TP 2 de programmation fonctionnelle en Objective Caml

Christian Rinderknecht

4 février 2015

L'objectif est de présenter la curryfication et la possibilité de l'évaluation partielle des fonctions qu'il implique, puis le filtrage et les paramètres fonctionnels. Les exemples relèvent du calcul symbolique.

1 Curryfication

La curryfication est la correspondance entre une fonction à n arguments et une fonction à un argument, lequel est un n-uplet. Par exemple, une fonction qui ajoute deux entiers peut s'écrire

```
let ajoute x y = x + y
```

ou

```
let somme (x, y) = x + y
```

La première forme est dite *curryfiée*, et la deuxième *non curryfiée*.

La première prend deux arguments qui sont chacun de type `int`. Elle a le type `int` → `int` → `int`. L'opérateur → est associatif à droite, c'est-à-dire que ce type doit être lu `int` → (`int` → `int`). Nous avons donc une fonction qui prend en entier en argument, et renvoie une fonction des entiers vers les entiers. Par exemple,

```
let ajoute2 = ajoute 2
```

définit une fonction qui ajoute 2 à son argument ; elle a le type `int` → `int`. Il en découle que

```
ajoute2 3
```

a le type `int` : cette expression est un entier.

La seconde prend un argument, qui est une paire d'entiers. Elle a donc le type `int` × `int` → `int`. L'opérateur × a priorité sur l'opérateur →, c'est-à-dire que ce type doit être lu (`int` × `int`) → `int`. Par exemple,

```
somme (2, 3)
```

a le type `int`. Il est impossible de réaliser une application partielle comme dans le premier cas. L'expression `somme 2` (qui est équivalente à `somme(2)`) provoque une erreur de typage car la fonction `somme` attend une paire d'entiers et reçoit un entier à la place.

```
# somme 2;;
        ^
This expression has type int but is here used with type int * int
```

La version curryfiée est donc plus flexible, puisqu'elle autorise l'application partielle. Pour cette raison, elle sera souvent préférée.

— Écrire deux fonctions `curry` et `uncurry`. La première prend une fonction `f` sous forme non curryfiée et renvoie sa forme curryfiée ; la seconde fait le contraire.

2 Paramètres fonctionnels

— Écrire une fonction `fun_prod` qui prend deux fonctions f et g en argument et renvoie une fonction des paires dans les paires, qui applique f à la première composante et g à la seconde.

— Écrire une fonction `iter` qui prend un entier n, une fonction f, un entier x et calcule $f^n(x)$.

— Application : écrire une fonction `power` qui, étant donnés deux entiers m et n, calcule m^n.

— Application (bis). Soit $(F_n)_{n \in \mathbb{N}}$ la suite de Fibonnacci définie par

$$\forall n \in \mathbb{N}^+, F_n = F_{n-1} + F_{n-2}; F_0 = F_1 = 1$$

1. Écrire le programme récursif calqué sur la définition mathématique.
2. Calculer le nombre d'appels récursifs nécessaires pour calculer F_n.
3. Pour diminuer le temps d'exécution, calculer la fonction f telle que $(F_{n+2}, F_{n+1}) = f(F_{n+1}, F_n)$ et en déduire une définition de f dont le coût est linéaire en n.

— Écrire une fonction `iter_prod` dont les paramètres sont f, g et n, et calcule une fonction des paires dans les paires, qui à (x, y) associe $(f^n(x), g^n(y))$. On utilisera `fun_prod` et `iter`. On écrira `iter_prod` de deux façons différentes (nommer `iter_prod_bis` la variante).

— Écrire une fonction `loop` telle que la valeur de `loop` $f\,p\,z$ soit le premier élément de la suite $(f^k(z))_{k \in N}$ qui satisfasse le prédicat p (un prédicat est une fonction à valeurs dans les booléens.).

— Utiliser `loop` pour écrire une fonction `modulo`, à deux arguments entiers positifs x et y, qui calcule $x \bmod y$.

— Réécrire la fonction `iter` (nommer `iter_bis` la variante) à partir de la fonction `loop`. Pour cela, calculer la suite $(f^k(x), n - k)_{k \geq 0}$ et s'arrêter lorsque la deuxième composante s'annule. On obtient donc la paire $(f^n(x), 0)$, dont on extrait la première composante.

3 Filtrage

Le filtrage permet de faire un choix en fonction de la *forme* (i.e. de la *structure*) d'une valeur. Un filtre est une expression. Sa syntaxe est

`match` e `with` p_1 `->` e_1 `|` p_2 `->` e_2 `|` ... `|` p_n `->` e_n

L'évaluation de cette expression débute par le calcul de la valeur v de l'expression e. Ensuite, on compare, dans l'ordre, la forme de v aux différents motifs. Si p_i est le premier motif qui *filtre* v, alors le résultat renvoyé sera la valeur de e_i. Cette construction correspond aux définitions par cas des fonctions mathématiques. Exemple (Fibonacci, méthode naïve) :

```
let rec fib n =
  match n with
    0 -> 1
  | 1 -> 1
  | _ -> fib(n-1) + fib(n-2)
```

Un motif peut contenir des variables, qui sont alors *liées* à la sous-valeur qu'elles filtrent. Exemple :

```
let implique bool1 bool2 =
  match (bool1, bool2) with
    (true, x) -> x
  | (false, _) -> true
```

Cette fonction prend deux booléens en argument et indique si le premier implique le second. Le `match` porte sur la paire `(bool1, bool2)`, ce qui permet de prendre en compte les deux valeurs à la fois si on le désire. La première ligne du `match` s'applique lorsque `bool1` vaut `true` ; on attribue alors le nom `x` au deuxième composant de la paire (`x` est liée à la valeur de `bool2`) et on renvoie `x`, *qui est liée par le motif* (à la valeur de `bool2`). La deuxième ligne s'applique lorsque `bool1` vaut `false`.

Une variable ne peut apparaître qu'une seule fois par motif. De plus, un motif ne peut pas faire référence à des variables définies précédemment. Dans les deux cas, cela demanderait d'effectuer des comparaisons implicites, ce que le compilateur refuse. Par exemple,

```
let equal p =
  match p with
    (x, x) -> true
  | _ -> false
```

est incorrect parce que `x` apparaît deux fois ;

```
let equal x y =
  match y with
    x -> true
  | _ -> false
```

est correcte, mais *renvoie toujours true*. En effet, `match y with x -> ...` ne signifie pas « si y est égal à x alors... » mais « si y est de la forme x alors... », ce qui est toujours vrai puisqu'une variable de motif (ici le second `x`) est toujours de la forme de n'importe qu'elle valeur (ici la valeur résultant de l'évaluation de la variable `y`). C'est là la différence entre *comparaison* et *filtrage*.

— Écrire la fonction `factorielle` avec un filtrage par motifs.
— Écrire la fonction transposée (`transpose`) d'une matrice carrée de dimension 2 représentée par une paire de paires de flottants.
— Écrire une fonction `prod` qui calcule le produit de deux telles matrices.
— Écrire une fonction `is_const` qui teste si une telle matrice est constante, c.-à-d. possède quatre composantes égales.

— Écrire une fonction `trig_sup` qui teste si une telle matrice est triangulaire supérieure.