Lambda_Calcul_en_OCaml

May 21, 2019

1 Table of Contents

2 Lambda-calcul implémenté en OCaml

Ce notebook est inspiré de ce post de blog du Professeur Matt Might, qui implémente un mini langage de programmation en λ-calcul, en Python. Je vais faire la même chose en OCaml.

2.1 Expressions

On rappelle que les expressions du Lambda calcul, ou λ-calcul, sont les suivantes :

$$\begin{cases} x, y, z & \text{(des variables)} \\ uv & \text{(application de deux termes } u, v \text{)} \\ \lambda x.v & \text{(lambda-function prenant la variable } x \text{ et le terme } v \text{)} \end{cases}$$

2.2 But ?

Le but ne va pas être de les représenter comme ça avec des types formels en Caml, mais plutôt d'utiliser les constructions de Caml, respectivement `u(v)` et `fun x -> v` pour l'application et les fonctions anonymes, et encoder des fonctionnalités de plus haut niveau dans ce langage réduit.

2.3 Grammaire

Avec une grammaire BNF, si `<var>` désigne un nom d'expression valide (on se limitera à des noms en miniscules consistitués des 26 lettres `a,b,..,z`) :

```
<exp> ::= <var>
| <exp>(<exp>)
| fun <var> -> <exp>
| (<exp>)
```

2.4 L'identité

```
In [3]: let identite = fun x -> x ;;

Out[3]: val identite : 'a -> 'a = <fun>


In [4]: let vide = fun x -> x ;;

Out[4]: val vide : 'a -> 'a = <fun>
```

2.5 Conditionnelles

La conditionnelle est `si cond alors valeur_vraie sinon valeur_fausse`.

```
In [13]: let si = fun cond valeur_vraie valeur_fausse -> cond valeur_vraie valeur_fausse ;;

Out[13]: val si : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

C'est très simple, du moment qu'on s'assure que `cond` est soit `vrai` soit `faux` tels que définis par leur comportement :

```
si vrai e1 e2 == e1
si faux e1 e2 == e2
```

```
In [14]: let vrai = fun valeur_vraie valeur_fausse -> valeur_vraie ;;
         let faux = fun valeur_vraie valeur_fausse -> valeur_fausse ;;

File "[14]", line 1, characters 28-41:
Warning 27: unused variable valeur_fausse.
```

```
Out[14]: val vrai : 'a -> 'b -> 'a = <fun>


File "[14]", line 2, characters 15-27:
Warning 27: unused variable valeur_vraie.


Out[14]: val faux : 'a -> 'b -> 'b = <fun>
```

La négation est facile !

```
In [15]: let non = fun v x y -> v y x;;

Out[15]: val non : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c = <fun>
```

En fait, on va forcer une évaluation paresseuse, comme ça si l'une des deux expressions ne terminent pas, l'évaluation fonctionne quand même.

```
In [16]: let vrai_paresseux = fun valeur_vraie valeur_fausse -> valeur_vraie () ;;
         let faux_paresseux = fun valeur_vraie valeur_fausse -> valeur_fausse () ;;

File "[16]", line 1, characters 38-51:
Warning 27: unused variable valeur_fausse.


Out[16]: val vrai_paresseux : (unit -> 'a) -> 'b -> 'a = <fun>


File "[16]", line 2, characters 25-37:
Warning 27: unused variable valeur_vraie.


Out[16]: val faux_paresseux : 'a -> (unit -> 'b) -> 'b = <fun>
```

Pour rendre paresseux un terme, rien de plus simple !

```
In [17]: let paresseux = fun f -> fun () -> f ;;

Out[17]: val paresseux : 'a -> unit -> 'a = <fun>
```

2.6 Nombres

La représentation de Church consiste a écrire n comme $\lambda f.\lambda z.f^n z$.

```
In [18]: type 'a nombres = ('a -> 'a) -> 'a -> 'a;;  (* inutilisé *)
         type entiers_church = (int -> int) -> int -> int;;
```

```
Out[18]: type 'a nombres = ('a -> 'a) -> 'a -> 'a
```

```
Out[18]: type entiers_church = (int -> int) -> int -> int
```

0 est trivialement $\lambda f.\lambda z.z$:

```
In [34]: let zero = fun (f : ('a -> 'a)) (z : 'a) -> z ;;
```

```
File "[34]", line 1, characters 16-17:
Warning 27: unused variable f.
```

```
Out[34]: val zero : ('a -> 'a) -> 'a -> 'a = <fun>
```

1 est $\lambda f.\lambda z.fz$:

```
In [35]: let un = fun (f : ('a -> 'a)) -> f ;;
```

```
Out[35]: val un : ('a -> 'a) -> 'a -> 'a = <fun>
```

Avec l'opérateur de composition, l'écriture des entiers suivants est facile.

```
In [36]: let compose = fun f g x -> f (g x);;
```

```
Out[36]: val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

```
In [37]: let deux = fun f -> compose f f;;  (* == compose f (un f) *)
         let trois = fun f -> compose f (deux f) ;;
         let quatre = fun f -> compose f (trois f) ;;
         (* etc *)
```

```
Out[37]: val deux : ('a -> 'a) -> 'a -> 'a = <fun>
```

```
Out[37]: val trois : ('a -> 'a) -> 'a -> 'a = <fun>
```

```
Out[37]: val quatre : ('a -> 'a) -> 'a -> 'a = <fun>
```

On peut généraliser ça, avec une fonction qui transforme un entier (`int`) de Caml en un entier de Church :

```
In [38]: let rec entierChurch (n : int) =
             fun f z -> if n = 0 then z else f ((entierChurch (n-1)) f z)
         ;;
```

```
Out[38]: val entierChurch : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

Par exemple :

```
In [39]: (entierChurch 0) (fun x -> x + 1) 0;; (* 0 *)
         (entierChurch 7) (fun x -> x + 1) 0;; (* 7 *)
         (entierChurch 3) (fun x -> 2*x) 1;; (* 8 *)

Out[39]: - : int = 0


Out[39]: - : int = 7


Out[39]: - : int = 8
```

Et une fonction qui fait l'inverse (note : cette fonction n'est *pas* un λ-terme) :

```
In [40]: let entierNatif c : int =
             c (fun x -> x + 1) 0
         ;;

Out[40]: val entierNatif : ((int -> int) -> int -> int) -> int = <fun>
```

Un petit test :

```
In [41]: entierNatif (si vrai zero un);; (* 0 *)
         entierNatif (si faux zero un);; (* 1 *)

Out[41]: - : int = 0


Out[41]: - : int = 1


In [42]: entierNatif (entierChurch 100);; (* 100 *)

Out[42]: - : int = 100
```

2.7 Test d'inégalité

On a besoin de pouvoir tester si $n \leq 0$ (ou $n = 0$) en fait.

```
In [43]: (* prend un lambda f lambda z. ... est donne vrai ssi n = 0 ou faux sinon *)
         let estnul = fun n -> n (fun z -> faux) (vrai);;
```

```
File "[43]", line 2, characters 29-30:
Warning 27: unused variable z.
```

```
Out[43]: val estnul : (('a -> 'b -> 'c -> 'c) -> ('d -> 'e -> 'd) -> 'f) -> 'f = <fun>
```

```
In [44]: (* prend un lambda f lambda z. ... est donne vrai ssi n > 0 ou faux sinon *)
         let estnonnul = fun n -> n (fun z -> vrai) (faux);;
```

```
File "[44]", line 2, characters 32-33:
Warning 27: unused variable z.
```

```
Out[44]: val estnonnul : (('a -> 'b -> 'c -> 'b) -> ('d -> 'e -> 'e) -> 'f) -> 'f =
           <fun>
```

On peut proposer cette autre implémentation, qui "fonctionne" pareil (au sens calcul des β-réductions) mais est plus compliquée :

```
In [45]: let estnonnul2 = fun n -> non (estnul n);;
```

```
Out[45]: val estnonnul2 :
           (('a -> 'b -> 'c -> 'c) -> ('d -> 'e -> 'd) -> 'f -> 'g -> 'h) ->
           'g -> 'f -> 'h = <fun>
```

```
In [46]: entierNatif (si (estnul zero) zero un);; (* 0 *)
         entierNatif (si (estnul un)   zero un);; (* 1 *)
         entierNatif (si (estnul deux) zero un);; (* 1 *)
```

```
Out[46]: - : int = 0
```

```
Out[46]: - : int = 1
```

```
Out[46]: - : int = 1
```

```
In [47]: entierNatif (si (estnonnul zero) zero un);; (* 0 *)
         entierNatif (si (estnonnul un)   zero un);; (* 1 *)
         entierNatif (si (estnonnul deux) zero un);; (* 1 *)
```

```
Out[47]: - : int = 1
```

```
Out[47]: - : int = 0
```

```
Out[47]: - : int = 0


In [48]: entierNatif (si (non (estnul zero)) zero un);; (* 0 *)
         entierNatif (si (non (estnul un))   zero un);; (* 1 *)
         entierNatif (si (non (estnul deux)) zero un);; (* 1 *)

Out[48]: - : int = 1


Out[48]: - : int = 0


Out[48]: - : int = 0
```

2.8 Successeurs

Vue la représentation de Churc, $n+1$ consiste a appliquer l'argument f une fois de plus : $f^{n+1}(z) = f(f^n(z))$.

```
In [49]: let succ = fun n f z -> f ((n f) z) ;;

Out[49]: val succ : (('a -> 'b) -> 'c -> 'a) -> ('a -> 'b) -> 'c -> 'b = <fun>


In [50]: entierNatif (succ un);; (* 2 *)

Out[50]: - : int = 2


In [51]: deux;;
         succ un;;

Out[51]: - : ('a -> 'a) -> 'a -> 'a = <fun>


Out[51]: - : ('_a -> '_a) -> '_a -> '_a = <fun>
```

On remarque qu'ils ont le même typage, mais OCaml indique qu'il a moins d'informations à propos du deuxième : ce `'_a` signifie que le type est *contraint*, il sera fixé dès la première utilisation de cette fonction.

C'est assez mystérieux, mais il faut retenir le point suivant : `deux` était écrit manuellement, donc le système a vu le terme en entier, il le connaît et saît que `deux = fun f -> fun x -> f (f x))`, pas de surprise. Par contre, `succ un` est le résultat d'une évaluation *partielle* et vaut `fun f z -> f ((deux f) z)`. Sauf que le système ne calcule pas tout et laisse l'évaluation partielle ! (heureusement !)

Si on appelle `succ un` à une fonction, le `'_a` va être contraint, et on ne pourra pas s'en reservir :

```
In [54]: let succ_de_un = succ un;;

Out[54]: val succ_de_un : ('_a -> '_a) -> '_a -> '_a = <fun>


In [55]: (succ_de_un) (fun x -> x + 1);;

Out[55]: - : int -> int = <fun>


In [56]: (succ_de_un) (fun x -> x ^ "0");;


         File "[56]", line 1, characters 23-24:
    Error: This expression has type int but an expression was expected of type
             string
       1: (succ_de_un) (fun x -> x ^ "0");;



In [57]: (succ un) (fun x -> x ^ "0");;
         (* une valeur fraîchement calculée, sans contrainte *)

Out[57]: - : string -> string = <fun>
```

2.9 Prédecesseurs

Vue la représentation de Church, $\lambda n.n - 1$ n'existe pas... mais on peut tricher.

```
In [30]: let pred = fun n ->
             if (entierNatif n) > 0 then entierChurch ((entierNatif n) - 1)
             else zero
         ;;

Out[30]: val pred : ((int -> int) -> int -> int) -> ('a -> 'a) -> 'a -> 'a = <fun>


In [31]: entierNatif (pred deux);; (* 1 *)

Out[31]: - : int = 1


In [32]: entierNatif (pred trois);; (* 2 *)

Out[32]: - : int = 2
```

2.10 Addition

Pour ajouter n et m, il faut appliquer une fonction f n fois puis m fois : $f^{n+m}(z) = f^n(f^m(z))$.

```
In [33]: let somme = fun n m f z -> n(f)( m(f)(z));;

Out[33]: val somme : ('a -> 'b -> 'c) -> ('a -> 'd -> 'b) -> 'a -> 'd -> 'c = <fun>


In [34]: let cinq = somme deux trois ;;

Out[34]: val cinq : ('_a -> '_a) -> '_a -> '_a = <fun>


In [35]: entierNatif cinq;;

Out[35]: - : int = 5


In [36]: let sept = somme cinq deux ;;

Out[36]: val sept : (int -> int) -> int -> int = <fun>


In [37]: entierNatif sept;;

Out[37]: - : int = 7
```

2.11 Multiplication

Pour multiplier n et m, il faut appliquer le codage de n exactement m fois : $f^{nm}(z) = (f^n(f^n(...(f^n(z))...))$.

```
In [38]: let produit = fun n m f z -> m(n(f))(z);;

Out[38]: val produit : ('a -> 'b) -> ('b -> 'c -> 'd) -> 'a -> 'c -> 'd = <fun>
```

On peut faire encore mieux avec l'opérateur de composition :

```
In [39]: let produit = fun n m -> compose m n;;

Out[39]: val produit : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>


In [40]: let six = produit deux trois ;;

Out[40]: val six : ('_a -> '_a) -> '_a -> '_a = <fun>


In [41]: entierNatif six;;
```

```
Out[41]: - : int = 6


In [42]: let huit = produit deux quatre ;;

Out[42]: val huit : ('_a -> '_a) -> '_a -> '_a = <fun>


In [43]: entierNatif huit;;

Out[43]: - : int = 8
```

2.12 Paires

On va écrire un constructeur de paires, `paire a b` qui sera comme `(a, b)`, et deux destructeurs, `gauche` et `droite`, qui vérifient :

```
gauche (paire a b) == a
droite (paire a b) == b
```

```
In [75]: let paire = fun a b -> fun f -> f(a)(b);;

Out[75]: val paire : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c = <fun>


In [76]: let gauche = fun p -> p(fun a b -> a);;
         let droite = fun p -> p(fun a b -> b);;

File "[76]", line 1, characters 30-31:
Warning 27: unused variable b.


Out[76]: val gauche : (('a -> 'b -> 'a) -> 'c) -> 'c = <fun>


File "[76]", line 2, characters 28-29:
Warning 27: unused variable a.


Out[76]: val droite : (('a -> 'b -> 'b) -> 'c) -> 'c = <fun>


In [77]: entierNatif (gauche (paire zero un));;
         entierNatif (droite (paire zero un));;

Out[77]: - : int = 0


Out[77]: - : int = 1
```

2.13 Prédécesseurs, deuxième essai

Il y a une façon, longue et compliquée (source) d'y arriver, avec des paires.

```
In [78]: let pred n suivant premier =
             let pred_suivant = paire vrai premier in
             let pred_premier = fun p ->
                 si (gauche p)
                 (paire faux premier)
                 (paire faux (suivant (droite p)))
             in
             let paire_finale = n pred_suivant pred_premier in
             droite paire_finale
         ;;

Out[78]: val pred :
           ((((('a -> 'b -> 'a) -> 'c -> 'd) -> 'd) ->
            ((('e -> 'e -> 'e) ->
              ((('f -> 'g -> 'g) -> 'c -> 'h) -> 'h) ->
              ((('i -> 'j -> 'j) -> 'k -> 'l) -> 'l) -> 'm) ->
             'm) ->
            ('n -> 'o -> 'o) -> 'p) ->
           (((((('f -> 'g -> 'g) -> 'c -> 'h) -> 'h) ->
             ((('i -> 'j -> 'j) -> 'k -> 'l) -> 'l) -> 'm) ->
            'k) ->
           'c -> 'p = <fun>
```

Malheureusement, ce n'est pas bien typé.

```
In [79]: entierNatif (pred deux);; (* 1 *)


        File "[79]", line 1, characters 18-22:
    Error: This expression has type
             ((('a -> 'b -> 'a) -> 'c -> 'd) -> ('a -> 'b -> 'a) -> 'c -> 'd) ->
             (('a -> 'b -> 'a) -> 'c -> 'd) -> ('a -> 'b -> 'a) -> 'c -> 'd
           but an expression was expected of type
             ((('a -> 'b -> 'a) -> 'c -> 'd) -> 'd) ->
             ((('e -> 'e -> 'e) ->
               ((('f -> 'g -> 'g) -> 'c -> 'h) -> 'h) ->
               ((('i -> 'j -> 'j) -> 'k -> 'l) -> 'l) -> 'm) ->
              'm) ->
             ('n -> 'o -> 'o) -> 'p
           The type variable 'd occurs inside ('a -> 'b -> 'a) -> 'c -> 'd
       1: entierNatif (pred deux);; (* 1 *)
```

2.14 Listes

Pour construire des listes (simplement chaînées), on a besoin d'une valeur pour la liste vide, `listevide`, d'un constructeur pour une liste `cons`, un prédicat pour la liste vide `estvide`, un accesseur `tete` et `queue`, et avec les contraintes suivantes (avec `vrai`, `faux` définis comme plus haut) :

```
estvide (listevide) == vrai
estvide (cons tt qu) == faux

tete (cons tt qu) == tt
queue (cons tt qu) == qu
```

On va stocker tout ça avec des fonctions qui attendront deux arguments (deux fonctions - rappel tout est fonction en λ-calcul), l'une appellée si la liste est vide, l'autre si la liste n'est pas vide.

```
In [58]: let listevide = fun survide surpasvide -> survide;;

File "[58]", line 1, characters 28-38:
Warning 27: unused variable surpasvide.


Out[58]: val listevide : 'a -> 'b -> 'a = <fun>


In [59]: let cons = fun hd tl -> fun survide surpasvide -> surpasvide hd tl;;

Out[59]: val cons : 'a -> 'b -> 'c -> ('a -> 'b -> 'd) -> 'd = <fun>


File "[59]", line 1, characters 28-35:
Warning 27: unused variable survide.
```

Avec cette construction, `estvide` est assez simple : `survide` est `() -> vrai` et `surpasvide` est `tt qu -> faux`.

```
In [60]: let estvide = fun liste -> liste (vrai) (fun tt qu -> faux);;

File "[60]", line 1, characters 45-47:
Warning 27: unused variable tt.
File "[60]", line 1, characters 48-50:
Warning 27: unused variable qu.


Out[60]: val estvide : (('a -> 'b -> 'a) -> ('c -> 'd -> 'e -> 'f -> 'f) -> 'g) -> 'g =
           <fun>
```

Deux tests :

```
In [61]: entierNatif (si (estvide (listevide)) un zero);; (* estvide listevide == vrai *)
         entierNatif (si (estvide (cons un listevide)) un zero);; (* estvide (cons un listevi

Out[61]: - : int = 1


Out[61]: - : int = 0
```

Et pour les deux extracteurs, c'est très facile avec cet encodage.

```
In [62]: let tete = fun liste -> liste (vide) (fun tt qu -> tt);;
         let queue = fun liste -> liste (vide) (fun tt qu -> qu);;

File "[62]", line 1, characters 45-47:
Warning 27: unused variable qu.


Out[62]: val tete : (('a -> 'a) -> ('b -> 'c -> 'b) -> 'd) -> 'd = <fun>


File "[62]", line 2, characters 43-45:
Warning 27: unused variable tt.


Out[62]: val queue : (('a -> 'a) -> ('b -> 'c -> 'c) -> 'd) -> 'd = <fun>


In [69]: entierNatif (tete (cons un listevide));;
         entierNatif (tete (queue (cons deux (cons un listevide))));;
         entierNatif (tete (queue (cons trois (cons deux (cons un listevide)))));;

Out[69]: - : int = 1


Out[69]: - : int = 1


Out[69]: - : int = 2
```

Visualisons les types que Caml trouve a des listes de tailles croissantes :

```
In [70]: cons un (cons un listevide);;  (* 8 variables pour une liste de taille 2 *)

Out[70]: - : '_a ->
             ((('_b -> '_b) -> '_b -> '_b) ->
              ('_c ->
               ((('_d -> '_d) -> '_d -> '_d) -> ('_e -> '_f -> '_e) -> '_g) -> '_g) ->
              '_h) ->
             '_h
         = <fun>
```

```
In [71]: cons un (cons un (cons un (cons un listevide)));;  (* 14 variables pour une liste de

Out[71]: - : '_a ->
             ((('_b -> '_b) -> '_b -> '_b) ->
              ('_c ->
               ((('_d -> '_d) -> '_d -> '_d) ->
                ('_e ->
                 ((('_f -> '_f) -> '_f -> '_f) ->
                  ('_g ->
                   ((('_h -> '_h) -> '_h -> '_h) -> ('_i -> '_j -> '_i) -> '_k) -> '_k) ->
                  '_l) ->
                 '_l) ->
                '_m) ->
               '_m) ->
              '_n) ->
             '_n
           = <fun>


In [72]: cons un (cons un (cons un (cons un (cons un (cons un (cons un (cons un listevide))))))

Out[72]: - : '_a ->
             ((('_b -> '_b) -> '_b -> '_b) ->
              ('_c ->
               ((('_d -> '_d) -> '_d -> '_d) ->
                ('_e ->
                 ((('_f -> '_f) -> '_f -> '_f) ->
                  ('_g ->
                   ((('_h -> '_h) -> '_h -> '_h) ->
                    ('_i ->
                     ((('_j -> '_j) -> '_j -> '_j) ->
                      ('_k ->
                       ((('_l -> '_l) -> '_l -> '_l) ->
                        ('_m ->
                         ((('_n -> '_n) -> '_n -> '_n) ->
                          ('_o ->
                           ((('_p -> '_p) -> '_p -> '_p) -> ('_q -> '_r -> '_q) -> '_s) ->
                           '_s) ->
                          '_t) ->
                         '_t) ->
                        '_u) ->
                       '_u) ->
                      '_v) ->
                     '_v) ->
                    '_w) ->
                   '_w) ->
                  '_x) ->
                 '_x) ->
```

```
             '_y) ->
            '_y) ->
           '_z) ->
          '_z
      = <fun>
```

Pour ces raisons là, on se rend compte que le type donné par Caml à une liste de taille k croît linéairement *en taille* en fonction de k !

Aucun espoir donc (avec cet encodage) d'avoir un type générique pour les listes représentés en Caml.

Et donc nous ne sommes pas surpris de voir cet essai échouer :

```
In [68]: let rec longueur liste =
             liste (zero) (fun t q -> succ (longueur q))
         ;;


        File "[68]", line 2, characters 44-45:
    Error: This expression has type 'a but an expression was expected of type
             (('b -> 'b) -> 'b -> 'b) -> ('c -> 'a -> 'd) -> 'e
           The type variable 'a occurs inside
           (('b -> 'b) -> 'b -> 'b) -> ('c -> 'a -> 'd) -> 'e
       1: let rec longueur liste =
       2:     liste (zero) (fun t q -> succ (longueur q))
       3: ;;
```

En effet, `longueur` devrait être bien typée et `liste` et `q` devraient avoir le même type, or le type de `liste` est strictement plus grand que celui de `q`...

On peut essayer de faire une fonction `ieme`. On veut que `ieme zero liste = tete` et `ieme n liste = ieme (pred n) (queue liste)`.

En écrivant en haut niveau, on aimerait pouvoir faire :

```
In [87]: let pop liste =
             si (estvide liste) (listevide) (queue liste)
         ;;


        File "[87]", line 2, characters 42-47:
    Error: This expression has type
             ('a -> 'b -> 'a) ->
             ('c -> 'd -> 'e -> 'f -> 'f) -> ('g -> 'h -> 'g) -> 'i -> 'j
           but an expression was expected of type
             ('a -> 'a) -> ('k -> 'l -> 'l) -> 'm
           The type variable 'a occurs inside 'b -> 'a
       1: let pop liste =
       2:     si (estvide liste) (listevide) (queue liste)
```

```
3: ;;
```

```
In [86]: let ieme n liste =
             tete (n pop liste)
         ;;

File "[86]", line 2, characters 12-15:
Error: Unbound value pop
   1: let ieme n liste =
   2:     tete (n pop liste)
   3: ;;
```

2.15 La fonction U

C'est le premier indice que le λ-calcul peut être utilisé comme modèle de calcul : le terme $U : f \to f(f)$ ne termine pas si on l'applique à lui-même.

Mais ce sera la faiblesse de l'utilisation de Caml : ce terme ne peut être correctement typé !

```
In [55]: let u = fun f -> f (f);;

File "[55]", line 1, characters 19-22:
Error: This expression has type 'a -> 'b
       but an expression was expected of type 'a
       The type variable 'a occurs inside 'a -> 'b
   1: let u = fun f -> f (f);;
```

A noter que même dans un langage non typé (par exemple Python), on peut définir U mais son exécution échouera, soit à caude d'un dépassement de pile, soit parce qu'elle ne termine pas.

2.16 La récursion via la fonction Y

La fonction Y trouve le point fixe d'une autre fonction. C'est très utile pour définir des fonctions par récurrence.

Par exemple, la factorielle est le point fixe de la fonction suivante : "$\lambda f.\lambda n.1$ si $n \leq 0$ sinon $n * f(n-1)$" (écrite dans un langage plus haut niveau, pas en λ-calcul).

Y satisfait ces contraintes : $Y(F) = f$ et $f = F(f)$. Donc $Y(F) = F(Y(F))$ et donc $Y = \lambda F.F(Y(F))$. Mais ce premier essai ne marche pas.

```
In [56]: let rec y = fun f -> f (y(f));;

Out[56]: val y : ('a -> 'a) -> 'a = <fun>
```

```
In [57]: let fact = y(fun f n -> si (estnul n) (un) (produit n (f (pred n))));;


        File "[57]", line 1, characters 63-64:
    Error: This expression has type
             ((('a -> 'b -> 'a) -> 'c -> 'd -> 'e -> 'e) -> 'd -> 'e -> 'e) ->
             ((('f -> 'f -> 'f) ->
               ((('g -> 'h -> 'h) -> 'c -> 'i) -> 'i) ->
               ((('j -> 'k -> 'k) -> 'l -> 'm) -> 'm) -> 'n) ->
              'o ->
              ('f -> 'f -> 'f) ->
              ((('g -> 'h -> 'h) -> 'c -> 'i) -> 'i) ->
              ((('j -> 'k -> 'k) -> 'l -> 'm) -> 'm) -> 'n) ->
             (('p -> 'p) -> 'p -> 'p) -> 'q -> 'r
           but an expression was expected of type
             ((('a -> 'b -> 'a) -> 'c -> 'd -> 'e -> 'e) -> 'd -> 'e -> 'e) ->
             ((('f -> 'f -> 'f) ->
               ((('g -> 'h -> 'h) -> 'c -> 'i) -> 'i) ->
               ((('j -> 'k -> 'k) -> 'l -> 'm) -> 'm) -> 'n) ->
              'n) ->
             ('s -> 't -> 't) -> 'u
           The type variable 'n occurs inside
           'o ->
           ('f -> 'f -> 'f) ->
           ((('g -> 'h -> 'h) -> 'c -> 'i) -> 'i) ->
           ((('j -> 'k -> 'k) -> 'l -> 'm) -> 'm) -> 'n
      1: let fact = y(fun f n -> si (estnul n) (un) (produit n (f (pred n))));;
```

On utilise la η-expansion : si e termine, e est équivalent (ie tout calcul donne le même terme) à $\lambda x.e(x)$.

```
In [58]: let rec y = fun f -> f (fun x -> y(f)(x));;

Out[58]: val y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
```

Par contre, le typage n'arrive toujours pas à trouver que l'expression suivante devrait être bien définie :

```
In [59]: let fact = y(fun f n -> si (estnul n) (un) (produit n (f (pred n))));;


        File "[59]", line 1, characters 63-64:
    Error: This expression has type
             ((('a -> 'b -> 'a) -> 'c -> 'd -> 'e -> 'e) -> 'd -> 'e -> 'e) ->
             ((('f -> 'f -> 'f) ->
               ((('g -> 'h -> 'h) -> 'c -> 'i) -> 'i) ->
```

```
         ((('j -> 'k -> 'k) -> 'l -> 'm) -> 'm) -> 'n) ->
        'o ->
        ('f -> 'f -> 'f) ->
        ((('g -> 'h -> 'h) -> 'c -> 'i) -> 'i) ->
        ((('j -> 'k -> 'k) -> 'l -> 'm) -> 'm) -> 'n) ->
       (('p -> 'p) -> 'p -> 'p) -> 'q -> 'r
     but an expression was expected of type
       ((('a -> 'b -> 'a) -> 'c -> 'd -> 'e -> 'e) -> 'd -> 'e -> 'e) ->
       ((('f -> 'f -> 'f) ->
         ((('g -> 'h -> 'h) -> 'c -> 'i) -> 'i) ->
         ((('j -> 'k -> 'k) -> 'l -> 'm) -> 'm) -> 'n) ->
        'n) ->
       ('s -> 't -> 't) -> 'u
     The type variable 'n occurs inside
     'o ->
     ('f -> 'f -> 'f) ->
     ((('g -> 'h -> 'h) -> 'c -> 'i) -> 'i) ->
     ((('j -> 'k -> 'k) -> 'l -> 'm) -> 'm) -> 'n
1: let fact = y(fun f n -> si (estnul n) (un) (produit n (f (pred n))));;
```

2.17 Conclusion

Je n'ai pas réussi à traduire intégralement la prouesse initiale, écrite en Python, par Matt Might. Dommage, le typage de Caml est trop strict pour cet exercice.