Chapitre 2 : Les bases de Haskell, partie 1

Carole Porrier
carole.porrier@univ-paris13.fr

Département d'informatique
IUT de Villetaneuse

21 mars 2024
Paradigmes de développement universels

Table des matières

Table des matières

Les types en Haskell

- Haskell est un langage à typage **fort et statique**:
  - Chaque variable a un type **précis** qui ne change pas;
  - Plusieurs contraintes lorsqu'on **combine** des types.
- **Avantage:**
  - Beaucoup d'erreurs sont détectées à la **compilation**;
  - Programmes plus **sûrs**;
  - Programmes **efficaces**, car il n'est pas nécessaire de **contrôler** les types à l'**exécution**.

Types de base

- `Bool`: valeurs booléennes (`True` ou `False`);
- `Char`: caractères;
- `String`: chaîne de caractères;
- `Int`: entiers bornés (32 bits ou 64 bits);
- `Integer`: entiers non bornés;
- `Float`, `Double`: nombres à virgule flottante.

Listes et tuples

- **Listes**:
  - `[a]`: liste dont les éléments sont de type `a`;
  - Regroupent des éléments **homogènes**;
  - La **longueur** est variable.
- **Tuples**:
  - `(a,b)`: couple dont le premier élément et de type `a` et le deuxième de type `b`;
  - `(a,b,c)`: triplet dont le premier élément et de type `a`, le deuxième de type `b` et le troisième de type `c`, etc.
  - Regroupent des éléments **hétérogènes**.
  - Longueur **fixe**.

Types de fonctions

- Les fonctions ont elles-mêmes un **type**;
- Autrement dit, la **signature** d'une fonction nous indique **son type**.

```
not :: Bool -> Bool    -- Negation logique
and :: [Bool] -> Bool  -- 'et' logique d'une liste
sort :: [a] -> [a]     -- Liste triee
length :: [a] -> Int   -- Longueur d'une liste
-- Combine deux listes
zip :: [a] -> [b] -> [(a, b)]
-- Applique une fonction à une liste
map :: (a -> b) -> [a] -> [b]
-- Conserve les éléments vérifiant une condition
filter :: (a -> Bool) -> [a] -> [a]
-- Additionne deux valeurs numériques
(+) :: Num a => a -> a -> a
-- La relation "strictement plus petit"
(<) :: Ord a => a -> a -> Bool
```

Types énumératifs

- Parfois, les **valeurs** possibles sont **finies**;
- Pour une meilleure **lisibilité**, on peut utiliser un type **énumératif**:

```
-- Directions
data Direction = Est | Nord | Ouest | Sud

-- Jour de la semaine
data Jour = Dimanche | Lundi | Mardi | Mercredi | Jeudi |
     Vendredi | Samedi
-- Version plus courte
data Jour = DIM | LUN | MAR | MER | JEU | VEN | SAM

-- Le type Bool est un type énumératif
data Bool = False | True
```

- Le nom du type et les valeurs possibles doivent **tous** commencer par une **majuscule**.

Synonymes (`type`)

- Une autre commande qui améliore la **lisibilité** est `type`;
- Elle permet de déclarer un type **synonyme** d'un autre (comme `typedef` en C);
- Les deux types sont alors **complètement équivalents**.

```
-- Le type String est synonyme de [Char]
type String = [Char]
-- Chemin vers un fichier
type FilePath = [Char]
-- Des points
type Point2D = (Float,Float)
type Point3D = (Float,Float,Float)
type Vecteur2D = (Float,Float)
type Angle = Float

-- Les fonctions deviennent plus lisibles
translate :: Vecteur2D -> Point2D -> Point2D
rotate :: Angle -> Point2D -> Point2D
```

Table des matières

Contraintes de types

- Plusieurs fonctions n'ont du sens que si le type satisfait une **contrainte** particulière:
- Prenons par exemple la fonction

```
-- Indique si un élément se trouve dans une liste
elem :: a -> [a] -> Bool
```

- On doit pouvoir dire si deux éléments de type `a` sont **égaux**.
- La **classe** des éléments "égalisables" en Haskell est identifiée par `Eq`.
- C'est pourquoi la signature **complète** de `elem` est

```
elem :: Eq a => a -> [a] -> Bool
```

Classes prédéfinies

- Nous verrons plus tard comment **définir** nos propres classes;
- Plusieurs classes sont **fournies** par défaut:
  - `Eq` : supporte `==` et `/=`;
  - `Ord` : supporte `==`, `/=`, `<=`, `<`, `>=` et `>`;
  - `Show` : supporte `show` (similaire à **`toString`** en Java):
  - `Read` : supporte `read` (l'inverse de **`show`**)
  - `Enum` : supporte `succ` et `pred` (peut être **énuméré**)
  - `Bounded` : supporte `minBound` et `maxBound` (des valeurs minimale et maximale)

Les fonctions `show` et `read`

```
Prelude> show 4
"4"
Prelude> show "paradigme"
"\"paradigme\""
Prelude> show [1,2,3]
"[1,2,3]"
Prelude> show (1.0,2.0)
"(1.0,2.0)"
Prelude> read "4"
*** Exception: Prelude.read: no parse
Prelude> read "4" :: Int
4
Prelude> read "2.5" :: Float
2.5
```

Classes numériques

- Les valeurs **numériques** (nombres) sont aussi organisées en **classes**;
- `Num` supporte `(+)`, `(*)`, **`abs`**, `signum`, `fromInteger` et `(-)`.
- `Fractional` : supporte `(/)` (inclut donc tous les nombres **rationnels**);
- `Integral` : inclut les nombres **entiers** (dont `Int` et `Integer`);
- `Floating` : inclut les nombres **à virgules** (dont `Float` et `Double`).
- `Complex` : représente les nombres **complexes**.

La fonction `fromIntegral`

```
Prelude> :t fromIntegral
fromIntegral :: (Num b, Integral a) => a -> b
```

- La signature indique que la fonction prend un type **entier** `a` pour produire un type **numérique** `b`;
- Bref, elle rend une valeur **plus générale**;
- Cela facilite les opérations entre **entiers** et **flottants** par exemple.

```
Prelude> let a = [1,2,3]
Prelude> sum a / length a
<interactive>:38:7:
    No instance for (Fractional Int) arising from a use of '/'
    In the expression: sum a / length a
    In an equation for 'it': it = sum a / length a
Prelude> sum a / (fromIntegral (length a))
2.0
```

Dérivation de classe

- On peut définir un type **énumératif** à l'aide de `data`:

```
data Mois = Jan | Fev | Mar | Avr | Mai | Jun |
            Jui | Aou | Sep | Oct | Nov | Dec
    deriving (Eq, Ord, Enum, Show, Read, Bounded)
```

- Le mot réservé `deriving` permet de **dériver automatiquement** des classes comme `Eq`, `Ord`, `Enum`, `Show`, `Read`, `Bounded`, etc.

- Le comportement **par défaut** est celui auquel on s'attend:

```
*Main> show (succ Mar)
"Avr"
*Main> read "Nov" :: Mois
Nov
*Main> Mai < Oct
True
*Main> Jun == Jui
False
*Main> (minBound :: Mois, maxBound :: Mois)
(Jan,Dec)
```

Table des matières

Syntaxe

- Pour **appliquer** (appeler) une fonction, on utilise l'**espace**, sans parenthèse, même s'il y a **plusieurs arguments**:

```
Prelude> max 3 5
5
```

- On **imbrique** des fonctions à l'aide de **parenthèses**

```
Prelude> max (min 3 5) (min 8 4)
4
```

- L'utilisation des parenthèses pour **imbriquer** des structures est un reproche souvent fait aux **langages fonctionnels** (Lisp, par exemple).

L'opérateur $

- Un opérateur **pratique** pour alléger l'emploi de parenthèses est le dollar (`$`);
- Il joue le même rôle que l'**espace**, mais en inversant l'**associativité** de droite à gauche:

```
Prelude> length ("paradigme" ++ " de " ++ "programmation")
26
Prelude> length $ "paradigme" ++ " de " ++ "programmation"
26
```

- Peut être utilisé **plusieurs fois**:

```
Prelude> length (replicate 5 (max 'a' 'e'))
5
Prelude> length $ replicate 5 $ max 'a' 'e'
5
```

Fonction polymorphique

- Lorsque le type est une **variable**, on dit que la fonction est **polymorphique**;
- **Exemple:**

```
sort :: [a] -> [a]      -- Liste triee
length :: [a] -> Int    -- Longueur d'une liste
```

- Les modules `Prelude`, `Data.List`, etc. offrent presque exclusivement des fonctions **polymorphiques**.
- On peut donc remplacer `a`, `b` par n'importe quel type, comme `Bool`, `Int`, `[Char]`, etc.

Fonctions d'ordre supérieur

- Une fonction peut en utiliser **une autre**:

```
-- Applique une fonction à chaque élément
map :: (a -> b) -> [a] -> [b]
-- Retient les éléments vérifiant une condition
filter :: (a -> Bool) -> [a] -> [a]
```

- **Exemple**:

```
Prelude> :t succ
succ :: Enum a => a -> a
Prelude> :t odd
odd :: Integral a => a -> Bool

Prelude> map succ [3,5,8,9]   -- Remplace par l'entier suivant
[4,6,9,10]
Prelude> filter odd [3,5,8,9] -- Retient les nombres impairs
[3,5,9]
```

Application partielle

- On peut appliquer **partiellement** certaines fonctions;
- Cela permet de construire **facilement** de nouvelles fonctions **à la volée**.

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude> :t (3+)
(3+) :: Num a => a -> a
Prelude> :t (+3)
(+3) :: Num a => a -> a
Prelude> map (+3) [5,8,10,7]
[8,11,13,10]
Prelude> map (3+) [5,8,10,7]
[8,11,13,10]
Prelude> map ('a' `elem`) ["pomme", "fraise", "orange", "kiwi"]
[False,True,True,False]
Prelude> filter (>0) [4,-2,5,3,8,-1,0,7]
[4,5,3,8,7]
Prelude> filter (<0) [4,-2,5,3,8,-1,0,7]
[-2,-1]
```

Composition de fonctions (1/3)

- Une autre opération **fondamentale** est la **composition** de fonctions;
- **Rappel**: Soient

$$f : A \to B \quad \text{et} \quad g : B \to C$$

deux fonctions.

- On définit la **composition** de f et g par

$$\begin{array}{rccc} g \circ f & : & A & \to & C \\ & & a & \mapsto & g(f(a)) \end{array}$$

- En Haskell, on utilise l'opérateur **point** (`.`).

Composition de fonctions (2/3)

```
-- Type de length
*Main> :t length
length :: [a] -> Int
-- Type de (<=4)
*Main> :t (<=4)
(<=4) :: (Ord a, Num a) => a -> Bool

-- Pratique pour des requêtes à la volée
*Main> filter ((<=4) . length) ["un", "deux", "trois", "quatre",
     "cinq", "six", "sept"]
["un","deux","cinq","six","sept"]
```

Composition de fonctions (3/3)

```
-- Synonymes
type Point = (Float,Float)
type Vector = (Float,Float)
type Angle = Float

-- Translation
translate :: Vector -> Point -> Point
translate (a,b) (x,y) = (x + a, y + b)

-- Rotation autour de l'origine
rotate0 :: Angle -> Point -> Point
rotate0 a (x,y) = (x * cos a - y * sin a,
                   x * sin a + y * cos a)

-- Rotation autour d'un point quelconque
rotate :: Point -> Angle -> Point -> Point
rotate (x,y) a = translate (x,y) . rotate0 a . translate (-x,-y)
```

```
*Main> rotate (1,2) (pi/2) (3,4)
(-1.0,4.0)
```

Table des matières

Définir/implémenter une règle de correspondance

- Maintenant que nous savons comment lire la **signature** d'une fonction, nous sommes en mesure de l'**implémenter**!
- Il existe plusieurs **mécanismes** à notre disposition;
- Essentiellement, nous avons
  - Plusieurs façons de définir des **structures conditionnelles**;
  - La **récursivité** et l'**appel** à d'autres fonctions pour les **structures répétitives** (boucles).

Par filtrage

- Une technique très utilisée est le **filtrage** (*pattern matching*);
- Il s'agit de décrire la **forme** à vérifier pour avoir une correspondance:
- Le caractère de **soulignement** correspond à n'importe quel motif (*wildcard*).

```
non :: Bool -> Bool
non True  = False
non False = True

multiplie :: Int -> Int -> Int
multiplie 1 y = y
multiplie x 1 = x
multiplie 0 _ = 0
multiplie _ 0 = 0
multiplie x y = x * y
```

Filtrage et récursivité

- Le filtrage est souvent combiné à la **récursivité**;
- On distingue un ou plusieurs **cas de base**;
- Puis une ou plusieurs **parties récursives**.

```
longueur :: [a] -> Int              -- Longueur d'une liste
longueur []     = 0                 -- Cas de base
longueur (_:xs) = 1 + longueur xs   -- Partie récursive

somme :: Num a => [a] -> a  -- Somme d'une liste
somme []     = 0            -- Cas de base
somme (x:xs) = x + somme xs -- Partie récursive

estPalindrome :: [Char] -> Bool
estPalindrome []  = True    -- Premier cas de base
estPalindrome [_] = True    -- Deuxième cas de base
estPalindrome s   = head s == last s &&
    estPalindrome (init (tail s))
                            -- Partie récursive
```

À l'aide de gardes

- Lorsque le **filtrage** ne suffit pas, il est possible d'utiliser des **gardes**;
- Celles ci permettent de vérifier des **conditions** supplémentaires.
- Il est recommandé d'**aligner** les symboles = pour une meilleure lisibilité:

```
-- Retourne le signe d'un entier
signe :: Int -> Int
signe x | x < 0     = -1
        | x > 0     = 1
        | otherwise = 0

-- Vérifie si une chaîne est un palindrome
estPalindrome :: [Char] -> Bool
estPalindrome s
    | length s <= 1    = True
    | head s /= last s = False
    | otherwise        = estPalindrome (tail (init s))
```

Variables/fonctions auxiliaires

- Il est souvent pratique de définir des **variables/fonctions** auxiliaires;
- Permet de rendre le code **plus lisible**;
- Ou d'éviter des **calculs répétitifs**;
- La syntaxe `where` permet de déclarer une variable/fonction auxiliaire **à la fin**;
- La syntaxe `let`...`in`... permet de déclarer une variable/fonction auxiliaire **au début**.

La syntaxe where (1/2)

```haskell
-- Coefficients d'un polynome de degré 2
type PolyQuad = (Float,Float,Float)

-- Racines réelles d'un polynome de degré 2
racines :: PolyQuad -> [Float]
racines (a,b,c)
    | delta < 0  = []
    | delta == 0 = [-b / (2 * a)]
    | otherwise  = [(-b + sqrt delta) / (2 * a),
                    (-b - sqrt delta) / (2 * a)]
    where delta = b * b - 4 * a * c
```

La syntaxe where (2/2)

```haskell
data Mois = Jan | Fev | Mar | Avr | Mai | Jun |
            Jui | Aou | Sep | Oct | Nov | Dec
    deriving (Eq)
type Jour = Int
type Annee = Int

-- Indique si une date est valide
dateValide :: Annee -> Mois -> Jour -> Bool
dateValide a m j = j >= 1 && j <= nbJours a m
    where nbJours a m
            | m `elem` [Jan,Mar,Mai,Jui,Aou,Oct,Dec] = 31
            | m `elem` [Avr,Jun,Sep,Nov]             = 30
            | estBissextile a                        = 29
            | otherwise                              = 28
          estBissextile a = a `mod` 4 == 0 &&
            (a `mod` 100 /= 0 || a `mod` 400 == 0)
```

La syntaxe `let/in`

```haskell
data Mois = Jan | Fev | Mar | Avr | Mai | Jun |
            Jui | Aou | Sep | Oct | Nov | Dec
    deriving (Eq)
type Jour = Int
type Annee = Int

-- Indique si une date est valide
dateValide :: Annee -> Mois -> Jour -> Bool
dateValide a m j =
    let nbJours a m
            | m `elem` [Jan,Mar,Mai,Jui,Aou,Oct,Dec] = 31
            | m `elem` [Avr,Jun,Sep,Nov]             = 30
            | estBissextile a                        = 29
            | otherwise                              = 28
        estBissextile a = a `mod` 4 == 0 &&
            (a `mod` 100 /= 0 || a `mod` 400 == 0)
    in  j >= 1 && j <= nbJours a m
```

Les syntaxes `case` et `if-then-else`

- Il est aussi possible d'utiliser la syntaxe **`case`**;
- De même que la syntaxe **`if/then/else`**;

```
-- Le filtrage (pattern matching)
premier :: [a] -> a
premier []    = error "Liste vide!"
premier (x:_) = x
-- est un sucre syntaxique pour la structure "case"
premier :: [a] -> a
premier xs = case xs of []    -> error "Liste vide!"
                        (x:_) -> x

-- Structure conditionnelle "classique"
estPositif :: Int -> Bool
estPositif x = if x >= 0 then True else False
```

Lecture et crédit

- Cours d'**aujourd'hui** et **prochain cours**: sections 3, 4, 5 et 6 de Learn You a Haskell for Great Good!.

Cours adapté de celui d'Alexandre Blondin Massé (UQAM), avec son aimable autorisation.