Algorithmique et Approche Fonctionnelle de la Programmation

Cours 6 : Ordre Supérieur et Polymorphisme

19 octobre 2009

Louis Mandel
Université Paris-Sud 11
louis.mandel@lri.fr

D'après des transparents de Sylvain Conchon

Ordre Supérieur

Les fonctions sont des valeurs à part entière

- Les fonctions sont des types de données comme les autres
- Rappel :

```
# let f x = x + 1;;
val f : int -> int = <fun>
```

  est équivalent à

```
# let f = fun x -> x + 1;;
val f : int -> int = <fun>
```

- Une fonction peut être :
  - ▷ stockée dans une structure de donnée (n-uplets, enregistrement, liste...)
  - ▷ passée en argument à une autre fonction
  - ▷ retournée comme résultat d'une autre 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

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

- Une enregistrement avec un champ fonctionnel :

```
# type t = { f : (int -> int);
             x : int; };;
# let _ = { f = (fun x -> x mod 2) ; x = 10; };;
- : t = {f = <fun>; x = 10}
```

- Liste contenant des fonctions

```
# let _ = [ (fun x -> x + 1); (fun x -> 4); (fun x -> x * 2) ];;
- : (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 somme (f, n) =
    let rec somme_aux (f, n) =
      if n = 1 then f 1
      else (f n) + somme_aux (f, n - 1)
    in
    somme_aux (f, n)
  ;;
val somme : (int -> int) * int -> int = <fun>
# let _ = somme ((fun x -> x * x), 10);;
- : int = 385
```

Exemple : la méthode dichotomique

- Si `f` est une fonction continue et monotone, on peut trouver un zéro de `f` sur un intervalle $[\mathtt{a}, \mathtt{b}]$ par la méthode dichotomique quand `f(a)` et `f(b)` sont de signe opposés

- Algorithme :
  - ▷ si ε est la précision souhaitée et que $|\mathtt{b} - \mathtt{a}| < \varepsilon$ alors on renvoie `a`
  - ▷ sinon, couper l'intervalle $[\mathtt{a}, \mathtt{b}]$ en deux et recommencer sur l'intervalle contenant 0

La méthode dichotomique

```
# 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 (a, c)
        else (c, b)
      in
      dichotomie (f, na, nb, epsilon)
  ;;
val dichotomie : (float -> float) * float * float * float -> float = <fun>
```

- La complexité de cette méthode est en $O(\log(|\mathrm{b} - \mathrm{a}|/\varepsilon))$

La méthode dichotomique

- Exemples :
  - on peut utiliser cette méthode pour trouver un encadrement de π en le calculant comme zéro de la fonction $\cos(x/2)$

```
# let _ = dichotomie ((fun x -> cos (x /. 2.0)), 3.1, 3.2, 1e-10);;
- : float = 3.1415926535613838 4
```

  - de même, on peut trouver un encadrement de e en le calculant comme zéro de la fonction $\log(x) - 1$

```
# let _ = dichotomie ((fun x -> log x -. 1.0), 2.0, 3.0, 1e-10);;
- : float = 2.7182818284491077 1
```

Fonction 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 = <fun>
```

- 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 = <fun>
```

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>
# let _ = plus2 10;;
- : int = 12
# let _ = plus2 100;;
- : int = 102
```

Fonctions en arguments et en résultat

- 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>
# let carre x = x *. x;;
val carre : float -> float = <fun>
# let carre' = derive (carre, 1e-10);;
val carre' : float -> float = <fun>
# let _ = carre' 1.0;;
- : float = 2.0000001654807 42
# let _ = carre' 2.0;;
- : float = 4.0000003309614 84
```

Fonctions en arguments et en résultat

- 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 = <fun>
```

- On fixe le paramètre `dx` par application partielle

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

- On peut alors définir par exemple la dérivée de la fonction sinus

```
# let sin' = derivation sin;;
val sin' : float -> float = <fun>
# let _ = sin' 1.0;;
- : float = 0.54030224738710330 7
# let _ = cos 1.0;;
- : float = 0.54030230586813976 5
```

Application partielle

On peut aussi faire des calculs avant de retourner une fonction

```
# let f x =
    Printf.printf "x = %d " x;
    (fun y -> Printf.printf "y = %d " y);;
val f : int -> int -> unit = <fun>
# let f1 = f 1;;
x = 1 val f1 : int -> unit = <fun>
# let _ = f1 2;;
y = 2 - : unit = ()
# let _ = f1 3;;
y = 3 - : unit = ()
# let _ = f 4 5;;
x = 4 y = 5 - : unit = ()
```

Différence avec les pointeurs de fonctions

```
# let f x =
    let x2 = x * x in
    fun y -> x2 + y * y;;
val f : int -> int -> int = <fun>
# let f5 = f 5;;
val f5 : int -> int = <fun>
# let f10 = f 10;;
val f10 : int -> int = <fun>
```

Les fonctions `f5` et `f10` ont chacune un état

- on parle de fermetures

Polymorphisme

Fonctions polymorphes

- Quel est le type de la fonction suivante ?

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

- Les appels suivants sont parfaitement corrects

```
# let _ = identite 4;;
- : int = 4
# let _ = identite "bonjour";;
- : string = "bonjour"
# let _ = identite (true, (fun x -> x mod 2 = 0));;
- : bool * (int -> bool) = (true, <fun>)
```

Variables de type et fonctions polymorphes

- Laissons OCaml nous indiquer son type :

```
# let identite x = x;;
val identite : 'a -> 'a = <fun>
```

- `'a` est une variable de type et elle peut être remplacée par n'importe quel type (de base, fonctionnel, u défini par le programmeur)
- 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 -> bool)` :
    `int * (int -> bool) -> int * (int -> bool)`
- 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
# let _ = Nil;;
- : 'a liste = Nil
# let _ = Cons(1, Nil);;
- : int liste = Cons (1, Nil)
# let _ = Cons(true, Nil);;
- : bool liste = Cons (true, Nil)
```

Synthèse de types

- 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>
```

Rappel des règles de typage

$$\frac{\text{si} \quad H \vdash e : \texttt{bool} \quad \text{et} \quad H \vdash e_1 : t \quad \text{et} \quad H \vdash e_2 : t}{\text{alors} \quad H \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t} \qquad \frac{H(x) = t}{H \vdash x : t}$$

$$\frac{H \vdash e_1 : t_1 \qquad x : t_1, H \vdash e_2 : t_2}{H \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : t_2} \qquad \frac{x : t_1, H \vdash e : t_2}{H \vdash \texttt{(fun}\ x\ \texttt{->}\ e\texttt{)} : t_1\ \texttt{->}\ t_2}$$

$$\frac{H \vdash e_1 : t_2\ \texttt{->}\ t_1 \qquad H \vdash e_2 : t_2}{H \vdash \texttt{(}e_1\ \ e_2\texttt{)} : t_1} \qquad \frac{H \vdash e_1 : \texttt{int} \qquad H \vdash e_2 : \texttt{int}}{H \vdash e_1\ \texttt{+}\ e_2 : \texttt{int}}$$

Preuve du type de `compose`

$$\dfrac{\dfrac{}{H \vdash \texttt{f} : \texttt{('a -> 'b)}} \qquad \dfrac{H \vdash \texttt{g} : \texttt{('c -> 'a)} \qquad H \vdash \texttt{x} : \texttt{'c}}{H \vdash \texttt{g x} : \texttt{'a}}}{\overbrace{\texttt{f} : \texttt{('a -> 'b)}, \texttt{g} : \texttt{('c -> 'a)}, \texttt{x} : \texttt{'c}}^{H} \vdash \texttt{f (g x)} : \texttt{'b}}$$

$$\dfrac{\texttt{f} : \texttt{('a -> 'b)}, \texttt{g} : \texttt{('c -> 'a)} \vdash \texttt{fun x -> f (g x)} : \texttt{'c -> 'b}}{\texttt{f} : \texttt{('a -> 'b)} \vdash \texttt{fun g -> fun x -> f (g x)} : \texttt{('c -> 'a) -> 'c -> 'b}}$$

$$\dfrac{}{\emptyset \vdash \texttt{fun f -> fun g -> fun x -> f (g x)} : \texttt{('a -> 'b) -> ('c -> 'a) -> 'c -> 'b}}$$

Synthèse de types

- Comment OCaml a-t-il fait pour découvrir le type de la fonction `compose` ?
  - ▷ on affecte des variables différentes au type de chaque paramètre
  - ▷ on raffine la valeur 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 ;;
```

Ordre supérieur et polymorphisme

- Le mélange ordre supérieur et polymorphisme permet d'écrire du code plus général et donc plus réutilisable

Ordre supérieur et polymorphisme

```
# let rec print_string_list l =
    match l with
    | [] -> ()
    | x :: xs -> print_string x; print_string_list xs
  ;;
val print_string_list : string list -> unit = <fun>
```

```
# let rec print_int_list l =
    match l with
    | [] -> ()
    | x :: xs -> print_int x; print_int_list xs
  ;;
val print_int_list : int list -> unit = <fun>
```

Ordre supérieur et polymorphisme

```
# let rec iter f l =
    match l with
    | [] -> ()
    | x :: xs -> f x; iter f xs
  ;;
val iter : ('a -> 'b) -> 'a list -> unit = <fun>
# let print_string_list l =
    iter print_string l;;
val print_string_list : string list -> unit = <fun>
# let print_int_list l =
    iter print_int l;;
val print_int_list : int list -> unit = <fun>
```