ()
    printInt = putStrLn ∘ show
```

UNIVERSITÉ DE SHERBROOKE

La do-notation III

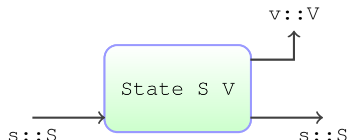
Version « plus » fonctionnelle

```
main :: IO ()
main = do c ← getChar
          putStrLn ""
          unless (isDigit c) (fail "erreur")
          c' ← getChar
          putStrLn ""
          unless (isDigit c') (fail "erreur")
          print $ charsToInt [c, c']
  where
    charsToInt = foldr ((+) ∘ ord) 0
```

Généralisation du concept

La monade `State`

- Gère un état implicite
- Monade `IO` où le *monde* est une valeur d'Haskell



UNIVERSITÉ DE SHERBROOKE

De très nombreuses monades

De nombreuses structures sont des monades

- calcul non déterministe `[a]`
- calcul dans un environnement `e -> a`
- calcul avec un journal `(a,w)`
- calcul pouvant échouer `Maybe a`
- calcul avec une alternative `Either e a`
- calcul par continuation `r -> a -> a`

Applications

Success stories

- Les bibliothèques de combinateurs
  - analyseur syntaxique (Hutton and Meijer, 1998)
  - producteur d'affichage agréable (Hugues, 1995)
- Les languages spécifiques à un domaine
  - Programmation réactive
  - Langage de conception de matériel
  - Musique informatique (Hudak, 1996)
- Les processeurs de langage naturel

Décorer un arbre

```
'a'
|
+- 'b'
|  |
|  +- 'a'
|  |
|  +- 'c'
|  |
|  `- 'a'
|
`- 'c'
   |
   +- 'd'
   |
   +- 'a'
   |
   `- 'c'
```

Exigence

- À partir d'un arbre
- Générer un arbre décoré
- Avec trois techniques, de gauche à droite
  - chaque noeud est décoré d'un numéro distinct, selon l'ordre du parcours
  - chaque noeud est décoré d'un numéro indiquant le nombre d'instance de sa valeur déjà décoré
  - chaque noeud est décoré d'un numéro unique selon sa valeur, selon l'ordre du parcours

UNIVERSITÉ DE SHERBROOKE

Les guirlandes et les boules

Information dans GHCi

```
GHCi > :info Tree
data Tree a = Node {rootLabel :: a, subForest :: Forest a}
instance Show a => Show (Tree a)
instance Traversable Tree

GHCi > :info Forest
type Forest a = [Tree a]

GHCi > :info Traversable
class (Functor t, Foldable t) => Traversable t where
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
```

Les techniques I

Décoration 1 et 2

```
numerote :: a → State Int (Int, a)
numerote x = do i ← get
                put (i+1)
                return (i,x)

compte :: Ord a ⇒ a → State (Map a Int) (Int, a)
compte x = do v ← gets (Map.lookup x)
              let n = (fromMaybe 0 v) + 1
              modify (Map.insert x n)
              return (n,x)
```

UNIVERSITÉ DE SHERBROOKE

Les techniques II

Décoration 3

```
etiquette :: Ord a ⇒ a → State (Int, (Map a Int)) (Int, a)
etiquette x = do env ← get
                 let table = snd env
                     top = fst env
                     v = Map.lookup x table
                     (n,top') = if isJust v
                                then (fromJust v,top)
                                else (top,top+1)
                 put (top', Map.insert x n table)
                 return (n,x)
```

Décorer un arbre

Application des décorations et affichage

```
decorer :: (a → State s b) → s → Tree a → Tree b
decorer d e t = let m_tree = mapM d t
                in evalState m_tree e

printTree :: Show a ⇒ Tree a → IO ()

GHCi > printTree $ decorer numerote 0 charTree
```

Résultat

```
(0,'a')                  (1,'a')                  (1,'a')
|                        |                        |
+- (1,'b')               +- (1,'b')               +- (2,'b')
|  |                     |  |                     |  |
|  +- (2,'a')            |  +- (2,'a')            |  +- (1,'a')
|  |                     |  |                     |  |
|  +- (3,'c')            |  +- (1,'c')            |  +- (3,'c')
|  |                     |  |                     |  |
|  `- (4,'a')            |  `- (3,'a')            |  `- (1,'a')
|                        |                        |
`- (5,'c')               `- (2,'c')               `- (3,'c')
   |                        |                        |
   +- (6,'d')               +- (1,'d')               +- (4,'d')
   |                        |                        |
   +- (7,'a')               +- (4,'a')               +- (1,'a')
   |                        |                        |
   `- (8,'c')               `- (3,'c')               `- (3,'c')
```

Apports d'Haskell I

Un système de type souple et puissant

- Difficile de compiler sans comprendre les types
- Souplesse due à
  - Inférence de type efficace
  - Polymorphisme
  - Classe de type
- Pas de typage au *runtime*
- $\Rightarrow$ meilleures performances

UNIVERSITÉ DE SHERBROOKE

Apports d'Haskell II

La pureté

- Implique la paresse pour l'instant

  > *The next Haskell will be strict, but still pure*
  > — *Simon Peyton Jones*

- Augmente la lisibilité
- Facilite la parallélisation du code (à améliorer)
- Peut nuire au performance (temps et espace)

Apports d'Haskell III

Les monades

- Modèle pratique et intéressant
- Initialement pour les É/S
- Permettent de nombreuses manipulations impératives
- Offrent une programmation élégante
- ... sans contourner le typage fort

Haskell 2010 et GHC 7

Nouveautés dans Haskell 2010 (déjà dans GHC 6.x)

- Interface pour les fonctions étrangères (*FFI*)
- Types hiérarchiques (`Control.Concurrent.Chan`)
- Garde

Nouveautés dans GHC 7

- Nouveaux contrôleurs d'E/S
- Intégration du générateur LLVM
- Amélioration de l'inférence de type
- Amélioration des performances (au *runtime*)

À résoudre

Besoin

- Module de première classe (à la ML)
- Enregistrement (*record*)
- Déverminage et profilage

Bibliographie pertinente I

Paul Hudak.

« Haskore music tutorial », 1996.

Paul Hudak, John Hughes, Simon Peyton Jones et Philip Wadler.

« A history of Haskell : being lazy with class », 2007.

John Hughes.

« Why Functional Programming Matters », 1989.

Bibliographie pertinente II

John Hughes.

« The design of a pretty-printing library », 1995.

Graham Hutton et Erik Meijer.

« Monadic parsing in Haskell », 1998.

Philip Wadler.

« Monads for functional programming », 1995.

Tous à vos programmes !

```
developer :: (Coffee a) ⇒ [a] → IO Software
```