Programmation Fonctionnelle
Fonctions récursives

Luigi Santocanale
LIF, Aix-Marseille Université
Marseille, FRANCE

12 septembre 2013

Plan

Plan

Introduction

Plusieurs fonctions se définissent à partir d'autres :

```
factorial :: Int -> Int
factorial n = product [1..n]
```

> *`factorial` mappe un entier `n` vers le produit des entiers entre `1` et `n`.*

Les expressions sont évaluées par le processus qui consiste en l'application successive d'une fonction à ses arguments.

Par exemple :

```
factorial 4 = product [1..4]
            = product [1,2,3,4]
            = 1*2*3*4
            = 24
```

Fonctions récursives

En Haskell, une fonction peut se définir à partir de soi-même :

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

> *`factorial` mappe `0` vers `1`, et tout autre entier positif vers le produit de lui même et le factoriel de son prédécesseur.*

Une telle fonction est dite *récursive*.

Par exemple :

```
factorial 3
= 3 * factorial 2
= 3 * (2 * factorial 1)
= 3 * (2 * (1 * factorial 0))
= 3 * (2 * (1 * 1))
= 3 * (2 * 1)
= 3 * 2
= 6
```

Remarque(s) :

- La fonction récursive diverge sur les entiers < 0
  car le *cas base* de la récursion n'est jamais atteint :

```
> factorial (-1)
Error: Control stack overflow
```

- L'égalité

```
factorial 0 = 1
```

  est approprié car 1 est l'identité de la multiplication :

$$1 * x = x = x * 1 .$$

Pourquoi la récursion est utile ?

- Des fonctions, comme `factorial`, sont plus simples à définir à partir d'autres.
- Par contre, certaines fonctions peuvent se définir aisément en terme de soi-mêmes.
- Les propriétés des fonctions définies par récursion peuvent se prouver en utilisant le simple mais puissant méthode de l'*induction mathématique*.

En plus

- La récursion est l'outil principal pour définir la plupart des fonctions non élémentaires.

Pourquoi la récursion est utile ?

- Des fonctions, comme `factorial`, sont plus simples à définir à partir d'autres.
- Par contre, certaines fonctions peuvent se définir aisément en terme de soi-mêmes.
- Les propriétés des fonctions définies par récursion peuvent se prouver en utilisant le simple mais puissant méthode de l'*induction mathématique*.

En plus

- La récursion est l'outil principal pour définir la plupart des fonctions non élémentaires.

Plan

Récursion sur les listes

La récursion n'est pas restreinte aux nombres,
mais on peut aussi s'en servir utilisé pour définir des fonctions sur les listes :

```
product :: [Int] -> Int
product [] = 1
product (n:ns) = n * product ns
```

> *`product` mappe la liste vide vers* 1*,*
> *et toute liste non vide vers le résultat de la multiplication de la tête avec le produit de la queue.*

Par exemple :

```
product [2,3,4]
= 2 * product [3,4]
= 2 * (3 * product [4])
= 2 * (3 * (4 * product []))
= 2 * (3 * (4 * 1))
= 24
```

length

En utilisant la même technique comme dans `product` nous pouvons définir la fonction `length` sur les listes :

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

> *`length` mappe la liste vide vers* 0*, et toute liste non vide vers le successeur de la longueur de la queue.*

Par exemple :

```
length [1,2,3]
= 1 + length [2,3]
= 1 + (1 + length [3])
= 1 + (1 + (1 + length []))
= 1 + (1 + (1 + 0))
= 3
```

reverse

De façon semblable, nous pouvons définir la fonction *reverse* :

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

> *reverse mappe*
> *la liste vide vers elle même,*
> *et toute liste non vide vers la concaténation de deux listes : reverse de la queue xs et la liste [x], qui contient seulement la tête x.*

Par exemple :

```
reverse [1,2,3]
= reverse [2,3] ++ [1]
= (reverse [3] ++ [2]) ++ [1]
= ((reverse [] ++ [3]) ++ [2]) ++ [1]
= (([] ++ [3]) ++ [2]) ++ [1]
= [3,2,1]
```

Plan

Arguments multiples

Les fonctions de plusieurs arguments peuvent aussi être définie par récursion.

Par exemple :

- le « zip » des élements de deux listes :

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _          = []
zip _ []          = []
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
```

- Eliminer les premiers *n* élements d'une liste :

```
drop :: Int -> [a] -> [a]
drop 0 xs      = xs
drop n []      = []
drop n (_:xs) = drop (n-1) xs
```

- Concaténation de deux listes :

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs++ys)
```

Le tri rapide

Le *tri rapide* d'une liste d'entiers peut se spécifier par les deux regles suivantes :

- la liste vide est déjà triée ;
- on peut trier une liste non vide en
  1. triant d'abord les valeurs de la queue plus petits que la tête,
  2. triant les valeurs de la queue plus grands que la tête,
  3. en ajoutant ces deux listes triées à la gauche et à la droite de la tête.

En utilisant la récursion, cette spécification se traduit directement en une implementation :

```
qsort :: [Int] -> [Int]
qsort [] = []
qsort (x:xs) =
    qsort smaller ++ [x] ++ qsort larger
    where
      smaller = [a | a <- xs, a < x]
      larger  = [b | b <- xs, x < b]
```

Remarque :

- Il s'agit, avec toute probabilité, du plus simple implementation du tri vite dans un langage de programmation.

Par exemple :

```
q [3,2,4,1,5] = q [2,1] ++ [3] ++ q [4,5]
q [2,1] = q [1] ++ [2] ++ q []
q [4,5] = q [] ++ [4] ++ q [5]
```

Plan

Récursion mutuelle

Un exemple :

```
even :: Int -> Bool
even 0 = True
even n = odd (n-1)

odd :: Int -> Bool
odd 0  = False
odd n  = even (n-1)
```

Plan

Conseils autour de la récursion

Une méthodologie – proposée par G. Hutton – pour définir par récursion.

Procédez comme suit :

1. définissez le type de la fonction ;
2. dénombrez les cas à pendre en considération ;
3. définissez les cas simple (cas base) ;
4. définissez les autres cas ;
5. généralisez et simplifiez.

Attention : l'etape 2 se fait (le plus souvent) par filtrage.

Nous allons tester cette méthode avec le produit scalaire – voir TD précédent.

Étape 1 : le type

```
produitScalaire :: [Int] -> [Int] -> Int
```

Étape 2 : les cas

```
produitScalaire :: [Int] -> [Int] -> Int
produitScalaire [] ys =
produitScalaire xs [] =
produitScalaire (x:xs) (y:ys) =
```

Étape 3 : les cas simples

```
produitScalaire :: [Int] -> [Int] -> Int
produitScalaire [] ys = 0
produitScalaire xs [] = 0
produitScalaire (x:xs) (y:ys) =
```

Étape 4 : les autres cas

```
produitScalaire :: [Int] -> [Int] -> Int
produitScalaire [] ys = 0
produitScalaire xs [] = 0
produitScalaire (x:xs) (y:ys) =
          x*y + produitScalaire xs ys
```

Étape 5 : géneralisations et simplifications

```
produitScalaire :: (Num a) => [a] -> [a] -> a
produitScalaire [] _ = 0
produitScalaire _ [] = 0
produitScalaire (x:xs) (y:ys) =
          x*y + produitScalaire xs ys
```

Plan

Exercices I

1. Sans regarder la bibliothèque *prelude.hs*, définissez les fonctions suivantes par récursion :
   - Décider si tous les valeurs logiques d'une liste sont vrais :

     ```
     and :: [Bool] -> Bool
     ```

   - Concaténer une liste de listes :

     ```
     concat :: [[a]] -> [a]
     ```

   - Produire une liste avec *n* éléments identiques :

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

   - Sélectionner le *n*ième élément d'une liste :

     ```
     (!!) :: [a] -> Int -> a
     ```

   - Décider si un valeur est un élément d'une liste :

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

Exercices II

2. Définissez une fonction récursive

```
merge :: [Int] -> [Int] -> [Int]
```

qui mélange deux listes triées pour produire une seule liste triée. Par exemple :

```
> merge [2,5,6] [1,3,4]
[1,2,3,4,5,6]
```

3. Définissez une fonction récursive

```
msort :: [Int] -> [Int]
```

qui implémente le tri par mélange (merge sort). Cet algorithme de trie peut se spécifier par les deux règles suivantes :

- Les listes le longueur ≤ 1 sont déjà triées ;
- Les autres listes peuvent être triées en les découpant en deux moitiés, en triant de deux moitiés, et en mélangeant les listes résultantes.