Cycle ingénieur - 2ème année

Programmation fonctionnelle

Fonctions

2023-2024

Plan

Types fonctionnels

Curryfication

Opérateurs

Types fonctionnels

Toute fonction est un « citoyen de première classe »

(traduction littérale de *first-class citizen*)

Pierre angulaire de la programmation fonctionnelle

Signification

Une fonction a le même rang (ou citoyenneté*) que n'importe quel autre objet.*

Une fonction est donc une **expression typée**, i.e. elle peut être :

- utilisée comme une valeur ;
- passée en paramètre dans une autre fonction ;
- renvoyée comme résultat d'une autre fonction ;
- etc...

Fonctions

Type fonctionnel *T* → *R*

- *T* : type du paramètre en entrée
- *R* : type du résultat

Instantation : *paramètres* -> *corpsFonction*

Déclaration comme toute autre constante

- Déclaration globale ou locale (`let` ... `in` / `where`)
- Possibilité de déclarations locales à l'intérieur du corps

Cas des fonctions récursives

- Déclarations *croisées* possibles

Fonctions

Instantiation

```
\n -> n + 1
```

Déclaration

```
inc :: Int -> Int -- non obligatoire
inc = \n -> n + 1
```

NB: La déclaration du type est **fortement recommandée**.

Forme plus pratique :

```
inc :: Int -> Int
inc n = n + 1
```

Déclaration avec déclaration locale interne

```
inc :: Int -> Int
inc n = let next_n = n + 1 in next_n
```

ou

```
inc :: Int -> Int
inc n = next_n where next_n = n + 1
```

Fonctions

Déclaration récursive

```
fact :: Integer -> Integer
fact n =
  if n <= 0 then
    1
  else
    n * fact (n-1)
```

Déclaration récursive croisée

```
pair, impair :: Int -> Bool
pair   n = (n == 0) || impair (n - 1)
impair n = (n /= 0) &&   pair (n - 1)
```

Fonctions

Déclaration récursive terminale

```
fact  :: Integer -> Integer
fact n = fact' n 1
fact' :: Integer -> Integer -> Integer
fact' n acc =
  if n <= 0 then
    acc
  else
    fact' (n - 1) (n * acc)
```

On peut choisir de déclarer `fact'` *localement* :

```
fact :: Integer -> Integer
fact n = fact' n 1 where
  fact' n acc =
    if n <= 0 then
      acc
    else
      fact' (n - 1) (n * acc)
```

Fonctions : liste d'alternatives

Propriétés

- ≃ `switch` sur des expressions booléennes
- Éviter la lourdeur d'écriture de `if` … `then` … `else` imbriqués
- **Les règles d'exhaustivité et de séquentialité s'appliquent.**

Exemple

```haskell
-- Calcule les nombres de Fibonacci
fib :: Integer -> Integer
fib n
  | n <  0    = -1
  | n <= 1    = n
  | otherwise = fib (n - 2) + fib (n - 1)
```

Expression	Résultat
`fib (-1)`	`-1`
`fib 1`	`1`
`fib 6`	`8`

Curryfication

Curryfication

Et avec plusieurs paramètres ? plusieurs résultats ?

- Plusieurs paramètres : encapsulation des paramètres dans un tuple

```
sum (x, y) = x + y
```

- Plusieurs résultats : encapsulation des résultats dans un tuple

```
quorem (a, b) = (a `div` b, a `mod` b)
```

Donc on peut toujours de se ramener à un seul paramètre et un seul résultat.

Principe général de la curryfication

Transformer une fonction à « plusieurs » variables en une séquence de fonctions à une seule variable :

$$(T_1, T_2) \to R \rightsquigarrow T_1 \to T_2 \to R = T_1 \to (T_2 \to R)$$

$$(T_1, T_2, T_3) \to R \rightsquigarrow T_1 \to T_2 \to T_3 \to R = T_1 \to (T_2 \to (T_3 \to R))$$

Exemple : `sum (x, y) = x + y`

$\Longrightarrow$ `sum x y = x + y` i.e. `(sum x) y = x + y`

Curryfication : exemple

pow : calcul de x^n

- Deux paramètres : `n` (`Integer`) et `x` (`Double`)
  ↪ un seul paramètre de type `Integer * Double` : `(n, x)`
- Résultat : x^n (`Double`)

$\Longrightarrow$ Type de `pow` : `(Integer, Double) -> Double`

```
pow (n, x) = x^n
```

Forme curryfiée de pow

- Paramètre : `n`
- Résultat : fonction de type `Double -> Double` qui à `x` associe x^n :

$\Longrightarrow$ Type de `pow` : `Integer -> (Double -> Double)`

```
pow n = \x -> x^n
pow = \n -> (\x -> x^n)
```

Curryfication : exemples

```
pow :: Integer -> Double -> Double
pow n x = x^n
pow n = \x -> x^n
pow = \n -> (\x -> x^n)
```

Les trois expressions sont équivalentes.

Expression	Résultat
`pow 2`	`\x -> x^2`
`(pow 2) 4.0`	`16.0`
`pow 2 4.0`	`16.0`
`let cube = pow 3 in cube 5.0`	`125.0`

Opérateurs

Les opérateurs usuels sont des fonctions

Les opérateurs (comme la plupart des fonctions prédéfinies) sont curryifiés.

Utilisation d'un opérateur comme constante fonctionnelle

(i.e. passage d'une écriture infixe à une écriture préfixe)

- Opérateurs binaires : entourer l'opérateur avec des parenthèses :

  `(+)`, `(/)`, `(&&)`, `(==)`, `(<)`, ...
- Possibilité d'ajouter un des paramètres : `(+ 1)`, `(1 /)`, `(>= x)`, ...
- Opérateur unaire `-` : `negate`

Expression	Résultat
`(+) 3 5`	`8`
`negate 3`	`-3`

Expression	Résultat
`let inc = (+ 1) in inc 3`	`4`
`(>= 3) 5`	`True`
`(3 >=) 5`	`False`

Définition d'opérateurs (écriture infixe)

À partir d'une fonction à deux paramètres

(i.e. passage d'une écriture préfixe à une écriture infixe)

- Entourer la fonction d'« anti-quotes » : `` `div` ``, `` `mod` ``

```
implies :: Bool -> Bool -> Bool
implies p q = (not p) || q
```

Expression	Résultat
`False `implies` True`	`True`
`True `implies` False`	`False`

Un opérateur binaire est une fonction à deux paramètres

On le déclare donc comme tel :

```
(<=>) :: Bool -> Bool -> Bool
p <=> q = (p && q) || ((not p) && (not q))
```

Expression	Résultat
`False <=> False`	`True`
`True <=> False`	`False`

Opérations sur les fonctions (module `Data.Function`)

Fonctions particulières

- `id = (\x -> x)` : fonction identité
- `const c = (\x -> c)` : fonction constante égale à c
- `flip f = (\x y -> f y x)` : fonction avec ses deux paramètres inversés

Opérateurs `($) :: (a -> b) -> a -> b` et `(&) :: a -> (a -> b) -> b`

- `f $ x = x & f = f x`

Expression	Résultat
`(+1) $ (+1) $ 2`	4
`3 & (+1)`	4

Opérateur de composition `(.) :: (b -> c) -> (a -> b) -> (a -> c)`

- `(.) f g x = f (g x)` donc `f . g` ≡ f ∘ g

Expression	Résultat
`(+1) . (+1) $ 2`	4