OCaml mutable #1, files

Pierre Karpman

Lycée Champollion MP2I
https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

Fonctions de définition & manipulation

- `ref : 'a -> 'a ref` telle que `ref x` s'évalue en une (nouvelle) *référence* de valeur `x`
- `(!) : 'a ref -> 'a` opérateur unaire préfixe tel que `!r` s'évalue en la valeur associée à la référence `r`
- `(:=) : 'a ref -> 'a -> unit` opérateur binaire infixe tel que `r := x` *modifie* la valeur référencée par `r` en `x`

Remarque

- Une application de la fonction `(:=)` s'évalue en `unit`
- C'est généralement le cas de toutes les expressions *impures* d'OCaml (qui modifient un état ; un certain type d'*effet* (elles font autre chose que juste calculer une valeur))
- (Permet par ex. de parfois déterminer au typage si une fonction a des effets)

Démo

Références : comparaison avec les pointeurs en C

Similarités

Dans les deux cas :

- Manipulation d'un état *via* une « référence » vers cet état
- Permet d'implémenter un *passage par référence* (*duh!*)

Différences

En OCaml (références) :

- Pas de manipulation directe d'adresse
  - Pas d'équivalent de l'opérateur `&` de déréférencement
- Pas d'arithmétique de pointeur
- Pas de notion de pointeur nul
  - Mais implémentable (avec sûreté de manipulation en plus !) par une `'a option ref`

Références : intérêt

Assez évidemment

- Permet d'introduire de la mutabilité (*cf.* pile impérative)
- Nécessaire pour écrire du OCaml « impératif » (en conjonction avec des expressions de boucle, *cf.* plus tard)

Attention

- Hors besoin de mutabilité explicite, les références sont rarement très *utiles* en OCaml
- À utiliser avec parcimonie…

Table des matières

OCaml : enregistrements

OCaml permet de définir des types *enregistrement* qui sont essentiellement des types produits où :

- chaque élément possède un nom, et
- avec un support « natif » pour la mutabilité (sans utiliser de référence)

Enregistrements : syntaxe

Syntaxe de déclaration

- `type nom = {c1 : t1; c2 : t2; ... }`

où `nom` est le nom du type, `c1` etc. les noms des champs et `t1` etc. leurs types

- Le type peut éventuellement faire intervenir des paramètres pour le polymorphisme (même syntaxe que d'habitude : `type 'a ... nom = ...`)
- Un champ peut être défini comme mutable en préfixant son nom par `mutable`

Syntaxe de construction de valeur

- Par `{c1=v1 ; c2=v2 ; ...}` (et de très nombreuses variantes & sucres syntaxiques)

Exemples

```
type 'a slist2_e = { dat : 'a;
                     mutable nxt : 'a slist2_e option }

utop # { dat = 12 ; nxt = None }
- : int slist2_e = {dat = 12; nxt = None}
```

Enregistrements : syntaxe *bis*

Syntaxe d'accès des champs en lecture

- Par filtrage de motif (*guess how*)
- Par notation pointée : `r.c1`

Syntaxe d'accès des champs en écriture

Possible uniquement pour des champs mutables

- Par notation flèche sur notation pointée : `r.c1 <- v`
- Expression de type `unit`

Exemples

```
utop # let r = {dat = 12 ; nxt = None };;
utop # r.dat;;
- : int = 12
utop # r.dat <- 13;;
Error: The record field dat is not mutable
utop # r.nxt <- Some r;;
- : unit = ()
```

Enregistrements : intérêt

- Souvent plus agréable à manipuler que les types produits sur le long terme
  - Quand un même produit apparaît souvent, par ex. dans une structure de données
  - Facilite l'affichage du « bon » type par le système de type
- Fournit une mutabilité « générique » sans références

Attention

cependant aux conflits de noms entre champs entre différents types enregistrement…

Exercice

Réimplémentez les fonctions `ref`, `(!)` et `(:=)` avec un type enregistrement

Correction

```
type 'a ref = {mutable contents : 'a }
let ref x = {contents = x}
let ref_deref r = r.contents
let ref_assign r x = r.contents <- x

(* Bonus : redéfinition des opérateurs *)
let (!) = ref_deref
let (:=) = ref_assign
```

(Pas *exactement* la définition de ces fonctions dans la bibliothèque standard, mais compatible…)

Table des matières

OCaml : séquence

Motivation

- Les expressions *impures* de modification d'état ont type `unit` (leur seul « intérêt » est dans la modification d'état qu'elles effectuent)
- Dans du code en style impératif, on se retrouve souvent à « enchaîner » différentes expressions impures
- …ou des expressions impures avec des expressions pures (pour par ex. lire un état)

Solution existante : `let in`

Rappel : `let x = e1 in let y = e2 in ...`

- évalue `e1` et lie le résultat à `x` puis évalue `e2` et lie le résultat à `y` puis…
- `x`, `y`… sont plus généralement des motifs ; si les noms ne sont pas utilisés on peut utiliser `let _ = e1 in let _ = e2 in ...`
- ou `let () = e1 in let () = e2 in ...` si les expressions sont de type `unit`

Exemple

Exemple de pop d'une implémentation de pile impérative par référence sur une `'a list`

```
let pop (s:'a stack) : 'a
    = match !s with
    | [] -> failwith "empty stack"
    | x::xs -> let () = s := xs in x
```

OCaml : séquence (pour de vrai cette fois)

Syntaxe alternative aux `let in` : le « ; »

- On peut *séquencer* des expressions `e1`, `e2`… par `e1 ; e2 ; ... ; en`
- Sémantique : on évalue `e1`, `e2`… dans l'ordre, et l'évaluation de la séquence entière est celle de `en`
- N'a d'*intérêt* que si `e1`, `e2`… ont des *effets* (comme par ex. modifier un état)

Attention :

- Le `;` **sépare** deux *expressions* ; à ne pas confondre avec celui du C qui *termine une instruction* (un équivalent C serait la `,`)
- Une telle séquence serait fortement suspecte si `e1`, `e2`… n'étaient pas de type `unit`, le résultat de leurs évaluations étant perdu :

```
utop # 12 ; 13;;
Line 1, characters 0-2:
Warning 10 [non-unit-statement]: this expression
should have type unit.
```

Exemple

Exemple de pop d'une implémentation de pile impérative par référence sur une `'a list`

```
let pop (s:'a stack) : 'a
    = match !s with
    | [] -> failwith "empty stack"
    | x::xs -> s := xs ; x
```

Séquence : règles de priorité

Il peut être nécessaire de délimiter le début et la fin d'une séquence (quand elle fait partie d'une expression plus complexe). Deux possibilités :

- Parenthéser : `( e1 ; e2 ; ... ; en)`
- Avec `begin end` : `begin e1 ; e2 ; ... ; en end`

Attention

(Comme en général) l'absence de parenthésage (quand nécessaire) ou un mauvais parenthésage de séquence est une cause fréquente de bugs souvent détectés au typage, mais pas forcément avec des messages d'erreurs très explicites

Table des matières

Structure de données : File

En anglais : **queue**

Principe

Dans une file, on peut :

- ajouter (enfiler) un élément en queue de file
- si non vide, extraire (défiler) un élément

Avec la contrainte « FIFO » (*first in, first out*) qu'on défile l'élément encore présent dans la file depuis le plus longtemps.

Utilité

Permet de stocker des éléments à traiter, dans un ordre précis. Utile par exemple pour des parcours de graphe, d'arbre…

File fonctionnelle

Spécifications

- Une valeur (immuable) pour la file vide
  - Éventuellement une fonction *is_empty* qui teste si son argument est la file vide
- Un constructeur *push* : paramètres : une valeur x, une file q ; construit une *nouvelle* file où x a été ajouté en **queue** de q
- Une fonction *pop* : paramètre : une file q ; si q est non vide, renvoie le couple constitué de l'élément en tête de q et d'une *nouvelle* file identique au reste de q
- Éventuellement un accesseur *peek* : paramètre : une file q ; renvoie la tête de q si elle existe

File fonctionnelle : une implémentation inefficace (*cf.* TP)

On peut implémenter une file à partir d'une liste fonctionnelle (d'une pile) avec une concaténation de liste pour *push*

Illustration directement avec des `'a list` :

```
type 'a fq = 'a list
let is_empty q = (q = [])
let push x q = q @ [x]
let pop (x::xs) = x, xs (* match partiel ; pas autorisé pour vous *)
let peek (x::xs) = x    (* ditto *)
```

- Pour une implémentation usuelle de liste fonctionnelle, *push* a un coût linéaire en la taille de la file (pas terrible…)

File fonctionnelle : une implémentation *amortie* efficace (*cf.* TD)

On peut implémenter une file à partir de *deux* listes fonctionnelles (deux piles)

Implémentation

- La file vide correspond à deux piles vides s_1 et s_2
- On enfile les éléments en les empilant dans s_1
- Pour défiler :
  - si s_2 est vide, on dépile s_1 et l'empile dans s_2
  - si s_2 est non vide, on dépile s_2

Correction (esquisse)

Lemme :

- Transférer s_1 dans s_2 inverse l'ordre des éléments

Invariants :

- Les éléments dans s_1 sont rangés par ordre inverse d'arrivée (sommet : plus récent)
- Les éléments dans s_2 sont rangés par ordre d'arrivée (sommet : plus ancien)
- Les éléments dans s_2 sont plus anciens que les éléments dans s_1

File fonctionnelle : une implémentation *amortie* efficace (2)

Analyse de coût

Pour une pile fonctionnelle habituelle d'opérations en coût constant

- Pire cas standard :
  - *pop* peut avoir un coût linéaire en la taille de la file
  - *push* est en temps constant
  - Pas mieux que l'implémentation naïve :/
- *Amortie* (esquisse)
  - Chaque élément passant par la structure est au plus manipulé trois fois par les piles
  - Le coût de n opérations sur la file est un $O(n)$ opérations sur des piles donc $O(n)$
  - Le coût *amorti* de chaque opération est donc *constant* !
  - Donc en fait bien mieux que l'implémentation naïve :)

Exercice

Implémentez concrètement une telle file en utilisant le `type 'a queue = {s1 : 'a list ; s2 : 'a list}` (il suffit d'adapter les fonctions du TD à la syntaxe des enregistrements…)

Exercice : proposition de correction

En utilisant un filtrage dès que possible pour manipuler les enregistrements, et le sucre syntaxique des *champs implicites* (si un identifiant x a le nom d'un champ, `{ x }` équivaut à `{x = x}`)

```
type 'a queue = {s1 : 'a list ; s2 : 'a list}

let empty_queue = {s1 = [] ; s2 = []}

let push x {s1 ; s2} = {s1 = x::s1 ; s2}

let rec pop = function
    | {s1 = [] ; s2 = []} -> failwith "empty queue"
    | {s1 ; s2 = []} -> pop {s1 = [] ; s2 = List.rev s1}
    | {s1 ; s2 = x::xs} -> x, {s1 ; s2 = xs}
```

File impérative

Spécifications

- Un constructeur de file vide
- Une fonction *is_empty* qui teste si son argument est une file vide
- Un transformateur *push* : paramètres : une valeur x, une file q ; *modifie* q pour ajouter q en sa queue
- Un accesseur & transformateur *pop* : paramètre : une file q ; si q est non vide,
  - renvoie l'élément en tête de q
  - **et modifie q en supprimant celui-ci**
- Éventuellement un accesseur *peek* : paramètre : une file q ; renvoie l'élément en tête de q s'il existe

File impérative : implémentation 1

On utilise le module `Queue`…

- https://ocaml.org/manual/5.4/api/Queue.html

À savoir utiliser

- `create`
- `is_empty`
- `push`
- `pop`
- L'exception `Empty`

File impérative : implémentation 2 (*cf.* TP)

- Avec un tableau (éventuellement circulaire et/ou redimensionnable) avec un pointeur

File impérative : implémentation 3

- Avec des références sur deux piles fonctionnelles

Exercice

Proposez une telle implémentation, en utilisant le type enregistrement *mutable*

```
type 'a queue = {mutable s1 : 'a list ; mutable s2 : 'a list}
```

Exercice : proposition de correction

```
type 'a queue = {mutable s1 : 'a list ; mutable s2 : 'a list}

let make () = {s1 = [] ; s2 = []}

let push x q = q.s1 <- x::q.s1

let rec pop q
    = match q with
    | {s1 = [] ; s2 = []} -> failwith "empty queue"
    | {s1 ; s2 = []} -> q.s2 <- List.rev q.s1 ; q.s1 <- [] ; pop q
    | {s1 ; s2 = x::xs } -> q.s2 <- xs ; x
```

File impérative : implémentation 4

- Avec une liste impérative (en maintenant une référence vers la queue de la liste)
  - Pour être efficace, il faudra être capable de supprimer efficacement un élément arbitraire de la liste… *Cf.* TP de la semaine prochaine