Structures de données persistantes, septième et dernier cours

À la recherche du vecteur perdu: limites théoriques et conclusions

Xavier Leroy

2023-04-20

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

Objectifs du cours

En guise de conclusion de l'ensemble du cours, nous allons faire le point sur

- jusqu'où sommes-nous allés ?
  en termes d'efficacité en temps et en espace de structures de données persistantes ;
- jusqu'où pouvons-nous aller ?
  en termes de limitations fondamentales des modèles de calcul sans tableaux ou sans mutation en place.

Le tout sur le cas particulier des tableaux persistants.

Les tableaux

Les tableaux

Une structure de donnée essentielle en calcul numérique : vecteurs, matrices (denses), algèbre linéaire.

Les tableaux

Une structure de donnée essentielle en calcul numérique : vecteurs, matrices (denses), algèbre linéaire.

Un outil pour implémenter des structures éphémères efficaces : tables de hachage, tas (*heap*) implicite, matrices creuses, …

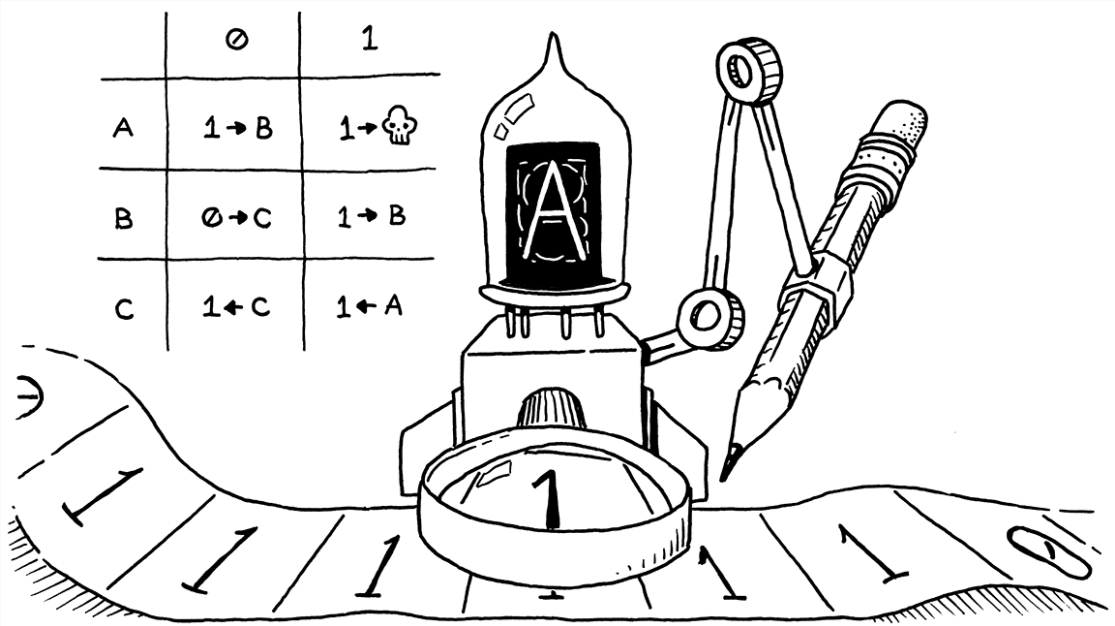
Les tableaux

Une structure de donnée essentielle en calcul numérique :
vecteurs, matrices (denses), algèbre linéaire.

Un outil pour implémenter des structures éphémères efficaces :
tables de hachage, tas (*heap*) implicite, matrices creuses, …

Un moyen pour simuler le modèle RAM (*Random-Access Machine*)
dans le langage (déclaratif) de notre choix,
et y exécuter tous les algorithmes connus dans le modèle RAM.

Un modèle de calcul : la machine de Turing



Une bande infinie portant des symboles + une tête de lecture/écriture.

Opération de base : lire le symbole sous la tête, écrire un symbole, déplacer la tête d'une case à droite ou à gauche.

Simuler la machine de Turing dans un langage fonctionnel pur

La bande = un triplet (r, x, f) où r est la liste «arrière»,
x le symbole sous la tête, f la liste «avant».

Opérations sur la bande :

$$\texttt{read}\ (r, x, f) = x \qquad \texttt{write}\ (r, x, f)\ x' = (r, x', f)$$

$$\texttt{right}\ (r, x, y :: f) = (x :: r, y, f) \qquad \texttt{right}\ (r, x, [\,]) = (x :: r, \mathbf{0}, [\,])$$

$$\texttt{left}\ (y :: r, x, f) = (r, y, x :: f) \qquad \texttt{left}\ ([\,], x, f) = ([\,], \mathbf{0}, x :: f)$$

Chaque transition de la machine est exécutée en temps $\mathcal{O}(1)$
(simulation temps-réel).

La machine à accès direct (RAM, *Random-Access Machine*)

Un nombre fixé de registres de travail $R_1, \ldots, R_k$
plus un nombre arbitraire de cases mémoires $M[1], M[2], \ldots$
qui contiennent des entiers en précision arbitraire.

Opérations typiques :

`load` $R_i, [R_j]$	lecture mémoire à l'adresse R_j
`store` $R_i, [R_j]$	écriture mémoire à l'adresse R_j
`inc` R_i / `dec` R_i	incrément, décrément
`add` R_i, R_j / `sub` R_i, R_j	addition, soustraction
`beqz` R_i, PC	branchement conditionnel (test à zéro)

Modèles de coût pour la RAM

Modèle uniforme : chaque opération prend un temps constant.

(Réaliste si on montre par ailleurs que les nombres manipulés par le programme restent petits.)

Modèle logarithmique : chaque opération prend un temps proportionnel à la taille en bits des données manipulées.

P.ex. `load` $R_i, [R_j]$ prend le temps $\|R_j\| + \|M[R_j]\|$

(taille de l'adresse mémoire + taille de la valeur lue).

Simuler la RAM dans un langage fonctionnel pur

Il suffit d'implémenter la mémoire M par un tableau persistant, avec l'opération `set` qui renvoie un nouveau tableau mis à jour.

Avec le modèle de coût logarithmique :

- Des implémentations avec `get` et `set` en $\mathcal{O}(\log n)$.
- D'où une simulation temps-réel de la RAM.

Avec le modèle de coût uniforme :

- Si implémentés avec des types algébriques : un ralentissement de $\Omega(\log n)$ est inévitable.
- Si implémentés au dessus de tableaux éphémères : $\mathcal{O}(1)$ possible avec les tableaux persistants de Baker.

Tableaux persistants : implémentations fonctionnelles pures

Les principales opérations des tableaux persistants

`get` : `int` $\rightarrow \alpha$ `parray` $\rightarrow \alpha$

`get` *i t* renvoie la valeur de l'indice *i* de *t*

`set` : `int` $\rightarrow \alpha \rightarrow \alpha$ `parray` $\rightarrow \alpha$ `parray`

`set` *i v t* renvoie un tableau identique à *t* sauf que l'indice *i* contient la valeur *v*

`make` : `int` $\rightarrow \alpha \rightarrow \alpha$ `parray`

`make` *n* v_0 initialise un tableau de taille *n* avec la valeur v_0.

`add` : $\alpha \rightarrow \alpha$ `parray` $\rightarrow \alpha$ `parray`

`add` *v t* agrandit le tableau *t* en ajoutant *v* à la fin. (Ou : autre mécanisme de redimensionnement.)

Implémentations purement fonctionnelles

Nous avons vu plusieurs implémentations purement fonctionnelles des tableaux persistants :

- Arbres binaires de recherche équilibrés (AVL, rouge-noir, etc) implémentant un dictionnaire indice $\mapsto$ valeur. (cours 2)
- Listes à accès direct. (cours 5)
- *Finger trees* annotés par des tailles. (cours 5 et 6)

Les plus efficaces sont les arbres préfixe et en particulier les arbres de Braun.

Toutes ces implémentations ont la même complexité $\mathcal{O}(\log n)$ pour `get` et `set`.

Les arbres de Braun

(W. Braun, M. Rem, *A logarithmic implementation of flexible arrays*, 1983.)

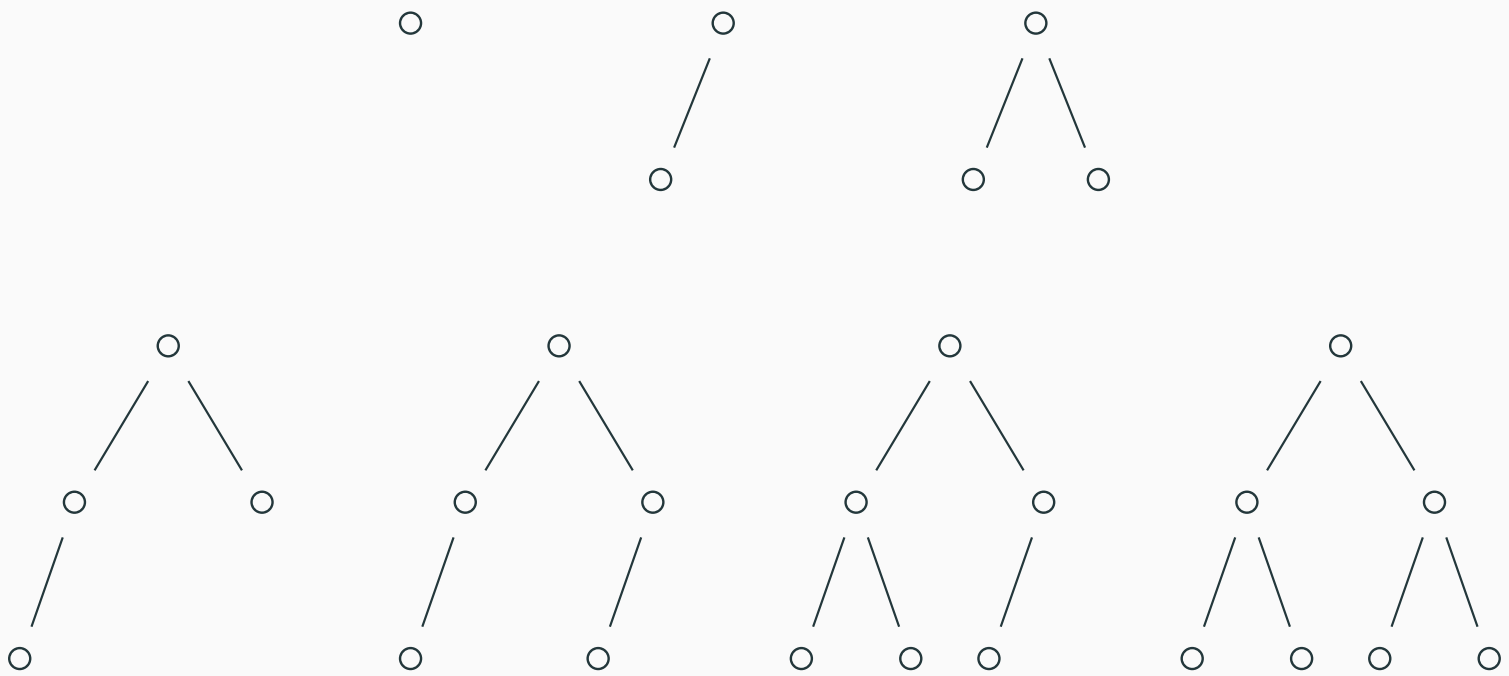
Des arbres binaires équilibrés par le poids selon un critère strict :

$$|G| = |D| \quad \text{ou} \quad |G| = |D| + 1 \quad \text{pour tout nœud } \langle G, x, D \rangle$$

avec $|G|$ la taille (le nombre de nœuds) de G.

Ce critère est tellement strict que la forme d'un arbre de Braun est entièrement déterminée par sa taille !

Les arbres de Braun de tailles 1, 2, …, 7



Les arbres de Braun comme tableaux persistants

Un arbre de Braun de taille n, portant des valeurs de type A aux nœuds, implémente un tableau persistant avec indices $1, \ldots, n$.

Lecture à l'indice i : (temps $\mathcal{O}(\log n)$)

$$\begin{aligned} \texttt{get}(1, \langle G, x, D\rangle) &= x \\ \texttt{get}(2i, \langle G, x, D\rangle) &= \texttt{get}(i, G) \\ \texttt{get}(2i+1, \langle G, x, D\rangle) &= \texttt{get}(i, D) \end{aligned}$$

Écriture de la valeur v à l'indice i : (temps $\mathcal{O}(\log n)$)

$$\begin{aligned} \texttt{set}(1, v, \langle G, x, D\rangle) &= \langle G, v, D\rangle \\ \texttt{set}(2i, v, \langle G, x, D\rangle) &= \langle \texttt{set}(i, v, G), x, D\rangle \\ \texttt{set}(2i+1, v, \langle G, x, D\rangle) &= \langle G, x, \texttt{set}(i, v, D)\rangle \end{aligned}$$

Les arbres de Braun comme tableaux persistants extensibles

Extension à gauche par ajout d'un premier élément v :

$$\begin{aligned} \texttt{addfirst}(v, \bullet) &= \langle \bullet, v, \bullet \rangle \\ \texttt{addfirst}(v, \langle G, x, D \rangle) &= \langle \texttt{addfirst}(x, D), v, G \rangle \end{aligned}$$

Temps $\mathcal{O}(\log n)$.

On vérifie sans peine que

$$\begin{aligned} \texttt{get}(1, \texttt{addfirst}(v, t)) &= v \\ \texttt{get}(i+1, \texttt{addfirst}(v, t)) &= \texttt{get}(i, t) \quad \text{pour tout } 1 \leq i \leq |t| \end{aligned}$$

Les arbres de Braun comme tableaux persistants extensibles

Extension à droite par ajout d'un dernier élément v :

$$\begin{aligned}
\texttt{addlast}(1, v, \bullet) &= \langle \bullet, v, \bullet \rangle \\
\texttt{addlast}(2n, v, \langle G, x, D \rangle) &= \langle \texttt{addlast}(n, v, G), x, D \rangle \\
\texttt{addlast}(2n+1, v, \langle G, x, D \rangle) &= \langle G, x, \texttt{addlast}(n, v, D) \rangle
\end{aligned}$$

L'argument n de $\texttt{addlast}(n, v, t)$ est initialisé à $n = |t| + 1$.

On peut stocker la taille $|t|$ avec l'arbre
(un tableau persistant = une paire (arbre, taille))

$\Rightarrow$ temps $\mathcal{O}(\log n)$.

Initialisation rapide d'un arbre de Braun

(C. Okasaki, *Three algorithms on Braun trees*, J. Func. Prog., 1997)

`make2` n v renvoie deux arbres de Braun de tailles respectives $n+1$ et n, où toutes les valeurs sont v, en temps $\mathcal{O}(\log n)$.

$$\texttt{make2}\ 0\ v = (\langle \bullet, v, \bullet \rangle, \bullet)$$
$$\texttt{make2}\ (2n+2)\ v = (\langle t_1, v, t_1 \rangle, \langle t_1, v, t_0 \rangle) \text{ avec } (t_1, t_0) = \texttt{make2}\ n\ v$$
$$\texttt{make2}\ (2n+1)\ v = (\langle t_1, v, t_0 \rangle, \langle t_0, v, t_0 \rangle) \text{ avec } (t_1, t_0) = \texttt{make2}\ n\ v$$

Note : temps $\mathcal{O}(\log n) \Rightarrow$ espace $\mathcal{O}(\log n) \Rightarrow$ partage.

Arbres préfixes avec nœuds de degré élevé

Deux moyens pour faire mieux que les arbres de Braun :

1. Augmenter le degré des nœuds de l'arbre préfixe :
   de 2 à p.ex. 16, 32 ou 64, ce qui correspond à traiter les indices par paquets de 4, 5 ou 6 bits.

2. Parcourir les indices en partant des bits de poids fort (représentation grand-boutiste), pour une meilleure localité spatiale lors des accès consécutifs i, $i + 1$, $i + 2$, …

   (En contrepartie : on perd l'extensibilité au début du tableau.)

Exemple d'arbres préfixe de degré 32

```
type 'a vect =
  | Leaf of 'a array
  | Node of 'a vect array

type 'a t = { length: int; height: int; vect: 'a vect }

let rec get1 i h v =
  match v with
  | Leaf t -> t.(i land 0x1F)
  | Node t -> get1 i (h - 5) t.((i lsr h) land 0x1F)

let get i m =
  if i >= 0 && i < m.length
  then get1 i m.height m.vect
  else raise Out_of_bounds
```

Tableaux persistants : implémentations impératives

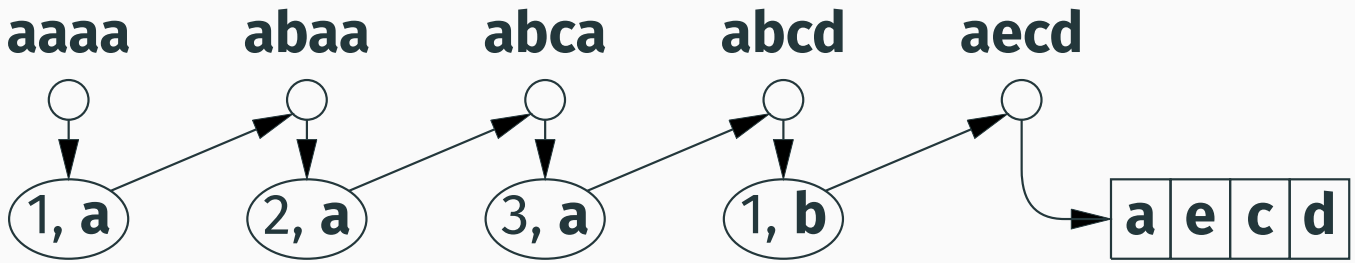
Tableaux persistants implémentés avec des tableaux impératifs

Au lieu de définir une structure de tableaux persistants dans un langage purement fonctionnel, supposons qu'elle est fournie comme un type prédéfini du langage.

On peut alors l'implémenter dans un langage impératif, en utilisant des tableaux éphémères.

Quelles performances peut-on atteindre dans ce cas ?

Les tableaux persistants de Baker (rappel du 4e cours)



Un tableau éphémère qui reflète la version courante du tableau persistant (celle produite par la dernière écriture),
plus des ≪deltas≫ (position, valeur) pour les autres versions.

Analyse de complexité

Espace : $\mathcal{O}(n + w)$ ✔

avec n = taille du tableau et w = nombre d'écritures.

Accès `get`/`set` à une version quelconque :

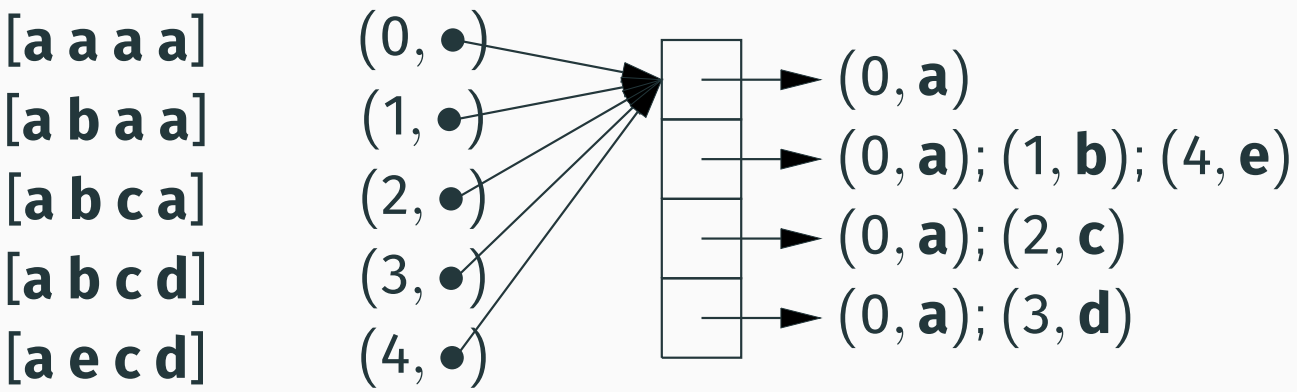
- ✘ temps $\mathcal{O}(w)$ garanti ;
- ✘ temps $\mathcal{O}(n)$ amorti si on fait une reconstruction globale quand l'historique dépasse la longueur n.

Accès à la version produite par le dernier `set` :

- ✔ temps $\mathcal{O}(1)$ garanti.

Parfait pour une utilisation linéaire du tableau,
comme la transcription d'un code impératif, ou …
la simulation d'une machine RAM !

Les tableaux de gros éléments (rappel du 4e cours)



Un tableau mutable avec pour chaque indice son historique : une liste de paires (date de modification, nouvelle valeur).

Affectation des dates : triviale en persistance partielle (0, 1, 2, …); en persistance complète, difficile ≪problème de la liste ordonnée≫, solution en $\mathcal{O}(1)$ amorti avec renumérotations.

Comment représenter les historiques ?

On peut reconstruire globalement le tableau après n écritures (n = taille du tableau). Chaque historique est donc de taille $\leq n$.

Opérations nécessaires :

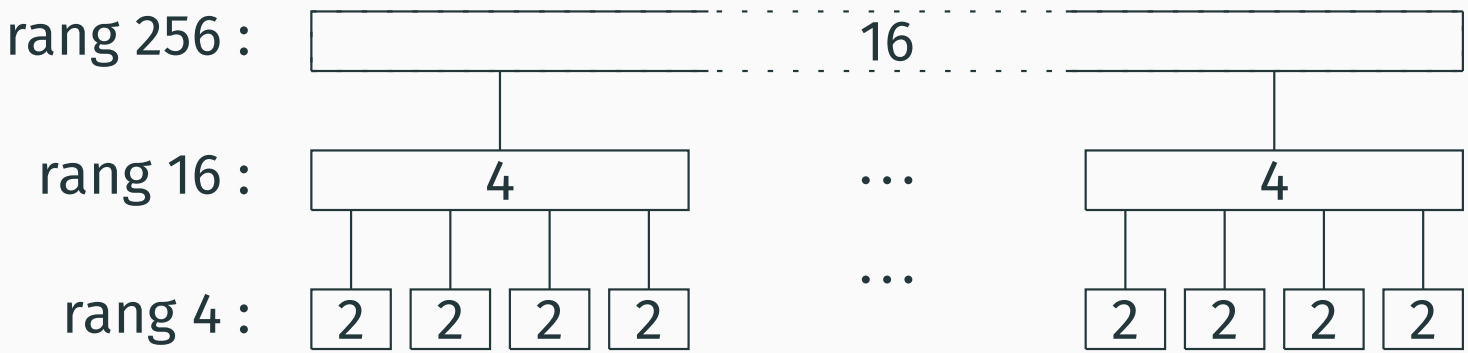
- pour la lecture à une date t : trouver l'élément (t', v) de l'historique avec $t' \leq t$ et t' maximal ;
- pour l'écriture : insérer un élément (t, v) en allouant un espace $\mathcal{O}(1)$.

Une solution simple : un arbre binaire de recherche éphémère, avec modification en place, et un index sur le max.
$\Rightarrow$ lecture et écriture en temps $\mathcal{O}(\log n)$ dans le pire cas,
$\mathcal{O}(1)$ pour la dernière version.

Les arbres de van Emde Boas (1975)

Des ensembles d'entiers entre 0 et M représentés par des arbres préfixes d'arité (très) variable :

un arbre de rang M = un tableau C de $\sqrt{M}$ arbres de rang $\sqrt{M}$
+ valeur min, valeur max
+ un arbre aux de rang $\sqrt{M}$
indiquant les cases non vides de C.



Analyse de complexité

La hauteur d'un arbre VEB de rang M est

$$H(M) = 1 + H(\sqrt{M}) = \mathcal{O}(\log \log M)$$

Opérations en temps $\mathcal{O}(\log \log M)$:

- Opérations ensemblistes : test d'appartenance, insertion, suppression.
- Opérations ordonnées : min, max, prédécesseur (plus grand $j \in S$ tel que $j < i$), successeur.

Espace prohibitif : $\mathcal{O}(M)$. ($M \gg$ nombre d'éléments)

(Aussi gros qu'un tableau de M bits… Celui-ci a les opérations ensemblistes en $\mathcal{O}(1)$ et les opérations ordonnées en $\mathcal{O}(M)$.)

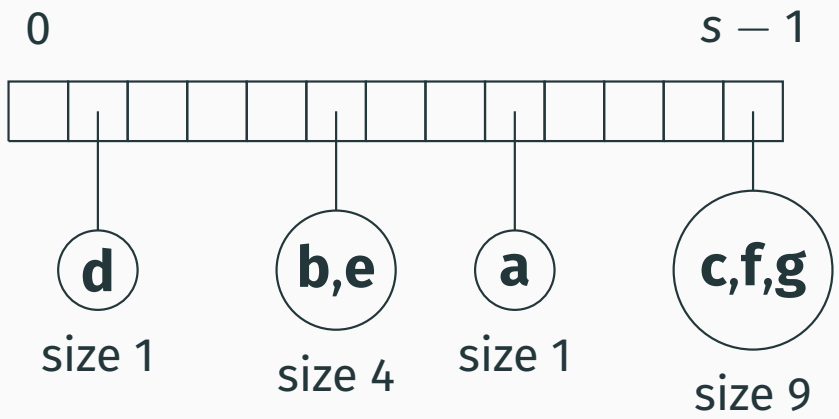
Arbres de van Emde Boas compacts

Idée : dans un arbre de rang M, on remplace le tableau de $\sqrt{M}$ sous-arbres, dont seulement $n \ll \sqrt{M}$ non vides, par une table de hachage (des indices $[0, \ldots, \sqrt{M}]$ dans les sous-arbres) de taille $\mathcal{O}(n)$.

Tables de hachage de base : recherche et insertion en temps $\mathcal{O}(1)$ si pas de collisions, se dégradant en $\mathcal{O}(n)$ si beaucoup de collisions.

Tables de hachage parfait : tirer des fonctions de hachage au hasard pour éviter les collisions ; opérations en temps $\mathcal{O}(1)$ amorti attendu.

Hachage parfait



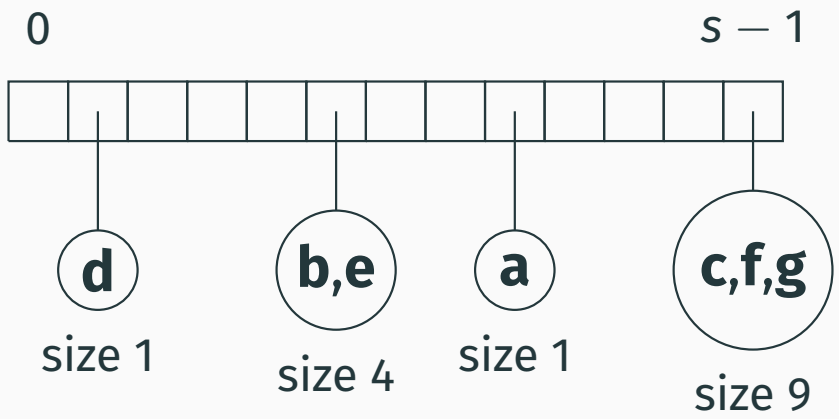
Le cas statique : (Fredman, Komlós, Szemerédi, 1984)

On répartit n éléments (connus à l'avance) en $s = 2(n-1)$ alvéoles (*buckets*) à l'aide d'une fonction de hachage H.

Si le *bucket* i contient $j > 1$ éléments, on en fait une table de taille j^2 avec une fonction de hachage H_i tirée au hasard jusqu'à ce qu'il n'y ait pas de collisions sur les j éléments.

La taille totale reste $\mathcal{O}(n)$ avec probabilité 1.

Hachage parfait



Le cas dynamique : (Dietzfelbinger, Karlin, Mehlhorn, Meyer, Rohnert, Tarjan, 1994)

Même principe, mais adaptations dynamiques lorsque de nouveaux éléments sont insérés :

- Lorsque une collision se produit dans le *bucket* i, tirer une nouvelle fonction H_i et re-hacher le *bucket*.
- Après un nombre suffisant d'insertions, reconstruire toute la table avec un s plus grand.

Les tableaux persistants de Dietz

(Paul F. Dietz, *Fully persistent arrays*, 1989.)

Des tableaux persistants avec accès en temps $\mathcal{O}(\log \log n)$ amorti attendu, et taille $\mathcal{O}(n + w)$ attendue.

Idée de base : un tableau éphémère de gros éléments, chaque gros élément étant un arbre VEB compact utilisant du hachage dynamique parfait.

Donne la complexité attendue pour la persistance partielle (dates = $0, 1, 2, \ldots$) mais pas pour la persistance complète : les renumérotations de dates entraînent des modifications de l'arbre VEB en temps $\mathcal{O}(\log n \cdot \log \log n)$.

Les tableaux persistants de Dietz

Deux raffinements :

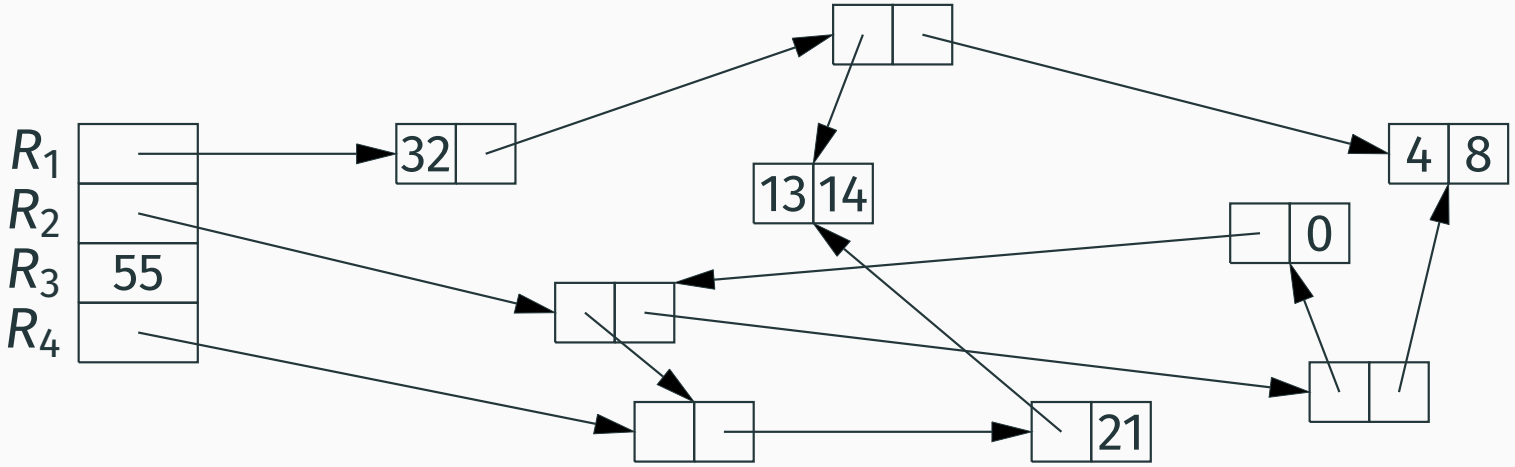
- Mettre aux feuilles de l'arbre VEB des arbres binaires de recherche contenant $\mathcal{O}(\log n)$ entrées de l'historique, avec opérations en temps $\mathcal{O}(\log \log n)$.
  (Ces A.B.R. n'ont pas besoin d'être modifiés lorsqu'on renumérote les dates, puisque l'ordre est préservé.)

- Prendre en compte ces groupes de dates dans l'algorithme de renumérotation, afin qu'il provoque seulement $\mathcal{O}(1)$ amorti opérations dans l'arbre VEB au lieu de $\mathcal{O}(\log n)$.

Cela rétablit la complexité annoncée : accès en temps $\mathcal{O}(\log \log n)$ amorti attendu, et taille $\mathcal{O}(n + w)$ attendue.

Modèle LISP vs. modèle RAM

Modèles de calcul : le modèle LISP / la machine à pointeurs



Un nombre fixé de registres de travail $R_1, \ldots, R_k$
plus un nombre arbitraire de cellules à 2 champs
accessibles via des pointeurs.

Chaque registre, chaque champ de cellule contient soit un pointeur soit une valeur d'un type de base T, p.ex. le type des entiers en précision arbitraire.

Pas de conversion entre valeurs de base et pointeurs : les pointeurs sont opaques.

Les opérations de la machine à pointeurs

Opérations sur les pointeurs :

$R_k := \texttt{cons}(R_i, R_j)$	création et initialisation d'une cellule
$R_j := \texttt{car}(R_i)$	lecture du 1er champ pointé par R_i
$R_j := \texttt{cdr}(R_i)$	lecture du 2e champ pointé par R_i
$\texttt{rplaca}(R_i, R_j)$	écriture de R_j dans le 1er champ pointé par R_i
$\texttt{rplacd}(R_i, R_j)$	écriture de R_j dans le 2e champ pointé par R_i
$R_j := \texttt{consp}(R_i)$	teste si R_i est un pointeur

Plus : les opérations sur le type de base T, p.ex. addition, soustraction, test à zéro.

Modèle de coût : opérations sur les pointeurs en temps constant ; opérations sur le type T généralement en temps constant (modèle uniforme).

Simuler la machine RAM avec la machine à pointeurs

On représente l'état mémoire de la RAM par une structure arborescente implémentant un dictionnaire adresse $\mapsto$ valeur, p.ex. un arbre binaire de recherche équilibré.

Traduction des instructions RAM :

instruction `load`	$\rightarrow$	opération `get` du dictionnaire
instruction `store`	$\rightarrow$	opération `set` du dictionnaire
autres instructions	$\rightarrow$	inchangées

Si le programme s'exécute en temps t et espace s sur la RAM, il s'exécute en temps $\mathcal{O}(t \log s)$ et espace $\mathcal{O}(s)$ sur la machine à pointeurs.

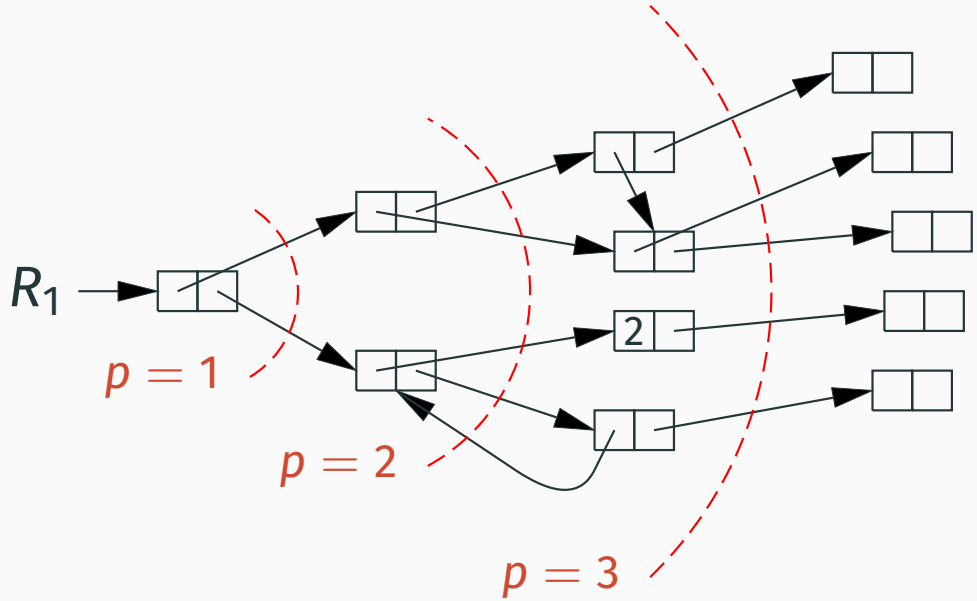
Une borne inférieure

(A. Ben-Amram, Z. Galil, *On pointers versus addresses*, 1992.)

Ben-Amram et Galil montrent que cette simulation est optimale :

La simulation d'un programme RAM de temps t et espace s sur une machine à pointeurs prend un temps $\Omega(t \log s)$ dès lors que le type de base T est incompressible.

Accessibilité des cases mémoire



Dans la machine à pointeurs, on appelle «case mémoire» un champ `car` ou `cdr` d'une cellule `cons`.

À partir des pointeurs contenus dans les registres $R_1, \ldots, R_k$ et en exécutant p instructions `car`, `cdr`, `rplaca`, `rplacd`, on peut lire ou écrire une case mémoire parmi au plus $2k(2^p - 1)$ cases.

Accessibilité des cases mémoire

Corollaire : si on travaille sur $s > 2k(2^p - 1)$ cases mémoire, il y en a au moins une qui nécessite $p + 1$ instructions pour être lue ou écrite, c.à.d. $\Omega(\log s)$ instructions.

On peut donc construire un programme RAM «adversarial», avec $\Omega(t)$ opérations `load` et `store`, qui chacune nécessite $\Omega(\log s)$ opérations de la machine à pointeur pour être simulée.

$\rightarrow$ Borne inférieure $\Omega(t \log s)$.

Mais : ceci suppose que les s cases mémoire de la RAM nécessitent $\Omega(s)$ cases mémoire de la machine à pointeurs.

Le problème de la compressibilité du type de base

On pourrait chercher à encoder l'état mémoire de la RAM $M[1] \ldots, M[s]$ en une (ou un petit nombre de) valeurs du type de base T.

Par exemple, un codage de Gödel :

$$2^{M[1]} \cdot 3^{M[2]} \cdot 5^{M[3]} \ldots p_s^{M[s]}$$

Bien sûr les opérations `get` et `set` sur ce codage sont très coûteuses (il faut factoriser le nombre).

Mais comment exclure la possibilité de compresser efficacement s valeurs du type de base en un petit nombre de valeurs ?

La notion d'incompressibilité

Ben-Amram et Galil disent qu'un type T est incompressible si pour tous $p > q$ il n'existe pas d'injection $T^p \to T^q$ qui soit facilement calculable.

Plus précisément, il n'existe pas de fonctions

$$f : T^p \to T^q \qquad g : T^q \to T^p$$

telles que $g(f(x)) = x$ pour tout x,
et telles que f et g sont calculables par la machine à pointeurs en temps très petit ($o(\|x\|)$ où $\|x\|$ est la taille de x en bits).

Ben-Amram et Galil montrent que s'il existe une simulation de la RAM par la machine à pointeurs en temps $o(t \log s)$, alors T est compressible.

Quelques types incompressibles

Les types finis, p.ex. $T =$ entiers sur k bits.

Par cardinalité : si $p > q$, $\text{card}(T^p) > \text{card}(T^q)$ et donc l'injection f n'existe pas.

Les entiers $\mathbb{N}$ avec addition, soustraction, multiplication

Tous les calculs passent au quotient modulo 2, donc une compression $f : \mathbb{N}^p \to \mathbb{N}^q$ donnerait une compression $\hat{f} : \{0,1\}^p \to \{0,1\}^q$.

Les entiers $\mathbb{N}$ avec $+$, $-$, $\times$, $/2$ et $<$

Voir Ben-Amram et Galil.

Les réels $\mathbb{R}$ avec des opérations continues

Voir Ben-Amram et Galil.

Un type compressible

(A. Ben-Amram, Z. Galil, *On the power of the shift instruction*, 1995.)

On considère les entiers en précision arbitraire avec des décalages (gauche et droite) de n bits en temps constant.

On peut facilement coder un nombre arbitraire d'entiers d bits en un seul entier V, avec accès en temps constant :

$$\texttt{get}(i, V) = V \gg di - (V \gg d(i+1) \ll d)$$
$$\texttt{set}(i, x, V) = V - \texttt{get}(i, V) \ll di + x \ll di$$

On peut aussi augmenter d par recodage en temps $\mathcal{O}(\log n)$, où n est le nombre d'entiers stockés dans V.

Ben-Amram et Galil étendent cela à un codage de n entiers arbitraires en $\alpha(n)$ entiers, avec lecture en temps $\mathcal{O}(\alpha(n))$ et écriture en temps $\mathcal{O}(1)$ amorti.

LISP pur vs. LISP impur

Deux modèles de machines à pointeurs

LISP pur = machine à pointeurs sans `rplaca` et `rplacd`.

Dans ce modèle, une cellule `cons` n'est jamais modifiée après création et initialisation. L'affectation aux registres reste possible.

Équivalent à Core ML sans références, ou Core Haskell strict.

LISP impur = machine à pointeurs avec `rplaca` et `rplacd`.

Les cellules peuvent être modifiées après initialisation.

Équivalent à Core-ML avec références mutables.

LISP pur vs. LISP impur

(N. Pippenger, *Pure versus impure Lisp*, TOPLAS, 1997.)

Pippenger (1997) pose le problème suivant :

> Étant donnée une permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$,
> et un flux infini de n-uplets $(x_1, \ldots, x_n); (x'_1, \ldots, x'_n); \ldots$
> produire ≪en temps réel≫ le flux des n-uplets permutés
> $(x_{\pi(1)}, \ldots, x_{\pi(n)}); (x'_{\pi(1)}, \ldots, x'_{\pi(n)}); \ldots$

LISP pur vs. LISP impur

(N. Pippenger, *Pure versus impure Lisp*, TOPLAS, 1997.)

Pippenger (1997) pose le problème suivant :

> Étant donnée une permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, et un flux infini de n-uplets $(x_1, \ldots, x_n); (x'_1, \ldots, x'_n); \ldots$ produire ≪en temps réel≫ le flux des n-uplets permutés $(x_{\pi(1)}, \ldots, x_{\pi(n)}); (x'_{\pi(1)}, \ldots, x'_{\pi(n)}); \ldots$

Ajoutant quelques contraintes supplémentaires, il montre que

- En LISP impur, on peut faire un prétraitement en temps $\mathcal{O}(n \log n)$, et chaque permutation est en temps $\mathcal{O}(n)$.
- En LISP pur, chaque permutation doit prendre un temps $\Omega(n \log n)$ (même argument que pour le tri), et aucun prétraitement n'est utile.

Le scénario de Pippenger

Un programme de type ≪serveur≫ avec entrées et sorties.
(On ajoute des primitives `read` et `write` aux 2 modèles LISP.)

> Lire un entier n
> Lire n entiers $\pi(1), \ldots, \pi(n)$
> Répéter pour toujours :
>> Lire n symboles $x_1, \ldots, x_n$
>> Calculer $(y_1, \ldots, y_n) = (x_{\pi(1)}, \ldots, x_{\pi(n)})$
>> Afficher $y_1, \ldots, y_n$

Le programme est interactif : il faut émettre la permutation du tuple courant avant de lire le prochain tuple.

Le programme est paramétrique en ses entrées : les entrées x_i ne sont pas des entiers ni des chaînes, mais des symboles abstraits sur lesquels on n'a aucune opération.

Une modélisation en OCaml

On modélise les entrées/sorties par des ≪séquences≫ de la bibliothèque standard (type `'a Seq.t`) : des listes possiblement infinies, évaluées à la demande, mais sans mémoisation.

```
let rec group n (s: 'a Seq.t) : 'a list Seq.t =
  fun () -> Seq.Cons (List.of_seq (Seq.take n s),
                      group n (Seq.drop n s))

let ungroup (s: 'a list Seq.t) : 'a Seq.t =
  Seq.concat (Seq.map List.to_seq s)

let transform n pi (input : 'a Seq.t) : 'a Seq.t =
  group n input |> Seq.map (permut pi) |> ungroup
```

Appliquer une permutation

Une solution naïve avec des listes : $\mathcal{O}(n^2)$

```
let permut pi xs =
  List.map (fun i -> List.nth xs i) pi
```

Une solution avec des tableaux natifs $\mathcal{O}(n)$
(pas disponibles ni en LISP pur ni en LISP impur) :

```
let permut pi xs =
  let a = Array.of_list xs in
  List.map (fun i -> a.(i)) pi
```

Appliquer une permutation

Une solution avec des tableaux fonctionnels (arbres de Braun) : $\mathcal{O}(n \log n)$

```
let permut pi xs =
  let a = List.fold_left Braun.addlast Braun.empty xs in
  List.map (fun i -> Braun.get i a) pi
```

Une solution avec prétraitement puis tri : $\mathcal{O}(n \log n)$

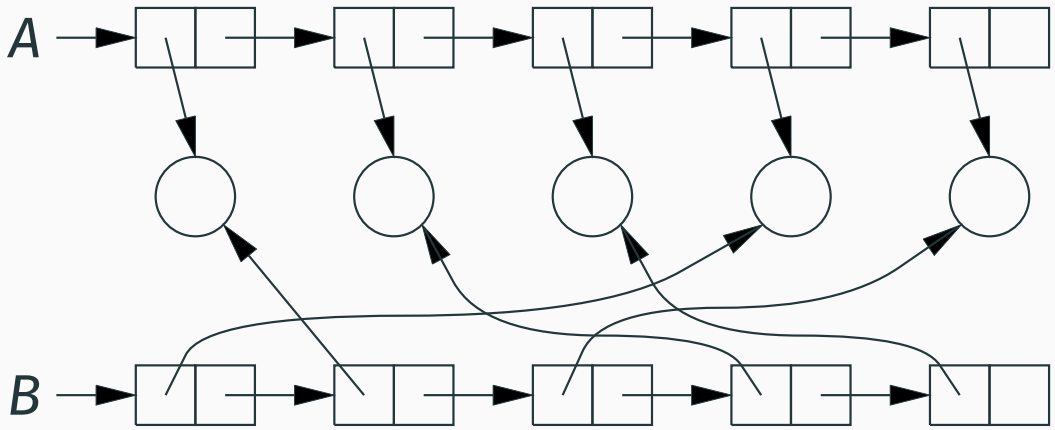
```
let permut invpi xs =
  List.combine invpi xs
  |> List.sort (fun (j1,x1) (j2,x2) -> Int.compare j1 j2)
  |> List.map snd
```

Il faut avoir précalculé l'inverse `invpi` de la permutation `pi`, mais le temps nécessaire est amorti sur tous les appels à `permut`.

La solution de Pippenger

Utilise des listes de références mutables.

Exprimable en LISP impur mais pas en LISP pur.



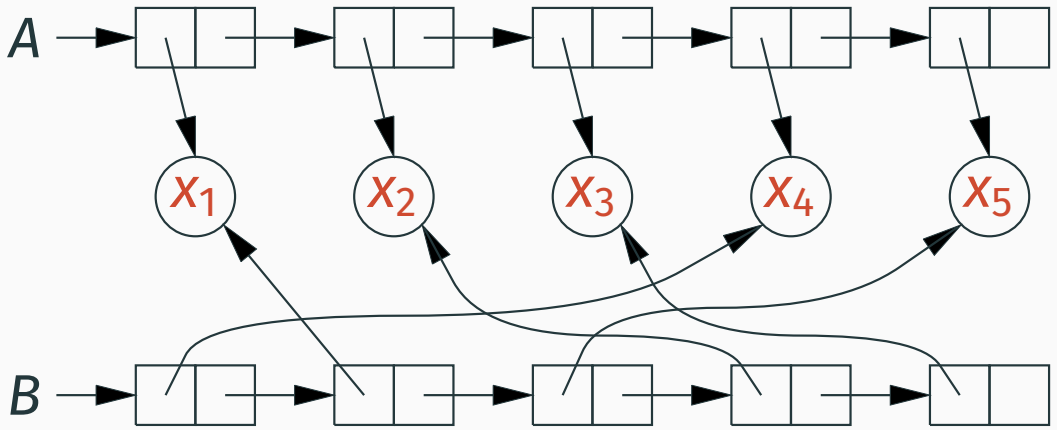
Prétraitement : on construit une liste A de n références distinctes, et sa permutation $B = \texttt{permut}\ \pi\ A$.

(Avec une fonction `permut` inefficace quelconque.)

La solution de Pippenger

Utilise des listes de références mutables.

Exprimable en LISP impur mais pas en LISP pur.



Permutation : on écrit les entrées $x_1, \ldots, x_n$ dans les références dans l'ordre de la liste A, puis on relit les références dans l'ordre de la liste B, obtenant les sorties $x_{\pi(1)}, \ldots, x_{\pi(n)}$.

Appliquer une permutation en LISP impur

```
let preprocess pi =
  let a = List.map (fun _ -> ref None) pi in
  let b = slow_permut pi a in
  (a, b)

let permut (a, b) xs =
  List.iter2 (fun r x -> r := Some x) a xs;
  List.map (fun r -> Option.get !r) b

let transform n pi (input : 'a Seq.t) : 'a Seq.t =
  let ab = preprocess pi in
  ungroup (Seq.map (permut ab) (group n input))
```

Temps : $\mathcal{O}(n)$ pour chaque appel à `permut`.

Analyse de la permutation en LISP pur

Par hypothèse de paramétricité, les symboles $x_1, \ldots, x_n$ lus peuvent être supposés nouveaux, distincts de tous les symboles vus précédemment.

Par hypothèse de pureté, les x_i ne peuvent pas être stockés dans des cellules allouées avant leur lecture.

Le calcul de permutation qui mène des x_i aux $y_i = x_{\pi(i)}$ a donc lieu entièrement entre la lecture des x_i et l'écriture des y_i.
(Pas de résultats précalculés utilisables.)

Analyse de la permutation en LISP pur

Le code qui permute un n-uplet doit donner le bon résultat pour n'importe laquelle des $n!$ permutations possibles.

Donc, il doit prendre au moins $\log_2 n! \sim n \log_2 n$ décisions binaires. (Même argument que pour le tri.)

Donc, le coût total de la permutation est $\Omega(n \log n)$, et il doit être payé à chaque cycle de lecture / permutation / écriture.

La revanche de l'évaluation paresseuse

(R. Bird, G. Jones, O. de Moor, *More haste, less speed : lazy versus eager evaluation*, JFP, 1997.)

En 1997, Bird et coauteurs publient une implémentation en Haskell du problème de Pippenger, en temps $\mathcal{O}(n)$ par cycle.

Cela ne contredit pas la borne inférieure de Pippenger, car la solution utilise l'évaluation paresseuse des listes.
Or, évaluation paresseuse $\Rightarrow$ mémoisation $\Rightarrow$ mutation.

Vers une solution paresseuse : une solution stricte non interactive

Supposons qu'on puisse traiter l'entrée ≪par lot≫ et non pas interactivement : on lit N tuples ; on les permute ; on les écrit.

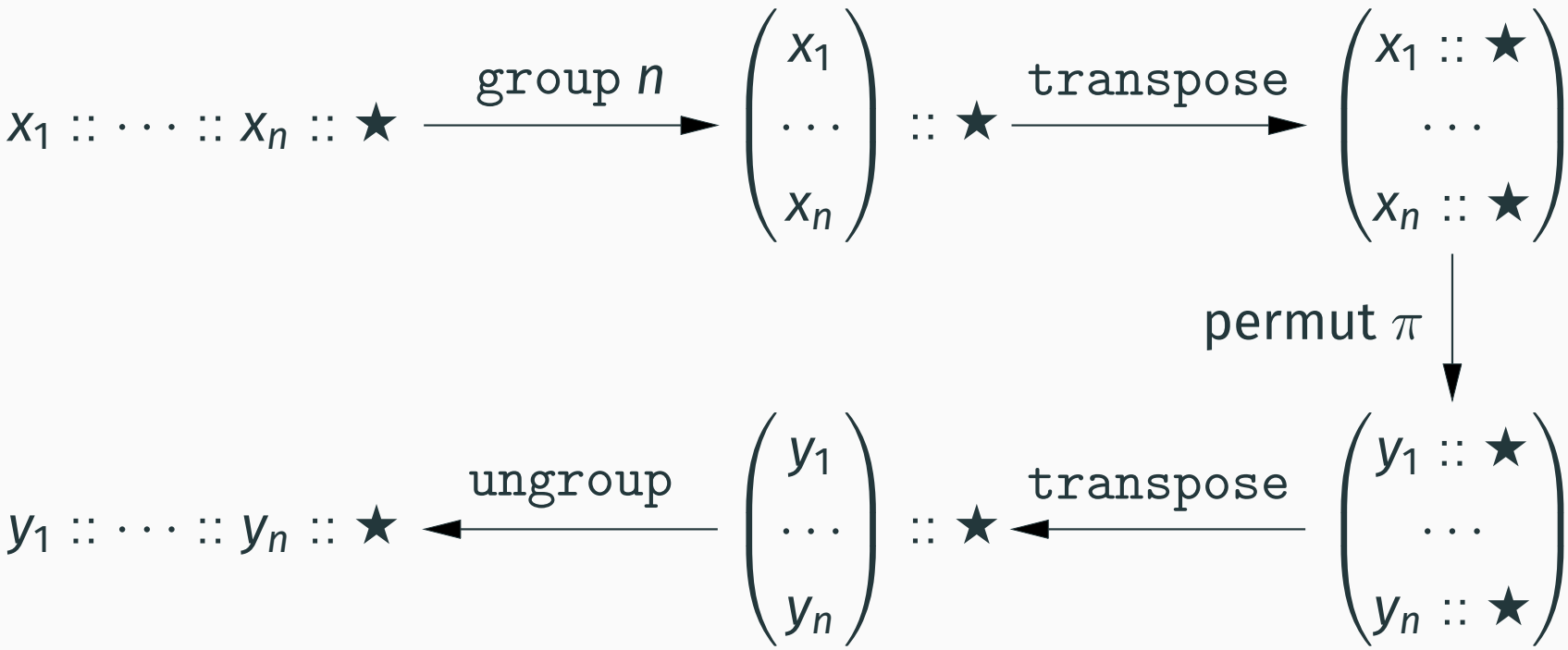
On peut alors faire comme suit :

- On transpose l'entrée (une liste de n-uplets) en un n-uplet de listes.
- On permute (une seule fois) ce n-uplet de listes.
- On transpose le n-uplet de listes résultant en une liste de n-uplets.

Le temps de calcul est dominé par la transposition, en temps $\mathcal{O}(n \cdot N)$, d'où un temps moyen de $\mathcal{O}(n)$ par tuple.

Retrouver l'interactivité avec des listes paresseuses

En utilisant des listes paresseuses, cette solution devient interactive : pour obtenir les n premiers éléments de la sortie, il suffit de lire les n premiers éléments de l'entrée.



(On a noté ★ les suspensions non évaluées).

Le modèle LISP paresseux

On voit apparaître un 3e modèle de calcul, en plus de Lisp pur et Lisp impur : Lisp paresseux, avec des cellules `cons` évaluées à la demande et mémoisées.

L'observation de Bird *et al* + le résultat de Pippenger
$\Rightarrow$ un *gap* de complexité entre Lisp pur et Lisp paresseux.

Y-a-t'il un *gap* entre Lisp paresseux et Lisp impur ?

Bibliographie

Bibliographie

Les tableaux persistants de Dietz :

- Paul F. Dietz. *Fully Persistent Arrays (extended abstract)*. Workshop WADS 1989, LNCS 382.
- Haim Kaplan, *Persistent Data Structures*, section 31.3.3 du *Handbook of data structures and applications*, Chapman&Hall / CRC Press, 2005.

Le *gap* entre machine RAM et machine à pointeurs :

- Amir M. Ben-Amram, Zvi Galil. *On Pointers versus Addresses*. J. ACM 39(3), 1992.
- Amir M. Ben-Amram, Zvi Galil. *On the Power of the Shift Instruction*. Inf. Comput. 117(1), 1995.

Le *gap* entre LISP pur et LISP impur :

- Nicholas Pippenger. *Pure Versus Impure Lisp*. ACM Trans. Program. Lang. Syst. 19(2), 1997.

Conclusions

Quelle algorithmique pour la programmation déclarative ?

Réponse 1 : les algorithmes abstraits classiques, en utilisant

- des structures de données persistantes au lieu des structures éphémères usuelles ;
- un style à passage d'état (ou une monade d'état) si nécessaire.

Des versions persistantes existent pour toutes les structures de données classiques (dictionnaire, ensemble, tas, pile, file, …)

- Souvent même complexité $\mathcal{O}$ que leurs versions éphémères. (Exceptions : tableaux, tables de hachage, union-find.)
- Facteurs constants raisonnables (entre 1 et 10).

Bonus : ces structures sont hautement réutilisables (encapsulation modulaire, bibliothèques ≪sur étagère≫).

Quelle algorithmique pour la programmation déclarative ?

Réponse 2 : une autre algorithmique est possible !

- Plus proche des définitions mathématiques ;
  moins obsédée par l'enchaînement des calculs et le stockage des états intermédiaires.
  (Ex : démonstration automatique, compilation, analyse statique.)
- Tirant meilleur parti du partage en mémoire entre résultats intermédiaires.
- Plus proche de la vérification formelle de l'algorithme.
  (Ex : utilisation de types dépendants.)

Là aussi, les structures persistantes nous aident :
partage élevé ; opérations de haut niveau (jointures, etc).

Bénéfices pour la programmation impérative

1. Le partage en mémoire !
2. La résistance aux ≪accidents≫ dans le flux de contrôle : retour en arrière, rattrapage d'erreurs, transactions.

Exemple : en OCaml on remplace souvent une table de hachage par une référence (mutable) sur une *map* (dictionnaire persistant).

```
Hashtbl.replace tbl k v   →   tbl := Map.add k v !tbl
```

Cela trivialise l'annulation des modifications sur une erreur :

```
let protect fn arg =
  let old = !r in
  try fn arg with exn -> r := old; raise exn
```

Bénéfices pour la programmation impérative

1. Le partage en mémoire !
2. La résistance aux ≪accidents≫ dans le flux de contrôle : retour en arrière, rattrapage d'erreurs, transactions.

Avec une référence atomique on peut aussi faire des transactions (mises à jour atomiques même en présence de parallélisme).

```
let rec transaction fn =
  let old = Atomic.get r in
  if Atomic.compare_and_set r old (fn old)
  then ()
  else transaction fn
```

(D'autres approches du parallélisme sont souvent préférables : verrouillage, structures de données *lock free*.)

Les implémentations purement fonctionnelles

Un style très algébrique, qui a de nombreux avantages :

- Implémentation aisée dans beaucoup de langages, y compris fonctionnel pur, y compris Agda / Coq / Lean.
- Présentation élégante et concise.
- Incite à fournir de nombreuses opérations : pas juste `mem`/`add`/`remove` mais aussi union, intersection, jointure.
- Itérateurs fiables et faciles à spécifier (pas d'invalidation).
- Vérification facile par raisonnement équationnel (pas besoin d'une logique de programmes).

L'utilisation de l'évaluation paresseuse préserve ces avantages tout en ajoutant des possibilités d'amortissement, d'ordonnancement explicite et de mémoïsation des calculs.

Les implémentations impératives de structures persistantes

Des algorithmes souvent très ingénieux, notamment

- les tableaux persistants de Baker, pour leur simplicité ;
- l'approche *fat nodes* de Driscoll et al, pour sa capacité à partager la mémoire encore mieux que l'approche fonctionnelle pure.

Difficilement utilisables en persistance complète (performances insuffisantes ou implémentations trop complexes).

Très intéressants dans des scénarios d'utilisation particuliers : utilisation linéaire (*single-threaded*), persistance partielle, semi-persistance.

Les tableaux persistants de Dietz nous interrogent sur les limites théoriques de l'approche.

FIN