Corrigé du TP 2 de programmation fonctionnelle en Objective Caml

Christian Rinderknecht

3 février 2014

1 Curryfication

```
let curry f = fun x y -> f (x, y);;
let uncurry f = fun (x, y) -> f x y;;
```

2 Paramètres fonctionnels

```
let fun_prod f g = fun (x, y) -> (f x, g y);;
```

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

On aurait pu écrire la dernière ligne : `iter (n-1) f (f x)`

La version naïve de Fibonnacci est

```
let rec fib n = if n < 2 then 1 else fib (n-1) + fib (n-2);;
```

Soit A_n le nombre d'appels récursifs pour calculer `fib(n)`. Alors nous avons

$$A_0 = A_1 = 1 \text{ et } \forall n > 1, A_n = 1 + A_{n-1} + A_{n-2}$$

Posons $B_n = A_n + 1$ et il vient :

$$B_0 = B_1 = 2 \text{ et } \forall n > 1, B_n = B_{n-1} + B_{n-2}$$

On prouve par récurrence sur n que $\forall n \in \mathbb{N}, B_n = 2F_n$. Par conséquent $A_n = 2F_n - 1$. Le temps d'exécution de `fib(n)` est donc asymptotiquement une exponentielle de n.

Si l'on pose $f : (x, y) \mapsto (x + y, x)$, alors $(F_{n+2}, F_{n+1}) = f(F_{n+1}, F_n)$.

La suite (F_{n+1}, F_n) pour tout $n \in \mathbb{N}$ vaut donc $(F_{n+1}, F_n) = f^n(F_1, F_0)$, soit $(F_{n+1}, F_n) = f^n(1, 1)$. Nous pouvons alors la programmer directement et efficacement (le nombre d'appels est linéaire en n maintenant) :

```
let fib n = fst (iter n (fun (x,y) -> (x+y,x)) (1,1));;
```

La fonction `fst` est prédéfinie mais peut se définir trivialement par :

```
let fst (x,y) = x;;

let power m n = iter n (fun x -> m * x) 1;;

let fib n = fst (iter n (fun (x,y) -> (x+y,x)) (1,0));;

let iter_prod f g n = iter n (fun_prod f g);;

let iter_prod_bis f g n = fun_prod (iter n f) (iter n g);;

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

let rec modulo x y = loop (fun v -> v - y) (fun v -> v < y) x;;

let iter_bis n f x =
  fst (loop (fun_prod f pred)
            (fun (_, k) -> k = 0)
            (x, n))
```

Dans la fonction `iter_bis`, on utilise `loop` pour calculer la suite $(f^k(x), n-k)_{k\geq 0}$ et on s'arrête lorsque la deuxième composante s'annule. On obtient donc la paire $(f^n(x), 0)$, dont on extrait la première composante grâce à `fst`. Cette fonction est prédéfinie mais peut se définir trivialement, ainsi que `pred` qui calcule le prédécesseur d'un entier par :

```
let fst (x,y) = x
let pred x = x - 1
```

3 Filtrage

Factorielle :

```
let rec fact n = match n with
  0 -> 1
| _ -> n * fact(n-1)
;;
```

ou bien

```
let rec fact n =
  match n with
    0 -> 1
  | _ -> n * fact(n-1)
;;
```

Opérations sur les matrices :

```
let transpose = fun ((x, y), (z, t)) -> ((x, z), (y, t));;

let prod ((x, y), (z, t)) ((x', y'), (z', t')) =
  ((x *. x' +. y *. z', x *. y' +. y *. t'),
   (z *. x' +. t *. z', z *. y' +. t *. t'))
```

```
;;

let is_const ((x, y), (z, t)) = (x = y) & (y = z) & (z = t)
;;
```

On ne peut pas utiliser un simple filtrage pour déterminer si les quatre éléments sont égaux (cf. énoncé).

```
let trig_sup m =
  match m with
    (_, (0.0, _)) -> true
  | _ -> false
;;
```