Structures de données persistantes, troisième cours

Concilier amortissement et persistance : de l'importance de la paresse

Xavier Leroy

2023-03-23

Collège de France, chaire de sciences du logiciel
xavier.leroy@college-de-france.fr

Amortissement

Analyser des suites d'opérations

Rappel du 1er cours :

Un programme =

des algorithmes abstraits

+ des structures de données efficaces adaptées

Un algorithme abstrait exécute plusieurs opérations de suite sur une même structure de données.

L'important n'est pas que chaque opération soit très efficace, mais que la séquence des opérations prenne peu de temps.

Exemple

n opérations en temps $\log n$ chacune $\rightarrow$ temps total $\mathcal{O}(n \log n)$

$n - 1$ opérations en temps constant, 1 opération en temps linéaire $\rightarrow$ temps total $\mathcal{O}(n)$

Une pile sous forme de tableau extensible

```
class Stack {
  private int[] stk;  private int sp;
  Stack () { stk = new int[1]; sp = 0; }
  void push(int v) {
      if (sp >= stk.length) {
          int[] newstk = new int[2 * stk.length];
          System.arraycopy(stk, 0, newstk, 0, stk.length);
          stk = newstk;
      }
      stk[sp] = v; sp++;
  }
  int top() { return stk[sp - 1]; }
  void pop() { sp--; }
}
```

Quand la pile est pleine, on réalloue un tableau deux fois plus grand, et on copie l'ancien tableau dans le nouveau.

Analyse d'une séquence d'opérations `push`

Considérons une suite de n `push`, avec $2^{k-1} < n \leq 2^k$.

On a fait n écritures `stk[sp] = v`, pour un temps total de n.

On a redimensionné le tableau $k-1$ fois :
de la taille 1 à la taille 2, …, de la taille 2^{k-1} à la taille 2^k.

Chaque redimensionnement $p \to 2p$ prend un temps p.

Temps total de redimensionnement : $\sum_{i=0}^{k-1} 2^i = 2^k - 1$.

Temps total pour n `push` : $2^k - 1 + n \leq 2n + n = 3n$.

Analyse d'une séquence d'opérations `push`

Considérons une suite de n `push`, avec $2^{k-1} < n \leq 2^k$.

On a fait n écritures `stk[sp] = v`, pour un temps total de n.

On a redimensionné le tableau $k-1$ fois :
de la taille 1 à la taille 2, …, de la taille 2^{k-1} à la taille 2^k.

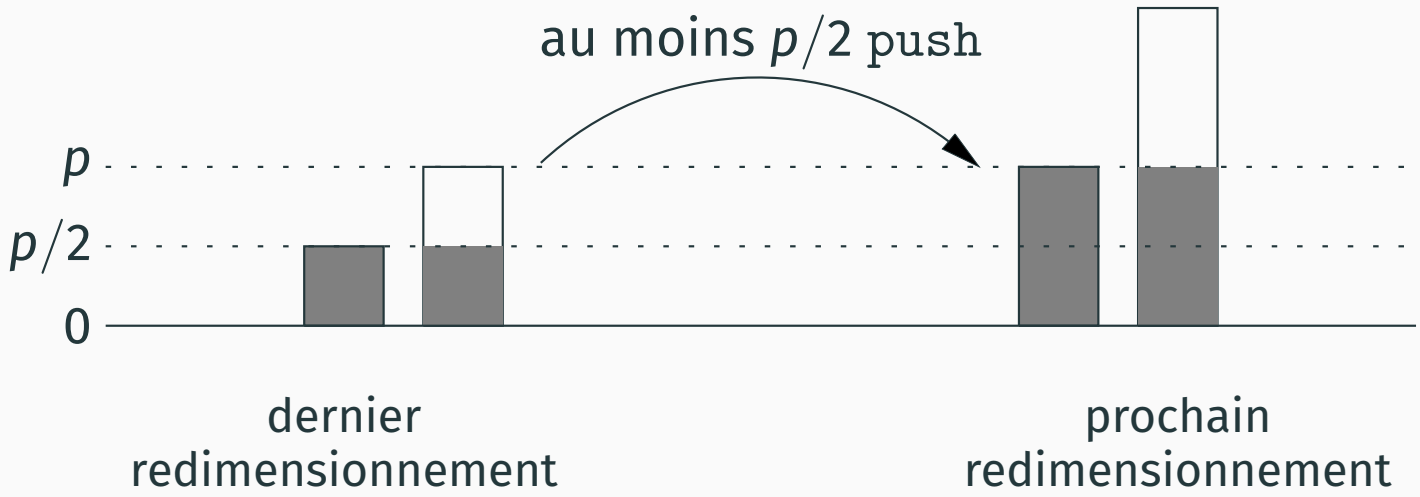
Chaque redimensionnement $p \to 2p$ prend un temps p.

Temps total de redimensionnement : $\sum_{i=0}^{k-1} 2^i = 2^k - 1$.

Temps total pour n `push` : $2^k - 1 + n \leq 2n + n = 3n$.

On dit que `push` s'exécute en temps amorti constant,
puisque toute séquence de n `push` prend au plus le temps kn
pour une certaine constante k.

Une analyse plus locale



Chaque redimensionnement laisse la pile à moitié vide.

Si p est la nouvelle taille, il faudra donc au moins $p/2$ opérations `push` avant d'avoir à la redimensionner (pour un coût p).

Le «surcoût» de 2 sur chacun des $p/2$ `push` permet donc d'anticiper le coût p du prochain redimensionnement.

La méthode du banquier (aussi appelée méthode comptable)

En termes financiers :

$$\text{coût facturé} \geq \text{coût réel} + \Delta\text{trésorerie}$$

En termes de temps d'exécution d'une opération :

$$\text{temps amorti} \geq \text{temps réel} + \Delta\text{crédits-temps}$$

La «trésorerie» (les crédits temps) doit toujours rester positive.

Dans l'exemple de la pile :

Opération	Temps amorti	Temps réel	Δcrédits
`top`, `pop`	1	1	0
`push` (en place)	3	1	$+2$
`push` (redim.)	3	$p+1$	$-p$

Il y a au moins $p/2$ `push` en place avant un `push` qui redimensionne, donc les crédits temps restent ≥ 0.

La méthode du physicien (aussi appelée méthode du potentiel)

Une fonction de potentiel Φ : état de la structure $\to \mathbb{R}_+$

Chaque opération doit vérifier

$$\text{temps amorti} \geq \text{temps réel} + \Delta\Phi$$

Pour la pile, on prend $\Phi = \max(2\texttt{sp} - p, 0)$
avec $p = \texttt{stk.length}$.

Opération	Temps amorti	Temps réel	$\Delta\Phi$
`top`	1	1	0
`pop`	1	1	0 ou -2
`push` (en place)	3	1	0 ou $+2$
`push` (redim.)	3	$p+1$	$-p+2$

Juste avant un redimensionnement, $\Phi = 2p - p = p$.
Juste après, $\Phi = 0$, et à la fin du `push` $\Phi = 2$.

Correction de l'analyse amortie

Sur une séquence d'opérations :

$$\sum \text{temps amortis} \geq \sum \text{temps réels} + \sum \Delta\Phi$$

Si $s_0, \ldots, s_n$ sont les états successifs de la structure,

$$\sum \Delta\Phi = (\Phi(s_n) - \Phi(s_{n-1})) + \cdots + (\Phi(s_1) - \Phi(s_0)) = \Phi(s_n) - \Phi(s_0)$$

Comme $\Phi \geq 0$ et $\Phi(s_0) = 0$ (généralement), on a

$$\text{temps de la séquence} = \sum \text{temps réels} \leq \sum \text{temps amortis}$$

Le temps d'exécution de la séquence est bien borné par le résultat de l'analyse amortie.

Méthode du physicien vs. méthode du banquier

La méthode du physicien est plus systématique :

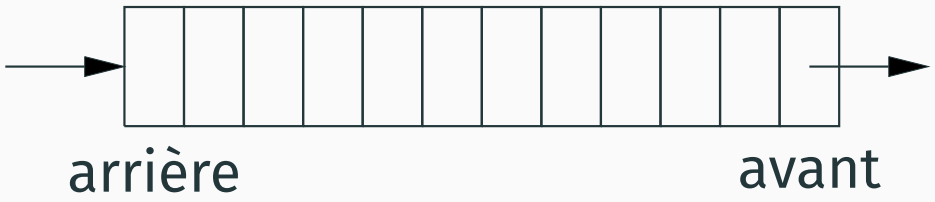
- Une fois choisie la fonction Φ, les coûts amortis se calculent presque automatiquement comme $\max(\text{coût réel} + \Delta\Phi)$.

La méthode du banquier est plus flexible :

- Peut prendre en compte l'historique des opérations et pas juste l'état courant de la structure.
- On peut répartir les crédits-temps sur plusieurs comptes attachés à différentes parties de la structure.

(Pour une formalisation des crédits temps en logique de séparation, voir l'exposé de F. Pottier, *Raisonner à propos du temps en logique de séparation*, 01/04/2021, Collège de France,)

Une file d'attente persistante

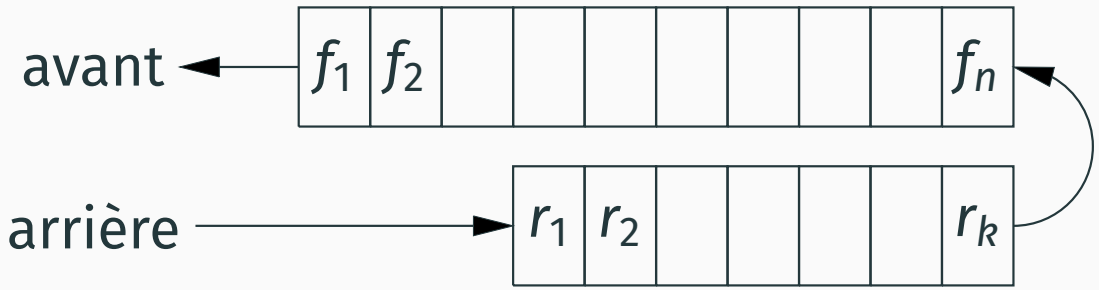


Implémentation triviale : une liste.

```
type 'a queue = 'a list
let empty = []
let isempty q = match q with [] -> true | _ -> false
let head q = match q with h :: t -> h | _ -> raise Empty
let tail q = match q with h :: t -> t | _ -> raise Empty
let add x q = q @ [x]
```

L'opération `add` est inefficace : temps $\mathcal{O}(n)$.

Une file d'attente persistante



Implémentation par deux listes :

- liste «avant» (f, *front*) : premier sorti en tête de liste ;
- liste «arrière» (r, *rear*) : dernier entré en tête de liste.

Lorsque la liste avant est vide, on peut la remplir en inversant la liste arrière : $([\,], r) \to (\texttt{rev}(r), [\,])$.

Cela prend un temps linéaire $\mathcal{O}(|r|)$, mais ce temps est amorti ! (par les $|r|$ insertions dans la file qui ont créé cette liste r).

Implémentation fonctionnelle de la file d'attente

```
type 'a queue = 'a list * 'a list
let empty = ([], [])
let isempty (f, r) = (f = [])
let head = function
  | ([], _) -> raise Empty
  | (x :: f, _) -> x
let tail = function
  | ([], _) -> raise Empty
  | (_ :: [], r) -> (List.rev r, [])
  | (_ :: f , r) -> (f, r)
let add x = function
  | ([], _) -> ([x], [])
  | (f, r) -> (f, x :: r)
```

Invariant : si la file n'est pas vide, $f \neq []$.

Autrement dit : si $f = []$ alors $r = []$ et la file est vide.

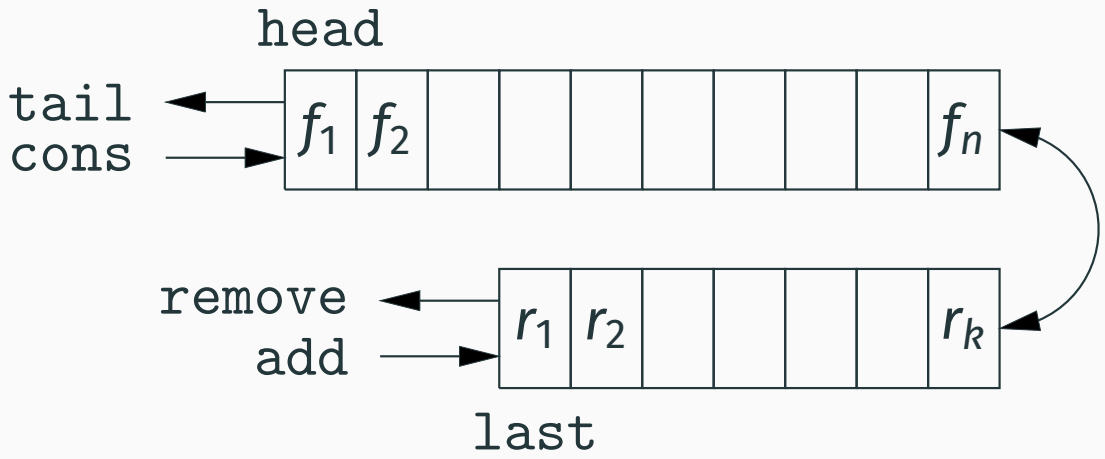
Analyse avec amortissement

Le potentiel est la longueur de la liste «arrière» :

$$\Phi((f, r)) \stackrel{def}{=} |r|$$

Opération	Temps amorti	Temps réel	$\Delta\Phi$
`add`	2	1	$+1$
`head`	1	1	0
`tail` (sans inversion)	1	1	0
`tail` (avec inversion)	1	$1 + \|r\|$	$-\|r\|$

Extension : file persistante à double entrée (*dequeue*)



On ajoute la possibilité d'insérer en tête de la file (`cons`) et de consulter et d'enlever la queue de la file (`last`, `remove`).

Même implémentation par deux listes (f, r) mais :

- $([\,], r) \to (\texttt{rev}\ r_2, r_1)$ avec $(r_1, r_2) = r$ coupé au milieu.
- $(f, [\,]) \to (f_1, \texttt{rev}\ f_2)$ avec $(f_1, f_2) = f$ coupé au milieu.

Potentiel pour l'analyse amortie : $\Phi((f, r)) = \Big|\ |f| - |r|\ \Big|$.

Problème avec l'utilisation persistante

Notre file est implémentée en style fonctionnel pur et peut donc être utilisée de manière persistante, c.à.d. en réutilisant plusieurs fois un état intermédiaire de la pile.

```
let q = add 3 (add 2 (add 1 empty)) in
let q1 = tail q and ... and qN = tail q in ...
```

L'état `q` est représenté par $f = [1]$ et $r = [3; 2]$.

Chacun des N appels à `tail q` va inverser la liste r, pour un surcoût total de $2N$.

Ce surcoût ne peut pas être amorti par les 3 opérations `add` : quel que soit leur coût amorti, il ne compense pas $2N$ quand $N \to \infty$.

Amortissement et persistance

$$\sum \text{temps amortis} \geq \sum \text{temps réels} + \sum \Delta\Phi$$

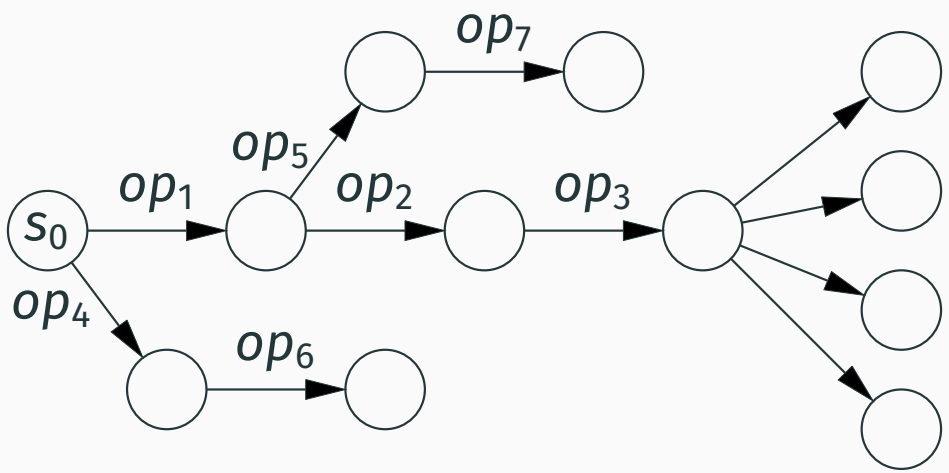
Si la structure est éphémère, ou persistante mais utilisée de manière linéaire (*single-threaded*), chaque état intermédiaire est utilisé une seule fois :

Alors on a bien $\sum \Delta\Phi = \Phi(s_n) - \Phi(s_0)$ et le temps réel de la séquence est bien borné par $\sum$ temps amortis.

Amortissement et persistance

$$\sum \text{temps amortis} \geq \sum \text{temps réels} + \sum \Delta\Phi$$

Une structure persistante peut aussi être utilisée de manière non linéaire, avec réutilisation d'états intermédiaires :



Dans ce cas, on a aucune garantie que $\sum \Delta\Phi \geq 0$, ni que le temps réel de la séquence est bien borné par $\sum$ temps amortis.

Évaluation paresseuse

Appel par valeur, appel par nom

Appel par valeur : l'argument d'un appel de fonction est entièrement évalué avant d'entrer dans le corps de la fonction.

$$(\lambda x.\, x + x)\ (\texttt{fib}\ 11) \xrightarrow{*} (\lambda x.\, x + x)\ 89 \xrightarrow{*} 89 + 89 \xrightarrow{*} 178$$

Appel par valeur, appel par nom

Appel par valeur : l'argument d'un appel de fonction est entièrement évalué avant d'entrer dans le corps de la fonction.

$$(\lambda x.\, x + x)\ (\texttt{fib}\ 11) \xrightarrow{*} (\lambda x.\, x + x)\ 89 \xrightarrow{*} 89 + 89 \xrightarrow{*} 178$$

Appel par nom : l'argument est passé non évalué à la fonction ; il est évalué à chaque fois qu'on en a besoin.

$$\begin{aligned}(\lambda x.\, x + x)\ (\texttt{fib}\ 11) &\xrightarrow{*} \texttt{fib}\ 11 + \texttt{fib}\ 11 \\ &\xrightarrow{*} 89 + \texttt{fib}\ 11 \xrightarrow{*} 89 + 89 \xrightarrow{*} 178\end{aligned}$$

Appel par valeur, appel par nom

Appel par valeur : l'argument d'un appel de fonction est entièrement évalué avant d'entrer dans le corps de la fonction.

$$(\lambda x.\, x + x)\ (\texttt{fib}\ 11) \xrightarrow{*} (\lambda x.\, x + x)\ 89 \xrightarrow{*} 89 + 89 \xrightarrow{*} 178$$

Appel par nom : l'argument est passé non évalué à la fonction ; il est évalué à chaque fois qu'on en a besoin.

$$\begin{aligned}(\lambda x.\, x + x)\ (\texttt{fib}\ 11) &\xrightarrow{*} \texttt{fib}\ 11 + \texttt{fib}\ 11 \\ &\xrightarrow{*} 89 + \texttt{fib}\ 11 \xrightarrow{*} 89 + 89 \xrightarrow{*} 178\end{aligned}$$

L'appel par nom est normalisant, mais pas l'appel par valeur : $(\lambda xy.\, y)\ \omega\ 0 \to 0$ par nom, diverge par valeur.

L'appel par nom duplique beaucoup de calculs.

Appel par nécessité et évaluation paresseuse

Appel par nécessité : l'argument est passé non évalué à la fonction ; il est évalué la première fois où on en a besoin, et le résultat mémoisé pour les prochaines utilisations.

$$(\lambda xy.\, y + y)\ \omega\ (\texttt{fib}\ 11) \xrightarrow{*} \underset{\texttt{fib}\ 11}{+} \xrightarrow{*} \underset{89}{+} \xrightarrow{*} 178$$

Dans un langage comme Haskell, cette évaluation paresseuse (*lazy evaluation*) s'applique aussi aux constructeurs, d'où des structures de données potentiellement infinies et évaluées à la demande.

```
numbers = 1 : map (+1) numbers
primes  = filter isprime numbers
```

Implémenter l'évaluation paresseuse dans un langage strict

En utilisant des références (trait impératif) pour la mémoisation.

```
type 'a susp = 'a status ref
and 'a status = Todo of unit -> 'a | Done of 'a

let force (s: 'a susp) : 'a =
  match !s with
  | Todo f -> let v = f () in s := Done v; v
  | Done v -> v
```

Pour suspendre le calcul de l'expression *e*, on écrit

`ref (Todo (fun () -> e))`

Notations pour l'évaluation paresseuse

Expressions : $e ::= \ldots \mid$ `lazy` e

Motifs (*patterns*) : $pat ::= \ldots \mid$ `lazy` pat

`lazy` e suspend l'évaluation de e

(comme `ref (Todo (fun () -> e))`)

`lazy` pat force une suspension et filtre sa valeur avec pat

(comme `match force susp with` pat)

Notations similaires dans le livre d'Okasaki et en OCaml :

	Ce cours	OCaml	Okasaki
Type des suspensions	`'a susp`	`'a Lazy.t`	`'a susp`
Suspendre e	`lazy` e	`lazy` e	`$`e
Forcer et filtrer avec p	`lazy` p	`lazy` p	`$`p
Forcer simplement	`force`	`Lazy.force`	`force`

Les listes paresseuses

```
type 'a stream = 'a cell susp
and 'a cell = Nil | Cons of 'a * 'a stream

let head = function lazy (Cons(h, t)) -> h | _ -> assert false
let tail = function lazy (Cons(h, t)) -> t | _ -> assert false

let rec map f l =
  lazy (match l with lazy Nil -> Nil
                   | lazy (Cons(h, t)) -> Cons(f h, map f t))

let rec numbers = lazy (Cons(1, map succ numbers))
```

Remarque : la fonction `map` est «incrémentale», au sens où pour produire les k premiers éléments de `map f l`, il suffit d'évaluer les k premiers éléments de `l`.

Un tri par fusion paresseux

```
let rec merge (s1: 'a stream) (s2: 'a stream) : 'a stream =
  lazy (match force s1, force s2 with
            | Nil, c2 -> c2
            | c1, Nil -> c1
            | Cons(h1, t1), Cons(h2, t2) ->
                if h1 <= h2
                then Cons(h1, merge t1 s2)
                else Cons(h2, merge s1 t2)

let rec mergesort (s: 'a stream) (len: int) : 'a stream =
  if len <= 1 then s else
    let (s1, s2) = split s (len/2) in
    merge (mergesort s1 (len/2)) (mergesort s2 (len - len/2))
```

Quand la paresse améliore l'efficacité algorithmique

Le tri se fait à la demande : dans `mergesort` s n, avec $n = |s|$,

- le premier élément de la liste triée (= le minimum de s) est produit en temps $\mathcal{O}(n)$;
- les éléments suivants en temps $\mathcal{O}(\log n)$ chacun.

On peut donc trouver les k plus petits éléments de la liste s en prenant les k premiers éléments de `mergesort` s n, temps $\mathcal{O}(n + k \log n)$.

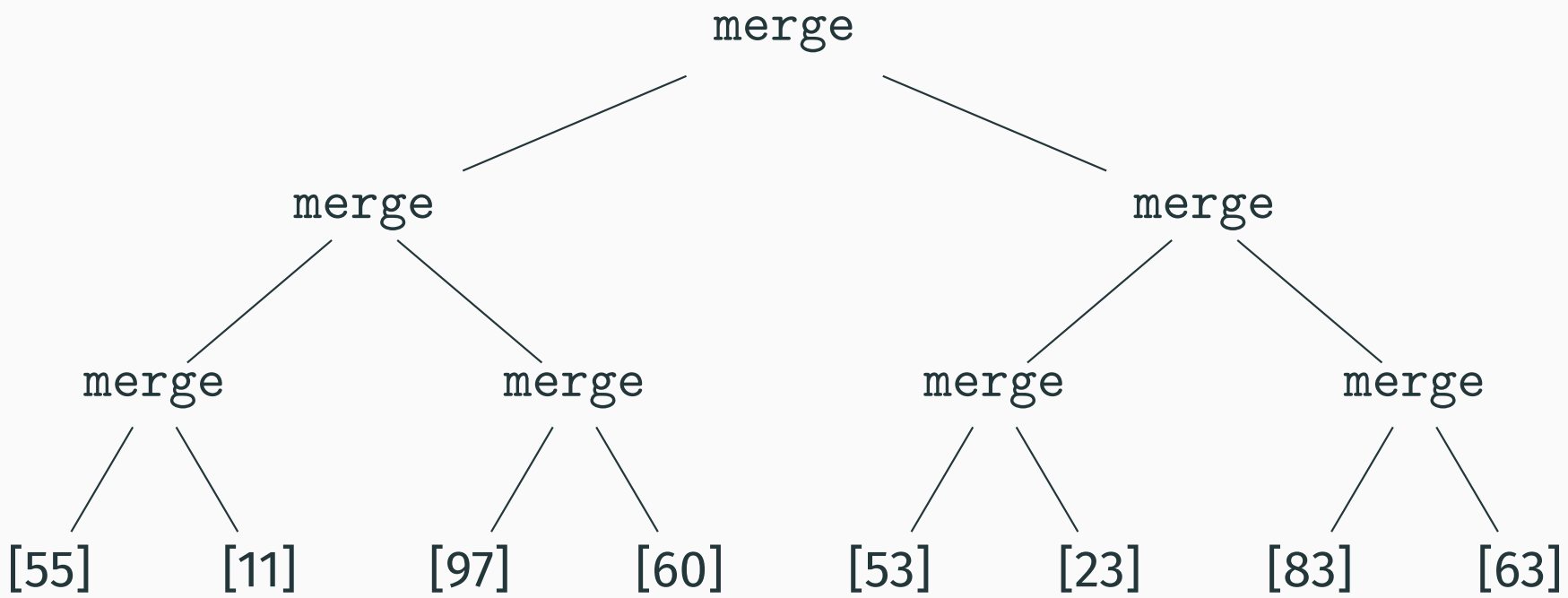
La fusion paresseuse

```
let rec merge (s1: 'a stream) (s2: 'a stream) : 'a stream =
  lazy (match force s1, force s2 with
          | Nil, c2 -> c2
          | c1, Nil -> c1
          | Cons(h1, t1), Cons(h2, t2) ->
              if h1 <= h2
              then Cons(h1, merge t1 s2)
              else Cons(h2, merge s1 t2)
```

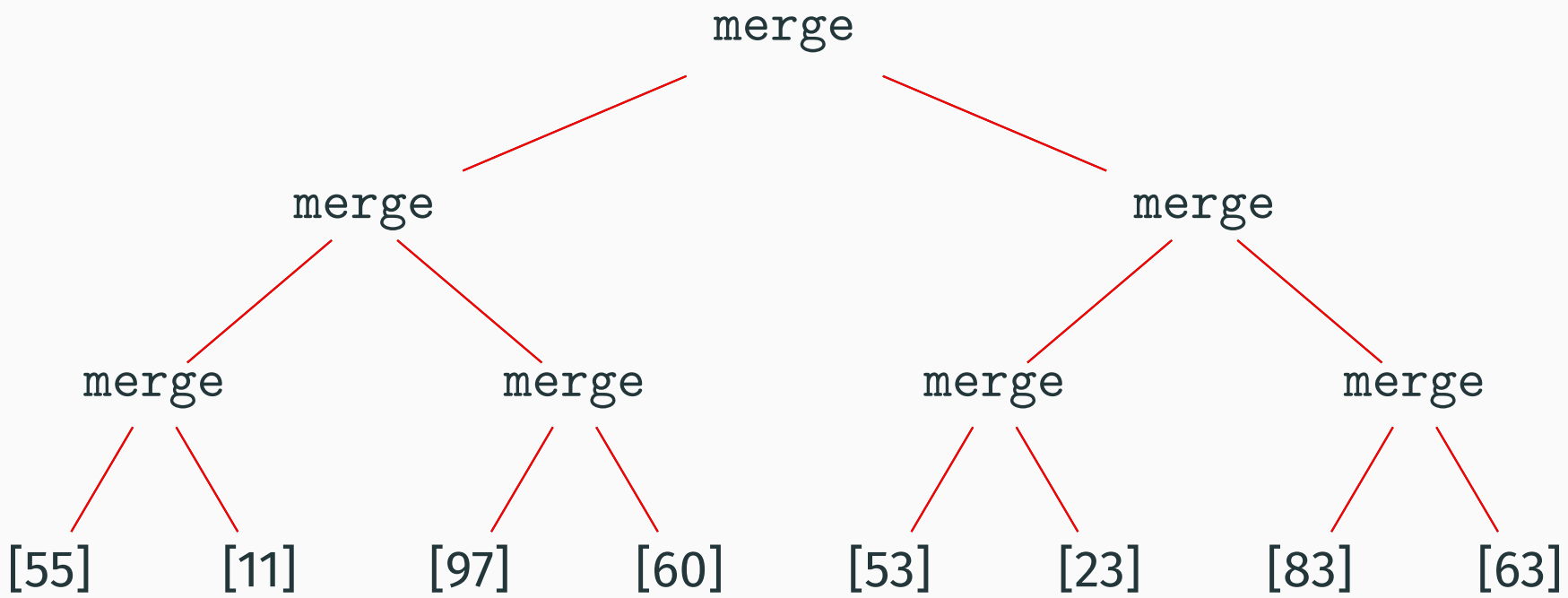
Pour obtenir le premier élément de `merge s1 s2`, il faut évaluer le premier élément de `s1` et celui de `s2`.

Pour obtenir chaque élément suivant de `merge s1 s2`, il faut évaluer un élément suivant d'une des deux entrées `s1` et `s2` (l'autre a déjà été évalué précédemment).

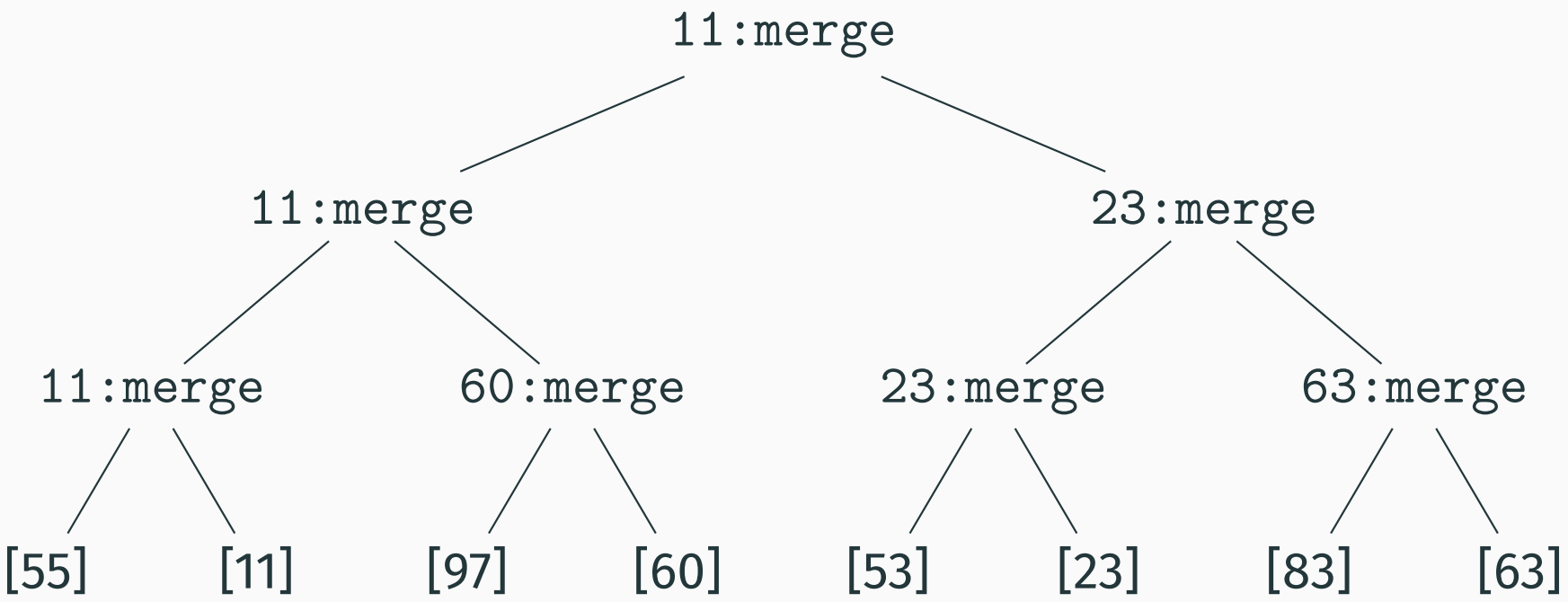
L’arbre des fusions



L’arbre des fusions

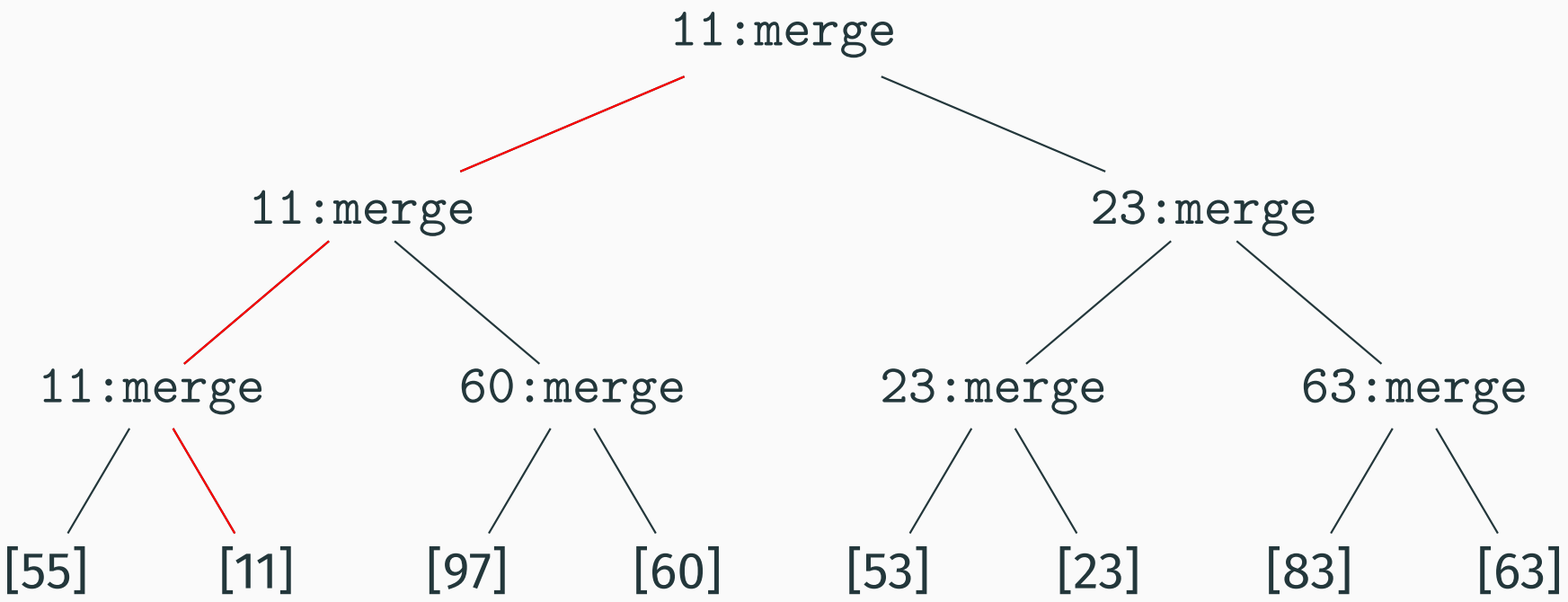


L’arbre des fusions



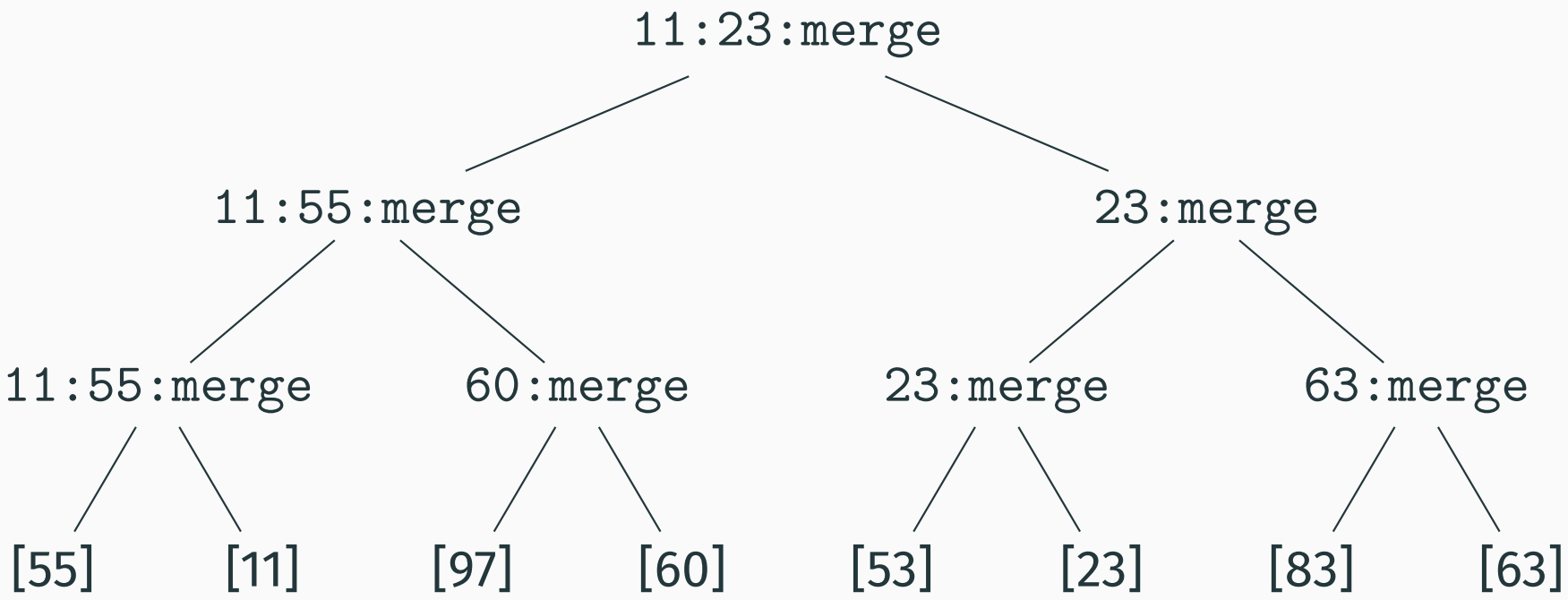
Premier résultat en évaluant les 7 `merge`.

L’arbre des fusions



Premier résultat en évaluant les 7 `merge`.

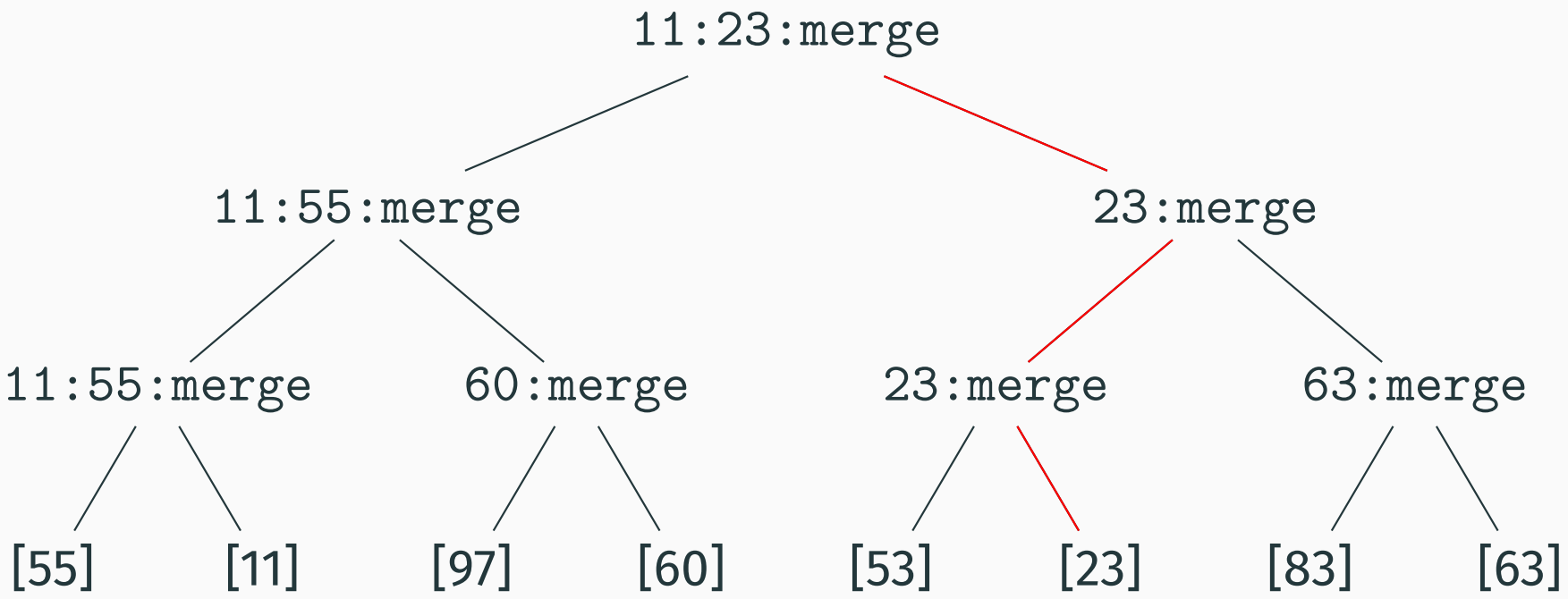
L’arbre des fusions



Premier résultat en évaluant les 7 `merge`.

Deuxième résultat en évaluant 3 `merge`.

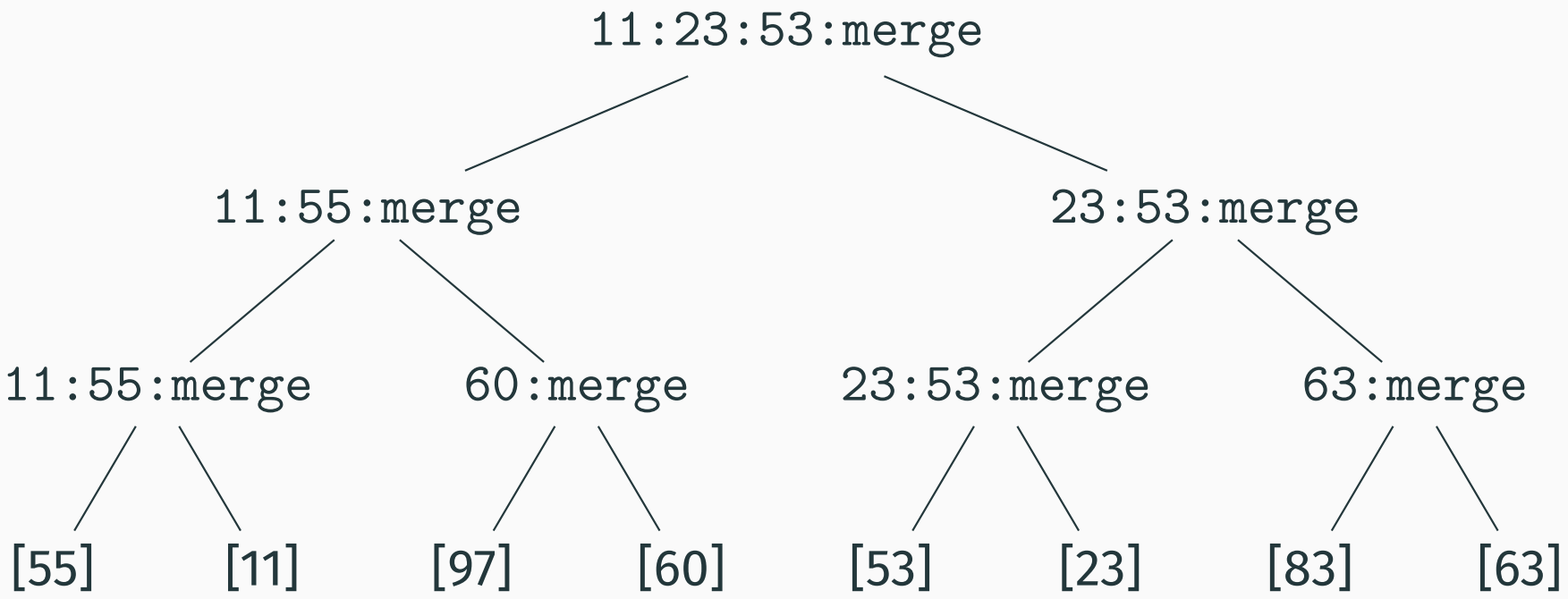
L’arbre des fusions



Premier résultat en évaluant les 7 `merge`.

Deuxième résultat en évaluant 3 `merge`.

L’arbre des fusions



Premier résultat en évaluant les 7 `merge`.

Deuxième résultat en évaluant 3 `merge`.

Troisième résultat en évaluant 3 `merge`.

Etc.

Concaténation ou inversion de listes paresseuses

```
let rec app (s1: 'a stream) (s2: 'a stream) : 'a stream =
  lazy (match s1 with lazy Nil -> force s2
                    | lazy(Cons(h, t)) -> Cons(h, app t s2))

let rev (l: 'a list) : 'a stream =
  lazy (List.fold_left (fun acc elt -> Cons(elt, lazy acc))
                       Nil l)
```

`app` est incrémentale : si chaque élément de `s1` et `s2` s'obtient en temps $\mathcal{O}(1)$, chaque élément de `app s1 s2` aussi.

`rev` n'est pas incrémentale : le premier élément de `rev` ℓ s'obtient en temps $\mathcal{O}(|\ell|)$, les autres en $\mathcal{O}(1)$.

Concilier amortissement et persistance

Inversion puis concaténation de listes paresseuses

Considérons la liste paresseuse $\texttt{app}\ \mathsf{s}\ (\texttt{rev}\ \ell)$, où $|\mathsf{s}| = |\ell| = n$.

L'évaluation complète de $\texttt{rev}\ \ell$ se produit lorsqu'on accède au $n+1^{\text{e}}$ élément de cette liste (de longueur $2n$).

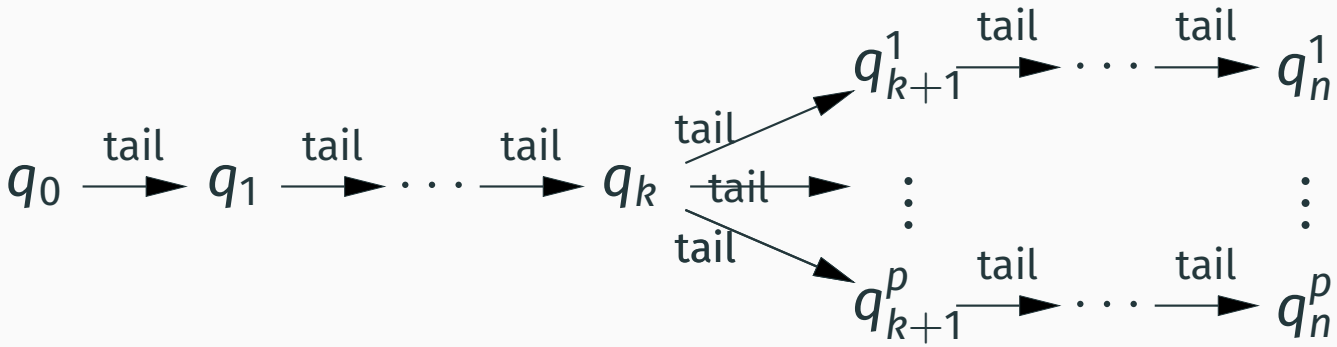
Pour évaluer complètement les i premiers éléments de cette liste, il faut un temps

$$\begin{cases} i & \text{si } i \leq n \\ n+i & \text{si } n < i \leq 2n \end{cases}$$

Dans les deux cas, le temps est $\leq 2i$.

On pourrait donc dire que chaque élément de la liste $\texttt{app}\ \mathsf{s}\ (\texttt{rev}\ \ell)$ s'évalue en temps amorti constant (2 unités).

Inversion puis concaténation persistante

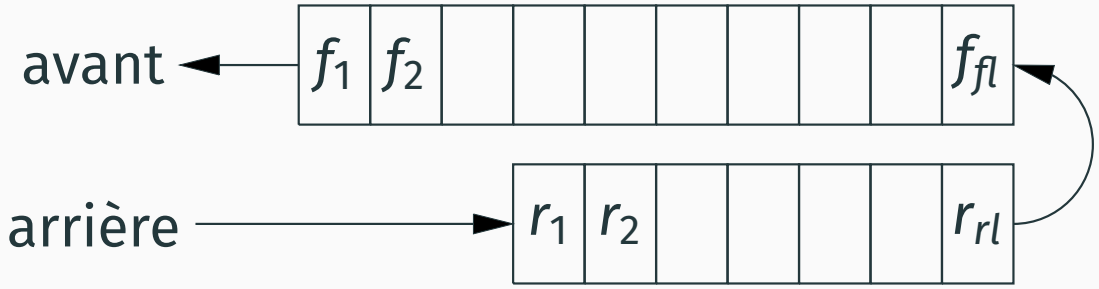


Soit $q_0 = \texttt{app}\ s\ (\texttt{rev}\ \ell)$. Supposons qu'on l'utilise non-linéairement : après k `tail` on la duplique p fois.

Grâce à la paresse, `rev` ℓ est évalué une seule fois, ainsi que les $n - k$ étapes de `app` restantes. Le coût amorti de chaque `tail` reste donc constant.

Ce ne serait pas le cas en appel par nom : `rev` ℓ serait évalué p fois. Si $k = n - 1$ par exemple, le coût total serait $n + pn$ pour $n + p$ opérations `tail`, d'où un coût amorti linéaire en p.

La file d'attente du banquier (C. Okasaki, 1995)




```
type 'a queue = int * 'a stream * int * 'a list
let empty = (0, lazy Nil, 0, [])
let is_empty (fl, f, rl, r) = (fl = 0)
```

Une file = un quadruplet (fl, f, rl, r) avec

- liste paresseuse «avant» f : premier sorti en tête de liste ;
- liste stricte «arrière» r : dernier entré en tête de liste ;
- $fl = |f|$ et $rl = |r|$.

Invariant : $fl \geq rl$, et donc la file est vide ssi $fl = 0$.

Les opérations de la file du banquier

```
let add x (fl, f, rl, r) =
    check (fl, f, rl + 1, x :: r)

let head (fl, f, rl, r) =
    match f with
    | lazy Nil -> raise Empty
    | lazy (Cons(x, _)) -> x

let tail (fl, f, rl, r) =
    match f with
    | lazy Nil -> raise Empty
    | lazy (Cons(_, f')) -> check (fl - 1, f', rl, r)
```

Essentiellement les mêmes opérations que pour la file non paresseuse usuelle, à l'opération `check` près…

Normalisation par rotation

La fonction `check` maintient l'invariant $fl \geq rl$ et prévoit d'inverser la liste arrière «suffisamment longtemps en avance».

```
let check ((fl, f, rl, r) as q) =
    if fl >= rl
    then q
    else (fl + rl, app f (rev r), 0, [])
```

Le calcul `app` f (`rev` r) est déclenché lorsque $|r| = |f| + 1$.

Comme esquissé précédemment, le coût $|r|$ d'évaluation de `rev` r va être amorti sur les opérations nécessaires pour déclencher ce calcul. L'analyse est compliquée car f pourrait encore contenir des `rev` r' non évalués provenant de rotations antérieures.

La méthode du banquier 2.0

Plus de comptes courants contenant des crédits temps, mais uniquement des comptes de dettes contenant des débits temps.

À chaque suspension est associé un compte de dette.

Lorsqu'on évalue `lazy` e, la dette initiale est supérieure ou égale au coût réel d'évaluation de e.

La dette peut être réduite à tout instant par transfert de crédits temps depuis le coût amorti facturé.

$$\text{temps amorti} \geq \text{temps réel} + \text{remboursements}$$

Lorsque la dette est tombée à zéro, on peut forcer la suspension et récupérer sa valeur, sans surcoût.

Les dettes d'une liste paresseuse

Pour une liste de n éléments, on a n dettes $d_1, \ldots, d_n \geq 0$.

La dette cumulée est $D(k) \stackrel{def}{=} \sum_{i=1}^{k} d_i$.

On peut accéder (sans surcoût) aux k premiers éléments de la liste si et seulement si $D(k) = 0$.

On peut diminuer une dette d_i de m, pour un coût amorti m :

$$\begin{array}{ccccccc} d_0 & \ldots & d_{i-1} & d_i & d_{i+1} & \ldots & d_n \\ & & & \downarrow & & & \\ d_0 & \ldots & d_{i-1} & d_i - m & d_{i+1} & \ldots & d_n \end{array}$$

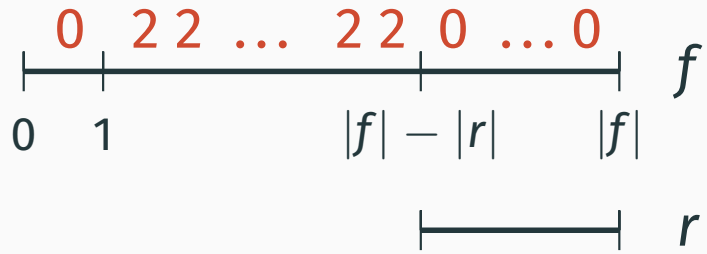
On peut transférer une dette vers un élément plus à gauche, pour un coût amorti nul :

$$\begin{array}{ccccccccccc} d_0 & \ldots & d_{i-1} & d_i & d_{i+1} & \ldots & d_{j-1} & d_j & d_{j+1} & \ldots & d_n \\ & & & \downarrow & & & & \downarrow & & & \\ d_0 & \ldots & d_{i-1} & d_i + m & d_{i+1} & \ldots & d_{j-1} & d_j - m & d_{j+1} & \ldots & d_n \end{array}$$

Les dettes de la file du banquier

Pour une file (f, r), la liste paresseuse f a les dettes

- 0 pour le premier élément ;
- 2 pour les $|f| - |r| - 1$ éléments suivants ;
- 0 pour les $|r|$ derniers éléments.

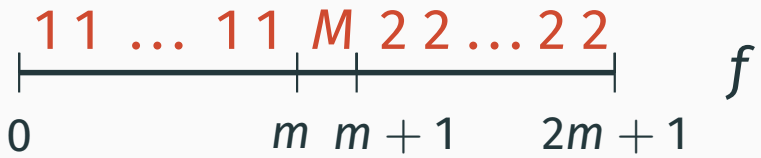


Le premier élément de f a la dette 0, donc est accessible gratuitement à tout instant (notamment par `head`).

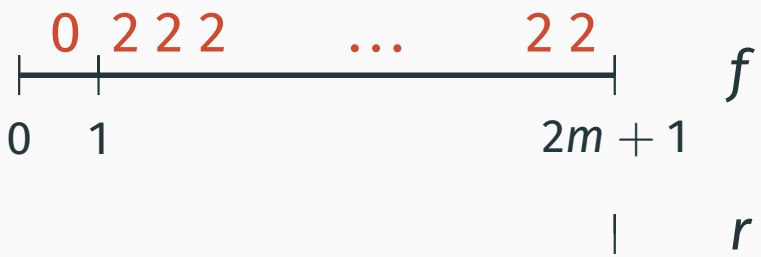
Analyse de l'opération `check`

La rotation se produit lorsque $|f| = m$ et $|r| = m + 1$.
Tous les éléments de f ont la dette 0.

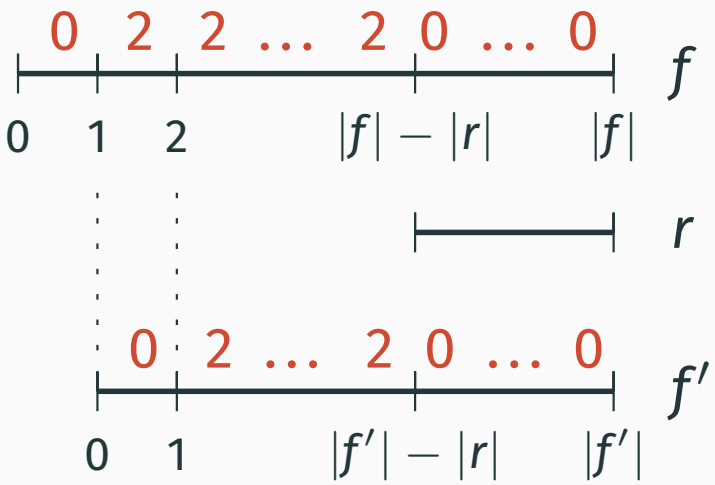
On forme la liste $\texttt{app}\ f\ (\texttt{rev}\ r)$ avec la dette 1 pour chacun des m `Cons` produits par `app`, la dette $M = m + 1$ pour le premier élément de `rev` r, et la dette 2 (surévaluée !) pour les suivants.



On déplace ensuite $m - 1$ unités de la dette M vers le début de la liste, et on paye 1 pour débloquer la tête de la liste.



Analyse de l'opération `tail`

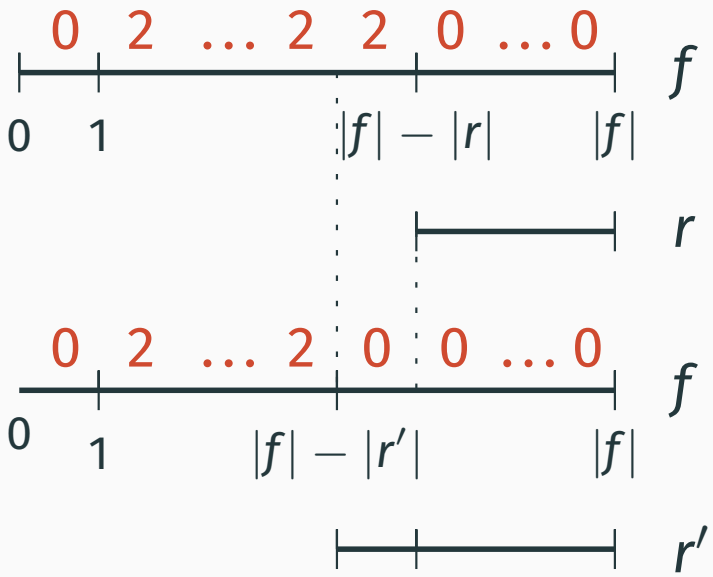


Si `check` ne fait pas de rotation, on passe de f à f' en supprimant le premier élément de f. Il faut débloquer le deuxième élément de f, en payant 2.

Si `check` fait une rotation, on a payé 1.

Le temps réel de `check` et `tail` est constant, donc le temps amorti de `tail` est constant.

Analyse de l'opération `add`



On passe de r à r' en ajoutant un élément à r. On fait passer l'élément en position $|f| - |r| - 1$ dans f de la dette 2 à la dette 0, en payant 2.

S'y ajoute éventuellement le coût de la rotation dans `check`.

Le temps réel de `check` et `add` est constant, donc le temps amorti de `add` est constant.

Tableau comparatif des banquiers

	Banquier classique	Banquier d'amortissement
Tient à jour	une trésorerie	des dettes
Raisonne en	crédits temps	débits temps
Amortit sur	des opérations passées	des opérations futures
Duplication ?	crédits non duplicables	dettes résiduelles duplicables
Utilisations ?	linéaire uniquement	persistante

La méthode du physicien 2.0

(Une simplification de la méthode du banquier 2.0, adaptée au cas où on a une seule suspension dans la structure persistante.)

Une fonction d'«anti-potentiel» Ψ : état de la structure $\to \mathbb{R}_+$

Un majorant de la somme des dettes de toutes les suspensions dans la structure.

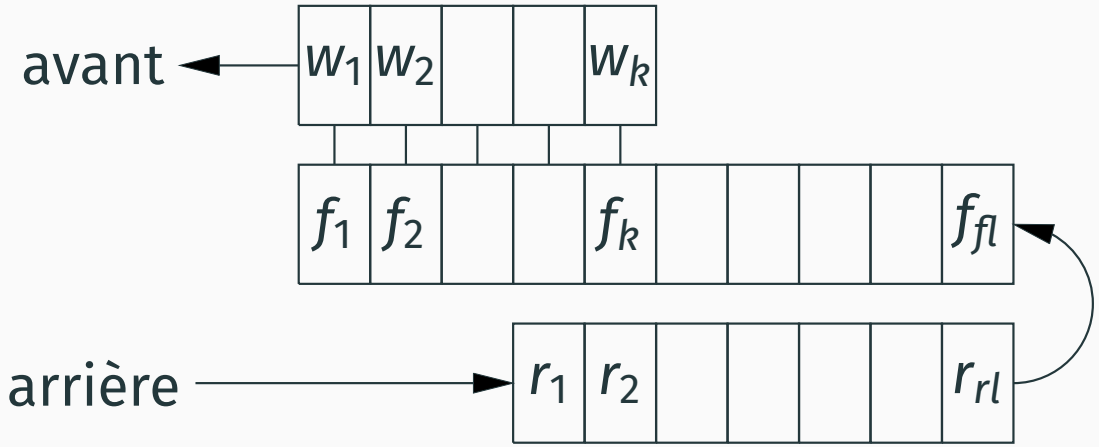
On peut accéder aux valeurs des suspensions ssi $\Psi(s) = 0$.

Chaque opération doit vérifier

$$\text{temps amorti} \geq \text{temps strict} - \Delta\Psi$$

Le temps strict est le temps réel + le temps d'évaluation des suspensions créées.

La file d'attente du physicien (C. Okasaki, 1996)




```
type 'a queue = 'a list * int * 'a list susp * int * 'a list
```

Une file = un quintuplet (w, fl, f, rl, r) avec

- liste de sortie w (un préfixe déjà évalué de f)
- liste stricte suspendue «avant» f
- liste stricte «arrière» r
- $fl = |f|$ et $rl = |r|$.

Invariants : $fl \geq rl$, et $w \neq [\,]$ sauf si la file est vide.

Les opérations de la file du physicien

```
let add x (w, fl, f, rl, r) =
    check (f, fl, rl + 1, x :: r)

let head (w, fl, f, rl, r) =
    match w with
    | [] -> raise Empty
    | x :: _ -> x

let tail (w, fl, f, rl, r) =
    match w with
    | [] -> raise Empty
    | _ :: w' ->
      check (w', fl - 1, lazy(List.tl (force f)), rl, r)
```

Comme la file du banquier, sauf que les extractions `tail` se font en parallèle sur le préfixe `w` et sur la liste avant suspendue `f`.

Normalisations de la file du physicien

Une première normalisation pour garantir que le préfixe `w` n'est pas vide, sauf si la file est vide.

```
let check_w ((w, fl, f, rl, r) as q) =
    if w = [] then (Lazy.force f, fl, f, rl, r) else q
```

Une seconde normalisation pour maintenir l'invariant $fl \geq rl$ et prévoir d'inverser la liste arrière ≪suffisamment longtemps en avance≫, tout comme pour la file du banquier.

```
let check ((w, fl, f, rl, r) as q) =
    if fl >= rl
    then check_w q
    else let f' = Lazy.force f in
         check_w (f', fl + rl, lazy (f' @ List.rev r), 0, [])
```

Analyse amortie

On prend comme anti-potentiel $\Psi(q) \stackrel{def}{=} \min(2|w|, |f| - |r|)$ de sorte que $\Psi = 0$ dans les deux cas de forçage d'une suspension.

```
let check_w ((w, fl, f, rl, r) as q) =
    if w = [] then (Lazy.force f, fl, f, rl, r) else q
```

Si w est vide, $\Psi(q) = 0$, on peut forcer `f`, le coût strict est 1 (`Lazy.force` ne coûte rien), et $\Delta\Psi \geq 0$

$\rightarrow$ coût amorti 1.

Analyse amortie

$$\Psi(q) \stackrel{def}{=} \min(2|w|, |f| - |r|)$$

```
let check ((w, fl, f, rl, r) as q) =
    if fl >= rl
    then check_w q
    else let f' = Lazy.force f in
         check_w (f', fl + rl, lazy (f' @ List.rev r), 0, [])
```

Si $|f| < |r|$, on a en fait $|f| = m$ et $|r| = m + 1$ pour un certain m.

$\Psi(q) = 0$ donc on peut forcer `f`, et le coût strict est 1 plus celui du calcul de `f' @ List.rev r`, à savoir $2m + 1$.

Le potentiel passe de 0 à $\min(2m, 2m + 1) = 2m$.

Le coût amorti est donc $(1 + 2m + 1) - (2m - 0) = 2$.

Analyse amortie

$$\Psi(q) \stackrel{def}{=} \min(2|w|, |f| - |r|)$$

```
let add x (w, fl, f, rl, r) =
    check (f, fl, rl + 1, x :: r)
```

$|r|$ augmente de 1, donc le potentiel décroît de 1 ou ne change pas. Le coût strict est constant $\rightarrow$ coût amorti constant.

```
let tail (w, fl, f, rl, r) =
    match w with
    | [] -> raise Empty
    | _ :: w' ->
      check (w', fl - 1, lazy(List.tl (force f)), rl, r)
```

$|w|$ décroît de 1 et $|f|$ de 1, donc le potentiel décroît de 1 ou 2. Le coût strict est constant $\rightarrow$ coût amorti constant.

Éliminer l’amortissement

Les applications ≪temps réel≫

Pour certaines applications, il faut absolument borner le temps que prend chaque opération :

- Systèmes temps-réel strict :
  contrôle-commande, asservissements, robotique, …
- Systèmes temps-réel souple :
  audio, vidéo, jeux, interfaces utilisateur, …

Exemple : une interface utilisateur où

100 opérations prennent 20ms chacune

est plus agréable à utiliser qu'une autre où

99 opérations prennent 1ms chacune et 1 opération prend 1s

même si la deuxième termine plus vite que la première.

Éliminer l'amortissement par l'ordonnancement

Une technique générale pour transformer une structure de donnée efficace amortie en structure efficace temps réel : l'ordonnancement (*scheduling*).

Idée : au lieu de retarder une opération coûteuse $\mathcal{O}(n)$ après n opérations bon marché $\mathcal{O}(1)$, on va

- incrémentaliser l'opération coûteuse pour qu'elle puisse s'effectuer en n petites étapes en $\mathcal{O}(1)$;
- ordonnancer une petite étape de l'opération coûteuse à chacune des n opérations bon marché.

Dès lors, toutes les opérations sont en $\mathcal{O}(1)$ dans le cas le pire, et non plus seulement $\mathcal{O}(1)$ amorti.

Incrémentaliser la concaténation et l'inversion de listes

Comment calculer $\texttt{app}\ f\ (\texttt{rev}\ r)$ de manière incrémentale ?

Généralisons le problème : calculer incrémentalement

$$\texttt{rotate}\ f\ r\ a \stackrel{def}{=} \texttt{app}\ f\ (\texttt{app}\ (\texttt{rev}\ r)\ a) \ \text{ quand } \ |r| = |f| + 1$$

On a les égalités suivantes :

$$\begin{aligned}
\texttt{rotate}\ [\,]\ [r_1]\ a &= r_1 :: a \\
\texttt{rotate}\ (f_1 :: fs)\ (r_1 :: rs)\ a &= \texttt{app}\ (f_1 :: fs)\ (\texttt{app}\ (\texttt{rev}(r_1 :: rs))\ a) \\
&= f_1 :: \texttt{app}\ fs\ (\texttt{app}(\texttt{rev}\ rs)(r_1 :: a)) \\
&= f_1 :: \texttt{rotate}\ fs\ rs\ (r_1 :: a)
\end{aligned}$$

Elles montrent que le calcul peut être incrémentalisé en utilisant une liste paresseuse pour le résultat : chaque élément est produit en temps $\mathcal{O}(1)$.

Ordonnancer le calcul d'une liste paresseuse

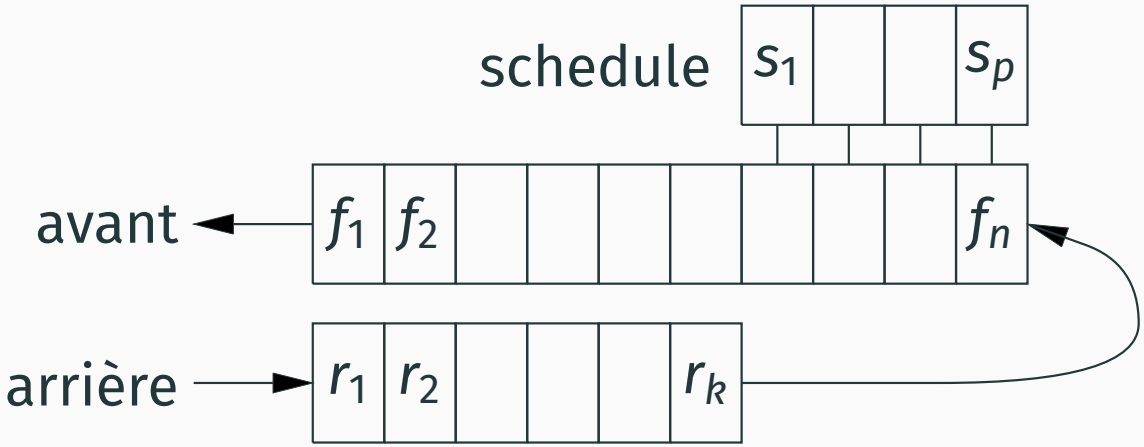
L'analyse amortie de la file du banquier montre qu'il ne suffit pas de laisser `tail` déclencher les calculs des éléments de `app` f (`rev` r). Chaque `add` et chaque `tail` doit calculer un élément de plus.

Il est facile d'ordonnancer les calculs des éléments d'une liste paresseuse f. Il suffit de considérer des paires (f, s) où s (le *schedule*) est initialement f puis plus tard un suffixe de ℓ.

```
let exec = function (f, lazy (Cons(x, s))) -> (f, s)
                  | (f, lazy Nil) -> (f, lazy Nil)
```

Chaque appel à `exec` force une étape de calcul, sans changer la valeur de la liste f.

La file temps-réel d'Okasaki (1995)




```
type 'a queue = 'a stream * 'a list * 'a stream
```

Une file est un triplet (f, r, s) avec

- f liste paresseuse «avant» (premier sorti en tête de liste);
- r liste stricte «arrière» (dernier entré en tête de liste);
- s (le *schedule*) est un suffixe de f ;
- $|f| = |r| + |s|$, ce qui implique $|f| \geq |r|$.

Ordonnancement des rotations

Si le *schedule* est vide, on démarre une nouvelle rotation de f, r.
Sinon, on calcule un élément de plus de la liste f.

```
let rec rotate f r a =  (* f fully evaluated, |r| = |f|+1 *)
  lazy (match f, r with
        | lazy Nil, [r1] -> Cons(r1, a)
        | lazy (Cons(f1, f')), r1 :: r' ->
            Cons(f1, rotate f' r' (lazy (Cons(r1, a))))
        | _ -> assert false)

let exec = function              (* |f|+1 = |r|+|s| on entry *)
  | (f, r, lazy(Cons(_, s))) -> (f, r, s)
  | (f, r, lazy Nil) -> let f' = rotate f r (lazy Nil)
                        in (f', [], f')
```

Toutes les suspensions construites par `rotate` s'évaluent en temps constant. Donc `exec` prend un temps constant.

Les opérations de la file temps-réel

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

let head = function
  | (lazy Nil, _, _) -> raise Empty
  | (lazy (Cons(x, _)), _, _) -> x

let tail = function
  | (lazy Nil, _, _) -> raise Empty
  | (lazy (Cons(_, f)), r, s) -> exec (f, r, s)
```

Les 3 opérations sont en temps constant !

(Il est possible d'implémenter une file d'attente temps-réel purement fonctionnelle, sans utiliser d'évaluation paresseuse : la file de Hood-Melville. Voir le séminaire de T. Nipkow.)

Point d’étape

Sur l'amortissement

Une approche qui mène à des structures de données simples mais étonnament efficaces.

Nous l'avons vu sur l'exemple des files d'attente, mais c'est vrai aussi pour les structures à base d'arbres équilibrés :

	$\mathcal{O}(\log n)$ cas le pire	$\mathcal{O}(\log n)$ amorti
A.B.R.	AVL, rouge-noir	*splay trees*
Tas	*leftist heaps*	*skew heaps*

Pour les structures amorties, les arbres ne portent pas d'information pour le rééquilibrage (hauteurs, couleurs, etc.). Chaque opération tente de réduire localement le déséquilibre.

Sur l'évaluation paresseuse

Un mécanisme très utile pour

- retarder des calculs coûteux jusqu'à ce qu'ils aient été amortis par suffisamment d'opérations ;
- partager le résultat du calcul, évitant ainsi de dupliquer les calculs en cas d'utilisation persistante (non linéaire) ;
- ordonnancer une suite d'étapes de calcul.

Une question conceptuelle : ces implémentations paresseuses sont-elles encore purement fonctionnelles, ou déjà impératives ?

Un constat : les raisonnements (correction fonctionnelle, complexité) sont plus faciles sur les implémentations paresseuses que sur les implémentations impératives en général.

Bibliographie

Bibliographie

Le principal support de ce cours :

- Chris Okasaki, *Purely Functional Data Structures*, chapitres 5, 6, 7.

L'article fondateur :

- Robert E. Tarjan, ≪Amortized Computational Complexity≫, *SIAM Journal on Algebraic and Discrete Methods* 6(2), 1985.

Une vérification de la file du banquier en logique de séparation :

- F. Pottier, A. Guéneau, J.-H. Jourdan, G. Mével, *Thunks and debits in separation logic with time credits*, mars 2023.