Programmation fonctionnelle

Notes de cours

Cours 3ter

12 Octobre 2011

Sylvain Conchon

sylvain.conchon@lri.fr

Itérateurs sur les listes

Schémas de définitions récursives

Les fonctions suivantes, vues dans le cours 3, ont toutes la même structure :

- la fonction `zeros`
- la fonction `recherche`
- la fonction `longueur`
- la fonction `append`
- la fonction `existe`

Laquelle ?

Schéma récursif en commun

Ces fonctions ont toutes le schéma récursif suivant (on note `l` la liste en entrée et `f` la fonction définie récursivement) :

1. si `l` est la liste vide, la valeur retournée par `f` ne dépend pas de `l` : c'est le cas de base de la récursion ;
2. sinon, `l` est de la forme `x::s` et la valeur retournée est calculée en effectuant une opération à partir de `x` et `f s`

Itération d'ordre supérieur

On peut capturer ce schéma à l'aide d'une fonction d'ordre supérieur prenant en argument une fonction `f` (à deux arguments), une liste `l` et un élément de départ `acc`

- L'argument `acc` représente la valeur retournée pour le cas de base de la récursion
- La fonction `f` est appliquée à chaque élément de la liste ainsi qu'au résultat de l'appel récursif

Exemple : la fonction somme

On montre comment abstraire le schéma d'une définition récursive à partir de la fonction somme suivante :

```
let rec somme l =
  match l with
   | [] -> 0
   | x::s -> x + (somme s)
```

Étape 1 : extraire l'opération récursive

```
let rec somme l =
  match l with
    | [] -> 0
    | x::s -> (fun a b -> a + b) x (somme s)
```

Étape 2 : abstraire l'opération récursive

```
let rec somme_fold f l =
  match l with
    | [] -> 0
    | x::s -> f x (somme_fold f s)

let somme l = somme_fold (fun a b -> a + b) l
```

Étape 4 : abstraire l'accumulateur

```
let rec somme_fold f l acc =
  match l with
    | [] -> acc
    | x::s -> f x (somme_fold f s acc)

let somme l = somme_fold (fun a b -> a + b) l 0
```

ou plus simplement

```
let somme l = somme_fold (+) l 0
```

Déroulons tout ça

```
somme [1;2;3]
 ⇒ somme_fold (+) [1;2;3] 0
 ⇒ (+) 1 (somme_fold (+) [2;3] 0)
 ⇒ (+) 1 ((+) 2 (somme_fold (+) [3] 0))
 ⇒ (+) 1 ((+) 2 ((+) 3 (somme_fold (+) [] 0)))
 ⇒ (+) 1 ((+) 2 ((+) 3 0))
 ⇒ (+) 1 ((+) 2 3)
 ⇒ (+) 1 5
 ⇒ 6
```

L'itérateur `fold_right`

La fonction `somme_fold` n'est pas spécifique au calcul de la somme d'une liste d'entiers. Son schéma récursif est capturé par la fonction suivante :

$$\begin{aligned} \texttt{fold\_right f [] acc} &= \texttt{acc} \\ \texttt{fold\_right f [}e_1\texttt{;}e_2\texttt{;...;}e_n\texttt{] acc} &= \texttt{f } e_1\ (f\ e_2\ (\cdots(f\ e_n\ acc)\cdots)) \end{aligned}$$

Cette fonction s'appelle `List.fold_right` dans la bibliothèque standard de OCaml :

```
let rec fold_right f l acc =
 match l with
   | [] -> acc
   | x::l -> f x (fold_right f l acc)
```

Son type est `('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

Attention : cette fonction n'est pas récursive terminale !

L'itérateur récursif terminal `fold_left`

Pour les fonctions dont l'ordre d'application de l'opération sur `x` et `g s` n'est pas important, on peut utiliser le parcours suivant :

$$\begin{aligned} \texttt{fold\_left f acc []} &= \texttt{acc} \\ \texttt{fold\_left f acc } [e_1; e_2; \ldots; e_n] &= \texttt{f } (\cdots (f\ (f\ acc\ e_1)\ e_2) \cdots)\ e_n \end{aligned}$$

Cette fonction s'appelle `List.fold_left` dans la bibliothèque standard de OCaml :

```
let rec fold_left f acc l =
  match l with
   | [] -> acc
   | x::s -> fold_left f (f acc x) s
```

Son type est `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

Cette fonction est récursive terminale !

Exemple 1 : somme d'une liste d'entiers (suite)

On peut écrire la fonction `somme` avec `List.fold_left`

```
let somme l = fold_left (+) 0 l

val somme : int list -> int
```

```
    somme [1;2;3]
=   fold_left (+) 0 [1;2;3]
⇒   fold_left (+) ((+) 0 1) [2;3]
=   fold_left (+) 1 [2;3]
⇒   fold_left (+) ((+) 1 2) [3]
=   fold_left (+) 3 [3]
⇒   fold_left (+) ((+) 3 3) []
=   fold_left (+) 6 []
⇒   6
```

exemple 2 : Longueur d'une liste

Rappel : version sans itérateur

```
let rec longueur l =
  match l with
    |  [] -> 0
    | x::s -> 1 + (longueur s)

val longueur : 'a list -> int
```

Version avec itérateur :

```
let longueur l = fold_left (fun acc x -> 1 + acc) 0 l

longueur : 'a list -> int
```

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

Évaluation de la fonction `longueur`

Évaluation de `longueur [3 ;2 ;1 ;4]`

On note `plus1` la fonction `(fun acc x -> 1 + acc)`

```
    longueur [3 ;2 ;1 ;4]
=   fold_left plus1 0 [3; 2; 1; 4]
⇒   fold_left plus1 (plus1 0 3) [2; 1; 4]
=   fold_left plus1 1 [2; 1; 4]
⇒   fold_left plus1 (plus1 1 2) [1; 4]
=   fold_left plus1 2 [1; 4]
⇒   fold_left plus1 (plus1 2 1) [1; 4]
=   fold_left plus1 3 [4]
⇒   fold_left plus1 (plus1 3 4) []
=   fold_left plus1 4 []
=   4
```

Exemple 3 : concaténation de deux listes

Rappel : version sans itérateur

```
let rec append l1 l2 =
  match l1 with
   | [] -> l2
   | x::s -> x::(append s l2)

append : 'a list -> 'a list -> 'a list
```

Version avec itérateur

```
# let append l1 l2 =
    fold_right (fun x acc -> x::acc) l1 l2;;

val append : 'a list -> 'a list -> 'a list = <fun>

# append [1;2;3] [4;5;6];;

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

Autres fonctions

```
let zeros = fold_left (fun acc x -> x=0 && acc) true

val zeros : int list -> bool = <fun>


let recherche n =
    fold_left (fun acc x -> x=n || acc) false

val recherche : 'a -> 'a list -> bool = <fun>


let existe p = fold_left (fun acc x -> p x || acc) false

val existe : ('a -> bool) -> 'a list -> bool = <fun>
```

Exemple 4 : liste des sous-listes d'une liste (1/2)

On souhaite écrire une fonction `sous_listes` pour calculer la liste des sous-listes d'une liste `l`

On commence par écrire une fonction `cons` qui ajoute un élément à toutes les listes d'une liste de listes :

```
let rec cons_elt x l =
  match l with
    | [] -> []
    | r::s -> (x::r)::(cons_elt x s)

cons_elt : 'a -> 'a list list -> 'a list list
```

Exemple 4 : liste des sous-listes d'une liste (2/2)

La fonction `sous_liste` s'écrit alors naturellement de la manière suivante :

```
let rec sous_listes l =
  match l with
   | [] -> [[]]
   | x::s -> let p = sous_listes s in (cons_elt x p)@p

sous_listes : 'a list -> 'a list list
```

```
# sous_listes [1;2;3];;

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

Exemple 5 : `sous_liste` avec itérateurs

On commence par définir l'itérateur `map` de type

```
('a -> 'b) -> 'a list -> 'b list
```

tel que `map f [`e_1 `; ... ;` e_n`] = [f` e_1 `; ... ; f` e_n`]`

```
let rec map f l =
   match l with
  | [] -> []
  | x::s -> let v = f x in v :: (map f s)
```

Le fonction `sous_liste` peut alors s'écrire (avec `@` l'opérateur de concaténation en OCaml) :

```
let sous_listes =
  fold_left (fun p x -> (map (fun l->x::l) p)@p) [[]]
```

Itérateurs vs. fonctions récursives (1/2)

Les versions sans itérateurs des fonctions `zeros`, `recherche` et `existe` sont plus efficaces que leurs versions avec itérateurs respectives

```
let rec existe p l =
  match l with
   | [] -> false
   | x::s -> p x || (existe p s)
```

On note `p` la fonction `fun x -> x=0`

$$
\begin{aligned}
& \texttt{existe p [3;0;2;1;4]} \\
\Rightarrow\ & \texttt{p 3 || existe p [0;2;1;4]} \\
=\ & \texttt{existe p [0;2;1;4]} \\
\Rightarrow\ & \texttt{p 0 || existe p [2;1;4]} \\
=\ & \texttt{true}
\end{aligned}
$$

Itérateurs vs. fonctions récursives (2/2)

La version avec itérateur va jusqu'au bout de la liste

```
let existe p = fold_left (fun acc x -> p x || acc) false
```

```
    existe p [3;0;2;1;4]
=   fold_left p false [0;2;1;4]
⇒   fold_left p (p 3 || false) [0;2;1;4]
=   fold_left p false [0;2;1;4]
⇒   fold_left p (p 0 || false) [2;1;4]
=   fold_left p true [0;2;1;4]
⇒   fold_left p (p 2 || true) [1;4]
=   fold_left p true [1;4]
⇒   fold_left p (p 1 || true) [4]
=   fold_left p true [4]
⇒   fold_left p (p 4 || true) []
=   fold_left p true []
=   true
```