



Les données mutables

CSI 3520
Amy Felty
University of Ottawa

Raisonner sur l'état mutable est difficile

ensemble mutable

```
insert i s1;  
f x;  
member i s1
```

ensemble immuable

```
let s1 = insert i s0 in  
f x;  
member i s1
```

member i s1 == true? ...

- Quand s1 est modifiable, il faut regarder f pour déterminer si elle modifie s1.
- Pire encore, il faut souvent résoudre *le problème du partage « aliasing »*.
- Pire encore, dans un contexte de la concurrence, il faut examiner *toutes les autres fonctions* qui *peuvent être exécutées par un autre « thread »* pour voir si elles modifient s1.

Jusqu'à présent...

On a considéré la partie (presque) purement fonctionnelle d'OCaml.

- On a vu quelques effets : imprimer et lever des exceptions.

Deux raisons pour cette importance:

- *Raisonner sur le code fonctionnel est plus facile.*
 - Raisonnement formel
 - en utilisant des équations et le modèle de substitution
 - et raisonnement informel
 - Les structures de données sont *persistantes*.
 - Ils ne changent pas – on en crée de nouveaux et le ramasse-miettes récupère ceux qui ne sont plus utilisés.
 - *Par conséquent, tout invariant prouvé reste vrai.*
 - par exemple, 3 est un membre de l'ensemble S.
- *Pour vous convaincre que vous n'avez pas besoin d'effets secondaires pour de nombreuses tâches*
 - Programmer avec des *données immuables de base telles que int, pair, list est simple.*
 - les types font beaucoup de tests pour vous!
 - ne craignez pas la récursion!
 - Vous pouvez planter des structures de données fonctionnelles *expressives et hautement réutilisables* telles que des « 2-3 arbres » polymorphes, des dictionnaires, des piles, des files d'attente, des ensembles, des expressions... La complexité d'espace et de temps reste raisonnable.

Mais hélas...

Les programmes purement fonctionnels ne peuvent pas tout faire.

- Parfois, on veut vraiment que le code ait un effet sur le monde.
- Par exemple, la boucle d'interaction niveau supérieur d'OCaml imprime les résultats.
 - Sans imprimer les résultats (un effet), comment pourrait-on savoir que les fonctions ont calculé le bon résultat?

Certains algorithmes ou structures de données exigent un état mutable.

- Les tables des adresses hash codée: l'accès et la mise à jour ont (essentiellement) un temps constant.
 - Les meilleurs dictionnaires fonctionnels ont:
 - soit accès logarithmique et mise à jour logarithmique
 - accès constant et mise à jour linéaire
 - mise à jour constante et accès linéaire
 - N'oubliez pas qu'il ya un coût:
 - On ne peut pas regarder les versions précédentes du dictionnaire. On peut le faire sur une version fonctionnelle.
- Algorithme d'unification de Robinson
 - Une partie essentielle du contrôle de type de l'OCaml
 - Également utilisé dans d'autres algorithmes d'analyse de programme
- l'algorithme de parcours en profondeur DFS (Depth First Search), etc...

Cependant, le code purement principalement fonctionnel est très productif

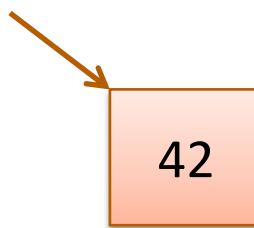
RÉFÉRENCES MUTABLES D'OCAML

Références

- Nouveau type: `t ref`
 - Considérez-le comme un pointeur sur une `case` contenant une valeur `t`.
 - Le contenu de la case peut être lu ou écrit.

Références

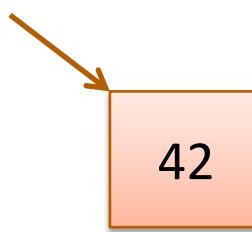
- Nouveau type: `t ref`
 - Considérez-le comme un pointeur sur une *case* contenant une valeur `t`.
 - Le contenu de la case peut être lu ou écrit.
- Pour créer une nouvelle case: `ref 42`
 - alloue une nouvelle case, initialise son contenu à 42 et renvoie un pointeur:



- `ref 42 : int ref`

Références

- Nouveau type: `t ref`
 - Considérez-le comme un pointeur sur une *case* contenant une valeur `t`.
 - Le contenu de la case peut être lu ou écrit.
- Pour créer une nouvelle case: `ref 42`
 - alloue une nouvelle case, initialise son contenu à 42 et renvoie un pointeur:



- `ref 42 : int ref`
- Pour lire le contenu: `!r`
 - si `r` pointe vers une case contenant 42, alors renvoie 42.
 - si `r : t ref` alors `!r : t`
- Pour écrire le contenu: `r := 5`
 - une modification de la case de `r`, il en contient maintenant 5.
 - si `r : t ref` alors `r := 5 : unit`

Exemple

```
let c = ref 0 in  
  
let x = !c in      (* x a la valeur 0 *)  
  
c := 42;  
  
let y = !c in      (* y a la valeur 42.  
                      x a toujours la valeur 0! *)
```

Un autre exemple

```
let c = ref 0

let next() =
    let v = !c in
    (c := v+1 ; v)
```

Un autre exemple

```
let c = ref 0

let next() =
    let v = !c in
    (c := v+1 ; v)
```

Si $e_1 : \text{unit}$
et $e_2 : t$ alors
 $(e_1 ; e_2) : t$

On peut aussi l'écrire comme ceci:

```
let c = ref 0

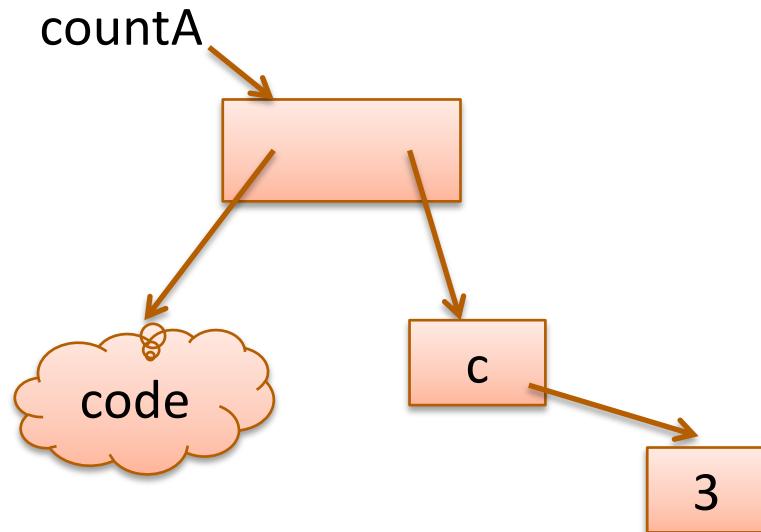
let next() =
    let v = !c in
    let _ = c := v+1 in
    v
```

Un autre idiome

Référence mutable modifiable

```
let c = ref 0

let next () : int =
  let v = !c in
  (c := v+1 ; v)
```



Référence mutable locale

```
let counter () =
  let c = ref 0 in
  fun () ->
    let v = !c in
    (c := v+1 ; v)
```

```
let countA = counter() in
let countB = counter() in
countA(); (* 0 *)
countA(); (* 1 *)
countB(); (* 0 *)
countB(); (* 1 *)
countA(); (* 2 *)
```

Boucles impératives

```
(* print n .. 0 *)
let count_down (n:int) =
  for i = n downto 0 do
    print_int i;
    print_newline()
done
```

```
(* print 0 .. n *)
let count_up (n:int) =
  for i = 0 to n do
    print_int i;
    print_newline()
done
```

```
(* sum of 0 .. n *)
let sum (n:int) =
  let s = ref 0 in
  let current = ref n in
  while !current > 0 do
    s := !s + !current;
    current := !current - 1
  done;
  !s
```

Boucles imperatives?

```
(* print n .. 0 *)
```

```
let count_down (n:int) =
  for i = n downto 0 do
    print_int i;
    print_newline()
done
```

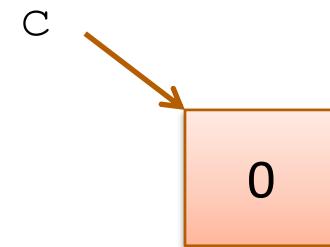
```
(* for i=n downto 0 do f i *)
```

```
let rec for_down
  (n : int)
  (f : int -> unit)
  : unit =
  if n >= 0 then
    (f n; for_down (n-1) f)
  else
    ()
```

```
let count_down (n:int) =
  for_down n (fun i ->
    print_int i;
    print_newline()
  )
```

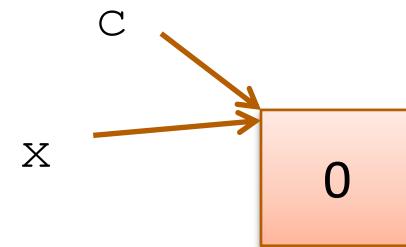
Partage « Aliasing »

```
let c = ref 0  
  
let x = c  
  
x := 42 ;  
  
!c
```



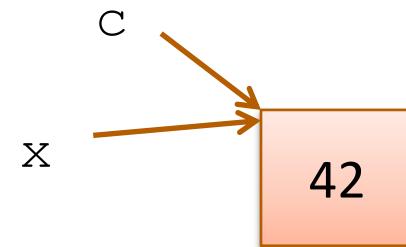
Partage « Aliasing »

```
let c = ref 0  
  
let x = c  
  
x := 42 ;  
  
!c
```



Partage « Aliasing »

```
let c = ref 0  
  
let x = c  
  
x := 42 ;  
  
!c
```



LES RÉFÉRENCES ET LES MODULES

Les types et les références

Les types concrets de premier ordre fournissent beaucoup d'informations sur les structures de données

- int ==> immuable
- int ref ==> mutable
- int * int ==> immuable
- int * (int ref) ==> 1^{er} composant immuable, 2^{ème} mutable
- ... etc

Les types d'ordre supérieur?

- int -> int ==> la fonction ne peut pas être modifiée
- ==> qu'est-ce qui se passe quand on l'exécute?

Les types abstraits?

- stack, queue? stack * queue?

Piles fonctionnelles

```
module type STACK =
  sig
    type 'a stack
    val empty : unit -> 'a stack
    val push : 'a -> 'a stack -> 'a stack
    val pop : 'a stack -> 'a stack
    val top : 'a stack -> 'a option
    ...
  end
```

Piles fonctionnelles

```
module type STACK =  
  sig  
    type 'a stack  
  
    val empty : unit -> 'a stack  
    val push : 'a -> 'a stack -> 'a stack  
    val pop : 'a stack -> 'a stack  
    val top : 'a stack -> 'a option  
    ...  
  end
```

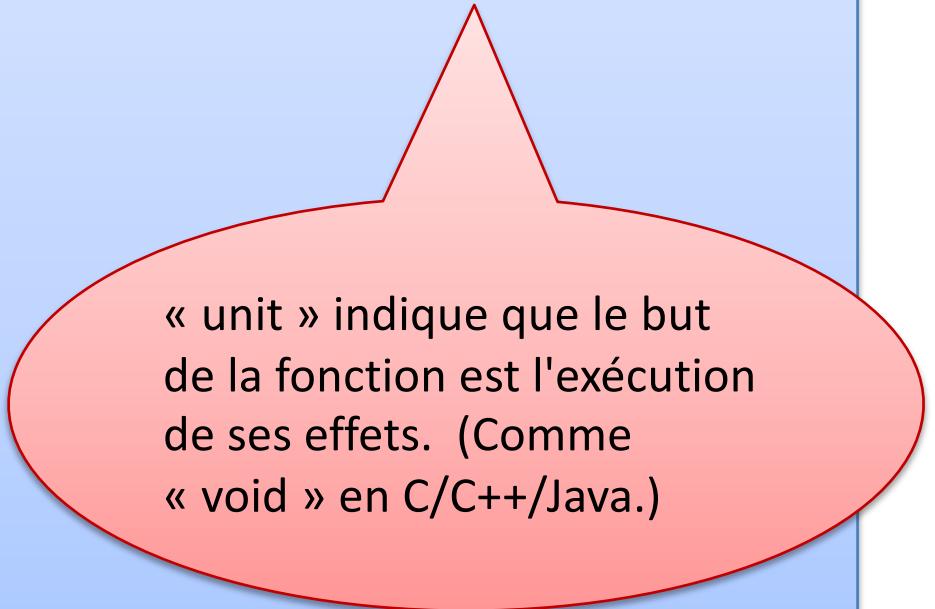
Dans une interface fonctionnelle, il y a des arguments d'entrée et les opérations produisent de nouveaux résultats

Piles impératives

```
module type IMP_STACK =  
sig  
  type 'a stack  
  val empty : unit -> 'a stack  
  val push : 'a -> 'a stack -> unit  
  ...  
end
```

Piles impératives

```
module type IMP_STACK =  
sig  
  type 'a stack  
  val empty : unit -> 'a stack  
  val push : 'a -> 'a stack -> unit  
  ...  
end
```



« unit » indique que le but de la fonction est l'exécution de ses effets. (Comme « void » en C/C++/Java.)

Piles impératives

```
module type IMP_STACK =  
sig  
  type 'a stack  
  val empty : unit -> 'a stack  
  val push : 'a -> 'a stack -> unit  
  val pop : 'a stack -> 'a option  
end
```

Malheureusement, il n'est pas toujours clair qu'il y aura des effets. C'est une bonne idée de les documenter si l'utilisateur peut les détecter.

Piles impératives

```
module type IMP_STACK =  
sig  
  type 'a stack  
  val empty : unit -> 'a stack  
  val push : 'a -> 'a stack -> unit  
  val pop : 'a stack -> 'a option  
end
```

Malheureusement, il n'est pas toujours clair qu'il y aura des effets. C'est une bonne idée de les documenter si l'utilisateur peut les détecter.

Parfois, on utilise des références à l'intérieur d'un module mais les structures de données ont une sémantique fonctionnelle (persistante, immuable)

Piles impératives

```
module type IMP_STACK =  
sig  
  type 'a stack  
  val empty : unit -> 'a stack  
  val push : 'a -> 'a stack ->  
  val pop : 'a stack -> 'a option  
end
```

Malheureusement, il n'est pas toujours clair qu'il y aura des effets. C'est une bonne idée de les documenter si l'utilisateur peut les détecter.

C'est une excellente façon d'utiliser des références en ML. Cherchez ces opportunités.

Parfois, on utilise des références à l'intérieur d'un module mais les structures de données ont une sémantique fonctionnelle (persistante, immuable)

Piles impératives

```
module ImpStack : IMP_STACK =
  struct
    type 'a stack = ('a list) ref

    let empty() : 'a stack = ref []

    let push (x:'a) (s:'a stack) : unit =
      s := x::(!s)

    let pop (s:'a stack) : 'a option =
      match !s with
      | [] -> None
      | h::t -> (s := t ; Some h)

  end
```

Piles impératives

```
module ImpStack : IMP_STACK =
  struct
    type 'a stack = ('a list) ref

    let empty() : 'a stack = ref []

    let push (x:'a) (s:'a stack) =
      s := x::(!s)

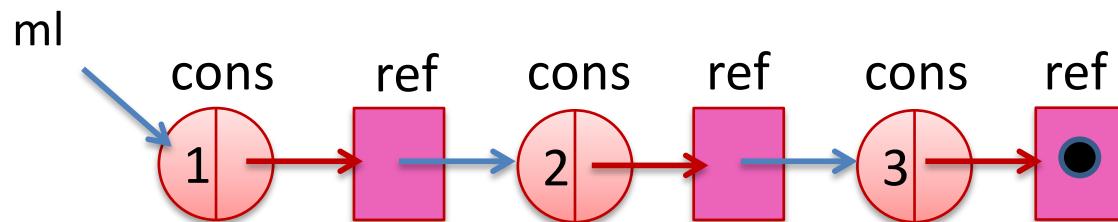
    let pop (s:'a stack) =
      match !s with
        | [] -> None
        | h::t -> (s := t ; Some h)

  end
```

Note: Il n'est pas nécessaire que *tout* soit mutable. La liste est immuable, mais stockée dans une seule cellule mutable.

Listes Complètement Mutables

```
type 'a mlist =  
    Nil | Cons of 'a * ('a mlist ref)  
  
let ml = Cons(1, ref (Cons(2, ref  
    (Cons(3, ref Nil))))))
```

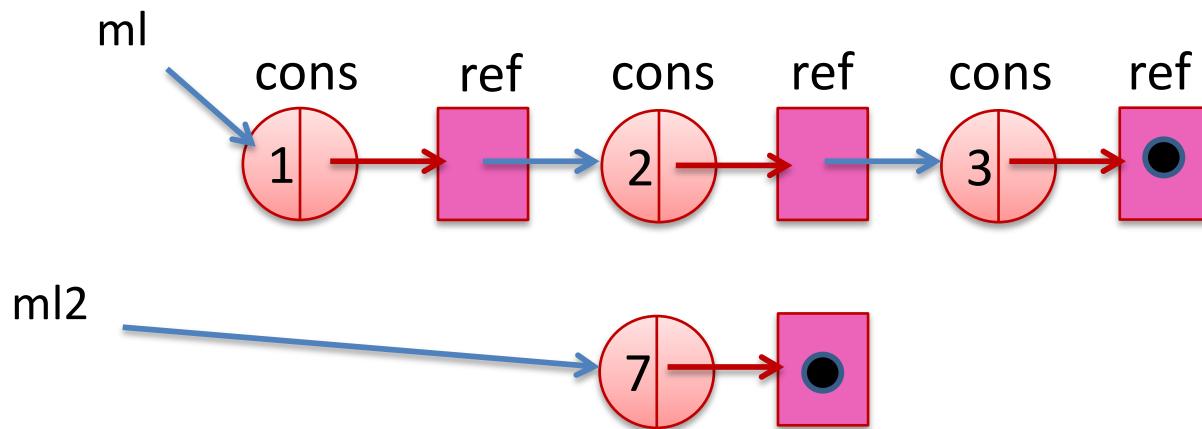


Listes Complètement Mutables

```
type 'a mlist =  
    Nil | Cons of 'a * ('a mlist ref)
```

```
let ml = Cons(1, ref (Cons(2, ref  
(Cons(3, ref Nil))))))
```

```
let ml2 = Cons(7, ref Nil)
```

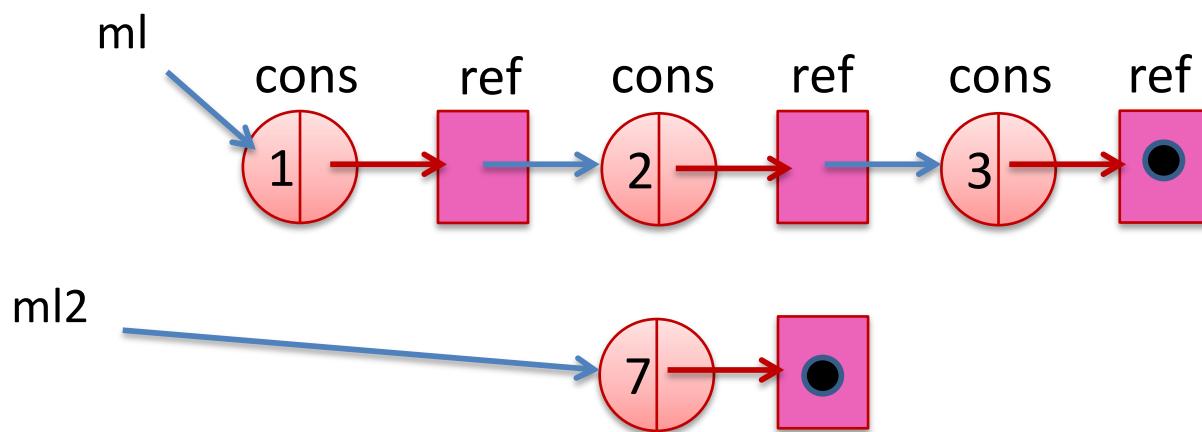


Listes Complètement Mutables

```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec fudge(l:'a mlist)
  (m:'a mlist) : unit =
  match l with
  | Nil -> ()
  | Cons(h,t) -> t := m ; ()
```

fudge ml ml2

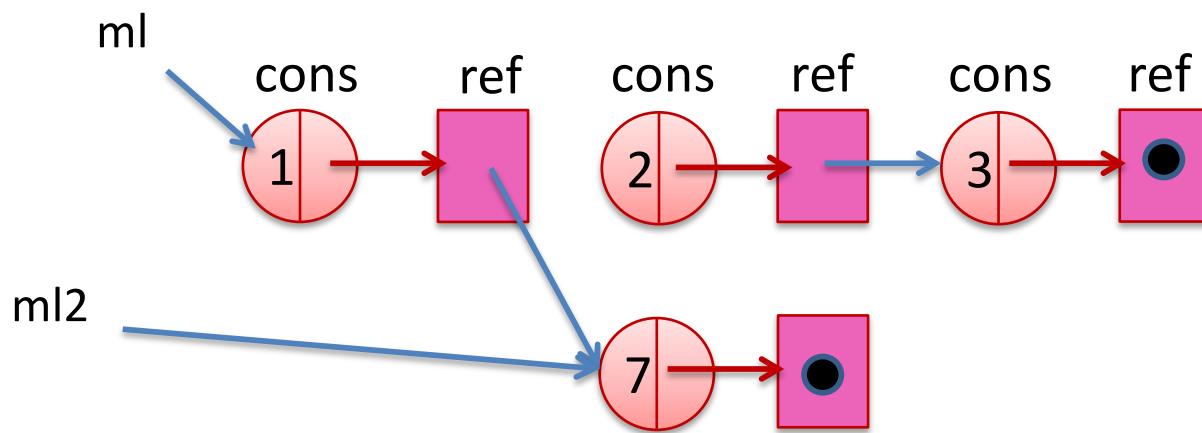


Listes Complètement Mutables

```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec fudge(l:'a mlist)
  (m:'a mlist) : unit =
  match l with
  | Nil -> ()
  | Cons(h,t) -> t := m ; ()
```

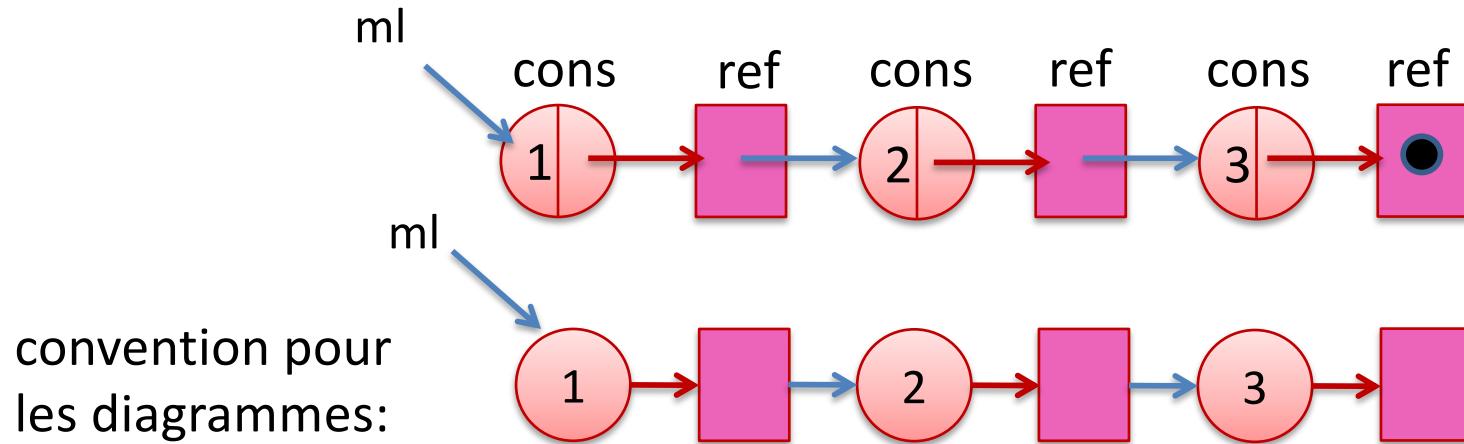
fudge ml ml2



Listes Complètement Mutables

```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec mlength(m:'a mlist) : int =
  match m with
  | Nil -> 0
  | Cons(h,t) -> 1 + mlength(!t)
```



Plein de risques

```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec mlength(m:'a mlist) : int =
  match m with
  | Nil -> 0
  | Cons(h,t) -> 1 + length(!t)

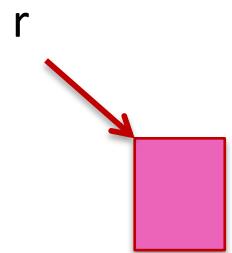
let r = ref Nil in
let m = Cons(3,r) in
(r := m ;
mlength m)
```

Plein de risques

```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec mlength(m:'a mlist) : int =
  match m with
  | Nil -> 0
  | Cons(h,t) -> 1 + length(!t)

let r = ref Nil in
let m = Cons(3,r) in
(r := m ;
mlength m)
```

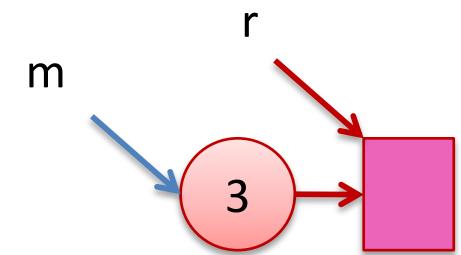


Plein de risques

```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec mlength(m:'a mlist) : int =
  match m with
  | Nil -> 0
  | Cons(h,t) -> 1 + length(!t)

let r = ref Nil in
let m = Cons(3,r) in
(r := m ;
mlength m)
```

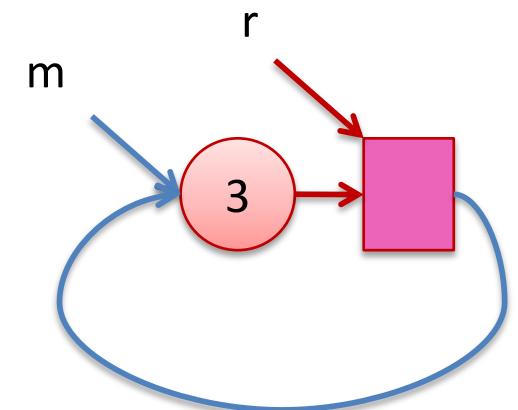


Plein de risques

```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec mlength(m:'a mlist) : int =
  match m with
  | Nil -> 0
  | Cons(h,t) -> 1 + length(!t)

let r = ref Nil in
let m = Cons(3,r) in
(r := m ;
mlength m)
```



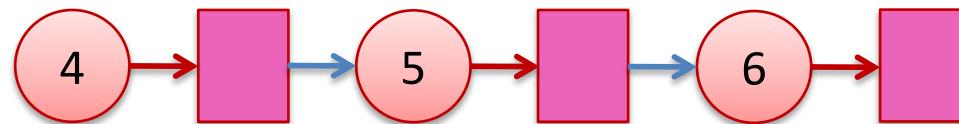
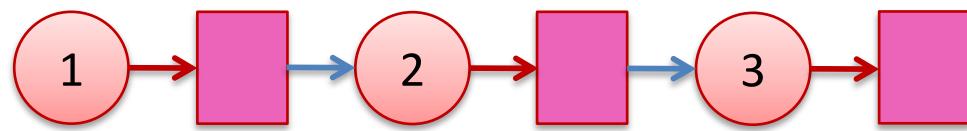
Un autre exemple

```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec mappend xs ys =
  match xs with
  | Nil -> ()
  | Cons(h,t) ->
    (match !t with
     | Nil -> t := ys
     | Cons(_,_) as m -> mappend m ys)
```

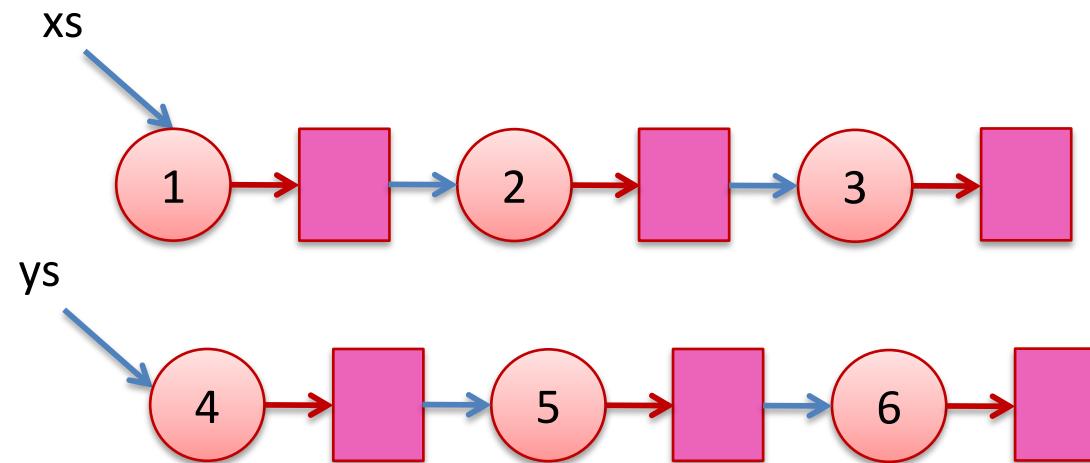
Exemple: « append » mutable

```
let rec mappend xs ys =
  match xs with
  | Nil -> ()
  | Cons(h, t) ->
    (match !t with
     | Nil -> t := ys
     | Cons(_, _) as m -> mappend m ys) ;;
let x = Cons(1, ref (Cons(2, ref (Cons(3, ref Nil))))) ;;
let y = Cons(4, ref (Cons(5, ref (Cons(6, ref Nil))))) ;;
mappend x y ;;
```



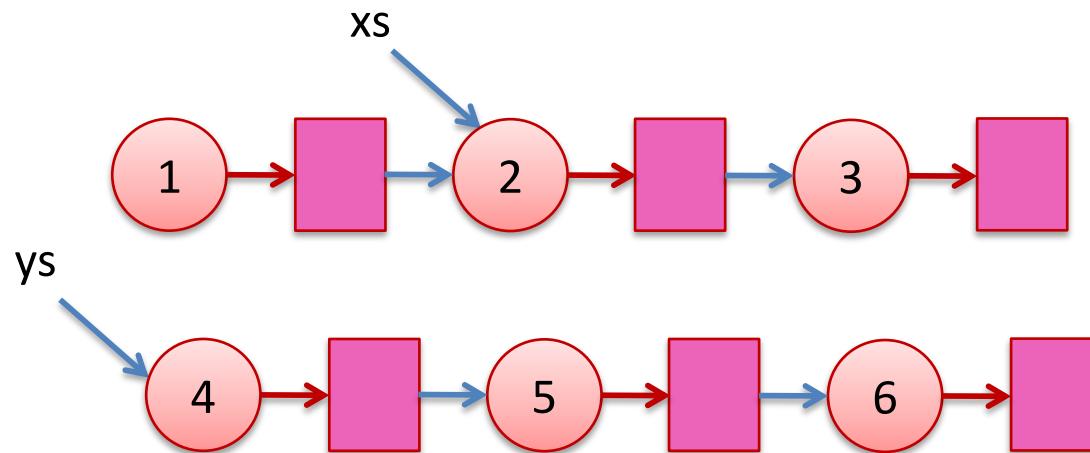
Exemple: « append » mutable

```
let rec mappend xs ys =
  match xs with
  | Nil -> ()
  | Cons(h, t) ->
    (match !t with
     | Nil -> t := ys
     | Cons(_, _) as m -> mappend m ys) ;;
let x = Cons(1, ref (Cons (2, ref (Cons (3, ref Nil)))))) ;;
let y = Cons(4, ref (Cons (5, ref (Cons (6, ref Nil)))))) ;;
mappend x y ;;
```



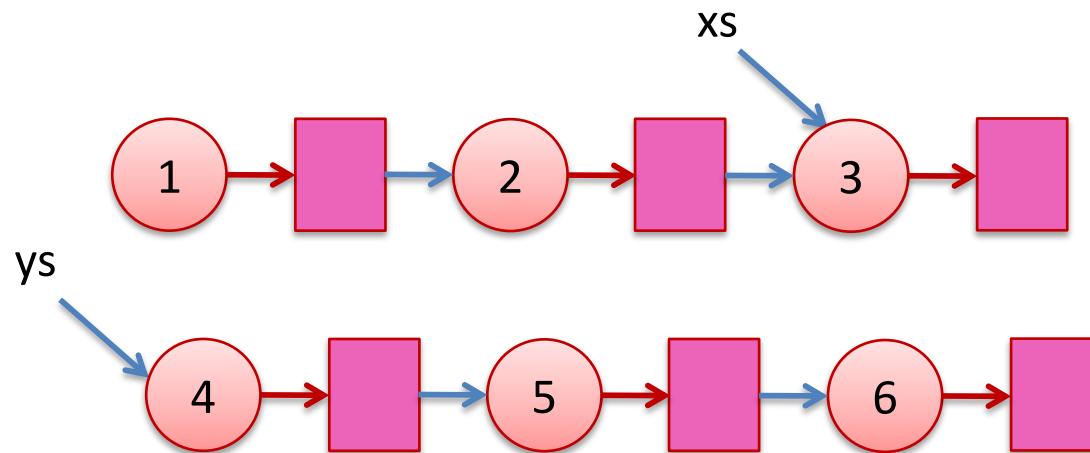
Exemple: « append » mutable

```
let rec mappend xs ys =
  match xs with
  | Nil -> ()
  | Cons(h, t) ->
    (match !t with
     | Nil -> t := ys
     | Cons(_, _) as m -> mappend m ys) ;;
let x = Cons(1, ref (Cons (2, ref (Cons (3, ref Nil)))))) ;;
let y = Cons(4, ref (Cons (5, ref (Cons (6, ref Nil)))))) ;;
mappend x y ;;
```



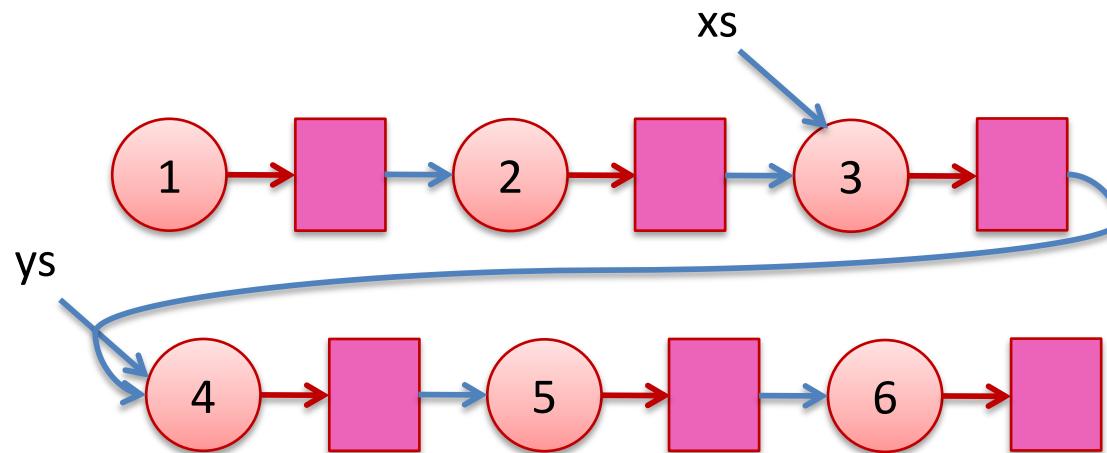
Exemple: « append » mutable

```
let rec mappend xs ys =
  match xs with
  | Nil -> ()
  | Cons(h, t) ->
    (match !t with
     | Nil -> t := ys
     | Cons(_, _) as m -> mappend m ys) ;;
let x = Cons(1, ref (Cons (2, ref (Cons (3, ref Nil)))))) ;;
let y = Cons(4, ref (Cons (5, ref (Cons (6, ref Nil)))))) ;;
mappend x y ;;
```



Exemple: « append » mutable

```
let rec mappend xs ys =
  match xs with
  | Nil -> ()
  | Cons(h, t) ->
    (match !t with
     | Nil -> t := ys
     | Cons(_, _) as m -> mappend m ys) ;;
let x = Cons(1, ref (Cons (2, ref (Cons (3, ref Nil)))))) ;;
let y = Cons(4, ref (Cons (5, ref (Cons (6, ref Nil)))))) ;;
mappend x y ;;
```



Ajouter judicieusement la mutabilité

Deux types:

```
type 'a very_mutable_list =
  Nil
  | Cons of 'a * ('a very_mutable_list ref)
```

```
type 'a less_mutable_list = 'a list ref
```

Le premier permet les listes cycliques, mais pas le deuxième

- le deuxième évite certains types d'erreurs
- On dit *une conception correcte par construction*
 - « *correct-by-construction design* »

Est-il possible d'éviter tout état?

- Oui! (dans les programmes « single threaded »)
 - L'état est une entrée et une sortie de chaque fonction...
mais ce n'est pas nécessairement efficace

Considérez la différence entre nos piles fonctionnelles et nos piles impératives :

- fnl_push : `'a -> 'a stack -> 'a stack`
- imp_push : `'a -> 'a stack -> unit`

RÉSUMÉ

Comment / quand utiliser des données mutables?

- Une question compliquée!
- En général, essayez d'écrire d'abord la version fonctionnelle.
 - par exemple, un prototype
 - On peut ignorer le partage et les modifications
 - On peut ignorer les situations de compétition « race conditions »
 - le raisonnement est plus facile (le modèle de substitution est valide)
- Parfois, on ne peut pas se permettre la version fonctionnelle pour des raisons d'efficacité
 - exemple: les tables de routage doivent être rapides dans un commutateur
 - un temps constant pour l'accès et la mise à jour (les tables des adresses hash codée)
- Lorsque vous utilisez des données mutables, essayez de les *encapsuler* à l'aide d'une interface.
 - essayez de réduire le nombre d'erreurs qu'un client peut voir
 - conception correcte par construction
 - le réalisateur de module doit penser explicitement au partage et aux invariants
 - notez les invariants, tester-les
 - si ces tests sont encapsulés dans un module, ils peuvent être localisés
 - *la plupart de votre code devrait être fonctionnel*

Résumé

Des structures de données mutables peuvent *améliorer l'efficacité*.

- par exemple, les tables des adresses hash codée, mémoïsation, algorithme de parcours en profondeur « depth-first search »

Mais ils sont *beaucoup* plus difficiles à utiliser correctement, alors ne les choisissez pas prématurément

- *la mise à jour des données à un endroit peut avoir un effet sur d'autres endroits*
- *écrire et imposer des invariants devient plus important*
 - pourquoi? parce que les types ont moins de puissance...
- *les cycles dans les données (autres que dans les fonctions) ne peuvent pas se produire avant* l'introduction des références
 - Il faut définir les opérations beaucoup plus soigneusement pour éviter les boucles
 - plus de cas à traiter et le compilateur ne vous aide pas!
- L'ajout de « multi-threading » introduit encore plus de complications

Donc, utilisez des références lorsque vous devez, mais essayez de les éviter.