# Aspects impératifs du langage Caml, continuation

Solange Coupet Grimal

Cours Logique, Déduction, Programmation 2002–2004*

## 1 Champs modifiables dans les enregistrements

**Enregistrements simples.** C'est un type produit dont les composantes peuvent avoir des types différents.

```
# type date = {mois:int;annee:int};;
type date = { mois : int; annee : int; }
```

`mois` et `annee` sont les étiquettes de l'enregistrement.

```
# let m = { mois=1; annee=1952 };;
val m : date = {mois = 1; annee = 1952}
# m.mois;;
- : int = 1
# type individu = { nom:string; arrivee:date};;
type individu = { nom : string; arrivee : date; }
# let toto = { nom="toto";
               arrivee={ mois=1; annee=1952 }
             };;
    val toto : individu = {nom = "toto"; arrivee = {mois = 1; annee = 1
# toto.arrivee.annee;;
- : int = 1952
```

On peut utiliser le filtrage sur les enregistrements :

```
# let bissextile = fun
    {mois = _; annee=x } -> x mod 4 = 0;;
  val bissextile : date -> bool = <fun>
# bissextile { mois=1; annee=1952};;
- : bool = true
# bissextile { mois=1; annee=1997};;
- : bool = false
```

*Texte révisé par Luigi Santocanale le 22 octobre 2005, et adapté au langage Objective Caml.



**Enregistrements à champs modifiables.** Certains champs de l'enregistrement peuvent être mutables. Ceci permet de modifier la valeur de ce champ, comme pour les vecteurs.

De même qu'il est possible de modifier des éléments d'un vecteur, il est possible de modifier la valeur des champs d'un enregistrement à condition qu'ils aient été déclarés de type mutable.

Par exemple, un point du plan peut être repéré par ses coordonnées ;

```
# type coord = { mutable abs:float; mutable ord:float};;
type coord = { mutable abs : float; mutable ord : float; }
```

Ceci est équivalent à la définition :

```
# type coord = {abs:float ref ; ord:float ref};;
type coord = { abs : float ref; ord : float ref; }
```

Les objets de ces 2 types sont strictement isomorphes. Les 2 champs contiennent des adresses. Seule la syntaxe diffère. Les déclarations avec champs mutables induisent une syntaxe plus simple que celles avec références, comme illustré par ce qui suit.

Définissons la translation de vecteur de coordonnées `(a,b)` :

```
# type coord = { mutable abs:float; mutable ord:float};;
type coord = { mutable abs : float; mutable ord : float; }
# let translation (a,b) pt =
  pt.abs <- pt.abs+.a; pt.ord <- pt.ord+.b;;
val translation : float * float -> coord -> unit = <fun>
# let point={abs=0. ; ord=3.};;
val point : coord = {abs = 0.; ord = 3.}
# translation (1.,1.) point;;
- : unit = ()
# point;;
- : coord = {abs = 1.; ord = 4.}
```

Faisons subir un changement à `p1` et examinons le résultat sur `p2` et `p3` définis ci-après :

```
# let p1 = {abs= 2.; ord= 2.};;
val p1 : coord = {abs = 2.; ord = 2.}
# let p2 = {abs= 2.; ord= 2.};;
val p2 : coord = {abs = 2.; ord = 2.}
# let p3 = p1;;
val p3 : coord = {abs = 2.; ord = 2.}
# translation (1.,1.) p1;;
- : unit = ()
# p1;;
- : coord = {abs = 3.; ord = 3.}
# p2;;
- : coord = {abs = 2.; ord = 2.}
# p3;;
- : coord = {abs = 3.; ord = 3.}
```

## 2 Exemple : les listes chaînées circulaires

Les listes chaînées sont constituées de cellules qui sont des enregistrements à 2 champs : l'un portant une information, l'autre l'adresse de la cellule suivante. Le second champ étant une adresse, il sera donc mutable.

```
# type 'a cellule={info:'a; mutable suiv:'a cellule};;
type 'a cellule = { info : 'a; mutable suiv : 'a cellule; }
```

La liste étant circulaire, la dernière cellule contiendra donc l'adresse de la première.

Création d'une cellule.

```
# #print_depth 5;;
# let rec toto = {info= 1; suiv=toto};;
val toto : int cellule =
  {info = 1;
   suiv =
    {info = 1;
     suiv =
      {info = 1;
       suiv = {info = 1; suiv = {info = 1; suiv = {info = ...; suiv = .
```

Création d'une liste circulaire

```
# let make_list e = let rec x={info=e;suiv=x} in x;;
val make_list : 'a -> 'a cellule = <fun>
# let l= make_list "d'amour";;
val l : string cellule =
  {info = "d'amour";
   suiv =
    {info = "d'amour";
     suiv =
      {info = "d'amour";
       suiv =
        {info = "d'amour";
         suiv = {info = "d'amour"; suiv = {info = ...; suiv = ...}}}}}}
```

L'idée est de repérer la liste par la dernière cellule. Ainsi, on peut faire des insertions en tête en rajoutant une cellule juste après la dernière. Les déclarations ci-dessous donnent les informations portées respectivement par la dernière et la première cellule d'une liste chaînée `l`.

```
# let last l = l.info;;
val last : 'a cellule -> 'a = <fun>
# let first l = l.suiv.info;;
val first : 'a cellule -> 'a = <fun>
```

On peut se servir de ces deux fonctions pour écrire une fonction pour imprimer une liste circulaire. On se servira de cette fonction pour afficher les listes circulaires dans l'environnement interactif de OCaml.



```
# let print_clist l =
  let rec print_clist_acc l beg  =
    if l.suiv == beg then print_endline (first l)
    else
      (print_string ((first l)^",");
       print_clist_acc l.suiv beg)
  in
    print_clist_acc l l;;
              val print_clist : string cellule -> unit = <fun>
# #install_printer print_clist;;
```

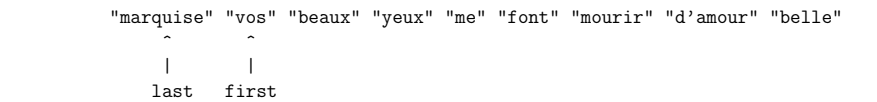
Insertion en tête :

```
# let insert l e =
  l.suiv <- {info=e;suiv=l.suiv}; l;;
  val insert : 'a cellule -> 'a -> 'a cellule = <fun>
# let l= insert l "mourir";;
mourir,d'amour
val l : string cellule =
# let l = insert l "font" in insert l "me";;
me,font,mourir,d'amour
- : string cellule =
# let l = insert l "yeux" in
let l= insert l "beaux" in
  insert l "vos";;
    vos,beaux,yeux,me,font,mourir,d'amour
- : string cellule =
```

Insertion en queue :

```
# let insert_tail l e = l.suiv <- {info=e;suiv=l.suiv}; l.suiv;;
val insert_tail : 'a cellule -> 'a -> 'a cellule = <fun>
(* Variante *)
# let insert_tail l e = let x = {info=e;suiv=l.suiv} in
  (l.suiv<- x;x);;
  val insert_tail : 'a cellule -> 'a -> 'a cellule = <fun>
# let l= insert_tail l "belle";;
vos,beaux,yeux,me,font,mourir,d'amour,belle
val l : string cellule =
# let l = insert_tail l "marquise";;
vos,beaux,yeux,me,font,mourir,d'amour,belle,marquise
val l : string cellule =
```

À cette étape, la liste possède cette structure :




Suppression de la tête :

```
# let elim_head l = l.suiv <- l.suiv.suiv; l;;
val elim_head : 'a cellule -> 'a cellule = <fun>
# elim_head l;;
beaux,yeux,me,font,mourir,d'amour,belle,marquise
- : string cellule =
```

Attention, ça n'élimine la tête que si `l` contient au moins 2 éléments.

```
# let make_list e = let rec x={info=e;suiv=x} in x;;
val make_list : 'a -> 'a cellule = <fun>
# let m = make_list 1;;
1
val m : int cellule =
# elim_head m;;
1
- : int cellule =
```

Question : que se passe-t'il si on ne redéfinit pas la fonction `make_list` ?

Comment reconstituer la liste chaînée `l` définie précédemment à partir de la liste :

```
["vos";"beaux";"yeux";"me";"font";"mourir";"d'amour";"belle";"marquise"] ?
```

On crée une liste chaînée `l` de premier élément `"vos"`. Puis on insère en queue chaque élément de la liste.

Rappel :

```
(List.fold_left f a [b1;...;bn]) renvoie (f....(f (f a b1) b2)...bn)
```

```
# let l= let x= make_list "vos" in
  List.fold_left insert_tail
    x
    ["beaux";"yeux";"me";"font";"mourir";"d'amour";"belle";"marquise"];
      vos,beaux,yeux,me,font,mourir,d'amour,belle,marquise
val l : string cellule =
# let circ_list_of_list = function
    [] -> failwith "circ_list_of_list"
  | (a::l) ->let x= make_list a in
      (List.fold_left insert_tail x l);;
```



```
      val circ_list_of_list : 'a list -> 'a cellule = <fun>
# let q = circ_list_of_list
          ["vos";"beaux";"yeux";"me";"font";"mourir";"d'amour";
           "belle";"marquise"];;
    vos,beaux,yeux,me,font,mourir,d'amour,belle,marquise
val q : string cellule =
```

Réciproquement, la liste des éléments d'une liste circulaire peut être obtenue de la façon suivante :

```
# let list_of_circ_list l =
  let rec l_of_c p =
    if p==l then [] else
      first p :: l_of_c p.suiv
  in
    (first l :: l_of_c l.suiv);;
          val list_of_circ_list : 'a cellule -> 'a list = <fun>
# list_of_circ_list q;;
- : string list =
["vos"; "beaux"; "yeux"; "me"; "font"; "mourir"; "d'amour"; "belle";
 "marquise"]
```

## 3 Boucles

Pour finir, rappelons que Caml offre la possibilité de construire des boucles à l'aide des mots clés `for` et `while`. La syntaxe est la suivante :

```
for nom = expr1 to expr2 do expr3 done
for nom = expr1 downto expr2 do expr3 done
```

où `expr1,expr2 : int` et `expr3 : unit`. Par exemple :

```
# let liste i n =
  let ls = ref [] in
    for j = n downto i
    do
      ls := j::!ls
    done ;
    !ls;;
            val liste : int -> int -> int list = <fun>
# liste 1 10;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

La syntaxe de `while` est la suivante :

```
while expr1 do expr2 done
```

où `expr1 : bool` et `expr2 : unit`. Un exemple d'utilisation est :

```
# let until predicat changer x =
  let y = ref x in
    while not (predicat !y)
    do
      y :=  changer(!y)
    done;
    !y;;
            val until : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a = <fun>
# until (fun x -> x > 10) (fun x -> x+1) 0;;
- : int = 11
```

Pour être cohérent, le corps de la boucle (`expr3` pour `for` et `expr2` pour `while`) doit avoir le type `unit`. Autrement Ocaml donne un avertissement :

```
# let until predicat changer x =
  let y = ref x in
    while not (predicat !y)
    do
      y :=  changer(!y) ; 1
    done;
    !y;;
            Characters 92-113:
Warning: this expression should have type unit.
        y :=  changer(!y) ; 1
        ^^^^^^^^^^^^^^^^^^^^^
val until : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a = <fun>
```

En effet l'expression `y :=  changer(!y) ; 1` a type `int`, comme on peut aisément s'en convaincre :

```
# let y = ref 0;;
val y : int ref = {contents = 0}
# y:= !y + 1; 1;;
- : int = 1
```