Programmation Fonctionnelle

Stefano Guerrini
stefano.guerrini@univ-paris13.fr
http://www.lipn.univ-paris13.fr/~guerrini

LIPN - Institut Galilée, Université Paris Nord 13

Licence Info 3

février 2013

Listes polymorphes

Listes polymorphes

- Objective Caml permet de générer des listes d'objets de même type sans que ce type soit précisé.
- On parle pourtant de listes polymorphes.
- A partir d'un type quelconque α on peut construire les listes d'éléments de type α.
- Les listes d'éléments de type α ont type : `'a list`

> On peut construire des listes d'un type quelconque, mais dans une liste tous les éléments ont le même type.

- Le type `'a list` est en fait un schéma de type prédéfini.
- La plupart des fonctions de manipulation des listes sont définies dans la bibliothèque `List`.

Construction de listes (1/2)

- La liste vide `[]` est une valeur de type polymorphe

```
# [];;
- : 'a list = []
```

- L'opérateur infixé `::` ou cons ajoute en élément en tête à une liste

```
# 1 :: [];;
- : int list = [1]
# true :: [];;
- : bool list = [true]
```

- On rappel qu'on n'a pas le droit de mélanger des éléments de types différents dans une liste.

```
# 1 :: [true];;
Characters 6-10:
  1 :: [true];;
       ^^^^
Error: This expression has type bool but an expression was expected of type
         int
```

- On note `[e1;...;en]` la liste `e1 :: (e2 :: ... (en :: []) ... )`

```
# 1 :: 2 :: -2 :: 0 :: 4 :: [];;
- : int list = [1; 2; -2; 0; 4]
```

Construction de listes (2/2)

- L'opérateur infixé `@` (opérateur de concaténation) concatène deux listes

```
  # [1;2] @ [3;4;5];;
- : int list = [1; 2; 3; 4; 5]
```

- Attention à la différence entre

```
# [1;2] @ [];;
- : int list = [1; 2]
```

et

```
# [1;2] :: [];;
- : int list list = [[1; 2]]
```

- Les éléments d'une listes peuvent être des listes.

```
# [4;5] :: [[1;2];[];[6];[7;8;5;2]];;
- : int list list = [[4; 5]; [1; 2]; []; [6]; [7; 8; 5; 2]]
```

- Les listes de listes polymorphes ont type : `'a list list`

```
# [[]];;
- : 'a list list = [[]]
```

- et aussi bien

```
# [[[]]];;
- : 'a list list list = [[[]]]
# [[[2;3]; [1]];[[]]; [[3]]];; [[[2;3]; [1]];[[]]; [[3]]];;
- : int list list list = [[[2; 3]; [1]]; [[]]; [[3]]]
```


Déstructuration des listes (1/3)

- La déstructuration des listes (l'extraction des éléments de la liste) se fait par pattern matching.

```
# let f = function
    [] -> 0
  | x :: l -> x+1;;
    val f : int list -> int = <fun>
```

- A la place de donner le nom de l'argument de la fonction, on donne des motifs (patterns) qui correspondent à des valeurs possibles pour l'argument de la fonction.
- A chaque motif `m` est associée une expression `e`

```
m -> e
```

  Si le motif `m` correspond à la valeur de la variable de la fonction alors le résultat est la valeur de l'expression `e`
- Par exemple `[] -> 0` indique que si l'argument de `f` est la liste vide, alors le résultat est `0`.

```
# f [];;
- : int = 0
```


Déstructuration des listes (2/3)

- Les pairs motif->expression (`p -> e`) sont séparées par une barre verticale

```
p1 -> e1 | p2 -> e2.
```

- Un motif peut contenir des variables qui, si le motif correspond à la valeur de l'argument de la fonction, seront associés à une partie de la valeur de l'argument.
- Par exemple, le motif de

```
x :: l -> x+1
```

  correspond à une liste non vide.
  - La valeur de tête de la liste est associée à la variable `x`.
  - La queue de la liste (c'est-à-dire ce qui reste de la liste sans la tête) est associée à la variable `l`.
- Les valeurs associés aux variables dans un motif sont utilisées pour le calcul de l'expression associée au motif.
- Pourtant, le motif précédent dit que le résultat de la fonction sur une liste non vide est la valeur de la tête plus 1.

```
# f [3;2;7];;
- : int = 4
```


Déstructuration des listes (3/3)

- Pour prendre les éléments d'une liste on peut également utiliser le fonctions `hd` (head) e `tl` (tail) de la librairie `List`.

```
# List.hd [1;3;5];;
 - : int = 1
# List.tl [1;3;5];;
- : int list = [3; 5]
```

- Pourtant

```
# (function l -> List.hd l :: List.tl l) [1;3;5];;
- : int list = [1; 3; 5]
```

- La fonction `f` qu'on viens de voir on pourrait l'écrire

```
# let f l =
  if (l == [])
  then 0
  else List.hd l + 1;;
      val f : int list -> int = <fun>
```

Motifs

- Les motifs peuvent être beaucoup plus compliqués des cas qu'on a vu.
- Par exemple,
  - pour définir une fonction qui prend une liste de listes d'entiers et ajoute 1 au premier éléments de la tête de l'argument (et si cet élément n'existe pas alors la fonction renvoie une constante)
  - il faut que le motif analyse la tête d'une liste non vide.

```
# let f = function
    [] -> 0
  | [] :: l -> 1
  | (x :: l1) :: l2 -> x+1;;
      val f : int list list -> int = <fun>

# f [ []; [1;2] ];;
- : int = 1
# f [ [3;2]; []; [1;4] ];;
- : int = 4
```

- On analysera en détail le pattern matching plus avant.

Fonctions récursives sur les listes

- La somme des éléments d'une liste d'entiers.

```
# let rec somme = function
    [] -> 0
  | x :: l -> x + somme l;;
    val somme : int list -> int = <fun>
# somme [3;2;7];;
- : int = 12
```

- La conjonction des éléments d'une liste de booléens.

```
# let rec conj = function
    [] -> true
  | x :: l -> x & conj l;;
    val conj : bool list -> bool = <fun>
```


La longueur d'une liste

- La longueur d'une liste est l'exemple classique de fonction polymorphe sur les listes : pour la calculer on n'a pas besoin de connaitre le type des éléments de la liste.

```
# let rec longueur = function
    [] -> 0
  | _ :: l -> longueur l + 1;;
    val longueur : 'a list -> int = <fun>

# longueur [true; false];;
- : int = 2
# longueur [[2]];;
- : int = 1
```

- En utilisant la fonction `List.tl`, on peut écrire aussi :

```
# let rec longueur l =
  if l == []
  then 0
  else 1 + longueur (List.tl l);;
      val longueur : 'a list -> int = <fun>
```

- On peut enfin utiliser la fonction `length` de la bibliothèque `List`. Attention, même la fonction de la bibliothèque prend un temps proportionnel à la longueur de la liste.

```
# List.length [1;4;5];;
 - : int = 3
```


La fonctionnelle map (1/2)

- Considérons l'exemple suivant, qui ajoute 1 à tout élément d'une liste.

```
# let rec succl = function
    [] -> []
  | x :: l -> (x+1) :: succl l;;
    val succl : int list -> int list = <fun>
# succl [2;1;5];;
- : int list = [3; 2; 6]
```

- Voici maintenant une fonction qui, à une liste d'entiers `[n1; . . .; nk]`, associe la liste de booléens `[b1; . . .;bk]` telle que `bi` soit `true` si et seulement si `ni` est pair.

```
# let rec pairl = function
    [] -> []
  | x :: l -> ((x mod 2) = 0) :: pairl l;;
    val pairl : int list -> bool list = <fun>
#  pairl [1;4;6;3;8];;
- : bool list = [false; true; true; false; true]
```

La fonctionnelle map (2/2)

- La librairie `List` définie la fonctionnelle

```
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

telle que

`List.map f[e1;...;en] = [f e1;...;f en]`

- `map` peut s'écrire très facilement

```
# let rec map f = function
    [] -> []
  | x :: l -> f x :: map f l;;
    val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- On peut alors réécrire les expressions vues plus haut en

```
# List.map (function x->x+1) [3; 2; 6];;
- : int list = [4; 3; 7]

# List.map (function x->(x mod 2)=0) [1;4;6;3;8];;
- : bool list = [false; true; true; false; true]
```


Fonctionnelles d'itérations sur les listes (1/4)

- La librairie `List` définie aussi deux fonctionnelles d'itération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>

# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

- Ces deux fonctionnelles permettent l'écriture compacte d'une fonction de calcul sur les éléments d'une liste comme la somme des éléments.
- Les effets de `fold_right` et `fold_left` sont les suivants.

`fold_right f [e1;e2;...;en] a = (f e1 (f e2 ... (f en a)...))`

`fold_left f a [e1;e2;...;en] = (f ... (f (f a e1) e2) ... en)`

- La `fold_right` correspond à un schéma général de récursion sur les listes.
- En fait, `let g l = fold_rigth f l a` definie une fonction telle que

`g [] = a`

`g (x :: l) = f x (g l)`

- La `fold_left` correspond plutôt à une iteration sur les élements de la liste de l'externe vers l'interne.

Fonctionnelles d'itérations sur les listes (2/4)

- `fold_right` et `fold_left` permettent d'écrire très rapidement et dans une forme concise les fonction récursives sur les listes.
- Par exemple, la fonction qui somme les éléments d'une liste peut s'écrire

```
# let somme l = List.fold_right (function x -> function y -> x+y) l 0;;
val somme : int list -> int = <fun>

# somme [3;1;9];;
- : int = 13
```

ou sinon

```
# let somme l =
  let add x y = x+y in
    List.fold_right add l 0;;
    val somme : int list -> int = <fun>
```

- L 'écriture directe de `fold_right` peut se faire de la façon suivante.

```
# let rec fold_right f ls a = match ls with
    [] -> a
  | x :: l -> f x (fold_right f l a);;
    val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```


Fonctionnelles d'itérations sur les listes (3/4)

- On peut également redéfinir la fonctionnelle `map`.

```
# let map f ls = List.fold_right (fun x l -> (f x) :: l) ls [];;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- De même, la fonction qui concatène deux listes peut s'écrire :

```
# let append l1 l2 = List.fold_right (fun x ls -> x :: ls) l1 l2;;
val append : 'a list -> 'a list -> 'a list = <fun>

# append [1;2;3] [4;5;6;7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
```

- La fonction `append` est prédéfinie dans la bibliothèque `List`.

```
# List.append [1;2;3] [4;5;6;7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
```

Fonctionnelles d'itérations sur les listes (4/4)

- L 'écriture directe de `fold_left` peut se faire de la façon suivante.

```
# let rec fold_left f a = function
    [] -> a
  | x :: l -> fold_left f (f a x) l;;
    val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

- En utilisant `fold_left` on peut définir la function qui inverse une liste.

```
# let rev = List.fold_left (fun l x -> x :: l) [];;
val rev : '_a list -> '_a list = <fun>

# rev [1;2;3;4];;
- : int list = [4; 3; 2; 1]
```

- La fonction

```
# List.rev;;
- : 'a list -> 'a list = <fun>
```

  est prédéfinie dans la librairie `List`.

Quelques autres fonctions du module `List`

```
# List.mem;;
- : 'a -> 'a list -> bool = <fun>
# List.flatten;;
- : 'a list list -> 'a list = <fun>
# List.find;;
- : ('a -> bool) -> 'a list -> 'a = <fun>
# List.split;;
- : ('a * 'b) list -> 'a list * 'b list = <fun>
# List.combine;;
- : 'a list -> 'b list -> ('a * 'b) list = <fun>
# List.assoc;;
- : 'a -> ('a * 'b) list -> 'b = <fun>
```