

Algorithmique et Approche Fonctionnelle

Cours 6 : Ordre Supérieur et Polymorphisme

20 Octobre 2009

Ordre Supérieur

les fonctions sont des valeurs à part entière

les fonctions sont des types de données comme les autres

une fonction peut être :

- stockée dans une **structure de donnée** (n-uplets, enregistrements, listes etc.)
- passée en **argument** à une autre fonction
- retournée comme **résultat** d'une fonction

les fonctions prenant des fonctions en arguments ou rendant des fonctions en résultat sont dites **d'ordre supérieur**

exemples de structures de données contenant des fonctions

un n-uplet avec des composantes fonctionnelles :

```
# ( (fun x-> x+1) , 4 , (fun x -> x::['a']) );;  
- : (int -> int) * int * (char -> char list) = (<fun>,4,<fun>)
```

un enregistrement avec une étiquette fonctionnelle :

```
# type t = { f : int -> int ; x : int };;  
type t = { f : int -> int ; x : int ; }  
# { f = (fun x -> x+1) ; x=10 };;  
- : t = {f = <fun> ; x = 10}
```

une liste contenant des fonctions :

```
# [(fun x-> x+1) ; (fun x-> x*2) ; (fun x-> 4)];;  
- : (int -> int) list = [<fun> ; <fun> ; <fun>]
```

- certaines fonctions prennent naturellement des fonctions en arguments
- par exemple, les notations mathématiques telles que la sommation $\sum_{i=1}^n f(i)$ se traduisent immédiatement si l'on peut utiliser des arguments fonctionnels

```
# let rec somme (f,n) =
  if n<=0 then 0
  else (f n) + somme (f,n-1);;
val somme : (int -> int) * int -> int = <fun>
```

```
# somme ((fun x->x*x),10);;
- : int = 385
```

si f est une fonction continue et monotone, on peut trouver un zéro de f sur un intervalle $[a, b]$ par la méthode dichotomique quand $f(a)$ et $f(b)$ sont de signes opposés :

- si ϵ est la précision souhaitée et que $|b - a| < \epsilon$ alors on renvoie a
- sinon, couper l'intervalle $[a, b]$ en deux et recommencer sur l'intervalle contenant 0

```
# let rec dichotomie (f,a,b,epsilon) =
  if abs_float(b -. a) < epsilon then a
  else
    let c = (a+.b) /. 2.0 in
    let na,nb = if (f a)*(f c)>0.0 then (c,b) else (a,c) in
    dichotomie (f,na,nb,epsilon)
val dichotomie :
(float -> float) * float * float * float -> float = <fun>
```

la complexité de cette méthode est en $O(\log(|b - a|/\epsilon))$

par exemple, on peut utiliser cette fonction pour trouver un encadrement de π en le calculant comme zéro de la fonction $\cos(x/2)$

```
# dichotomie ((fun x->cos (x/.2.0)),3.1,3.2,1e-10);;
- : float = 3.14159265356138384
```

les fonctions à **plusieurs arguments** sont en fait des fonctions d'ordre supérieur qui rendent des **fonctions en résultat**

```
# let plus x y = x+y;;
val plus : int -> int -> int
```

il faut lire le type de cette fonction de la manière suivante

```
int -> (int -> int)
```

de manière équivalente, on peut écrire la fonction `plus` de la façon suivante afin de souligner son résultat fonctionnel

```
# let plus x = (fun y -> x+y);;
val plus : int -> int -> int
```

les fonctions d'ordre supérieur rendant des fonctions en résultats peuvent être **appliquées partiellement**

```
# let plus2 = plus 2;;
val plus2 : int -> int = <fun>
```

```
# plus2 10;;
- : int = 12
```

on peut calculer de façon approximative la dérivée f' d'une fonction f avec un petit intervalle dx de la manière suivante :

```
# let derive (f,dx) = fun x -> (f(x +. dx) -. f(x))/ . dx;;
val derive :
  (float -> float) * float -> float -> float = <fun>
```

```
# derive ( (fun x->x*.x),1e-10) 1.;;
- : float = 2.000000165480742
```

on peut réécrire la fonction derive de la manière suivante

```
# let derive dx f = fun x -> (f(x +. dx) -. f(x))/ . dx;;
val derive : float -> (float -> float) -> float -> float
```

on fixe le paramètre dx par application partielle

```
# let derivation = derive 1e-10;;
val derivation : (float -> float) -> float -> float
```

on peut alors définir par exemple la dérivée de la fonction sinus

```
# let sin' = derivation sin;;
val sin' : float -> float

# sin' 1.;;
- : float = 0.540302247387103307
# cos 1.;;
- : float = 0.540302305868139765
```

Polymorphisme

quel est le type de la fonction suivante ?

```
# let identite x = x;;
```

les appels suivants sont parfaitement corrects

```
# identite 4;;
```

```
- : int = 4
```

```
# identite "bonjour";;
```

```
- : string = "bonjour"
```

```
# identite (4, (fun x->x+1));;
```

```
- : int * (int -> int) = (4, <fun>)
```

laissons OCAML nous indiquer son type

```
val identite : 'a -> 'a = <fun>
```

'a est une **variable de type** et elle peut être remplacée par n'importe quel type (de base, fonctionnel, ou défini par le programmeur)

la fonction identite a donc tous les types suivants (et bien plus encore)

- en remplaçant 'a par int

```
val identite : int -> int
```

- en remplaçant 'a par string

```
val identite : string -> string
```

- en remplaçant 'a par int * (int -> int)

```
val identite : int * (int -> int) -> int * (int -> int)
```

une fonction est **polymorphe** si son type contient des variables de type

définitions de types polymorphes

les types définis par le programmeur peuvent aussi être polymorphes

```
# type 'a liste = Nil | Cons of 'a * 'a liste;;
```

```
type 'a liste = Nil | Cons of 'a * 'a liste
```

```
# Nil;;
```

```
- : 'a liste = Nil
```

```
# Cons(1,Nil);;
```

```
- : int liste = Cons (1, Nil)
```

ordre supérieur et polymorphisme

le mélange **ordre supérieur+polymorphisme** permet d'écrire du code **plus général** et donc plus **réutilisable**

```
# let double x = 2 * x;;
```

```
# let carre x = x * x;;
```

on utilise ces fonctions pour définir une fonction qui quadruple un entier x et une autre qui calcule x^4

```
# let quadruple x = double (double x);;
```

```
# let puissance4 x = carre (carre x);;
```

ces fonctions sont similaires : elles appliquent **deux fois** une fonction

```
# let applique_deux_fois f x = f(f(x));;
```

```
val applique_deux_fois : ('a -> 'a) -> 'a -> 'a = <fun>
```

```
# let quadruple x = applique_deux_fois double x;;
```

```
# let puissance4 x = applique_deux_fois carre x;;
```

la fonction permettant de tester l'existence d'un élément dans une liste vérifiant une propriété quelconque p

```
#let rec existe p l =
  match l with
  | [] -> false
  | x::s -> p x || (existe p s);;
val existe : ('a -> bool) -> 'a list -> bool = <fun>
```

le test d'appartenance à une liste s'écrit facilement en utilisant existe de la manière suivante

```
# let appartient x = existe (fun y->x=y);;
val appartient : 'a -> 'a list -> bool = <fun>
```

```
# appartient 'a' ['o';'c';'a';'m';'l'];;
- : bool = true
```

la fonction map transforme une liste [e1 ; .. ; en] en une liste [f e1 ; .. ; f en] pour une fonction f quelconque

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

```
# map float_of_int [1;2;3;4];;
- : float list = [1.0; 2.0; 3.0; 4.0]
```

la fonction filtre filtre tous les éléments d'une liste vérifiant une certaine propriété p

```
#let rec filtre p l =
  match l with
  | [] -> []
  | x::s -> if p x then x::(filtre p s) else filtre p s;;
val filtre : ('a -> bool) -> 'a list -> 'a list = <fun>
```

```
# filtre (fun x->x mod 2=0) [1;2;3;4];;
- : int list = [2; 4]
```

la composition de fonctions s'écrit naturellement de la façon suivante :

```
# let compose f g = fun x -> f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

comment OCAML a-t-il fait pour découvrir le type de cette fonction ?

- on affecte des variables de type différentes à chaque paramètre
- on raffine le type de ces variables pour chaque contrainte apparaissant dans l'expression
- on obtient ainsi le type le **plus général**

exercices

quel est le type des fonctions suivantes ?

```
# let rec f a b c = if c <= 0 then a else f a b (b c);;
```

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

```
# let f g x = g ( g x );;
```

```
# let rec f g x = f ( g x );;
```

```
# let rec f x = f x ;;
```

- le **type de départ** de la fonction compose est

```
'a -> 'b -> 'c -> 'd
```

- la sous-expression $(g\ x)$ indique que **g est une fonction** et que **x a le type d'entrée de g**

```
'a -> ('e -> 'f) -> 'e -> 'd
```

- la sous-expression $f\ (g\ x)$ indique que **f est une fonction** et que le type du résultat de $(g\ x)$ est le **type d'entrée de f**

```
('g -> 'h) -> ('e -> 'g) -> 'e -> 'd
```

- dernière contrainte : le type de $f(g\ x)$ est le **type de retour de compose**

```
('g -> 'h) -> ('e -> 'g) -> 'e -> 'h
```

qui est bien le résultat donné par OCAML (après renommage des variables 'g par 'a, 'h par 'b et 'e par 'c)