

Programmation Fonctionnelle Avancée

5 : Queues revisitées : les vertues de la paresse

Pierre Letouzey

Université Paris Cité
UFR Informatique
Institut de Recherche en Informatique Fondamentale
letouzey@irif.fr

13 mars 2023

Plan du cours

- ▶ Les Queues du chapitre 3 : efficacité ou persistance ?
- ▶ Les Queues revisitées : persistantes et $O(1)$ amorti
- ▶ Queues temps réel : persistantes et $O(1)$ constant
- ▶ Pour en savoir plus

Reprenons nos files fonctionnelles *efficaces* du chapitre 3

- ▶ Files d'attente, réalisées par une pile d'entrée et une pile de sortie
- ▶ Nous avons trouvé un coût amorti constant, avec la méthode du banquier : le crédit obtenu pour l'ajout d'un élément est dépensé pour le renverser vers la pile de sortie, et pour la suppression.
- ▶ Est-ce qu'il y avait des hypothèses derrière cette analyse ?

Exemples (queue1.ml)

```
exception Empty
```

```
module type FIFO = sig
```

```
  type 'a t
```

```
  val empty : 'a t
```

```
  val add : 'a -> 'a t -> 'a t
```

```
  val remove : 'a t -> 'a * 'a t
```

```
  (** leve l'exception [Empty] sur une file vide *)
```

```
end
```

Exemples (queue2.ml)

```
module FifoDL : FIFO = struct
  type 'a t = 'a list * 'a list
  let empty = ([], [])
  let add x (l1, l2) = (x::l1, l2)
  let remove (l1, l2) = match l2 with
  | a::l -> (a, (l1, l))
  | [] -> match List.rev l1 with
  | [] -> raise Empty
  | a::l -> (a, ([], l))
end
```

Le module instrumenté (queue3.ml) I

On va maintenant *instrumenter* le code pour compter à la fois le nombre d'appels de fonctions *add* et *remove*, et pour accumuler le coût de ces opérations.

```
module FifoDLCount = struct
  (* instrumented to trace cost *)
```

```
type 'a t = 'a list * 'a list
let empty = ([], [])
```

```
let ncalls = ref 0 (* number of function calls *)
let cost = ref 0 (* accumulated cost *)
let incr c n = c := !c + n
let reset () = cost := 0; ncalls := 0
```

Le module instrumenté (queue3.ml) II

```

let add x (l1, l2) =
  incr ncalls 1; incr cost 1; (x::l1, l2)

let remove (l1, l2) =
  incr ncalls 1;
  match l2 with
  | a::l → incr cost 1; (a, (l1, l))
  | [] →
    incr cost (List.length l1);
    match List.rev l1 with
    | [] → raise Empty
    | a::l → incr cost 1; (a, ([], l))
end

```

Exemples (queue4.ml)

```

open FifoDLCount
let rec iterate_add n q =
  if n=0 then q
  else iterate_add (n-1) (add n q)
let rec iterate_remove n q =
  if n=0 then q
  else iterate_remove (n-1) (snd (remove q))

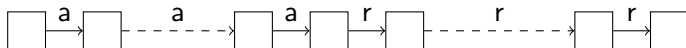
let test n =
  reset ();
  let _ = iterate_remove n (iterate_add n empty)
  in (!ncalls ,!cost)

let _ = test 10;; (* as expected *)

```


Séquence des opérations (queue4.ml)

Utilisation non-persistente de la queue.



Exemples (queue5.ml)

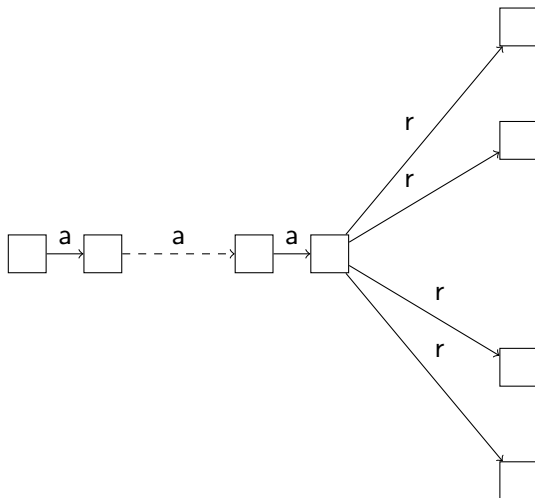
```
let rec repeat_remove n q =  
  if n=0 then q  
  else let _ = remove q in repeat_remove (n-1) q
```

```
let test2 n =  
  reset ();  
  let _ = repeat_remove n (iterate_add n empty)  
  in (!ncalls ,!cost)
```

```
let _ = test2 10  
(* ne correspond pas *)
```

Séquence des opérations (queue5.ml)

Utilisation persistente de la queue.



Observation dans le cas d'usage persistant

- ▶ Le coût n'est plus linéaire si on utilise la structure de façon persistante !
- ▶ Avec n `remove` *sur la même file*, qui coûtent *n fois de suite* un `List.rev` sur une liste de n éléments, on fabrique une séquence de n opérations avec coût $O(n^2)$, et pas $O(n)$ comme vu auparavant.
- ▶ Où est l'erreur ?
- ▶ Réponse : notre analyse excluait ce genre de séquences d'opérations !

L'hypothèse cachée dans l'analyse de coût amorti

- ▶ L'argument utilisé au chapitre 3 :
Quand on exécute un `remove` qui fait appel à `List.rev`, on a n éléments sur la pile d'entrée, donc n crédits qu'on utilise pour le `List.rev`.
- ▶ Ce raisonnement *n'est plus valable* si on fait un usage persistant de la structure de données : les crédits sur la deuxième liste peuvent avoir été déjà consommés lors d'un appel précédent !
- ▶ Notre structure est persistante, mais notre analyse de complexité était basée sur un usage *non*-persistant.
- ▶ Comment obtenir une complexité linéaire même pour un usage persistant ?

D'où vient le problème ?

Considérons le scénario suivant :

- ▶ On crée une première file f , avec tous les éléments sur la pile d'entrée.
- ▶ On construit $f_1 = \text{snd}(\text{remove } f)$, obtenue en renversant la pile d'entrée de f vers la pile de sortie de f_1 .
- ▶ On construit $f_2 = \text{snd}(\text{remove } f)$, obtenue aussi en renversant la pile d'entrée de f vers la pile de sortie de f_2 .
- ▶ On a maintenant créé deux copies indépendantes de la même structure, chacune obtenue en renversant la pile d'entrée de f .
- ▶ On a perdu le “lien” entre les deux copies qui permettrait de profiter dans une copie du travail fait dans l'autre.

Exemple

Comparer les effets de :

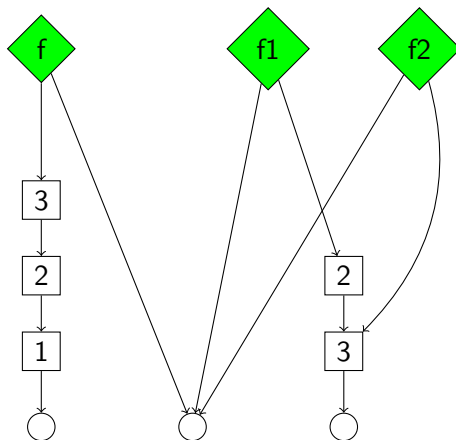
1. Usage non-persistant :

```
let f = add 3 (add 2 ( add 1 empty))  
let f1 = snd (remove f)  
let f2 = snd (remove f1)
```

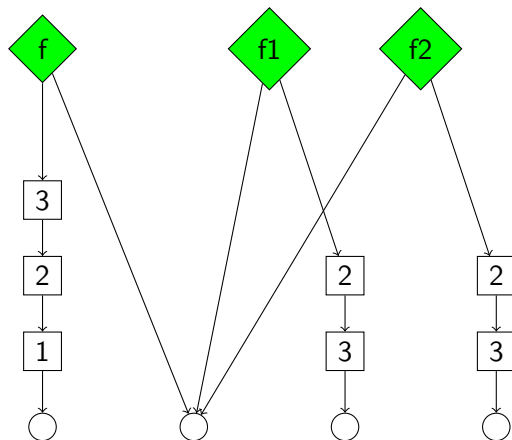
2. Usage persistant :

```
let f = add 3 (add 2 ( add 1 empty))  
let f1 = snd (remove f)  
let f2 = snd (remove f)
```

Structures en mémoire : le cas non-persistent



Structures en mémoire : le cas persistant



Comment faire mieux dans le cas d'usage persistant ?

- ▶ Pour avoir des files efficaces même avec un usage persistant, il faut éviter la duplication “gratuite” des opérations chères (`List.rev`) comme dans l'exemple précédent.
- ▶ La paresse est la solution ! Pourquoi ?
 - ▶ On partage des structures qui ont la même origine.
 - ▶ Avec la paresse, les structures de données peuvent contenir des calculs suspendus (structures d'un type α `Lazy.t`).
 - ▶ Plusieurs structures partageant un calcul suspendu peuvent profiter de l'avancement du calcul suspendu.

Mise en pratique : changement de la représentation interne

La paresse à la rescousse !

- ▶ On remplace les deux listes par deux streams.
- ▶ On garde trace de la taille de chaque stream.
- ▶ Le type de la file d'attente devient donc :

```
open Stream (* voir le cours 4 *)  
type 'a t = int * 'a stream * int * 'a stream;;
```

- ▶ On a d'abord le flot d'entrée et sa longueur, puis le flot de sortie et sa longueur.

Comment adapter les opérations add et remove ?

- ▶ Il y a une subtilité : quand doit-on faire le renversement (éventuellement suspendu) du flot d'entrée vers le flot de sortie ?
- ▶ Si on le fait trop tard (quand le flot de sortie est complètement vide), on risque de créer *plusieurs* renversements suspendus qui sont *indépendants*. On ne pourra donc pas profiter du partage !
- ▶ Si on le fait trop tôt (par exemple après chaque add), on perd tout l'avantage introduit au cours 3.

Comment gérer les calculs suspendus ?

La clef est de garantir dans tous les cas au moins l'une des deux conditions suivantes :

- ▶ soit on partage entre toutes les copies la même opération coûteuse, qui est exécutée une seule fois ; pour cela, on utilisera des structures paresseuses, qui nous permettent de partager du calcul, comme nous l'avons vu ;
- ▶ soit on s'assure qu'au moment de la copie, il y aura assez d'opérations élémentaires avant l'opération chère, pour couvrir le coût de l'opération chère ; pour cela, on évitera de trop laisser grandir la pile d'entrée.

Mise en pratique : la taille des streams

- ▶ Invariant sur la taille des deux streams :
 - ▶ La stream de sortie est *toujours* au moins aussi longue que la stream d'entrée ;
 - ▶ dès qu'une opération peut violer cet invariant, on retourne *paresseusement* la stream d'entrée et on la concatène *paresseusement* à la stream de sortie.
- ▶ Cette opération est réalisée par la fonction check suivante :

```
let check ((nin , sin , nout , sout) as q) =  
  if nin <= nout then q  
  else  
    (0, lazy Nil , nout+nin , sout ++ reverse sin)
```

```

open Stream (* voir le cours 4 *)
module FileDS : FIFO = struct
  type 'a t = int * 'a stream * int * 'a stream

  let empty = (0, lazy Nil, 0, lazy Nil)

  let check ((nin, sin, nout, sout) as q) =
    if nin <= nout then q
    else (0, lazy Nil, nin + nout, sout ++ reverse sin)

  let add x (nin, sin, nout, sout) =
    check (nin + 1, lazy (Cons (x, sin)), nout, sout)

  let remove (nin, sin, nout, sout) =
    match Lazy.force sout with
    | Nil -> raise Empty
    | Cons (x, sout') -> x, check (nin, sin, nout - 1, sout')
end

```

Discussion sur le coût

On observe les faits suivants :

- ▶ La fonction `check` a un coût constant : les opérations `++` et `reverse` sont indiquées, mais pas exécutées (fonctions paresseuses).
- ▶ La fonction `add` a un coût constant : elle appelle `check` après avoir ajouté un seul élément.
- ▶ La fonction `remove` a un coût constant *sauf* quand le `Lazy.force` déclenche le calcul d'un `reverse`, qui est monolithique.

Pour un usage non persistant de la structure de données, on peut faire le même calcul qu'avant, et obtenir un temps amorti en $O(1)$.

Pour un usage *persistant* de la structure de données, le temps amorti est *aussi* en $O(1)$, mais c'est plus dur à montrer ; nous donnons ici l'argument de façon informelle.

Preuve informelle de coût amorti $O(1)$ non-persistant

- Considérons une file f_0 avec les deux streams de même taille m , et une suite de $m + 1$ opérations

$$\begin{aligned} f_1 &= \text{snd}(\text{remove } f_0) \\ &\vdots \\ f_{m+1} &= \text{snd}(\text{remove } f_m) \end{aligned}$$

- La première opération `remove` introduit un `reverse r` de taille m .
- La dernière opération `remove` a besoin d'accéder au premier élément de `reverse r` et donc force son calcul (de coût m) ; ce coût est amorti par les `remove` précédents peu chers.
- Pour l'analyse de coût avec un usage non persistant, on s'arrête là.

Preuve informelle de coût $O(1)$ persistant, cas 1

- Considérons une file f_0 avec les deux streams de même taille m , et une suite de m opérations

$$f_1 = \text{snd}(\text{remove } f_0)$$

$$f_2 = \text{snd}(\text{remove } f_1)$$

$$\vdots$$

$$f_m = \text{snd}(\text{remove } f_1)$$

- La première opération `remove` introduit un `reverse r` de taille m .
- Le `reverse r` est donc déjà dans le stream de sortie de f_1 , et f_2, \dots, f_m partagent ce même `reverse r`.
- Le calcul sera fait une seule fois et partagé entre toutes les $m - 1$ copies (propriété des structures paresseuses partagées) ; donc le coût total du `reverse` est toujours m .

Preuve informelle de coût $O(1)$ persistant, cas 2

- Considérons une file f_0 avec les deux streams de même taille m , et une suite de m opérations

$$f_1 = \text{snd}(\text{remove } f_0)$$

$$\vdots$$

$$f_m = \text{snd}(\text{remove } f_0)$$

- La première opération `remove` introduit un `reverse r` de taille m . C'est donc pareil pour les autres.
- On a créé m copies d'un `reverse` suspendu, chacune sur un flot de taille m !
- Chacune est le deuxième argument d'une concaténation paresseuse de flots, le premier arguments ayant la taille m .
- Pour arriver à forcer le `reverse` *d'une seule* copie il faudra faire d'abord m opérations de `remove`. Ce sont donc ces `remove` qui paient pour le `reverse` !

L'expérience

- ▶ On crée un programme `runlazy.ml` qui contient les modules `Stream`, `FileDS`, les `test` et `test2` précédent, et :

```
let _ =
  match Sys.argv.(1), int_of_string (Sys.argv.(2)) with
  | "test", n -> test n
  | "test2", n -> test2 n
  | exception _ | _ ->
      failwith ("usage: _ ^ Sys.argv.(0) ^ \"_(test | test2) _<n>")
```

- ▶ Compiler avec `ocamlopt runlazy.ml -o runlazy`, et exécuter par exemple `time ./runlazy test2 1000000`.
- ▶ Voir également `runlist.ml` pour des tests similaires sur la version à base de listes. Dans ce cas, `test2 50000` est déjà très long.

Considérations sur l'efficacité

L'implémentation avec les streams est *plus chère* que celle avec les listes :

- ▶ on paye le surcoût des structures paresseuses
- ▶ on peut avoir plusieurs reverse suspendus dans la sortie
- ▶ Combien de reverse peut-on fabriquer au maximum lors d'une séquence de n opérations ?
- ▶ C'est le prix à payer pour avoir un coût amorti constant *avec un usage persistant*.
- ▶ Pour les usages non persistantes (aussi appelés éphémères), l'implémentation avec la double liste est plus efficace.

Le cas du temps réel

- ▶ Nos structures de données ont maintenant un temps d'exécution *amorti* constant, même pour un usage persistant.
- ▶ Cependant, de temps à l'autre, il faut effectuer une opération coûteuse (le reverse) qui peut prendre un temps important, et surtout *non borné* : le reverse prendra $O(n)$, et n peut être arbitrairement grand.
- ▶ Dans certaines conditions, par exemple dans les systèmes temps réel, on a besoin de s'assurer qu'*aucune* opération ne prend un temps de calcul non borné, et on est prêt à payer un surcoût en complexité de code, ou même une performance moindre en moyenne pour y arriver.

Les files temps réel

- ▶ Il est possible de définir une implémentation des files qui a un temps d'exécution *constant*, pour toutes les opérations.
- ▶ La partie coûteuse de notre code précédent, qui peut déclencher un reverse "monolithique" :

```
let check ((nin, sin, nout, sout) as q) =  
    if nin <= nout then q  
    else (0, lazy Nil, nin+nout, sout ++ reverse sin)
```

- ▶ On cherche du code équivalent, mais qui soit incrémental.
- ▶ L'idée est de retourner sin *progressivement* pendant qu'on consomme sout en sachant que :

à l'appel du else, on a $nin = nout + 1$.

Exemples (rotate.ml)

```
(* rotate f r acc = f ++ (reverse r) ++ acc *)
(* Hypothesis: length(r) = length(f)+1 *)
```

```
let rec rotate f r acc =
  match Lazy.force f, Lazy.force r with
  | Nil, Cons(y, _) -> lazy (Cons (y, acc))
  | Cons (x, xs), Cons(y, ys) ->
    let acc' = lazy (Cons (y, acc)) in
    lazy (Cons (x, rotate xs ys acc'))
  | _, _ -> assert false (* cas impossible *)
```


L'opération de rotation de la file

- ▶ On veut avoir, aux suspensions près :

`rotate f r acc = f ++ (reverse r) ++ acc`

- ▶ Pour bien comprendre, observons qu'à chaque appel récursif
 - ▶ on déplace dans le stream résultat un élément de `f`
 - ▶ on déplace un élément de la tête de `r` sur l'accumulateur `acc`
- ▶ Comme la longueur de `r` est égale à celle de `f+1`, on a le temps de retourner tout le stream `r` avant d'avoir besoin de son premier élément, et on ne déclenche donc pas de calcul monolithique.

Un exemple d'exécution simplifié

rotate est une suspension, et le calcul est déclenché seulement quand on consomme un élément x du résultat (on le marque \times). On abrège ici lazy (Cons ...) en $::$ et lazy Nil en nil.

rotate (1::~~2~~::3::~~4~~::nil) (7::~~6~~::5::~~4~~::nil) nil

↓

~~1~~::~~2~~ rotate (2::~~3~~::nil) (6::~~5~~::4::~~3~~::nil) (7::~~6~~::nil)

↓

~~1~~::~~2~~~~3~~ rotate (3::~~4~~::nil) (5::~~4~~::3::~~2~~::nil) (6::~~5~~::7::~~6~~::nil)

↓

~~1~~::~~2~~::~~3~~~~4~~ rotate nil (4::~~3~~::nil) (5::~~4~~::6::~~5~~::7::~~6~~::nil)

↓

~~1~~::~~2~~::~~3~~::~~4~~::5::~~6~~::7::~~8~~::nil

Dans cet exemple, le coût de chaque appel est bien constant.

Un exemple d'exécution réaliste

- ▶ En réalité, la stream de sortie n'est pas aussi linéaire : si on garde l'ancien code en remplaçant juste `(f ++ reverse r)` par `(rotate f r nil)`, l'insertion de 7 éléments de suite donne plutôt cette configuration :

```
rotate (rotate (rotate nil (1::nil) nil)
              (3::2::nil) nil)
      (7::6::5::4::nil) nil
```

- ▶ Et si on enfile n éléments à la suite, le nombre d'appels imbriqués à `rotate` grandit arbitrairement (combien ?)
- ▶ Pour obtenir le premier élément du résultat, on peut avoir à calculer $O(\log n)$ étapes de `rotate` !
- ▶ C'est mieux que le $O(n)$ qu'on avait avec `reverse`, mais ce n'est pas le coût constant qu'on cherchait ! On doit trouver un moyen pour ne pas accumuler les `rotate` imbriqués...

Dégeler progressivement les suspensions avant rotation

- ▶ Avant : on avait dans la structure la pile de sortie (éventuellement avec plusieurs `rotate` suspendues), et la pile d'entrée.
- ▶ L'idée lumineuse de Okasaki : on change la représentation en

$(pile_entree, pile_sortie, s)$

où la pile de sortie contient au plus une suspension qui est s .

- ▶ s est une “copie partagée” de la partie de la pile de sortie qui est une suspension, tel que forcer s fait aussi avancer le calcul de la pile de sortie.
- ▶ s sert seulement à faire ce forcing du calcul de la pile de sortie.
- ▶ Ainsi, l'exécution de `rotate` se fera toujours comme dans le cas de l'exemple simplifié vu auparavant.

L'ordonnanceur

- ▶ Invariant de (en, out, s) : $|out| - |s| = |en|$.
- ▶ Il y a une fonction `exec` qui sert d'ordonnanceur, et fait avancer le calcul de `s` d'une étape tant que c'est possible.
- ▶ Quand on a fini de forcer `s`, `exec` recommence avec le retournement de `en`.
- ▶ Il faut exécuter l'ordonnanceur chaque fois qu'on viole l'invariant : dans les cas `add` et `remove`.
- ▶ Du coup, plus besoin de connaître les longueurs des piles.
- ▶ En plus, la pile d'entrée peut être une liste ordinaire.

Les files temps réel I

open Stream

```
module RealTimeQueue : FIFO = struct
  type 'a t = 'a list * 'a stream * 'a stream
  let mkqueue out = ([], out, out)
  let empty = mkqueue (lazy Nil)

  let rec rotate f en acc =
    match Lazy.force f, en with
    | Nil, [y] -> lazy (Cons (y, acc))
    | Cons (x, xs), y :: ys ->
      let acc' = lazy (Cons (y, acc)) in
      lazy (Cons (x, rotate xs ys acc'))
    | _, _ -> assert false
```

Les files temps réel II

```
let  exec (en,out,s) = match Lazy.force s with
    | Cons (x, s') → (en, out, s')
    | Nil → mkqueue (rotate out en (lazy Nil))
```

```
let  add x (en,out,s) = exec (x::en, out, s)
```

```
let  remove (en,out,s) = match out with
    | lazy Nil → raise Empty
    | lazy (Cons(x,out')) → x, exec (en, out', s)
```

```
end
```

Remarque sur la complexité

- ▶ Ces files temps réel ont une complexité constante même dans le pire des cas, et même pour un usage persistant : elles sont plus efficaces que les files avec les streams que nous avons vu précédemment.
- ▶ Cependant, il y des surcoûts :
 - ▶ en temps, lié aux différentes suspensions qui sont mises en jeu,
 - ▶ en lisibilité du code, pour qui souhaite l'adapter
- ▶ Pour un usage non persistant, et non temps réel, les files avec les deux listes peuvent rester plus intéressantes.
- ▶ Pour des tests en pratique, voir `runreal.ml` sur le même modèle que `runlazy.ml`

Pour en savoir plus



Chris Okasaki.

Purely functional data structures.

Cambridge University Press, 1999.

Voir les chapitres 6 et 7.



Gerth Stølting Brodal and Chris Okasaki.

Optimal purely functional priority queues.

J. Funct. Program., 6(6) :839–857, 1996.