

Programmation Fonctionnelle

Types et Classes

Luigi Santocanale
LIF, Aix-Marseille Université
Marseille, FRANCE

15 septembre 2017

Plan

Types élémentaires, listes et tuplets

Types des fonctions

Polymorphisme

Surcharge et classes

Prémisse

Haskell est un langage fortement typé, statiquement typé, avec mécanisme d'inférence de type.

Rappel :

	Typage forte	Typage statique	Inférence de type
C	Non	Oui	Non
Java	Oui	Oui	Non
Python	Oui	Non	Non
Haskell	Oui	Oui	Oui

Qu'est ce que un type ?

Un type est une collection de valeurs. ¹

Par exemple, en Haskell, le type élémentaire `Bool` contient les deux valeurs `False` et `True`.

Haskell est un langage fortement typé!!!

1. Vous vous rappelez la différence entre expressions et valeurs?

Erreurs de type

Si on applique une fonction à un ou plus arguments du mauvais type on obtient un erreur de type.

```
> 1 + False
ERROR - Cannot infer instance
*** Instance      : Num Bool
*** Expression   : 1 + False
```

1 est un nombre et False est une valeur logique, mais + demande deux nombres.

Les types de Haskell I

- Si l'évaluation d'une expression e produit une valeur de type t , on dit que e possède type t .
On écrit cela par

$$e :: t$$

Par exemple :

$$\text{False} :: \text{Bool} \quad \text{et} \quad \text{False} \ \&\& \ \text{True} :: \text{Bool}.$$

- Toute expression bien formée a un type, qui peut se calculer pendant la compilation – en utilisant un algorithme connu sous le nom d'*inférence de type*.
- Tous les erreurs de type sont détectés à la compilation ; ainsi, les programmes deviennent plus surs et performants (pas de vérifications de type pendant l'exécution).

Les types de Haskell II

- La commande `:type` calcule le type d'une expression, sans l'évaluer :

```
> not False
True
> :type not False
not False :: Bool
```

Plan

Types élémentaires, listes et tuplets

Types des fonctions

Polymorphisme

Surcharge et classes

Types élémentaires

Haskell possède un nombre de types élémentaires (ou basiques), incluant :

Bool	valeurs logiques
Char	simples caractères
Int	entiers à précision fixe
Integer	entiers à précision arbitraire
Float	nombres flottants

Le suivant

String	chaînes de caractères
--------	-----------------------

est un faux type élémentaire.

Le type des listes

- Une liste est une séquence de valeurs ayant tous même type :

```
[False, True, False] :: [Bool]
['a', 'b', 'c', 'd'] :: [Char]
```

- En général :

[t] est le type des listes dont tous les objets ont type t.

Remarque :

- Le type des listes ne porte pas d'information sur la longueur d'une liste :

```
[False, True] :: [Bool]
[False, True, False] :: [Bool]
```

- Il n'y a pas de restrictions sur le type des éléments d'une liste. Par exemple, on peut avoir des listes de listes :

```
[['a'], ['b', 'c']] :: [[Char]]
```

- En Haskell, les chaînes de caractères sont des listes de Char :

```
String == [Char] .
```

Remarque : types dérivés et expressions de type

- On construit des nouveaux types à partir de types donnés.
- Nous avons besoins d'opérateurs et expressions pour le types, à ne pas confondre avec les expressions usuelles, même si la syntaxe est souvent très proche :

[Bool]

Le type dont les elements appartiennent au type Bool

[False , True , True]

La liste composée par les trois Booléens False, True, True, qui est appartient donc au type Bool.

Tuplets

- Un tuple est une séquence de valeurs, possiblement de types différents :

```
(False, True) :: (Bool, Bool)
```

```
(False, 'a', True) :: (Bool, Char, Bool)
```

- En general :

(t_1, t_2, \dots, t_n) est le type des n -tuplets dont le i -ème composante a type t_i , pour $i = 1, \dots, n$.

Remarque :

- Le type d'un tuple est aussi sa taille :

```
(False, True) :: (Bool, Bool)
```

```
(False, True, False) :: (Bool, Bool, Bool)
```

- Le type de la composante ne fait pas l'objet de restrictions :

```
('a', (False, 'b')) :: (Char, (Bool, Char))
```

```
(True, ['a', 'b']) :: (Bool, [Char])
```

Plan

Types élémentaires, listes et tuplets

Types des fonctions

Polymorphisme

Surcharge et classes

Types des fonctions

- Chaque fonction a un type de valeurs d'entrée (domaine, type des arguments) et un type de valeurs de sortie (codomaine, type des valeurs calculés).

Par exemple :

```
not :: Bool -> Bool
isDigit :: Char -> Bool
```

- Une fonction est donc une correspondance entre valeurs d'un type et valeurs d'un autre type.
- En général :

$t1 \rightarrow t2$ est le type des fonctions qui envoient les valeurs de type $t1$ vers les valeurs de type $t2$.

Remarque :

- Le type de l'argument et du résultat ne font pas l'objet de restrictions.

Par exemple, fonctions avec arguments (ou résultats) multiples sont possibles en utilisant les listes ou les tuplets :

```
add :: (Int, Int) -> Int
add (x, y) = x+y
```

```
zeroto :: Int -> [Int]
zeroto n = [0..n]
```

Fonctions Curryfiées

Fonctions avec arguments multiples sont aussi possibles en retournant fonctions comme résultats.

```
add' :: Int -> (Int -> Int)
add' x y = x+y
```

*add' prend un entier x et retourne la fonction $add' x$.
A son tour, cette fonction prend un entier y et retourne la valeur de $x + y$.*

Remarque :

- `add` et `add'` produisent le même résultat final, mais :
 - ▶ `add` prend ses arguments au même temps,
 - ▶ `add'` prend ses arguments un par un.

```
add  :: (Int,Int) -> Int
add' :: Int  -> (Int -> Int)
```

- Fonctions qui prennent leurs arguments un après l'autre sont appelée *fonctions Curryfiées* (depuis M. Haskell Curry).

- Fonctions avec plus que deux arguments peuvent être aussi Curryfiées :

```
mult :: Int -> (Int -> (Int -> Int))  
mult x y z = x*y*z
```

mult prend un entier *x* et retourne
la fonction *mult x*;
mult x prend un entier *y* et retourne
la fonction *mult x y*;
mult x y prend un entier *z* et retourne
la valeur de *x*y*z*.

Pourquoi utiliser les fonctions Curryfiées ?

Les fonctions Curryfiées sont plus ductiles que les fonctions sur les tuplets : des fonctions utiles peuvent se construire par application partielle des arguments à une fonction Curryfiée.

Par exemple :

```
add' 1 :: Int -> Int
take 5 :: [Int] -> [Int]
drop 5 :: [Int] -> [Int]
```

Conventions avec la curryfication I

Pour éviter un excès de parenthèses quand on utilise les fonctions Curryfiées, deux conventions simples sont adoptées :

- la flèche `->` est associative à droite :

```
Int -> Int -> Int -> Int
```

signifie

```
Int -> (Int -> (Int -> Int)).
```

- (par conséquence) l'application de fonctions est associative à gauche :

```
mult x y z
```

signifie

```
((mult x) y) z
```

Sauf si les tuplets sont nécessaires, toutes les fonctions en Haskell sont normalement définie en forme Curryfiée.

Plan

Types élémentaires, listes et tuplets

Types des fonctions

Polymorphisme

Surcharge et classes

Fonctions polymorphes

Une fonction est polymorphe (de la langue grecque ancienne : *de plusieurs formes*) si son type contient une ou plus variables de type.

```
length :: [a] -> Int
```

pour tout type a , $length$ prend une liste de valeurs de type a et retourne un entier.

Nous pouvons aussi avoir des expressions polymorphes :

```
[] :: [a]
```

et des types polymorphes :

```
[a]
```

Fonctions polymorphes

Une fonction est polymorphe (de la langue grecque ancienne : *de plusieurs formes*) si son type contient une ou plus variables de type.

```
length :: [a] -> Int
```

pour tout type a, length prend une liste de valeurs de type a et retourne un entier.

Nous pouvons aussi avoir des expressions polymorphes :

```
[] :: [a]
```

et des types polymorphes :

```
[a]
```

Remarque :

- Une variable de type peut être instanciée à des types différents, dans des contextes différents :

```
> length [False, True]
```

```
2
```

```
> length [1, 2, 3, 4]
```

```
4
```

```
a = Bool
```

```
a = Int
```

- Les variables de type débutent par une minuscule, elles sont nommées d'habitude `a`, `b`, `c`, etc.
- Plusieurs fonctions définies dans `prelude.hs` sont polymorphes.

```
fst :: (a,b) -> a
head :: [a] -> a
take :: Int -> [a] -> [a]
zip :: [a] -> [b] -> [(a,b)]
id :: a -> a
```

Recapitulatif : expressions de types

On construit les expressions de types

- à partir des types élémentaires :

`Bool`, `Int`, `Integer`, `Char`, ...

- à partir des variables de type :

`a`, `b`, `c`, ...

- et à l'aide des operateurs de type suivants :

<code>[...]</code>	listes
<code>(..., ..., ...)</code>	tuplets
<code>... -> ...</code>	fonctions

Plan

Types élémentaires, listes et tuplets

Types des fonctions

Polymorphisme

Surcharge et classes

Fonctions surchargées

Une fonction polymorphe est dite surchargée si son type contient une *contrainte de classe*.

```
sum :: Num a => [a] -> a
```

sum prend une liste de valeurs de ce type et retourne un valeur de type a,

à condition que a soit un type numérique !!!

Remarque :

- Une variable de type soumise à une contrainte de classe peut s'instancier à tout type satisfaisant cette contrainte :

```
> sum [1,2,3]
6
> sum [1.1,2.2,3.3]
6.6
```

- ...mais pas à d'autres :

```
> sum ['a','b','c']
ERROR
```

```
a = Int
a = Float
Char n'est pas un type numerique
```

C'est quoi une classe

- Chaque classe est caractérisée par ses **méthodes**.
- Une classe peut se considérer comme une **collection de types**.
- Un type appartient à une classe si on a le droit d'utiliser les méthodes de la classe.

Remarque : la notion de classe en Haskell a peu à voir avec la notion de classe en programmation orienté objet.

Exemple : la classe *Num*

- Les types

Integer, Int, Float, Double, Word
appartiennent à la classe Num ;

- Les methodes de cette classe sont :

+, *, -, abs, signum, fromInteger ;

- Chaque méthode a son type polymorphe, mais restraint par la contrainte Num :

```
(+) :: Num a => a -> a -> a
...
abs :: Num a => a -> a
```

- Les contraintes de classes se propagent des méthodes aux expressions complexes :

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

Langage

Synonymes :

- un type a satisfait une contrainte de classe,
- le type a appartient à cette classe.

Exemple :

- `Int` est un type numérique,
- `Int` appartient à `Num`,
- `Num Int`

D'autres classes

- Haskell possède un nombre de classes, incluant :

Num types numériques

Eq types avec égalité

Ord types avec une relation d'ordre standard.

- Methodes, exemple :

```
(+) :: Num a => a -> a -> a
```

```
(==) :: Eq a => a -> a -> Bool
```

```
(<) :: Ord a => a -> a -> Bool
```

- Expressions, exemple :

```
listEqual :: Num a => a -> a -> a
```

```
listEqual [] [] = True
```

```
listEqual (x:xs) (y:ys) =
```

```
    x == y && listEqual xs ys
```

```
listEqual _ _ = False
```

- plus sur les classes à suivre (voir ch10). □

Conseils

- Quand on définit une nouvelle fonction en Haskell, c'est utile d'écrire son type ;
- A l'intérieur d'un script, c'est une bonne habitude d'écrire le type de la fonction avant sa définition ;
- Quand on écrit le type d'une fonction polymorphe qui utilise les nombres, l'égalité, et l'ordre, prenez garde d'inclure les contraintes de classe nécessaires.

Exercices

1. Quel est le type des valeurs suivants ?

```
['a', 'b', 'c']  
( 'a', 'b', 'c' )  
[(False, '0'), (True, '1')]  
([False, True], ['0', '1'])  
[tail, init, reverse]
```

2. Quel est le type des fonctions suivantes :

```
second xs = head (tail xs)  
swap (x,y) = (y,x)  
pair x y = (x,y)  
double x = x*2  
palindrome xs = reverse xs == xs  
twice f x = f (f x)
```

3. Vérifiez vos réponses en utilisant ghci.