Persistent data structures, seventh and last lecture

In search of the lost vector: lower bounds and conclusions

Xavier Leroy

2023-04-20

Collège de France, chair of Software sciences
`xavier.leroy@college-de-france.fr`

Course objectives

To conclude the whole course, let's see

- how far did we go?
  in terms of time and space efficiency
  of persistent data structures;
- how far can we go?
  in terms of fundamental limitations of computation models
  that have no arrays or no in-place update.

The running example will be persistent arrays.

Arrays

Arrays

A data structure that is crucial for numerical computing: vectors, (dense) matrices, linear algebra.

Arrays

A data structure that is crucial for numerical computing: vectors, (dense) matrices, linear algebra.

A tool for implementing efficient ephemeral structures: hash tables, implicit heaps, sparse matrices, …

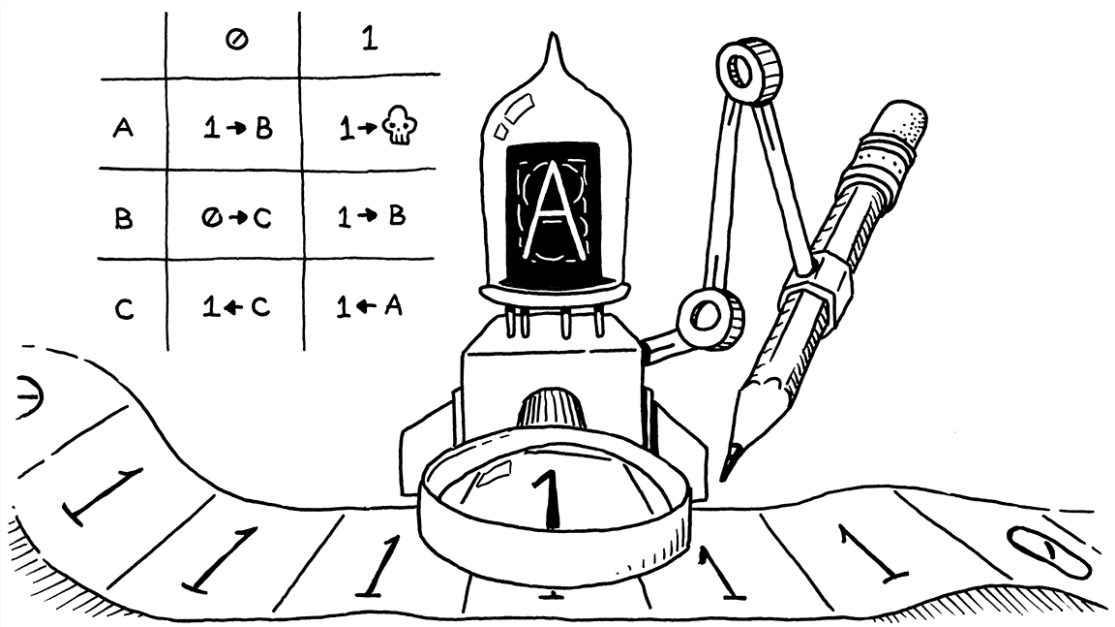
Arrays

A data structure that is crucial for numerical computing: vectors, (dense) matrices, linear algebra.

A tool for implementing efficient ephemeral structures: hash tables, implicit heaps, sparse matrices, …

A way to simulate the RAM model (*Random-Access Machine*) in the (declarative) language of our choice, enabling this language to execute all algorithms known in the RAM model.

A model of computation: Turing machines



An infinite tape carrying symbols + a read/write head.

Basic operation: read the symbol under the head; write a symbol; move the head one step left or right.

Simulating a Turing machine in a pure functional language

The tape = a triple (r, x, f) with x the symbol under the head, r the rear list (to the left of the head), f the front list (to the right).

Operations on the tape:

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

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

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

This is a real-time simulation: each transition of the machine is executed in $\mathcal{O}(1)$ time.

Another model of computation: the Random-Access Machine

A fixed number of registers $R_1, \ldots, R_k$
plus an arbitrary number of memory locations $M[1], M[2], \ldots$
Both contain arbitrary-precision integers.

Typical operations:

`load` $R_i, [R_j]$	memory read at address R_j
`store` $R_i, [R_j]$	memory write at address R_j
`inc` R_i / `dec` R_i	increment, decrement
`add` R_i, R_j / `sub` R_i, R_j	addition, subtraction
`beqz` R_i, PC	conditional branch (if equal to zero)

Cost models for the RAM

Uniform model: each operation takes constant time.

(This is realistic only if we can put a fixed upper bound on the numbers manipulated by the program.)

Logarithmic model: each operation takes time proportional to the bit size of the data it manipulates.

E.g. `load` $R_i, [R_j]$ takes time $\|R_j\| + \|M[R_j]\|$

(bit size of the memory address $+$ bit size of the value read).

Simulating the RAM in a purely functional language

It suffices to implement the memory M by a persistent array, with a `set` operation that returns a new, updated array.

In the logarithmic cost model:

- We have persistent arrays with `get` and `set` in $\mathcal{O}(\log n)$ time.
- This provides a real-time simulation of the RAM.

In the uniform cost model:

- If the memory is implemented with algebraic data types only: we'll see that a slowdown by a factor $\Omega(\log n)$ is unavoidable.
- If the memory is implemented on top of ephemeral arrays: real-time simulation is possible using e.g. Baker's arrays.

Persistent arrays: purely functional implementations

The main operations on persistent arrays

$\texttt{get} : \texttt{int} \to \alpha\ \texttt{parray} \to \alpha$

`get` *i t* returns the value at index *i* in *t*

$\texttt{set} : \texttt{int} \to \alpha \to \alpha\ \texttt{parray} \to \alpha\ \texttt{parray}$

`set` *i v t* returns an array identical to *t* except that index *i* contains value *v*

$\texttt{make} : \texttt{int} \to \alpha \to \alpha\ \texttt{parray}$

`make` *n* v_0 returns an array of size *n* initialized with value v_0.

$\texttt{add} : \alpha \to \alpha\ \texttt{parray} \to \alpha\ \texttt{parray}$

`add` *v t* grows the array *t*, adding *v* as last element. (Or: any other resizing mechanism.)

Purely functional implementations

We saw previously several purely functional implementations of persistent arrays:

- Balanced binary search trees (AVL, red-black, etc) implementing finite maps index $\mapsto$ value. (lecture 2)
- Random-access lists. (lecture 5)
- Finger trees annotated with sizes. (lectures 5 and 6)

The most efficient are prefix trees, in particular Braun trees.

All these implementations have time complexity $\mathcal{O}(\log n)$ for `get` and `set` operations.

Braun trees

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

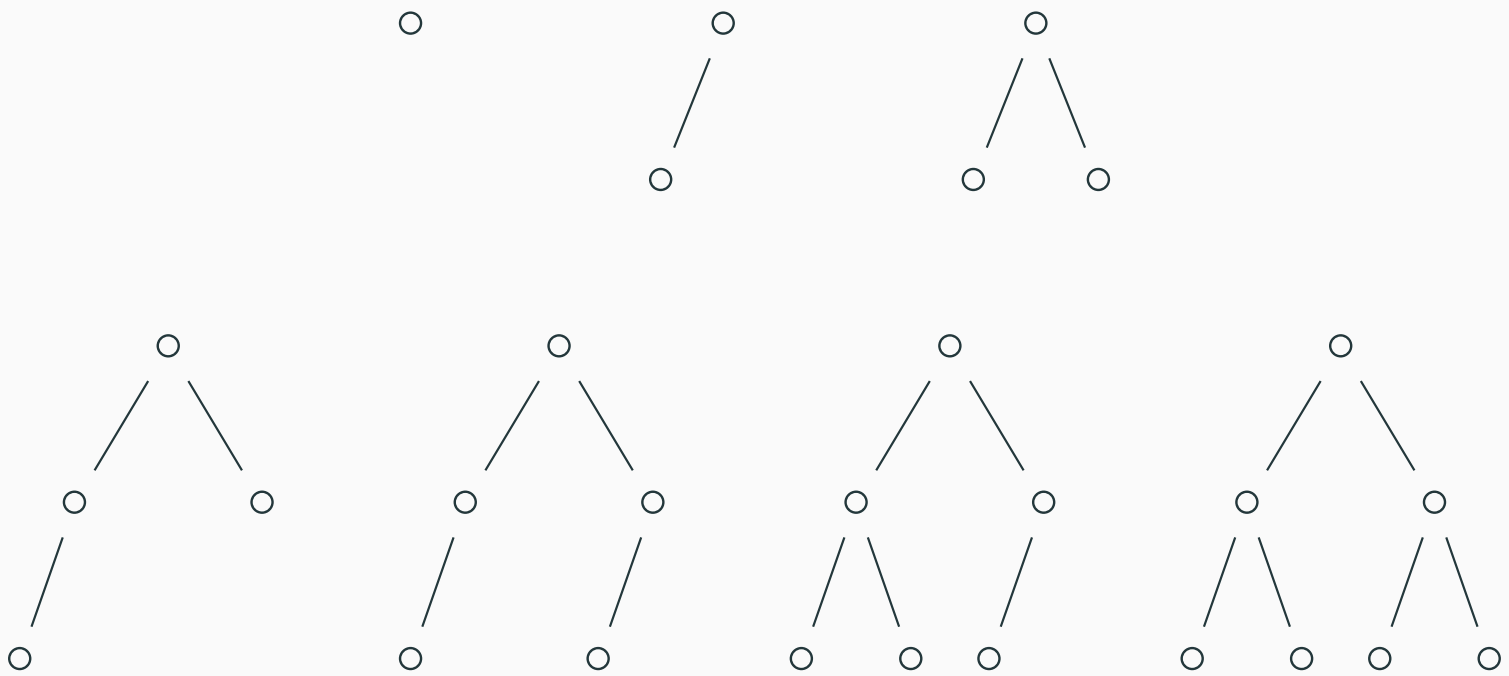
Weight-balanced binary trees with a strict balancing criterion:

$$\|G\| = \|D\| \quad \text{or} \quad \|G\| = \|D\| + 1 \quad \text{for each node } \langle G, x, D \rangle$$

where $\|G\|$ denotes the size (number of nodes) of tree G.

This balancing criterion is so strict that the shape of a Braun tree is entirely determined by its size!

Braun trees of sizes 1 to 7



Braun trees as persistent arrays

A Braun tree of size n, carrying values of type A at nodes, implements a persistent array with indices $1, \ldots, n$.

Reading at index i : (time $\mathcal{O}(\log n)$)

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

Writing value v at index i : (time and space $\mathcal{O}(\log n)$)

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

Braun trees as extensible persistent arrays

Extension to the left by adding a first element v :

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

Time and space $\mathcal{O}(\log n)$.

We can check easily that

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

Braun trees as extensible persistent arrays

Extension to the right by adding a last element v :

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

The n argument to $\texttt{addlast}(n, v, t)$ is initially $n = \|t\| + 1$.

We can store the size $\|t\|$ along with the Braun tree
(a persistent array = a pair (tree, size))

$\Rightarrow$ time and space $\mathcal{O}(\log n)$.

Fast initialization of a Braun tree

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

`make2` n v returns two Braun trees of respective sizes $n+1$ and n, where all values are equal to v, in time $\mathcal{O}(\log n)$.

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

Note: time $\mathcal{O}(\log n) \Rightarrow$ space $\mathcal{O}(\log n) \Rightarrow$ efficient sharing.

Prefix trees with high-degree nodes

Two possible improvements over Braun trees:

1. Increase the degree of the nodes of the prefix tree:
   from 2 to e.g. 16, 32 or 64, which corresponds to processing indices by groups of 4, 5 or 6 bits.

2. Scan the bits of indices starting with the most significant bits (big-endian representation), so as to increase spatial locality when accessing consecutive indices i, $i + 1$, $i + 2$, ...

   (A downside: we lose extensibility at the beginning of the array.)

Example of prefix trees of degree 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
```

Persistent arrays: imperative implementations

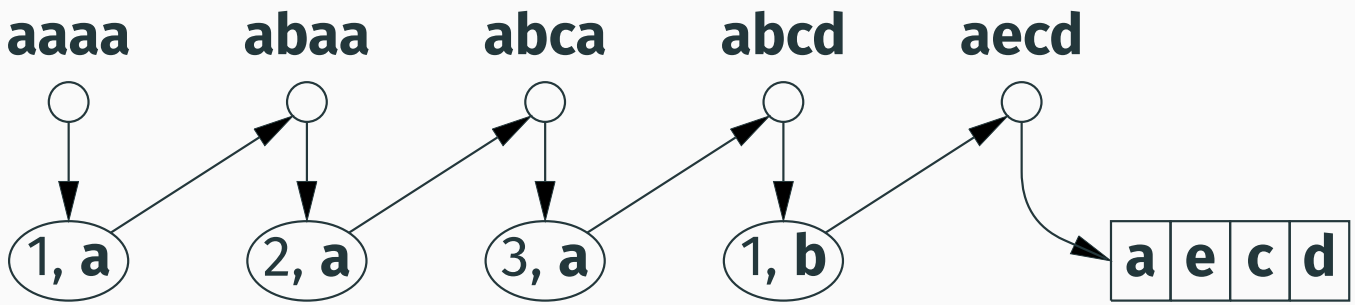
Persistent arrays implemented using ephemeral arrays

Instead of defining a persistent array structure in our purely functional language, let's assume that this structure is provided as a built-in type of the language.

We can, then, implement it as part of the runtime system of the language, in an imperative language, using ephemeral arrays.

What performance can we reach in this case?

Baker’s persistent arrays (from the 4th lecture)



An ephemeral array that reflects the current version of the persistent array (the one produced by the latest write), plus “deltas” (position + value) to represent the other versions.

Complexity analysis

Space: $\mathcal{O}(n + w)$ ✔

where n = size of the array and w = number of writes.

Accessing an arbitrary version with `get`/`set`:

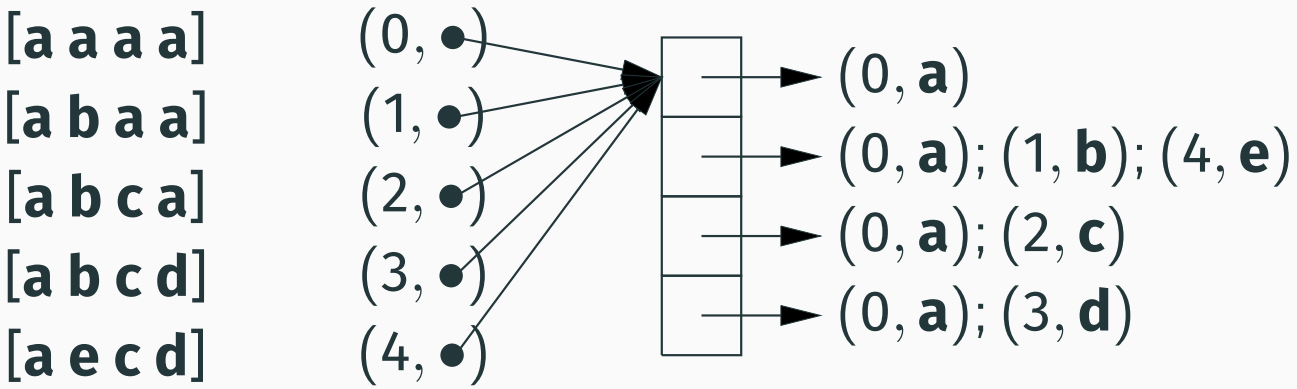
- ✘ $\mathcal{O}(w)$ worst-case time;
- ✘ $\mathcal{O}(n)$ amortized time if we use global rebuilding.

Accessing the version produced by the most recent `set`:

- ✔ $\mathcal{O}(1)$ worst-case time.

Perfect for single-threaded uses of the array,
as in the case of the transcription of imperative code, or …
the simulation of a RAM !

Arrays of fat elements (from the 4th lecture)



A mutable array containing, for each index, its history:
a list of pairs (modification date, new value at that date).

Assigning dates: trivial (0, 1, 2, ...) for partial persistence;
for total persistence, a difficult problem (the order maintenance problem) that can be solved in amortized $\mathcal{O}(1)$ time by on-demand renumbering of the dates.

How to represent histories?

We can globally rebuild the array after n writes ($n =$ array size).
Each history, therefore, has size $\leq n$.

Operations on histories:

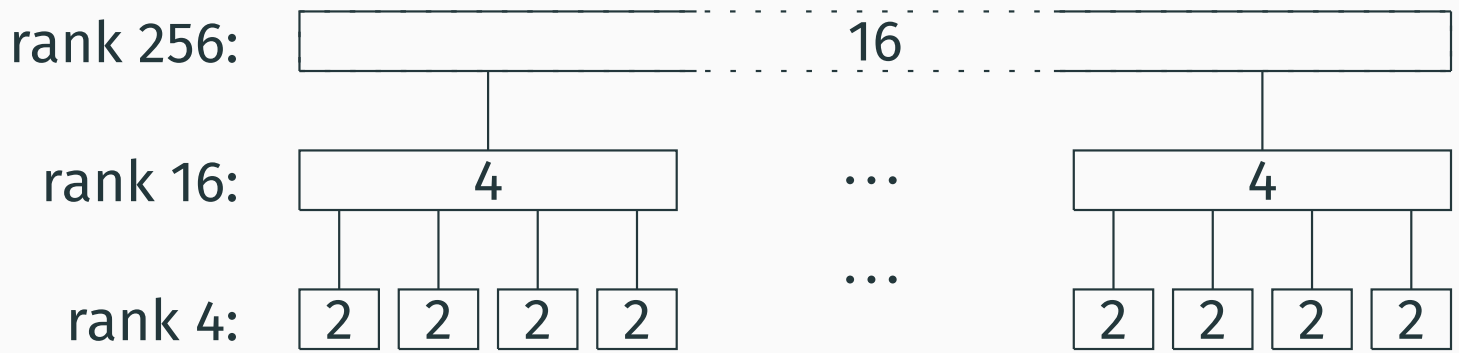
- for reading at date t: find the history element (t', v) with $t' \leq t$ and t' maximal;
- for writing: insert an element (t, v), using only $\mathcal{O}(1)$ space.

Simple solution: an ephemeral binary search tree, with in-place updates, and a finger pointing to the rightmost element.
$\Rightarrow$ reads and writes in $\mathcal{O}(\log n)$ worst-case time,
$\mathcal{O}(1)$ time for the latest version.

van Emde Boas trees (1975)

A representation of sets of integers between 0 and M as prefix trees of highly variable arity:

a tree of rank M = an array C of $\sqrt{M}$ trees of rank $\sqrt{M}$
+ min and max values
+ an aux tree of rank rang $\sqrt{M}$
indicating the nonempty entries of C.



Complexity analysis

The height of a VEB tree of rank M is

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

Operations in time $\mathcal{O}(\log \log M)$:

- Set operations: membership, insertion, deletion.
- Ordered operations: min, max,
  predecessor (largest $j \in S$ such that $j < i$), successor.

Prohibitive space: $\mathcal{O}(M)$. ($M \gg$ number of elements)

As big as an array of M bits… Such a bit-vector implements set operations in $\mathcal{O}(1)$ but ordered operations in $\mathcal{O}(M)$.

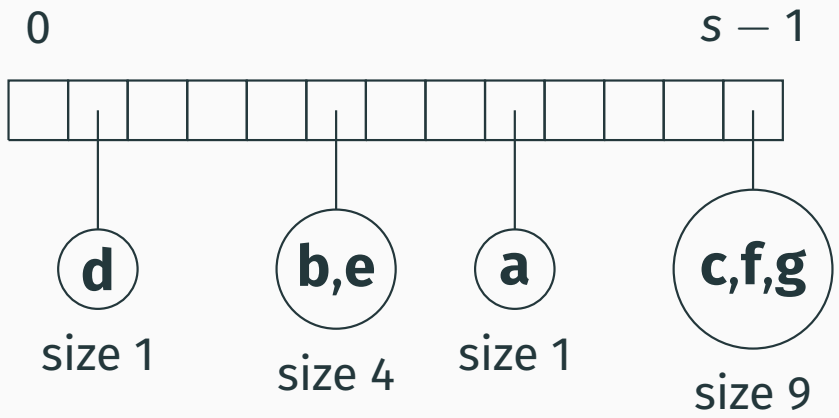
Compact van Emde Boas trees

Idea: in a tree of rank M, let's replace the array of $\sqrt{M}$ subtrees, of which $n \ll \sqrt{M}$ are non-empty, with a hash table (mapping indices $[0, \ldots, \sqrt{M})$ to subtrees) of size $\mathcal{O}(n)$.

Basic hash tables: search and insertion in time $\mathcal{O}(1)$ if no collisions, degrading to $\mathcal{O}(n)$ if many collisions.

Perfect hash tables: draw hash functions randomly to avoid collisions; operations in $\mathcal{O}(1)$ amortized expected time.

Perfect hashing



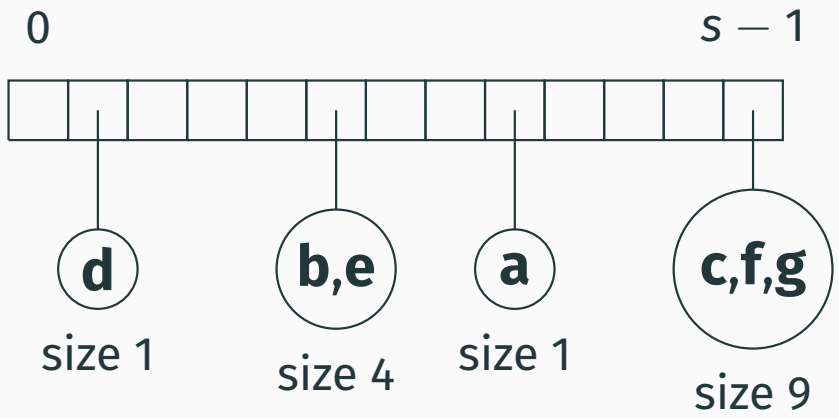
The static case: (Fredman, Komlós, Szemerédi, 1984)

We divide n elements (known in advance) among $s = 2(n-1)$ buckets using a hash function H.

If bucket i contains $j > 1$ elements, we turn it into a hash table of size j^2, using a hash function H_i randomly drawn until there are no collisions on the j elements.

The total size remains $\mathcal{O}(n)$ with probability 1.

Perfect hashing



The dynamic case: (Dietzfelbinger, Karlin, Mehlhorn, Meyer, Rohnert, Tarjan, 1994)

Same approach, but must adapt dynamically to the insertion of new elements.

- When a collision occurs in bucket i, draw a new hash function H_i and re-hash the bucket.
- After sufficiently many insertions, globally rebuild the table with a bigger value of s.

Dietz's persistent arrays

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

Persistent arrays with accesses in $\mathcal{O}(\log \log n)$ amortized expected time, and expected size $\mathcal{O}(n + w)$.

Basic idea: an ephemeral array of fat elements,
each fat element being a compact VEB tree using dynamic perfect hashing.

Delivers the desired complexity for partial persistence (with dates = $0, 1, 2, \ldots$) but not for full persistence:
renumbering the dates requires updates to the VEB trees that take $\mathcal{O}(\log n \cdot \log \log n)$ time.

Dietz's persistent arrays

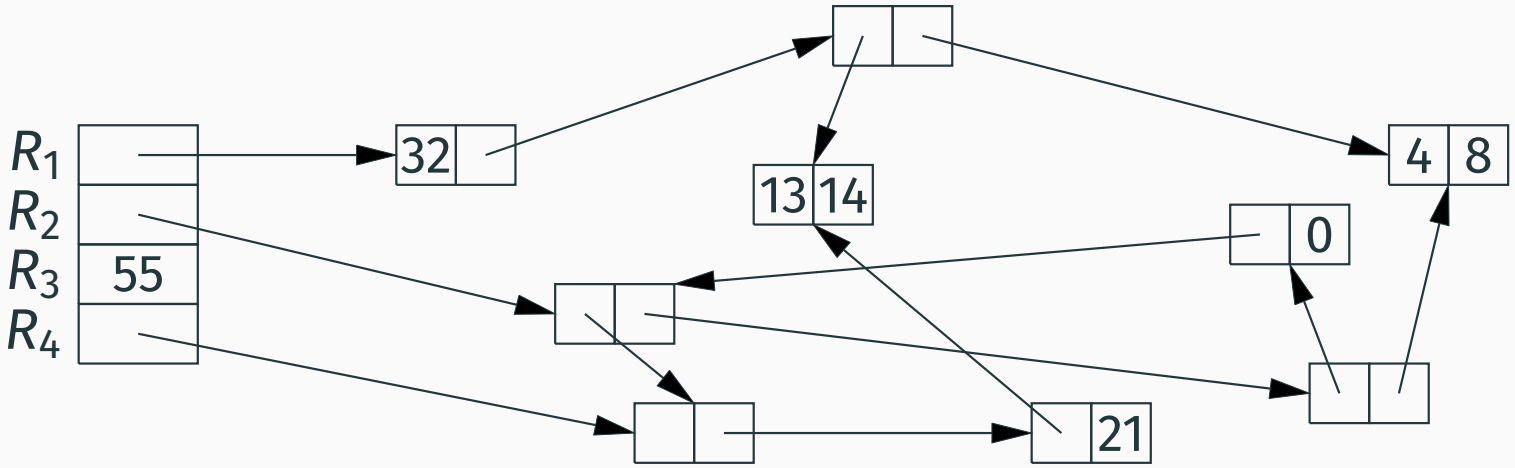
Two improvements:

- At the leaves of the VEB trees, put not just history entries but binary search trees containing $\mathcal{O}(\log n)$ history entries, with operations in time $\mathcal{O}(\log \log n)$.
  (These BST require no modifications when renumbering dates, since the relative order of dates is preserved.)
- Take these groups of dates into account in the renumbering algorithm, so that renumbering triggers only $\mathcal{O}(1)$ amortized operations in the VEB trees instead of $\mathcal{O}(\log n)$.

This restores the desired complexity: accesses in $\mathcal{O}(\log \log n)$ amortized, expected time, and $\mathcal{O}(n + w)$ expected size.

The LISP model vs. the RAM model

A model of computation: the LISP model / the pointer machine



A fixed number of registers $R_1, \ldots, R_k$
plus arbitrarily-many two-field cells accessible only via pointers.

Each register, each cell field contains either a pointer or a value of a base type T (such as arbitrary-precision integers).

No conversions between base values and pointers.
Pointers are opaque.

The operations of the pointer machine

Operations over pointers:

$R_k := \texttt{cons}(R_i, R_j)$	allocate and initialize a cell
$R_j := \texttt{car}(R_i)$	read the first field pointed by R_i
$R_j := \texttt{cdr}(R_i)$	read the second field pointed by R_i
$\texttt{rplaca}(R_i, R_j)$	write R_j in the first field pointed by R_i
$\texttt{rplacd}(R_i, R_j)$	write R_j in the second field pointed by R_i
$R_j := \texttt{consp}(R_i)$	test whether R_i is a pointer

Plus: operations over the base type T such as addition, subtraction, test against zero.

Cost model: pointer operations take constant time;
operations over T generally take constant time (uniform model).

Simulating the RAM with a pointer machine

Let's represent the RAM memory state by a tree-shaped structure implementing a finite map adress $\mapsto$ value,
(e.g. a balanced binary search tree).

Translating the RAM instructions:

`load` instructions	$\to$	the `get` operation of the finite map
`store` instructions	$\to$	the `set` operation of the finite map
other instructions	$\to$	unchanged

If the program runs on the RAM in time t and space s,
it runs on the pointer machine in time $\mathcal{O}(t \log s)$ and space $\mathcal{O}(s)$.

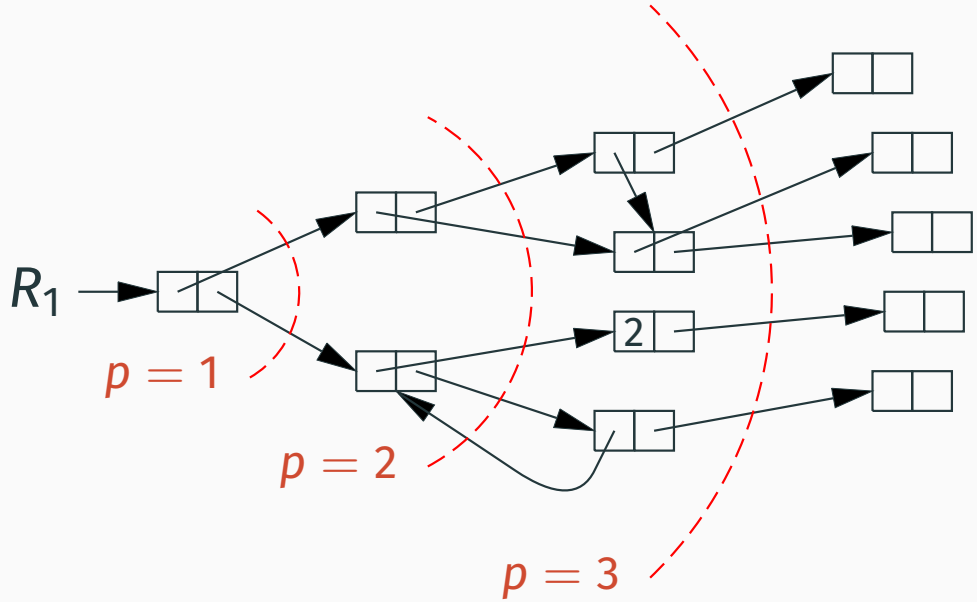
A lower bound

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

Ben-Amram and Galil show that this simulation is optimal:

Simulating a RAM program that uses time t and space s on a pointer machine takes time $\Omega(t \log s)$,
as long as the base type T is incompressible.

Reachability of memory words



In the pointer machine, let's call "memory word"
a `car` or `cdr` field of a `cons` cell.

Starting with the pointers contained in registers $R_1, \ldots, R_k$
and executing p instructions `car`, `cdr`, `rplaca`, `rplacd`,
we can read or write a word among at most $2k(2^p - 1)$ words.

Reachability of memory words

Corollary: if we work with $s > 2k(2^p - 1)$ memory words, there exists one word that requires at least $p + 1$ instructions to be read or written. (That is, $\Omega(\log s)$ instructions.)

We can, therefore, construct an "adversarial" RAM program containing $\Omega(t)$ `load` and `store` operations, each requiring $\Omega(\log s)$ pointer machine operations to be simulated.

$\rightarrow$ Lower bound $\Omega(t \log s)$.

But: this assumes that the s memory words of the RAM must be represented by $\Omega(s)$ memory words of the pointer machine.

The issue with compressible base types

We could try to encode the RAM memory state $M[1] \ldots, M[s]$ by one (or a small number of) values of base type T.

For example, we could use integers and a Gödel encoding:

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

Of course, `get` and `set` operations on this encoding are expensive (we have to factor the number).

But how can we rule out the possibility of efficiently compressing s values of the base type in a much smaller number of values?

A notion of incompressibility

Ben-Amram and Galil say that a type T is incompressible if, for all $p > q$, there exists no easily-computable injection $T^p \to T^q$.

More precisely: there exists no functions

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

such that $g(f(x)) = x$ for all x,
and such that f and g are computable by the pointer machine in time independent of x.

Ben-Amram and Galil show that if there exists a simulation of the RAM by the pointer machine in time $o(t \log s)$, then T is compressible.

Examples of incompressible types

Finite types such as $T = k$-bit integers.

Cardinality argument: if $p > q$, then $\text{card}(T^p) > \text{card}(T^q)$, therefore the injection f does not exist.

Integer numbers $\mathbb{N}$ with addition, subtraction, multiplication

All operations can be quotiented modulo 2, hence a compression $f : \mathbb{N}^p \to \mathbb{N}^q$ would give a compression $\hat{f} : \{0,1\}^p \to \{0,1\}^q$.

Integer numbers $\mathbb{N}$ with $+$, $-$, $\times$, $/2$, and $<$

See Ben-Amram and Galil.

Real numbers $\mathbb{R}$ with continuous operations

See Ben-Amram and Galil.

A compressible type

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

Consider arbitrary-precision integers with left and right shifts by n bits in constant time.

We can easily encode any number of d-bit integers in a single integer V, with constant-time access:

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

We can also increase d by recoding in time $\mathcal{O}(\log n)$, where n is the number of integers stored in V.

Ben-Amram and Galil extend this to an encoding of n unbounded integers into $\alpha(n)$ integers, with reads in time $\mathcal{O}(\alpha(n))$ and writes in time $\mathcal{O}(1)$ amortized.

Pure LISP vs impure LISP

Two flavors of pointer machines

Pure LISP = pointer machine without `rplaca` nor `rplacd`.

In this model, a cons cell is never modified after allocation and initialization. Assignment to registers remain possible.

Equivalent to Core ML without references, or to Core Haskell with strict evaluation.

Impure LISP = pointer machine with `rplaca` and `rplacd`.

Cells can be modified in-place after initialization.

Equivalent to Core-ML with mutable references.

Pure LISP vs. impure LISP

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

Pippenger (1997) considers the following problem:

> Given a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$,
> and an infinite stream of n-tuples $(x_1, \ldots, x_n); (x'_1, \ldots, x'_n); \ldots$
> output "in real time" the stream of permuted tuples
> $(x_{\pi(1)}, \ldots, x_{\pi(n)}); (x'_{\pi(1)}, \ldots, x'_{\pi(n)}); \ldots$

Pure LISP vs. impure LISP

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

Pippenger (1997) considers the following problem:

> Given a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$,
> and an infinite stream of n-tuples $(x_1, \ldots, x_n); (x'_1, \ldots, x'_n); \ldots$
> output "in real time" the stream of permuted tuples
> $(x_{\pi(1)}, \ldots, x_{\pi(n)}); (x'_{\pi(1)}, \ldots, x'_{\pi(n)}); \ldots$

Adding some extra constraints, he shows that

- In impure LISP, we can preprocess the problem in time $\mathcal{O}(n \log n)$, then each permutation takes $\mathcal{O}(n)$ time.
- In pure LISP, each permutation must take $\Omega(n \log n)$ time (same argument as for sorting), and no preprocessing helps.

Pippenger's scenario

A server-style program, with input/output.

(We add `read` and `write` primitives to the two LISP models.)

Read a number n

Read n numbers $\pi(1), \ldots, \pi(n)$

Repeat forever:

Read n symbols $x_1, \ldots, x_n$

Compute $(y_1, \ldots, y_n) = (x_{\pi(1)}, \ldots, x_{\pi(n)})$

Write $y_1, \ldots, y_n$

The program is interactive: we must output the permutation of the current tuple before reading the next tuple.

The program is parametric over its inputs: the x_i inputs are neither integers nor character strings, but abstract symbols for which we have no operations.

An OCaml implementation

We represent input/output by “sequences” from the OCaml standard library (type `'a Seq.t`): possibly infinite lists, evaluated on-demand but without memoization.

```
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
```

Applying a permutation

A naive solution using lists: $\mathcal{O}(n^2)$

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

A solution using primitive arrays $\mathcal{O}(n)$
(not available in pure LISP nor in impure LISP):

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

Applying a permutation

A solution using functional arrays (Braun trees): $\mathcal{O}(n \log n)$

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

A solution with preprocessing, then sorting: $\mathcal{O}(n \log n)$

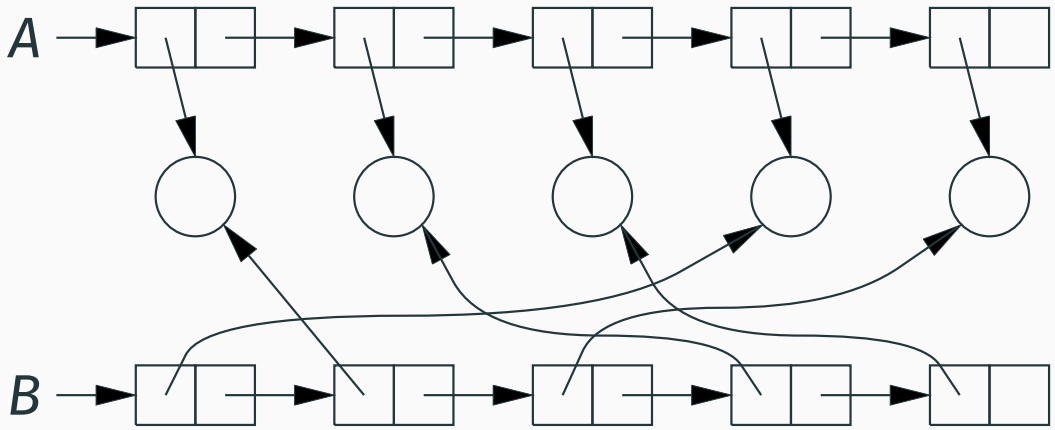
```
let permut invpi xs =
  List.combine invpi xs
  |> List.sort (fun (j1,x1) (j2,x2) -> Int.compare j1 j2)
  |> List.map snd
```

We need to precompute the inverse `invpi` of permutation `pi`, but the time it takes is amortized over all calls to `permut`.

Pippenger's solution

Uses lists of mutable references.

Can be expressed in impure LISP but not in pure LISP.



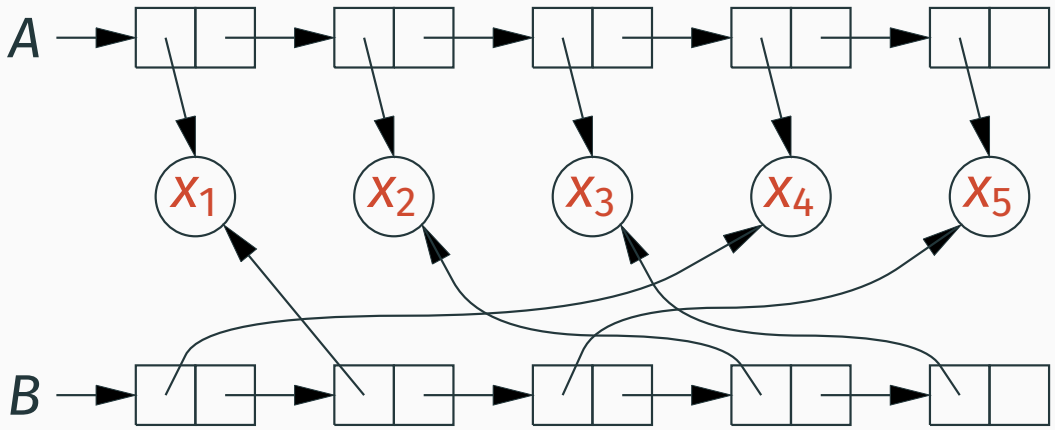
Preprocessing: construct a list A of n fresh references and its permutation $B = \texttt{permut}\ \pi\ A$.

(Using any inefficient `permut` function.)

Pippenger's solution

Uses lists of mutable references.

Can be expressed in impure LISP but not in pure LISP.



Permutation: store the inputs $x_1, \ldots, x_n$ in the references in the order of list A, then read back the references in the order of list B, obtaining the outputs $x_{\pi(1)}, \ldots, x_{\pi(n)}$.

Applying a permutation in impure LISP

```
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))
```

Time: $\mathcal{O}(n)$ for each call to `permut`.

An analysis of permutation in pure LISP

By parametricity hypothesis, the symbols $x_1, \ldots, x_n$ read from the input can be assumed distinct from all symbols read before.

By purity hypothesis, the x_i cannot be stored in cons cells allocated before the x_i were read.

The computation of the permutation from x_i to $y_i = x_{\pi(i)}$ therefore takes place entirely between the reading of the x_i and the writing of the y_i. No precomputed quantities can be reused.

An analysis of permutation in pure LISP

The code that permutes a n-tuple must give the correct result for any of the $n!$ possible permutations.

Therefore, it must take at least $\log_2 n! \sim n \log_2 n$ binary decisions. (Same argument as for sorting.)

Hence, the cost of the permutation is $\Omega(n \log n)$, and it must be paid in full at each read/permute/write cycle.

The revenge of lazy evaluation

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

In 1997, Bird and coauthors published a Haskell implementation of Pippenger's problem that runs in time $\mathcal{O}(n)$ per cycle.

This does not invalidate Pippenger's lower bound, since Bird's solution uses lazy evaluation of lists.
Recall that lazy evaluation $\Rightarrow$ memoization $\Rightarrow$ in-place updates.

Towards a lazy solution: a strict, non-interactive solution

Assume we can process the input in batches instead of interactively: read N tuples; permute them; write the results.

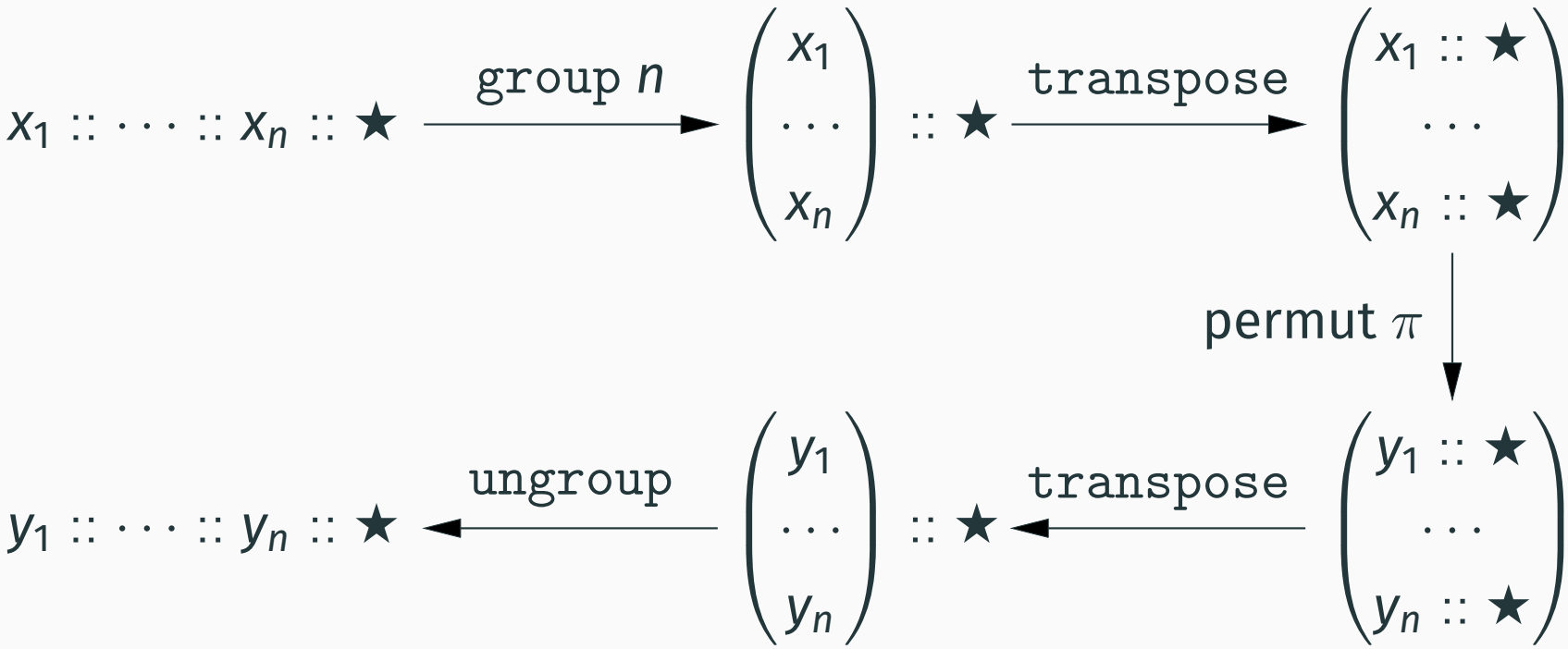
We could, then, process as follows:

- Transpose the input (a list of n-tuples), obtaining a n-tuple of lists.
- Permute (once) this n-tuple of lists.
- Transpose the permuted n-tuple of lists, obtaining a list of n-tuples.

Running time is dominated by the transposition steps, which take time $\mathcal{O}(n \cdot N)$, hence a $\mathcal{O}(n)$ amortized time per tuple.

Recovering interactivity with lazy lists

Using lazy lists, this solution becomes interactive: to obtain the first n elements of the output, it suffices to read the first n elements of the input.



(We write ★ for unevaluated suspensions.)

The lazy LISP model

A third model of computation appears, beyond pure LISP and impure LISP: lazy LISP, with cons cells evaluated on-demand and memoized.

Bird *et al*'s observation + Pippenger's result
$\Rightarrow$ a complexity gap between pure LISP and lazy LISP.

Is there a gap between lazy LISP and impure LISP?

References

References

Dietz's persistent arrays:

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

The gap between RAM and pointer machines:

- 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.

The gap between pure LISP and impure LISP:

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

Conclusions

Which algorithms for declarative programming?

First answer: the abstract algorithms we know and love, combined with

- persistent data structures instead of the usual ephemeral structures;
- state-passing style or a state monad.

We have persistent versions of all familiar data structures (finite maps, finite sets, heaps, stacks, queues, …)

- Often: same $\mathcal{O}$ complexity as the ephemeral versions.
  (Notable exceptions: arrays, hash tables, union-find.)
- Constant factors are reasonable (between 1 and 5).

Bonus: these persistent structures are highly reusable (modular encapsulation, “off-the-shelf” libraries).

Which algorithms for declarative programming?

Second answer: another algorithmic style is possible!

- Closer to mathematical definitions; less obsessed with sequencing computations and storing intermediate states.
  (Examples: theorem proving, compilation, static analysis.)
- Exploiting memory sharing between intermediate results.
- Closer to the formal verification of the algorithm.
  (Ex: using dependent types to reflect proof steps in the code.)

Persistent structures help here as well:
lots of sharing; high-level operations (joins, etc).

Benefits for imperative programming

1. Memory sharing!
2. Resilience to hazards in control flow:
   backtracking, error handling, transactional updates.

Example: in OCaml, I often replace a hash table by a mutable reference to a persistent map.

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

This makes backtracking on error trivial:

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

Benefits for imperative programming

1. Memory sharing!
2. Resilience to hazards in control flow:
   backtracking, error handling, transactional updates.

Using an atomic reference, we can perform transactions (atomic updates even in the face of concurrent updates).

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

(Other approaches to concurrency can be preferred: locking, lock-free data structures.)

The purely functional implementations

A very algebraic style of programming, with many advantages:

- Easy implementation in many languages,
  including purely functional languages and Agda / Coq / Lean.
- Elegant, concise presentation.
- An encouragement to provide many operations: not just `mem`/`add`/`remove` but also union, intersection, joins.
- Iterators are reliable and easy to specify
  (no iterator invalidation).
- Easy verification using equational reasoning
  (no need for a program logic).

Using lazy evaluation preserves these advantages while adding new possibilities for amortization, explicit scheduling, and memoization.

The imperative implementations of persistent structures

Very clever algorithms, most notably

- Baker's persistent arrays, because of their simplicity;
- Driscoll et al's fat node approach, because it can share memory even more than the purely functional approach.

Difficult to use in full persistence scenarios
(too slow or too hard to implement).

Very interesting for specific use cases:
single-threaded uses, partial persistence, semi-persistence.

Dietz's persistent arrays raise the question: is $\Omega(\log \log n)$ a lower bound for fully persistent arrays?

THE END