Chapitre 3

Ordre supérieur

Introduction

Dans la plupart des langages impératifs comme `C`, les fonctions doivent être définies dans le texte du programme mais ne peuvent être créées dynamiquement durant l'exécution. En Caml si !

On peut en effet en Caml définir des fonctions anonymes qui sont des valeurs à parts entières au même titre qu'un entier ou une chaîne de caractères. Les valeurs fonctionnelles qui ne peuvent être affichées sont représentées à l'écran par le mot `<fun>` :

```
fun x->x+1;;
- : int -> int = <fun>

#2;;
- : int = 2
```

Ainsi au même titre que les autres objets du langage, les fonctions peuvent être passées en arguments d'autres fonctions (qu'on appelle alors des fonctionnelles) ou être le résultat de l'application d'une fonction (c'est toujours le cas lorsque l'on fait l'application partielle d'une fonction à plusieurs arguments).

Définition 1. *Une fonction est dite d'ordre supérieur si elle prend une autre fonction en argument ou renvoi une autre fonction comme résultat.*

Ces fonctions permettent de définir des outils généraux paramétrés par des traitements particuliers. Une fois définies, l'utilisation de ces fonctionnelles diminue le temps de programmation et permet le prototypage rapide.

3.1 Fonctionnelles générales

- La composition de fonctions :

  ```
  #let o f g = fun x -> f (g x);;
  o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
  ##infix "o";;  (* notation infix pour pouvoir écrire f o g *)
  ```

- Curryfication et Décurryfication :

  ```
  let curry f x y=f(x,y);;

  let uncurry f (x,y)= f x y;;
  ```

– Permutation

```
let perm f x y = f y x;;
```

```
perm : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c = <fun>
```

– Itération : appliquer `f n` fois :

```
let rec iter n f = fun x -> if n=0 then x else f (iter (n-1) f x);;
```

```
iter : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

– Iteration plus générale : appliquer`f` jusqu'à une condition d'arrêt :

```
let rec loop p f x = if (p x) then x else loop p  f (f x);;
```

Exemple 1 (utilisation de loop). Calcul par dichotomie du zéro d'une fonction dans un interval `[a,b]` avec une approximation de `e` :

```
let zero(f,a,b,e)=
      let assezbon (x,y) = x-.y<e and
          dicho (x,y) = let m=(x+.y)/.2.0 in if (f x)*.(f m)<0.0 then (x,m)
                                                                  else (m,y) in
      loop assezbon dicho (a,b);;
```

3.2 Fonctionnelles sur des structures de données particulières

.

3.2.1 Fonctionelles sur les liste

Il y a plein de fonctionnelles prédéfinies sur les listes, par exemple `map` qui prend en argument une liste `l` d'éléments de type `a`, une fonction `f` de type `a->b` et renvoie la liste d'éléments de type `b` formée en appliquant `f` aux éléments de `l`. soit :

```
let rec map f l= match l with
   []->[]
   |x::xs->(f x)::(map f xs);;
```

Consultez la doc en ligne pour trouver les autres fonctionelles disponible. A titre d'exercice vous pouvez aussi regarder le fichier `/usr/local/lib/caml-light/list.mli` et essayer de coder vous-même les fonctionelles qui apparaîssent puis regarder la solution choisie par les concepteurs de Caml dans le fichier `/usr/local/lib/caml-light/list.ml`.

3.2.2 Itérateurs

A regarder de plus près les fonctions écrites sur des types récursifs on constate des similarités de formes.

En fait il est possible d'associer à chaque type récursif un itérateur qui exprime le comportement du type récursif.

Pour les entiers de Peano[1], l'opérateur `nat_it` peut se définir par :

```
let rec nat_it f b n = match n with
                   O ->b
                  |S x -> f (nat_it f b x);;
```

Son type est :

```
('a -> 'a) -> 'a -> nat -> 'a
```

Il peut être utilisé pour définir l'addition, la multiplication...

```
let addition x = nat_it S x;;
let multiplication x  = nat_it (addition x) O;;
```

Pour les listes, l'opérateur `liste_it` prédéfini en Caml peut s'écrire :

```
let rec list_it f b l = match l with
   []->b
  |x::xs->f x (list_it f b xs);;
```

Son type est :

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

Il peut être utilisé pour définir toute les fonctions sur les listes.

Par exemple on peut définir la longueur d'une liste

```
let longueur = list_it (fun x->succ) 0;;
```

La fonctionnelle `map` peut être définie par :

```
let map f = list_it (fun x l ->(f x)::l) [];;
```

[1] cf TD