Introduction à la programmation fonctionnelle

Notes de cours

Cours 4

27 mars 2020

Sylvain Conchon

sylvain.conchon@lri.fr

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**

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>]
```

Fonctions comme arguments

- Certaines fonctions prennent naturellement des fonctions en arguments
- Par exemple, les notations mathématiques telles que la sommation $\Sigma_{i=1}^{n} f(i)$ 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
```

Exemple : la méthode dichotomique (1/2)

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

La méthode dichotomique (2/2)

```
# 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>
```

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
```

Fonctions en résultat

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
```

Application partielle

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
```

Fonctions en arguments et en résultat (1/2)

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 = 2.00000016548074
```

Fonctions en arguments et en résultat (2/2)

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

Fonctions polymorphes

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>)
```

Variables de type et fonctions polymorphes

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

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

- en remplaçant `'a` par `int` : `int -> int`
- en remplaçant `'a` par `string` : `string -> string`
- en remplaçant `'a` par `int * (int -> int)` :
  `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;;
```

Exemples : fonctions génériques sur les listes (1/3)

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
```

Exemples : fonctions génériques sur les listes (2/3)

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]
```

Exemples : fonctions génériques sur les listes (3/3)

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 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
```

Comment OCaml a-t-il fait pour découvrir le type de cette fonction ?

Synthèse de type (2/2)

- 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

Synthèse de type pour la fonction `compose`

- 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`)

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 ;;
```

Tri de listes

Tri pas Insertion : principe

Cet algorithme de tri suit de manière naturelle la structure récursive des listes

Soit `l` une liste à trier :

1. si `l` est vide alors elle est déjà triée
2. sinon, `l` est de la forme `x::s` et,
   - on trie récursivement la suite `s` et on obtient une liste triée `s'`
   - on insert `x` au bon endroit dans `s'` et on obtient une liste triée

Insertion

- La fonction `inserer` permet d'insérer un élément `x` dans une liste `l`
- Si la liste `l` est triée alors `x` est inséré au bon endroit
- On prend pour le moment <= comme relation d'ordre

```
# let rec inserer x l =
   match l with
     [] -> [x]
   | y::s -> if x<=y then x::l else y::(inserer x s);;

val inserer: 'a -> 'a list -> 'a list


# inserer 5 [3;7;10];;

- : int list = [3; 5; 7; 10]
```

Évaluation de la fonction `inserer`

Évaluation de `inserer 5 [3;7;10]`

$$
\begin{array}{rcl}
 & & \texttt{inserer 5 [3;7;10]} \\
[3;7;10] \neq [], y = 3, s = [7;10], 5 > 3 & \Rightarrow & \texttt{3}::\texttt{(inserer 5 [7;10])} \\
[7;10] \neq [], y = 7, s = [10], 5 <= 7 & \Rightarrow & \texttt{3}::\texttt{5}::\texttt{[7;10]} \\
 & \Rightarrow & \texttt{3}::\texttt{[5;7;10]} \\
 & \Rightarrow & \texttt{[3;5;7;10]}
\end{array}
$$

Trier une liste

On utilise la fonction `inserer` pour réaliser un tri par insertion d'une liste

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

```
# trier [6; 1; 9; 4; 3];;
- : int list = [1; 3; 4; 6; 9]
```

Évaluation de la fonction `trier`

Évaluation de `trier [6;4;1;5]`

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`
$x = 5, s = []$	$\Rightarrow$	`...(inserer 5 (trier []))))`
	$\Rightarrow$	`...(inserer 5 []))))`
	$\Rightarrow$	`inserer 6 (inserer 4 (inserer 1 [5]))`
	$\Rightarrow$	`inserer 6 (inserer 4 [1;5])`
	$\Rightarrow$	`inserer 6 [1;4;5]`
	$\Rightarrow$	`[1;4;5;6]`

Tri Rapide : principe

Soit une liste `l` à trier :

1. si `l` est vide alors elle est triée
2. sinon, choisir un élément `p` de la liste (le premier par exemple) nommé le pivot
3. partager `l` en deux listes `g` et `d` contenant les autres éléments de `l` qui sont plus petits (resp. plus grands) que la valeur du pivot `p`
4. trier récursivement `g` et `d`, on obtient deux listes `g'` et `d'` triées
5. on renvoie la liste `g'@[p]@d'` (qui est bien triée)

Partage d'une liste

La fonction suivante permet de partager une liste `l` en deux sous-listes `g` et `d` contenant les éléments de `l` plus petits (resp. plus grands) qu'une valeur donnée `p`

```
#let rec partage p l =
  match l with
     [] -> ([] , [])
    |x::s -> let (g, d) = partage p s in
               if x <= p then (x::g, d) else (g, x::d) ;;
val partage : 'a -> 'a list -> 'a list * 'a list = <fun>
```

```
# partage 5 [1;9;7;3;2;4];;
- : int list * int list = ([1; 3; 2; 4], [9; 7])
```

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

```
   partage 5 [1;9;3;7]
⇒  let (g1, d1) = partage 5 [9;3;7] in (1::g1, d1)
⇒  let (g2, d2) = partage 5 [3;7] in (g2, 9::d2)
⇒  let (g3, d3) = partage 5 [7] in (3::g3, d3)
⇒  let (g4, d4) = partage 5 [] in (g4, 7::d4)
⇒  [],[]
⇒  let (g4, d4) = ([],[]) in (g4, 7::d4)
⇒  ([], [7])
⇒  let (g3, d3) = ([],[7]) in (3::g3, d3)
⇒  ([3], [7])
⇒  let (g2, d2) = ([3],[7]) in (g2, 9::d2)
⇒  ([3], [9;7])
⇒  let (g1, d1) = ([3],[9;7]) in (1::g1, d1)
⇒  ([1;3], [9;7])
```

Tri rapide

```
# let rec tri_rapide l =
   match l with
      [] -> []
    | p::s -> let g , d = partage p s in
             (tri_rapide g)@[p]@(tri_rapide d)  ;;

val tri_rapide : 'a list -> 'a list = <fun>
```

```
# tri_rapide [5; 1; 9; 7; 3; 2; 4];;
- : int list = [1; 2; 3; 4; 5; 7; 9]
```