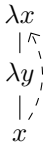
Programmation Avancée
Partiel mai 2015 : corrigé succinct

Documents de cours et TD autorisés – Durée : 3h.
Le barème est donné à titre indicatif (1 pt ≈ 9 min).
Les différents exercices sont indépendants et peuvent être traités dans n'importe quel ordre.

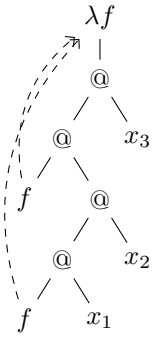
Exercice 1 : λ-calcul (3 pts)

1/ Dessiner l'arbre syntaxique du terme $u = \lambda xy.x$. La variable x est-elle libre ou liée dans u ?
2/ Dessiner l'arbre syntaxique du terme $t = \lambda f.((f\ ((f\ x_1)\ x_2))\ x_3)$. Quelles sont ses variables libres et quelles sont ses variables liées ?
3/ Réduire le terme $(t\ u)$ en forme normale. Bien faire apparaître les étapes de β-réduction.

Corrigé

1. 

x est liée par le λx.

2. 

f est liée. Les autres variables sont libres.

3.

$$\begin{aligned}(t\ u) =&(\lambda f.((f\ ((f\ x_1)\ x_2))\ x_3))(\lambda xy.x)\\ \to&(\lambda xy.x)\ (((\lambda xy.x)\ x_1)\ x_2)\ x_3\\ \to&(\lambda xy.x)\ ((\lambda y.x_1)\ x_2)\ x_3\\ \to&(\lambda xy.x)\ x_1\ x_3\\ \to&(\lambda y.x_1)\ x_3\\ \to&x_1\end{aligned}$$

Exercice 2 : Liste d'associations (3 pts + 1 bonus)

On considère ici des listes d'associations dont le deuxième élément est un entier, c'est-à-dire des listes de type `('a * int) list`.
1/ Écrire une fonction `add_to_key : 'a -> ('a * int) list -> ('a * int) list` qui ajoute 1 à l'entier associé à la clé donnée en argument. Par exemple,
`add_to_key "to" ["ti",2 ; "to",3 ; "ta",5]` renvoie `["to",4 ; "ti",2 ; "ta",5]`
Si la clé n'apparaît pas dans la liste initiale, on considère qu'elle est associée à `0`, par exemple
`add_to_key "to" ["ti",2 ; "ta",5]` renvoie `["to",1 ; "ti",2 ; "ta",5]`

2/ Écrire une fonction `count_occur : 'a list -> ('a * int) list` qui compte le nombre d'occurrences de chaque élément de la liste. Par exemple,
`count_occur ["to";"to";"ta";"ti";"to";"ti"]` renvoie `["ti",2 ; "to",3 ; "ta",1]`
(point de bonus pour une utilisation correcte d'un `List.fold`)

Corrigé

1.
```
let add_count c l =
  try
    let n = List.assoc c l in
      (c,n+1)::(List.remove_assoc c l)
  with
      Not_found -> (c,1)::l
```
Ou alors directement :
```
let rec add_count c =
  function
  | [] -> [(c,1)]
  | (c',i)::l when c = c' -> (c,i+1)::l
  | (c',i)::l -> (c',i)::(add_count c l)
```

2.
```
let count_occur =
  let rec go acc =
    function
    | [] -> acc
    | c::l -> go (add_count acc c) l
  in go  []
```
qui est récursif terminal.
Avec un `List.fold` :
```
let count_occur l =
  List.fold_right add_count l []
```

Attention. L'idée d'écrire une fonction `count_key k l` qui compte le nombre de `k` dans `l`, puis d'itérer cette fonction (appel récursif `x::xs -> (x,count_key x l)::(count_occur xs)` ou `List.fold`) ne marche pas. En effet, `count_key` est appelé *pour chaque* occurrence de la clé qui apparaît alors plusieurs fois dans le résultat.

Exercice 3 : map/reduce (3 pts)

Pour les questions de cet exercice, il est interdit d'utiliser `let rec`.

Il faut utiliser les itérateurs `List.map`, `List.fold_left`, `List.fold_right`, `List.combine`, `List.filter`, `List.for_all`, ...

On représente un brin d'ADN par une liste de caractères parmi les nucléotides `'A'`, `'C'`, `'G'`, ou `'T'` et un brin d'ARN en utilisant `'A'`, `'C'`, `'G'`, et `'U'`.

1/ Pour transcrire un brin d'ADN en brin d'ARN on le recopie en remplaçant les `'T'` par des `'U'`. Écrire une fonction `transcrire` qui transcrit un brin d'ADN passé en argument.

2/ Le contenu GC d'un brin d'ADN est la proportion de `'C'` ou `'G'` parmi les nucléotides de la séquence. Écrire une fonction `contenu_gc` qui retourne sous forme de flottant le contenu GC d'un brin d'ADN passé en argument.

Indication : utiliser `List.filter` pour compter le nombre de `'C'`.

3/ La distance de Hamming entre deux brins d'ADN de même longueur est le nombre de points de mutation entre la première et la deuxième séquence. C'est à dire, en alignant les séquences l'une au dessus de l'autre, le nombre de colonnes où les nucléotides différent. Écrire une fonction `hamming` qui prend en argument deux brins d'ADN de même longueur et retourne leur distance de Hamming.

Indication : utiliser `List.combine` puis compter le nombre de couples avec des éléments différents.

Corrigé

1.
```
let transfo c =
  match c with
  | 'T' -> 'U'
  | c -> c
```

```
let transcrire adn = List.map transfo adn
(* ou, par eta-contraction : *)
let transcrire' = List.map transfo
```

2.
```
let same_char c d = (c=d)
let count_char c l = List.length (List.filter (same_char c))

let contenu_gc adn =
    (float ((count_char 'C' adn) + (count_char 'G' adn))) /.
    (float (List.length l))
```
ou, pour éviter de parcourir la liste deux fois :
```
let is_gc c = (c='C')||(c='G')

let count_gc l = List.length (List.filter is_gc l)
let contenu_gc l = (float (count_gc l)) /. (float (List.length l))
```

3.
```
let mutation (a,b) = (a<>b)

let hamming l1 l2 = List.length (List.filter mutation (List.combine l1 l2))
```
ou, avec `List.map2` et `List.fold` :
```
let mutation a b = if a<>b then 1 else 0

let hamming l1 l2 = List.fold_right (+) (List.map2 mutation l1 l2) 0
```
ou, directement avec `List.fold_right2` :
```
let add_mutation a b n = if a<>b then n+1 else n

let hamming l1 l2 = List.fold_right2 add_mutation l1 l2 0
(* et directement avec une fonction anonyme : *)
let hamming' l1 l2 =
   List.fold_right2 (fun a b n -> if a<>b then n+1 else n) l1 l2 0
```

Exercice 4 : Récursion terminale : le retour de Fibonacci (4 pts)

La suite de Fibonacci, F_n est définie par

$$\begin{cases} F_n = 1 & \text{si } n \leq 1 \\ F_n = F_{n-1} + F_{n-2} & \text{sinon} \end{cases}$$

1/ Écrire une fonction récursive `fibo : int -> int` qui calcule F_n.

On défini aussi la suite de Fibonacci modifiée comme une suite de couples $F'_n = (a_n, b_n)$ par

$$\begin{cases} F'_n = (a_n, b_n) = (1, 1) & \text{si } n = 1 \\ F'_n = (a_n, b_n) = (b_{n-1}, a_{n-1} + b_{n-1}) & \text{sinon} \end{cases}$$

On remarque que $F'_n = (F_{n-1}, F_n)$.

2/ En utilisant les projections de Ocaml (`fst` et `snd`), exprimer F'_n en fonction de F'_{n-1} sans utiliser a_n ni b_n.

C'est-à-dire, compléter $F'_n = \ldots$ en utilisant uniquement F'_{n-1}, `fst` et `snd`.

3/ Écrire une fonction récursive `fibo' : int -> (int * int)` qui calcule F'_n.

4/ Écrire une fonction récursive terminale `fibo_aux : int -> (int * int) -> (int * int)` qui calcule F'_n de manière récursive terminale.

5/ En utilisant `fibo_aux`, en déduire une fonction qui calcule F_n de manière récursive terminale.

Corrigé

1. ```
   let rec fibo n =
     if n < 2
     then 1
     else (fibo (n-1)) + (fibo (n-2))
   ```
2. $F'_n = (\texttt{snd}\ F'_{n-1}, (\texttt{fst}\ F'_{n-1}) + (\texttt{snd}\ F'_{n-1}))$
3. ```
   let rec fibo' n =
     if n=1
     then (1,1)
     else let f=fibo' (n-1) in (snd f, (fst f)+(snd f))
   ```
   Attention, si on utilise pas le `let f` mais qu'on fait à la place 3 appels récursifs, on a une fonction qui calcule ce qu'il faut mais avec une complexité exponentielle au lieu de linéaire.
4. ```
   let rec fibo_aux n acc =
     if n = 1
     then acc
     else fibo_aux (n-1) (snd acc, (fst acc)+(snd acc))
   ```
5. ```
   let fibo_tr n = snd (fibo_aux n)
   ```

Exercice 5 : Expressions (9 pts)

On considère ici les expressions arithmétiques construites à partir de nombres réels. Elles sont représentées par le type

```
type expr =
  | Var of string       (* variable       *)
  | Num of Float        (* nombre         *)
  | Add of expr * expr  (* addition       *)
  | Sub of expr * expr  (* soustraction   *)
  | Mul of expr * expr  (* multiplication *)
  | Div of expr * expr  (* division       *)
```

Évaluation

1/ Comment est représenté en Caml l'expression $3 \times x + y$?
2/ Écrire une fonction `eval0 : expr -> float` qui évalue une expression sans variables (on lèvera une exception si on doit évaluer une variable).
3/ Comment peut-on gérer les variables pour évaluer une expression ? Donner le type Caml correspondant et expliquer son fonctionnement.
4/ Écrire une fonction `eval` qui évalue une expression contenant des variables.

Dérivation

On veut dériver une expression (par rapport à une variable x). On note $[\![\texttt{E}]\!]_x$ la dérivée[1] de l'expression `E` par rapport à la variable x et on rappelle les règles de dérivation $[\![\bullet]\!]_x$:

- $[\![x]\!]_x = 1$
- $[\![y]\!]_x = 0$ (où y est une variable différente de x)
- $[\![r]\!]_x = 0$ (où r est un nombre quelconque)
- $[\![\texttt{E} + \texttt{F}]\!]_x = [\![\texttt{E}]\!]_x + [\![\texttt{F}]\!]_x$
- $[\![\texttt{E} - \texttt{F}]\!]_x = [\![\texttt{E}]\!]_x - [\![\texttt{F}]\!]_x$
- $[\![\texttt{E} \times \texttt{F}]\!]_x = ([\![\texttt{E}]\!]_x \times \texttt{F}) + (\texttt{E} \times [\![\texttt{F}]\!]_x)$
- $[\![\frac{\texttt{E}}{\texttt{F}}]\!]_x = \frac{([\![\texttt{E}]\!]_x \times \texttt{F}) - (\texttt{E} \times [\![\texttt{F}]\!]_x)}{\texttt{F} \times \texttt{F}}$

5/ Que vaut $[\![(x \times x) + y]\!]_x$? $[\![(x \times x) + y]\!]_y$?
6/ Écrire une fonction `derive : string -> expr -> expr` telle que `derive "x" E` renvoie $[\![\texttt{E}]\!]_x$.

1. Ce qui correspond à la notation mathématique $\frac{d}{dx}E$, mais qui est vu ici comme la *sémantique* de la dérivation.

Corrigé

1. `Add (Mul (Num 3., Var "x"), Var "y")`
2.
```
let rec eval0 e =
  match e with
  | Var _ -> failwith "Question suivante."
  | Num r -> r
  | Add (e1,e2) -> (eval0 e1) +. (eval0 e2)
  | Sub (e1,e2) -> (eval0 e1) -. (eval0 e2)
  | Mul (e1,e2) -> (eval0 e1) *. (eval0 e2)
  | Div (e1,e2) -> (eval0 e1) /. (eval0 e2)
```
3. Utiliser un environnement, représenté par une liste d'association de type `(string * float) list`.
4. Comme l'environnement ne change pas, on peut faire une fonction annexe facilement.

```
let eval expr env =
  let rec eval_aux e =
    match e with
    | Var s -> List.assoc s env
    | Num r -> r
    | Add (e1,e2) -> (eval_aux e1) +. (eval_aux e2)
    | Sub (e1,e2) -> (eval_aux e1) -. (eval_aux e2)
    | Mul (e1,e2) -> (eval_aux e1) *. (eval_aux e2)
    | Div (e1,e2) -> (eval_aux e1) /. (eval_aux e2)
  in eval_aux expr
```

5. $[\![x \times x + y]\!]_x = 2 \times x$. $[\![x \times x + y]\!]_y = 1$.
   En restant plus proche de la sémantique donnée, on trouve
   $[\![x \times x + y]\!]_x = 1 \times x + x \times 1 + 0$ et $[\![x \times x + y]\!]_y = 0 \times x + x \times 0 + 1$
6.
```
let rec derive s expr =
  match expr with
  | Var v when v=s -> Num 1.
  | Var v -> Num 0.
  | Num _ -> Num 0.
  | Add (e1,e2) -> Add (derive s e1,derive s e2)
  | Sub (e1,e2) -> Sub (derive s e1,derive s e2)
  | Mul (e1,e2) -> Add (Mul (derive s e1,e2), Mul (e1,derive s e2))
  | Div (e1,e2) -> Div(Sub (Mul derive s e1,e2, Mul (e1,derive s e2)),
                       Mul (e2, e2))
```