Evaluation paresseuse et structure fonctionnelles persistantes efficaces

- Evaluation stricte et evaluation paresseuse
- Les Queues de la sémaine passée : efficacité ou persistance ?
- Les Queues révisitéés : persistantes et $O(1)$ amorti
- Queues temps réél : persistantes et $O(1)$ constant
- Pour en savoir plus

L'ordre d'évaluation

Quand on programme, on veut savoir dans quel ordre nos commandes ou expressions sont évalués.
Il n'est pas toujours simple de repondre, avec les langages actuels :

En C

```
int foo (int x, int y)
{ return x+y}

/* que fait ceci? */

foo(i++,j++)

/* et ceci? (+ n'est pas un sequence point) */

i=foo(i++,j)+foo(i--,j)
```

C'est pour traiter ces questions qu'on étudie la *sémantique* des langages de programmation !

La stratégie d'évaluation dans les langages fonctionnels

Pour les langages fonctionnels, une question majeure est de savoir ce qui se passe quand on évalue une fonction :

- est-ce que le compilateur essaye d'évaluer le corps d'une fonction avant qu'elle soit appliquée ?
  Si on écrit `fun x -> x+fact(100)`, est-ce que le factoriel est calculé à chaque appel ou une seul fois ?
- si on écrit une expression `e a` (`e` appliqué à `a`) est-ce qu'on commence par évaluer `e` ou `a` ?
- si on écrit `(fun x -> e) a`, est-ce qu'on évalue l'argument `a` d'abord, ou on fait le passage de paramètres d'abord ?

Il y a plusieurs choix possibles, et on peut tester facilement ceux faits dans OCaml.

Pas d'évaluation "sous le λ"

OCaml n'essaye pas d'évaluer le corps d'une fonction avant qu'elle ne soit appliquée à un argument

```
(* pas d'evaluation sans arguments ... *)
let id = fun x -> Printf.printf "<in fun>\\n%!";x;;
|| val id : 'a -> 'a = <fun>
```

Il y a plein de bonnes raisons d'arreter l'évaluation quand on rencontre une abstration (aussi appelée "lambda" du λ-calcul utilisé dans le cours de *Sémantique* pour traiter ces questions de façon générale).
Tous les langages fonctionnels font ce choix.

On évalue l'argument avant de le passer en paramètre

```
(* l'argument d'abord ... *)
let f = (fun x -> Printf.printf "<in fun>\\n%!";x + x)
        (Printf.printf "<in arg>\\n%!";35+24);;
|| <in arg>
|| <in fun>
|| val f : int = 118
```

On fait donc le calcul $35 + 24$ une seule fois, avant de le passer en argument à la fonction qui peut l'utiliser plusieurs fois.
Ce choix n'est pas le seul possible : d'autres langages fonctionnels, comme Haskell, passent d'abord l'argument, non évalué, en paramètre à la fonction, et ne lancent le calcule qu'au moment où on aura besoin de son résultat.
Cela permet à Haskell, par exemple, de ne jamais calculer `fact 100` dans une expression `(fun x -> 3) (fact 100)`

On évalue d'abord l'argument

Si on trouve une expression avec des applications imbriquées, comme `(e1 e2) e3`, OCaml évalue toujours d'abord l'argument e3, puis l'expression en position fonctionnelle, `(e1 e2)`.

```
(* a gauche d'abord ... *)
(Printf.printf "<in expr>\\n%!";
 (fun f x -> Printf.printf "<in fun>\\n%!";f x)
 (fun z -> z*2)
) (Printf.printf "<in arg>\\n%!";35);;
|| <in arg>
|| <in expr>
|| <in fun>
|| - : int = 70
```


OCaml fait de l'évaluation stricte

L'ensemble de ces choix s'appelle une *stratégie d'évaluation*

Celle utilisée par OCaml est appelée *évaluation stricte* (voir plus en profondeur dans le cours de Sémantique).

L'évaluation stricte n'est pas toujours adaptée

Precalculer les factoriels

Pour écrire un algorithme combinatoire, on a besoin du factoriel d'un entier naturel arbitraire.
Nous ne voulons pas le recalculer à chaque fois qu'on en a besoin ; on préfère garder la liste de tous les factoriels.

Pour cela, on écrit le code suivant, mais on s'aperçoit qu'il ne fait pas ce que l'on veut (essayez !) :

```
let rec fact = function 0 -> 1 | n -> n*(fact (n-1));;
let rec fact_from n = (fact n)::(fact_from (n+1));;
let fact_nat = fact_from 0;;
```

Evaluation paresseuse

Notre défi :

- on veut que la liste (infinie) ne soit pas calculée tout de suite
- mais seulement au coup par coup, quand on a besoin d'en prendre des éléments

En profitant du fait qu'en OCaml on n'évalue pas le corps d'une fonction, il est possible de simuler un calcul paresseux en utilisant des paramètres auxiliaires.

Evaluation paresseuse du pauvre, avec les fermetures

On change la définition du type des listes pour prévoir une 'queue ' de liste dont l'évaluation est bloquée sous une abstraction :

```
type 'a lazyltip = Nil | Cons of 'a * 'a lazyl
and 'a lazyl = unit -> 'a lazyltip;;
|| type 'a lazyltip = Nil | Cons of 'a * 'a lazyl
|| and 'a lazyl = unit -> 'a lazyltip

let rec fact_from n : 'a lazyl = fun () ->
        Cons(fact n,
             (fun () -> fact_from (n+1) ()));;
|| val fact_from : int -> int lazyl = <fun>
```


Evaluation paresseuse du pauvre, avec les fermetures

```
let rec take_l n s = match n with
   0 -> []
 | n -> match s() with Nil -> []
        | Cons(v,r) -> v::(take_l (n-1) r);;
|| val take_l : int -> 'a lazyl -> 'a list = <fun>


let fact_nat = fact_from 0;;
|| val fact_nat : int lazyl = <fun>


take_l 5 fact_nat;;
|| - : int list = [1; 1; 2; 6; 24]
```


Limites de notre evaluation paresseuse

La petite astuce avec les fermetures permet d'écrire des structures de données paresseuses, mais ce n'est pas encore ce que nous voulons !

```
let rec fact_from n = fun () ->
    Cons((Printf.printf "*";fact n),
         (fun () -> fact_from (n+1) ()));;

let fact_nat = fact_from 0;;
```

Limites de notre evaluation paresseuse

```
take_l 5 fact_nat;;
|| *****- : int list = [1; 1; 2; 6; 24]

take_l 5 fact_nat;;
|| *****- : int list = [1; 1; 2; 6; 24]
```

Notre code permet de décrire des listes infinies, *mais* chaque fois qu'on visite la même liste, on relance le calcul de ses éléments.
C'est très inefficace !

Paresse + *partage* !

Partager les résultat d'un calcul

Chaque fois qu'une partie d'une structure de données paresseuse est *dégélée* et calculée, on veut qu'elle soit remplacée, silencieusement, par le résultat de ce calcul dans la structure de donnée.
La prochaîne fois qu'on y accède, on veut trouver la résultat déjà évalué.

En OCaml

Le module `Lazy` fournit la mécanique nécéssaire pour la paresse et le partage.

Le module Lazy de la librairie OCaml

```
type 'a t
exception Undefined
val force : 'a t -> 'a
val lazy_is_val : 'a t -> bool
```

- une valeur de type `'a Lazy.t` est appelée une *suspension* et contient un calcul paresseux de type `'a`.
- on construit des valeurs de type `'a Lazy.t` avec le mot clé réservé `lazy` : l'expression `lazy (expr)` crée une suspension contenant le calcul `expr` *sans l'évaluer*
- `Lazy.force s` force l'évaluation de la suspension `s`, en renvoie le résultat, et remplace la valeur dans la structure de donnée : *si appelé sur une partie déjà dégélée, il ne refait pas le calcul*
- `Lazy.lazy_is_val s` teste si la suspension `s` est déjà dégélée

Streams : la liste des factoriels révisitée avec Lazy

```
type 'a streamtip = Nil | Cons of 'a * 'a stream
and  'a stream = 'a streamtip Lazy.t;;

let rec fact_from n : int stream =
    lazy (Cons((Printf.eprintf "*%!";fact n),
            lazy (Lazy.force (fact_from (n+1)))));;

let rec take_l n (s:'a stream) = match (n,s) with
      0,s -> []
    | n,s -> match (Lazy.force s) with Nil -> []
           | Cons(v,r) -> v :: (take_l (n-1) r);;
```

Notez bien qu'on a placé les `lazy` *exactement* là où on avait mis des fermetures `fun () ->`, et on a placé des `Lazy.force` *exactement* là où on avait forcé l'évaluation en appliquant à `()`.

Streams : la liste des factoriels révisitée avec Lazy

```
let fact_nat = fact_from 0;;
|| val fact_nat : int stream = <lazy>

take_l 5 fact_nat;;
|| *****- : int list = [1; 1; 2; 6; 24]

take_l 5 fact_nat;;
|| - : int list = [1; 1; 2; 6; 24]
```

La promesse est tenue : le calcul des premiers 5 éléments est fait *seulement la première fois*.
Jouez avec ce code pour vous familiariser avec l'idée des suspensions et de comment et où les forcer.
Exercice : modifiez le code de telle sorte que le calcul de chaque nouvel element de `fact_nat` utilise *une seule* multiplication.

Les opérations de base sur les streams ou flots

```
module type STREAM = sig
  type 'a streamhead = Nil | Cons of 'a * 'a stream
  and 'a stream = 'a streamhead Lazy.t

  (* concatenation de streams *)
  val (++) : 'a stream -> 'a stream -> 'a stream

  (* un stream contenant les premiers n elements *)
  val take : int -> 'a stream -> 'a stream

  (* le stream sans les premiers n elements *)
  val drop : int -> 'a stream -> 'a stream
  val reverse : 'a stream -> 'a stream
end;;
```


Implémentation des opérations de base I

En voici une implémentation en OCaml :

```
let (!$) = Lazy.force (* abbreviation *)

module Stream : STREAM = struct

type 'a streamhead = Nil | Cons of 'a * 'a stream
and 'a stream = 'a streamhead Lazy.t

(* concatenation *)
let (++) s1 s2 =
  let rec aux s = match !$s with
  | Nil -> !$s2
  | Cons (hd, tl) -> Cons (hd, lazy (aux tl))
in lazy(aux s1)
```


Implémentation des opérations de base II

```
(* copie paresseuse des premiers n elements *)
let take n s =
  let rec take' n s = match n, s with
  | 0, _ -> Nil
  | n, s -> match !$s with
            Nil -> Nil
          | Cons (hd, tl) ->
              Cons (hd, lazy (take' (n - 1) tl))
in lazy(take' n s)
```

Implémentation des opérations de base III

```
(* le stream apres n elements, monolithique! *)
let drop n s =
  let rec drop' n s =
    match n with 0 -> !$s
    | n -> match !$s with
           | Nil -> Nil
           | Cons (_, tl) -> (drop' (n - 1) tl) in
  match n with 0 -> s | n -> lazy (drop' n s)
```


Implémentation des opérations de base IV

```
(* retourne un stream, monolithique! *)
let reverse s =
  let rec reverse' acc s = match !$s with
    | Nil -> acc
    | Cons (hd, tl) ->
        reverse' (Cons (hd, lazy acc)) tl in
  lazy (reverse' Nil s)
end;;
```


Remarques

Les opérations `drop` et `reverse` sont *monolithiques*, dans le sens que, dès qu'on essaye de prendre le premier élément du stream résultant :

- pour `drop n s`, les premiers n éléments de s sont forcés
- pour `reverse s`, tout le stream s est forcé.

On ne suspend donc que le début de l'opération, mais une fois qu'on essaye d'en voir le résultat, toute l'opération est éxécutée d'un bloc.

ATTENTION Ne pas confondre avec le module `Stream` qui existe déjà dans la librairie standard OCaml et qui sert à construire des analyseurs recursifs descendants

Exercices

Réalisez les fonctions supplémentaires suivantes :

```
module type STREAMEXT = sig
  include module type of Stream
  val of_list : 'a list -> 'a stream
  val to_list : 'a stream -> 'a list
  val push : 'a -> 'a stream -> 'a stream
  val pop : 'a stream -> ('a * 'a stream) option
  val map : ('a -> 'b) -> 'a stream -> 'b stream
  val filter : ('a -> bool) -> 'a stream -> 'a stream
  val split : 'a stream -> 'a stream * 'a stream
  val join: 'a stream * 'a stream -> 'a stream
end;;
```

Assurez-vous que votre réalisation est bien paresseuse.

Syntaxe plus moderne

En OCaml il est possible d'utiliser le mot clé `lazy` même dans les motifs utilisés dans les définitions par cas ; cela permet souvent de se passer de l'usage de `Lazy.force`. Par exemple, un morceau de code tel que

```
match Lazy.force s with
  | Nil -> ...
  | Cons (_, tl) -> ...
```

peut s'écrire de façon équivalente comme :

```
match s with
  | lazy Nil -> ...
  | lazy (Cons (_, tl)) -> ...
```


Réprise et critique des files efficaces

Reprénons nos files fonctionnelles *efficaces* I

```
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;;
```

L'analyse de coût amorti donnait $O(1)$... *sous quelles hypothèses ?*

Instrumentons un peu le code ... I

```
module FifoDLCount = struct
  exception Empty
  type 'a t = 'a list * 'a list
  let empty = ([], [])
  let is_empty =
  function
    ([],[]) -> true
  |   _     -> false

  (* on ajoute de compteurs pour tracer le cout *)
  let ncalls = ref 0
  let cost = ref 0
  let incr c n = c:= !c +n
  let reset c = c := 0
```

Instrumentons un peu le code ... 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 -> (a, ([], l))
end;;
```


Instrumentons un peu le code ... III

```
open FifoDLCount
let e = empty;;
|| val e : 'a list * 'b list = ([], [])

let l = List.fold_left (fun l x -> add x l)
        e [1;2;3;4;5];;
|| val l : int list * 'a list = ([5; 4; 3; 2; 1], [])

ncalls;;
|| - : int ref = {contents = 5}

cost;;
|| - : int ref = {contents = 5}
```


Instrumentons un peu le code ... IV

Mais le cout n'est plus linéaire si on utilise la structure de facon persistante !

```
let (v1,l1) = remove l;;
|| val v1 : int = 1
|| val l1 : int list * int list = ([], [2; 3; 4; 5])

let (v2,l2) = remove l;;
|| val v2 : int = 1
|| val l2 : int list * int list = ([], [2; 3; 4; 5])

let (v3,l3) = remove l;;
|| val v3 : int = 1
|| val l3 : int list * int list = ([], [2; 3; 4; 5])
```


Instrumentons un peu le code ... V

```
let (v4,l4) = remove l;;
|| val v4 : int = 1
|| val l4 : int list * int list = ([], [2; 3; 4; 5])

let (v5,l5) = remove l;;
|| val v5 : int = 1
|| val l5 : int list * int list = ([], [2; 3; 4; 5])

ncalls;;
|| - : int ref = {contents = 10}

cost;;
|| - : int ref = {contents = 30}
```

Si on fait un usage persistant, ce n'est plus efficace !

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 *demontré* avant.
Ou est l'erreur ?
Reponse : notre analyse excluait ce genre de séquences d'opérations !

L'hypothèse cachée dans l'analyse de coût amorti

> *Quand on éxécute un remove qui fait appel à* `List.rev`*, on a n éléments sur la deuxième liste, donc $2n$ crédits ; on utilise n crédits pour payer le List.rev, 1 crédit pour payer l'enlévement d'un élément, et on laisse les $n-1$ crédits restants sur la première liste, qui contient $n-1$ éléments.*

Attention 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 étés déjà consommés lors d'un appel précédent !
Comme on vient de voir, en programmation fonctionnelle il est très naturel d'utiliser les structures de façon persistante ! Peut-on faire quelque chose ?

Files efficaces persistantes

Analyse des besoins

Pour avoir des files effices même avec un usage persistant, il faut éviter la duplication "gratuite" des opération chères (le `List.rev`) comme dans l'exemple vu auparavant.

Pour cela, on change la structure pour 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 pemettent 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 liste de droite.

Mise en pratique : réprésentation interne

Pour effectuer ces deux changement, on procède en plusieurs étapes.

Changement de la représentation interne

- on remplace les deux listes par deux streams,
- on garde trace de la taille de chaque stream, pour pouvoir contrôler facilement si ces tailles sont équilibrées

Le type de la file devient donc :

```
open Stream
type 'a t = int * 'a stream * int * 'a stream;;
```


Mise en pratique : la taille des streams

Invariant sur la taille des deux streams

- on garde le stream de tête de pile *toujours* plus long ou égal que le le stream de fond de pile
- dès qu'une opération peut violer cet invariant, on retourne *paresseusement* le stream de fond de pile et on le concatène *paresseusement* au stream de tête.

Cette opération est réalisée par la fonction `check` suivante :

```
let check (lenf, f, lenr, r as q) =
  if lenr <= lenf then q
  else (lenf + lenr, f ++ reverse r, 0, lazy Nil);;
```


La nouvelle file I

```
open Stream
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 (lenf, _, _, _) = lenf = 0

 let check (lenf, f, lenr, r as q) =
   if lenr <= lenf then q
   else (lenf + lenr, f ++ reverse r, 0, lazy Nil)

 let add x (lenf, f, lenr, r) =
     check (lenf, f, lenr + 1, lazy(Cons (x, r)))
```


La nouvelle file II

```
 let remove (lenf,f,lenr,r) =
  match Lazy.force f with
   | Nil          -> raise Empty
   | Cons (x, f') -> x,check (lenf - 1, f', lenr, r)
end;;
```

Discussion sur le côut

On observe les faits suivants :

- la fonction `check` a coût constant : les opérations ++ et `reverse` sont indiquées, mais pas éxécutées (fonctions paresseuses)
- la fonction `add` a coût constant : elle appelle `check` après avoir ajouté un seul élément
- la fonction `remove` a coût constant *sauf* si 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 prouver ; nous donnons ici l'argument de façon informelle.

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

Les idées essentielles de la preuve se voient en considérant une file f_0 avec les deux streams de même taille m, et une suite de $m+1$ opérations

$$f_i = snd(remove\ f_{i-1}), \quad 0 < i \leq m+1$$

La première operation `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 côut m) ; ce coût est amorti sur toute la séquence des m opérations pas chères précedentes.
Pour l'analyse de coût avec un usage non persistant, on s'arrétait là.

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

Si quelqun utilise la structure de façon persistante et fait k copies au milieu de cette séquence, on a deux cas :

- les k copies sont faites sur une des f_i avec $i > 0$; dans ce cas, le `reverse r` est déjà dans le stream de gauche de f_i, et *toutes* les copies partagent ce même `reverse r` ; le calcul sera fait une seul fois et partagé entre toutes les k copies (propriété des structures paresseuses partagées) ; donc le coût du `reverse` est toujours m et pas km comme dans le cas des listes ;
- les k copies sont faites sur f_0 : dans ce cas, le `reverse` n'est pas encore sur le stream de gauche, et il ne sera donc pas partagé entre les k copies ; mais pour arriver à forcer le `reverse`, pour chaque copie, il faudra faire m opérations, donc on pourra amortir le coût du `reverse` dans chaque copie séparément

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 à gauche

  Question : 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.

Files temps-réél

Le cas tu temps réél

Nos structures de données ont maintenant un temps d'éxécution *amorti* constant, même pour un usage persistant.

Cependant, de temps à autre, il faut effectuer une opération *chère (le `reverse r`)* 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-réél, on a besoin de s'assurer que *aucune* opération prend un temps de calcul non borné, et on est prêts à payer un surcoût en complexité de code, ou même un performance moindre en moyenne pour y arriver.

Les files temps réél

Il est possible de définir une implémentation des files qui a un temps d'éxécution *constant*, pour toutes les opérations.

Pour y arriver, nous dévons nous attaquer à la partie chère de notre code précédent, qui peut déclencher un `reverse` monolithique

```
let check (lenf, f, lenr, r as q) =
  if lenr <= lenf then q
  else (lenf + lenr, f ++ reverse r, 0, lazy Nil)
```

On cherche du code équivalent, mais qui soit, lui, incrémental.

L'idée est de retourner `r` *progressivement* pendant qu'on consomme `f` en sachant que :

- à l'appel du `else`, la longeur de `r` est égale à celle de `f` plus 1

L'opération de rotation de la file I

On écrit la fonction recursive `rotate(f,r,acc)` suivante

```
let rec rotate (f,r,acc) = 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"
;;
```

Pour la comprendre, observons qu'à chaque rappel récursif

- on déplace dans le stream résultat un élément de `f`
- on déplace un élément de la tête de `r` sur l'accumulateur `acc`

L'opération de rotation de la file II

Comme la longueur de `f` est égale à celle de `r`+1, on a le temps de retourner tout le stream `r` avant d'avoir besoin de son premier élément, et on ne declenche donc pas de calcul monolithique.

Un exemple d'éxécution simplifié

`rotate` est une suspension, et le calcul est declenché 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'éxécution réaliste

En réalité,le stream de gauche 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 prémier 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...

Dégéler progressivement les suspensions avant rotation

L'idée lumineuse de Okasaki est de garder une copie partagée du stream de gauche et de l'utiliser pour en forcer, *une à la fois* et à chaque opération sur la file, *toutes* les suspensions *avant* la prochaine rotation : ainsi, l'éxécution de `rotate` se fera toujours comme dans le cas de l'exemple simplifié vu auparavant.

La fonction clé pour obtenir ce résultat est la suivante, où `s` contient la partie de la copie du stream `f` dont les suspensions n'ont pas encore étés forcées.

```
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'
```

Invariant sur la taille de `f` et `r`

La *copie* du stream `f` est consommée progressivement, et remplit une double fonction :

- d'un cotè, il permet de forcer les suspensions dans `rotate` avant la prochaîne rotation, et d'éviter ainsi l'imbrication des `rotate` ;
- de l'autre coté, la longueur de cette copie est *toujours égale à* la longueur de `f` moins la longeur de `r` ; on peut donc l'utiliser pour savoir quand declencher une rotation : il suffit de le faire quand cette copie devient vide, ce qui indique une différence de 0 entre les tailles de `f` et `r` ; c'est grâce à cette propriété que l'on n'a plus besoin de garder l'information sur la longueur de `f` et `r`.

Simplification de la file

Avec ces fonctions, `rotate` et `exec`, on peut simplifier la représentation interne de la file :

- on n'a plus besoin de garder trace de la longeur de chaque stream,
- le stream de droite peut être remplacé par une liste
- on ajoute, à la fin, le stream qui sert de scheduler

Cela donne

```
type 'a t = 'a stream * 'a list * 'a stream
```

Et maintenant nos files temps réél s'écrivent simplement comme suit

Les files temps réél 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 réél 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 réél ont une complexité constante même dans le cas le pire, 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-réél, 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.

Attention Il existe une implémentation en OCaml des structures de Okasaki faite par Markus Mottl, mais l'implémentation des files avec les streams qu'on y trouve *n'est pas correcte*, avant la version 1.0.10 : le `reverse r` n'est pas une suspension, et n'est donc pas partagé.