Programmation Fonctionnelle Avancée 5 : Des vertues de la paresse - Structures de données fonctionnelles persistantes efficaces

Ralf Treinen

Université Paris Diderot
UFR Informatique
Institut de Recherche en Informatique Fondamentale

PARIS DIDEROT

treinen@irif.fr

17 octobre 2018

Plan du cours

- Les Queues du chapitre 3 : efficacité ou persistance ?
- Les Queues révisitéés : persistantes et $O(1)$ amorti
- Queues temps réél : 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)

```
module type FIFO = sig
  type 'a t
  exception Empty
  val empty : 'a t
  val is_empty : 'a t -> bool
  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
  exception Empty
  type 'a t = 'a list * 'a list
  let empty = ([], [])
  let is_empty = function
    |([],[]) -> true
    | _      -> false
  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 *)
  exception Empty
  type 'a t = 'a list * 'a list
  let empty = ([], [])
  let is_empty = function
    | ([],[]) -> true
    | _       -> false

  let ncalls = ref 0   (* number of function calls *)
  let cost = ref 0     (* accumulated cost *)
```


Le module instrumenté (queue3.ml) II

```
  let incr c n = c:= !c + n
  let reset () = cost := 0; ncalls :=0

  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);
```


Le module instrumenté (queue3.ml) III

```
        match List.rev l1 with
        | [] -> raise Empty
        | a::l -> (a, ([], l))
end;;
```

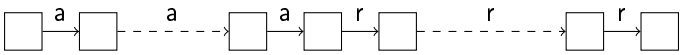
Exemples (queue5.ml)

```
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 let (_,q') = remove q in iterate_remove (n-1) q'

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

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


Séquence des opérations (queue5.ml)



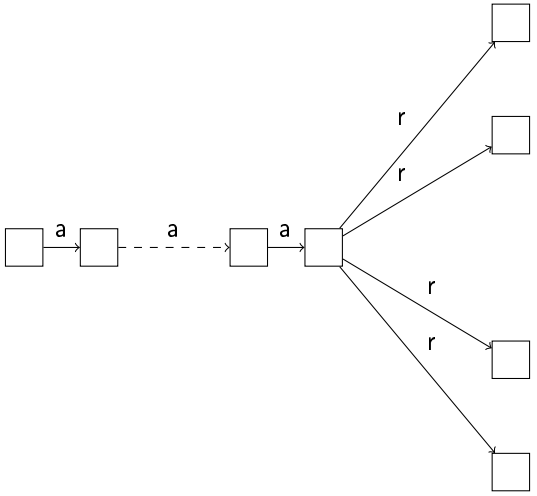
Exemples (queue6.ml)

```
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 (queue6.ml)



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 file d'origine*, 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 *démontré* avant.
  Ou 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 !
- Bien que notre structure est persistante, 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 $f_1 = snd(remove\ f)$, obtenue par renversant la pile d'entrée de f vers la pile de sortie de f_1.
- On construit $f_2 = snd(remove\ f)$, obtenue aussi par 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 par 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 f3 = snd (remove f)
```

(voir le dessin au tableau)

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 (le `List.rev`) comme dans l'exemple vu auparavant.
- La paresse est la solution ! Pourquoi ?
  - On a partage des structures qui ont le même origine.
  - Avec la paresse, les structures de données peuvent contenir des calculs suspendus (structures d'un type α `Lazy.t`).
  - En mettant les deux ensemble, on obtient la possibilité de partager un calcul suspendu !
  - 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, pour pouvoir contrôler facilement leur taille.
- 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 : il faut revenir à la question quand exactement on fait un 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 clèf 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 chère, 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 laisser grandir sans contrôle la pile d'entrée.

Mise en pratique : la taille des streams

- Invariant sur la taille des deux streams :
  - on garde le stream de sortie *toujours* plus long ou égal que le le stream d'entrée ;
  - dès qu'une opération peut violer cet invariant, on retourne *paresseusement* le stream d'entrée et on le concatène *paresseusement* au stream de sortie.
- Cette opération est réalisée par la fonction `check` suivante :

```
let check (l_in, s_in, l_out, s_out as q) =
if l_in <= l_out
   then q
   else (0, lazy Nil,
         l_out + l_in, s_out ++ reverse s_in);;
```



```
open Stream (* voir le cours 4 *)
module FileDS : FIFO = struct
 type 'a t = int * 'a stream * int * 'a stream
 exception Empty
 let empty = 0, lazy Nil, 0, lazy Nil
 let is_empty (_ , _, lout , _) = lout = 0

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

 let add x (lin , sin , lout , sout) =
     check (lin + 1, lazy(Cons (x, sin)), lout , sout)

 let remove (lin ,sin ,lout ,sout) =
  match sout with
   | lazy Nil               -> raise Empty
   | lazy (Cons (x, sout')) -> x,check (lin ,sin ,lout -1,sout')
end;;
```


Discussion sur le coût

On observe les faits suivants :

- La fonction `check` a coût constant : les opérations ++ et `reverse` sont indiquées, mais pas exécutées (fonctions paresseuses).
- La fonction `add` a coût constant : elle appelle `check` après avoir ajouté un seul élément.
- La fonction `remove` a 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 opérations

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

- 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` précédentes pas chères.
- 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

$$\begin{aligned} f_1 &= snd(remove\ f_0) \\ f_2 &= snd(remove\ f_1) \\ &\vdots \\ f_m &= snd(remove\ f_1) \end{aligned}$$

- 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 $f_2, \ldots, f_m$ partagent ce même `reverse r`.
- Le calcul sera fait une seul 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

$$\begin{aligned} f_1 &= snd(remove\ f_0) \\ &\vdots \\ f_m &= snd(remove\ f_0) \end{aligned}$$

- 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 qui contient les modules `Stream`, `FileDS`, et

```
let test2 n =
  let _ = repeat_remove n (iterate_add n empty) in ()
;;
let _ = test2 (int_of_string (Sys.argv.(1)))
```

- Compiler avec `ocamlopt runlazy.ml`, et exécuter `time ./a.out` avec les arguments 100000, 1000000, 10000000.

Considérations sur l'efficacité

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

- on paye le sur-coût 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 persistants (aussi appelés éphémères), l'implémentation avec la double liste est la 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 chère (le `reverse sin`) 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 sur-coût 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 chère de notre code précédent, qui peut déclencher un `reverse` "monolithique " :

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

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

  à l'appel du `else`, on a $lin = lout + 1$.

Exemples (rotate.ml)

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

let rec rotate (f,r,acc) =
  (* Hypothesis: length(r) = length(f)+1 *)

  match (!$f,!$r,acc) with
  | Nil, Cons(y,lazy Nil), a -> lazy (Cons (y, a))
  | Cons (x, xs), Cons(y,ys), a ->
    lazy (Cons (x,
                (rotate (xs, ys, lazy (Cons (y, a))))))
  | _, _, _ -> failwith "pattern impossible"
;;
```


L'opération de rotation de la file

- On veut avoir, aux suspensions près :

$$rotate(f, r, acc) = f + +(reverse\ r) + +acc$$

- Pour la 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 $*$).

rotate(1 : :2 : :3 : :[],7 : :6 : :5 : :4 : :[],[])
↓
~~1 : :~~(rotate(2 : :3 : :[], 6 : :5 : :4 : :[], 7 : :[]))
↓
~~1 : :2 : :~~(rotate(3 : :[], 5 : :4 : :[], 6 : :7 : :[]))
↓
~~1 : :2 : :3 : :~~(rotate([], 4 : :[], 5 : :6 : :7 : :[]))
↓
~~1 : :2 : :3 : :4 : :~~5 : :6 : :7 : :[]

Dans ce cas d'exemple, il semble bien que le coût de chaque appel est constant (pour simplicité on utilise la notation des listes).

Un exemple d'exécution réaliste

En réalité,le stream de sortie n'est pas aussi linéaire : après l'insertion de 7 éléments de suite, si on garde l'ancien code et on remplace juste `f ++ reverse r` avec `rotate(f,r,[])`, on se retrouve plutôt dans cette configuration :

rotate(rotate(rotate([],1 : :[],[]),3 : :2[],[]),7 : :6 : :5 : :4 : :[],[])

Et si on enfile n éléments à la suite, le nombre d'appels imbriqués à `rotate` grandit arbitrairement (Question : combien ?)

Ceci n'est pas bon du tout : 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 n'est pas le coût constant qu'on cherchait !
On doit trouver un moyen pour ne pas faire 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

$$(pile\_sortie, pile\_d'entree, s)$$

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 (f, r, s) : $|f| - |s| = |r|$.
- Il y a une fonction `exec` qui sert comme ordonnanceur, et fait avancer le calcul de s par une étape quand il reste du travail à faire.
- Quand la suspension s a terminé son travail, `exec` commence avec le retournement de r (la pile d'entrée).
- Il faut exécuter l'ordonnanceur chaque fois qu'on viole l'invariant : dans les cas `add` et `remove`.
- Du coup, on n'a plus besoin de connaître les longueurs des piles.
- En plus, la pile d'entrée peut être une liste ordinaire.

Les files temps réel I

```
module RealTimeQueue : FIFO = struct
type 'a t = 'a stream * 'a list * 'a stream
exception Empty

let empty = lazy Nil, [], lazy Nil

let is_empty (f, _, _) =
    match !$f with Nil -> true | _ -> false

let rec rotate (f,r,a) = match (!$f,r,a) with
  | Nil, [y], a -> lazy (Cons (y, a))
  | Cons (x, xs), y :: ys, a ->
      lazy (Cons (x,
                  (rotate (xs, ys, lazy (Cons (y, a))))))
```


Les files temps réel II

```
  | _, _, _ -> failwith "configuration illegale"

let exec (f,r,s) = match (f,r,!$s) with
  | f, r, Cons (x, s) -> f, r, s
  | f, r, Nil -> let f' = rotate (f, r, lazy Nil)
                 in f', [], f'

let add x (f, r, s) = exec (f, x :: r, s)

let remove (f,r,s) = match (!$f,r,s) with
  | Nil, _, _ -> raise Empty
  | Cons (x, f), r, s -> x, exec (f, r, 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 sur-coûts :
  - 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 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.