Persistent data structures, third lecture

Reconciling amortization and persistence: why laziness matters

Xavier Leroy

2023-03-23

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

Amortization

Analyzing sequences of operations

A program =

abstract algorithms

+ appropriate, efficient data structures

An abstract algorithm executes several operations in sequence on a given data structure.

What matters is not that each operation runs fast, but that the whole sequence of operation runs fast.

Example

n operations in time $\log n$ each $\rightarrow$ total time $\mathcal{O}(n \log n)$

$n - 1$ operations in constant time, 1 operation in linear time $\rightarrow$ total time $\mathcal{O}(n)$

A stack implemented as an extensible array

```
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--; }
}
```

When the stack is full, we reallocate an array twice as large, and copy the old array in the new array.

Analysis of a sequence of `push` operations

Consider a sequence of n `push`, where $2^{k-1} < n \leq 2^k$.

We performed n assignments `stk[sp] = v`, total time n.

We resized the array $k-1$ times:
from size 1 to size 2, ..., from size 2^{k-1} to size 2^k.

Each resizing $p \to 2p$ takes time p.

Total resizing time: $\sum_{i=0}^{k-1} 2^i = 2^k - 1$.

Total time for n `push`: $2^k - 1 + n \leq 2n + n = 3n$.

Analysis of a sequence of `push` operations

Consider a sequence of n `push`, where $2^{k-1} < n \leq 2^k$.

We performed n assignments `stk[sp] = v`, total time n.

We resized the array $k-1$ times:
from size 1 to size 2, ..., from size 2^{k-1} to size 2^k.

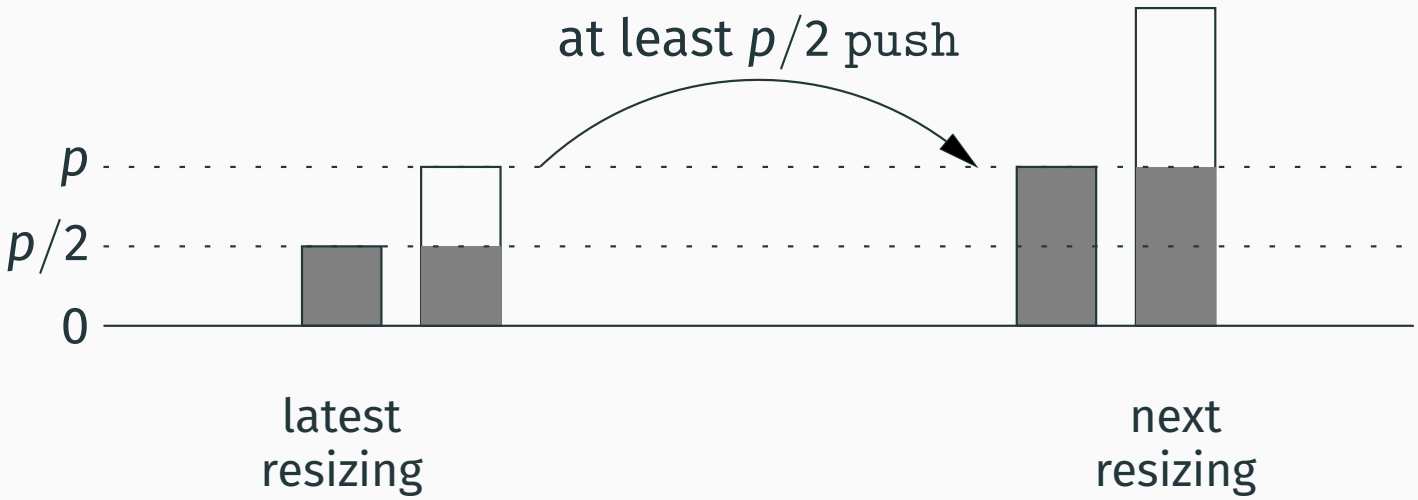
Each resizing $p \to 2p$ takes time p.

Total resizing time: $\sum_{i=0}^{k-1} 2^i = 2^k - 1$.

Total time for n `push`: $2^k - 1 + n \leq 2n + n = 3n$.

We say that `push` executes in amortized constant time,
since any sequence of n `push` takes time kn at most, for some constant k.

A more local analysis



Each resizing leaves the stack half empty.

If p is the new size, it will take at least $p/2$ `push` operations before the stack needs resizing again (for a cost of p).

The 2 cost added to each of the $p/2$ `push` pre-pays the p cost of the next resizing.

The banker's method (a.k.a. the accounting method)

In financial words:

$$\text{billed cost} \geq \text{actual cost} + \Delta\text{cash reserve}$$

In execution time words:

$$\text{amortized time} \geq \text{actual time} + \Delta\text{time credits}$$

The "cash reserve" (the time credits) must always remain ≥ 0.

In the stack example:

Operation	Amortized time	Actual time	Δtime credits
`top`, `pop`	1	1	0
`push` (no resizing)	3	1	$+2$
`push` (resizing)	3	$p+1$	$-p$

At least $p/2$ `push` without resizing take place before a resizing `push`. Therefore, time credits remain ≥ 0.

The physicist's method (a.k.a. the potential method)

A potential function Φ : state of the structure $\to \mathbb{R}_+$

Each operation must ensure that

$$\text{amortized time} \geq \text{actual time} + \Delta\Phi$$

For the stack example, we take $\Phi = \max(2\mathtt{sp} - p, 0)$
with $p = \mathtt{stk.length}$.

Operation	Amortized time	Real time	$\Delta\Phi$
`top`	1	1	0
`pop`	1	1	0 or -2
`push` (no resizing)	3	1	0 or $+2$
`push` (resizing)	3	$p+1$	$p+2$

Just before resizing, $\Phi = 2p - p = p$.
Just after, $\Phi = 0$, and at the end of the `push` $\Phi = 2$.

Soundness of amortized analysis

Consider a sequence of operations:

$$\sum \text{amortized times} \geq \sum \text{actual times} + \sum \Delta\Phi$$

If the structure is used in a single-threaded manner, and its successive states are $s_0, \ldots, s_n$,

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

Since $\Phi \geq 0$ and $\Phi(s_0) = 0$ (in general), we have

$$\text{time for the sequence} = \sum \text{actual times} \leq \sum \text{amortized times}$$

The running time for the sequence is correctly bounded by the result of amortized analysis.

The physicist's method vs. the banker's method

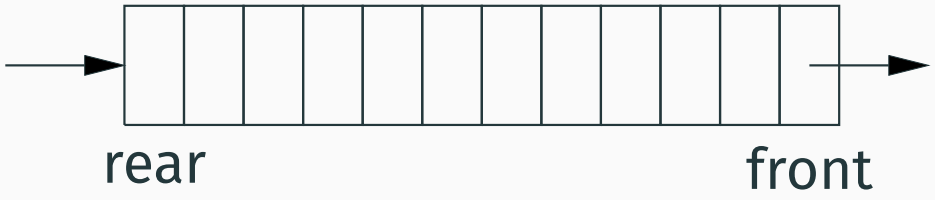
The physicist's method is more systematic:

- Once the Φ function is defined, amortized costs can be calculated almost automatically as $\max(\text{actual time} + \Delta\Phi)$.

The banker's method is more flexible:

- Credits can depend on the history of operations, not just the current state of the structure.
- Credits can be split across multiple accounts attached to different parts of the structure.

A persistent queue (FIFO)

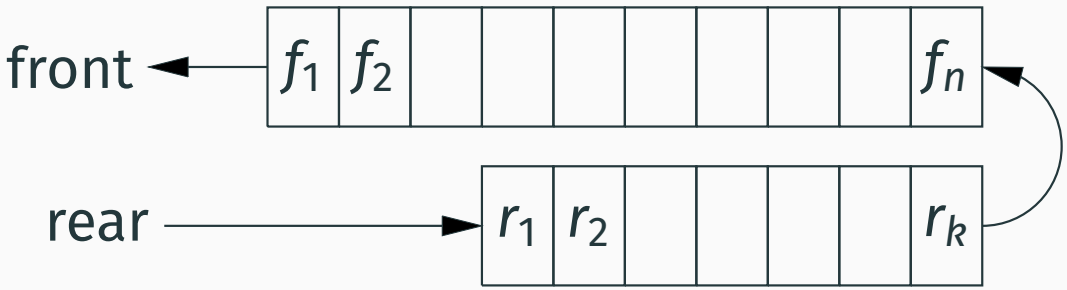


Trivial implementation: a singly-linked list.

```
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]
```

The `add` operation is inefficient: time $\mathcal{O}(n)$.

A persistent queue



Implemented as two lists:

- the "front" list f (head of f is first element out);
- the "rear" list r (head of r is last element in).

When the front list is empty, it can be refilled by reversing the rear list: $([\,], r) \to (\texttt{rev}(r), [\,])$.

This takes linear time $\mathcal{O}(\|r\|)$, but this time is amortized!
(over the $\|r\|$ insertions in the queue that created this list r).

Pure functional implementation of the queue

```
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: if the queue is not empty, $f \neq [\,]$.

In other words: if $f = [\,]$ then $r = [\,]$ and the queue is empty.

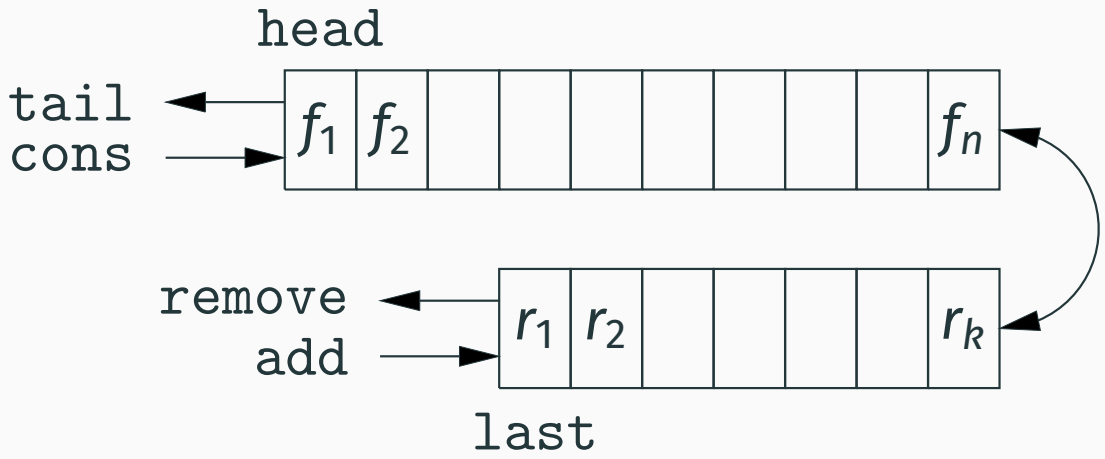
Amortized analysis

The potential is the length of the rear list:

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

Operation	Amortized time	Actual time	$\Delta\Phi$
`add`	2	1	$+1$
`head`	1	1	0
`tail` (no reverse)	1	1	0
`tail` (reverse)	1	$1 + \|r\|$	$-\|r\|$

Extension: a double-ended queue (dequeue)



We add operations to insert at head of queue (`cons`) and to lookup and extract at tail of queue (`last`, `remove`).

Same implementation by two lists (f, r), but:

- $([\,], r) \to (\texttt{rev}\ r_2, r_1)$ where $(r_1, r_2) = r$ split in the middle.
- $(f, [\,]) \to (f_1, \texttt{rev}\ f_2)$ where $(f_1, f_2) = f$ split in the middle.

Potential function for amortized analysis: $\Phi((f, r)) = \left|\ \|f\| - \|r\|\ \right|$.

Problem with persistent uses

Our queue is implemented in pure functional style and can therefore be used persistently, i.e. by reusing intermediate states of the queue.

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

The state `q` is represented as $f = [1]$ and $r = [3; 2]$.

Each of the N calls `tail q` must reverse list r, for a total overhead of $2N$.

This overhead cannot be amortized by the 3 `add` operations: no matter their amortized cost, this cost cannot account for $2N$ when $N \to \infty$.

Amortization and persistence

$$\sum \text{amortized times} \geq \sum \text{actual times} + \sum \Delta\Phi$$

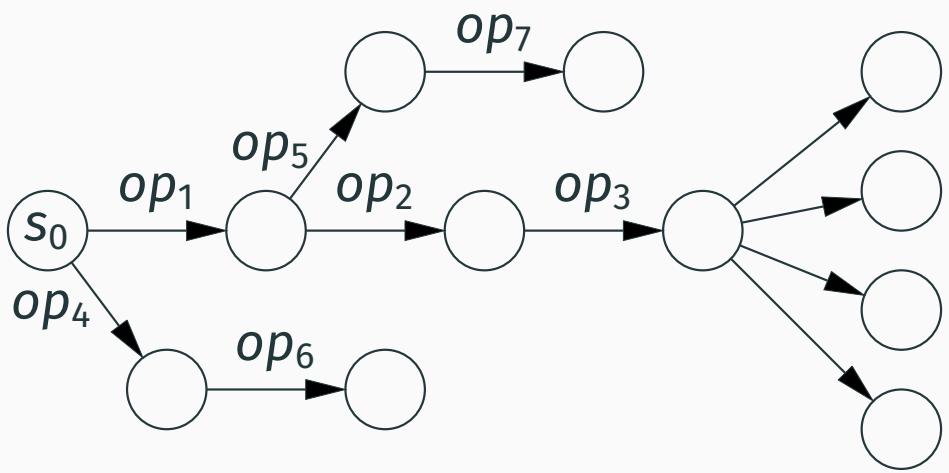
If the structure is ephemeral, or persistent but used in a single-threaded manner, each intermediate state is used only once:

In this case, we have $\sum \Delta\Phi = \Phi(s_n) - \Phi(s_0)$ and the actual running time is correctly bounded by $\sum$ amortized times.

Amortization and persistence

$$\sum \text{amortized times} \geq \sum \text{actual times} + \sum \Delta\Phi$$

A persistent structure can also be used in a non-single-threaded manner, with reuse of intermediate states:



We lose all guarantees that $\sum \Delta\Phi \geq 0$, and the actual running time of the sequence can exceed $\sum$ amortized times.

Lazy evaluation

Call by value, call by name

Call by value: the argument to a function call is evaluated before entering the function.

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

Call by value, call by name

Call by value: the argument to a function call is evaluated before entering the function.

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

Call by name: the argument is passed unevaluated to the function; it is evaluated every time its value is needed.

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

Call by value, call by name

Call by value: the argument to a function call is evaluated before entering the function.

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

Call by name: the argument is passed unevaluated to the function; it is evaluated every time its value is needed.

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

Call by name is normalizing, but not call by value:
$(\lambda xy.\, y)\ \omega\ 0 \to 0$ in CBN, diverges in CBV.

Call by name duplicates a lot of computations.

Call by need and lazy evaluation

Call by need: the argument is passed unevaluated to the function; it is evaluated the first time its value is needed, and the resulting value is memoized for future uses.

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

In a language like Haskell, this lazy evaluation also applies to data constructors, resulting in potentially infinite data structures that are evaluated on demand.

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

Implementing lazy evaluation in a strict language

Using references (an imperative feature) for memoization.

```
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
```

To suspend the evaluation of expression e, we write

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

Notations for lazy evaluation

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

Patterns: $pat ::= \ldots \mid$ `lazy` pat

`lazy` e suspends the evaluation of e

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

`lazy` pat forces a suspension and matches its value against pat

(like `match force susp with` pat)

Similar notations are used in Okasaki's book and in OCaml:

	This course	OCaml	Okasaki
Type of suspensions	`'a susp`	`'a Lazy.t`	`'a susp`
Suspend e	`lazy` e	`lazy` e	`$`e
Force and match against p	`lazy` p	`lazy` p	`$`p
Just force	`force`	`Lazy.force`	`force`

Lazy lists

```
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))
```

Note: the `map` function is "incremental", meaning that in order to produce the first k elements of `map f l`, it suffices to evaluate the first k elements of `l`.

Lazy merge sort

```
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))
```

When laziness improves algorithmic efficiency

Sorting takes place on demand: in `mergesort` s n, with $n = \|s\|$,

- the first element of the sorted list (= the minimum of s) is produced in time $\mathcal{O}(n)$;
- the following elements are produced in time $\mathcal{O}(\log n)$ each.

Hence, we can find the k smallest elements of list s by taking the first k elements of `mergesort` s n, in time $\mathcal{O}(n + k \log n)$.

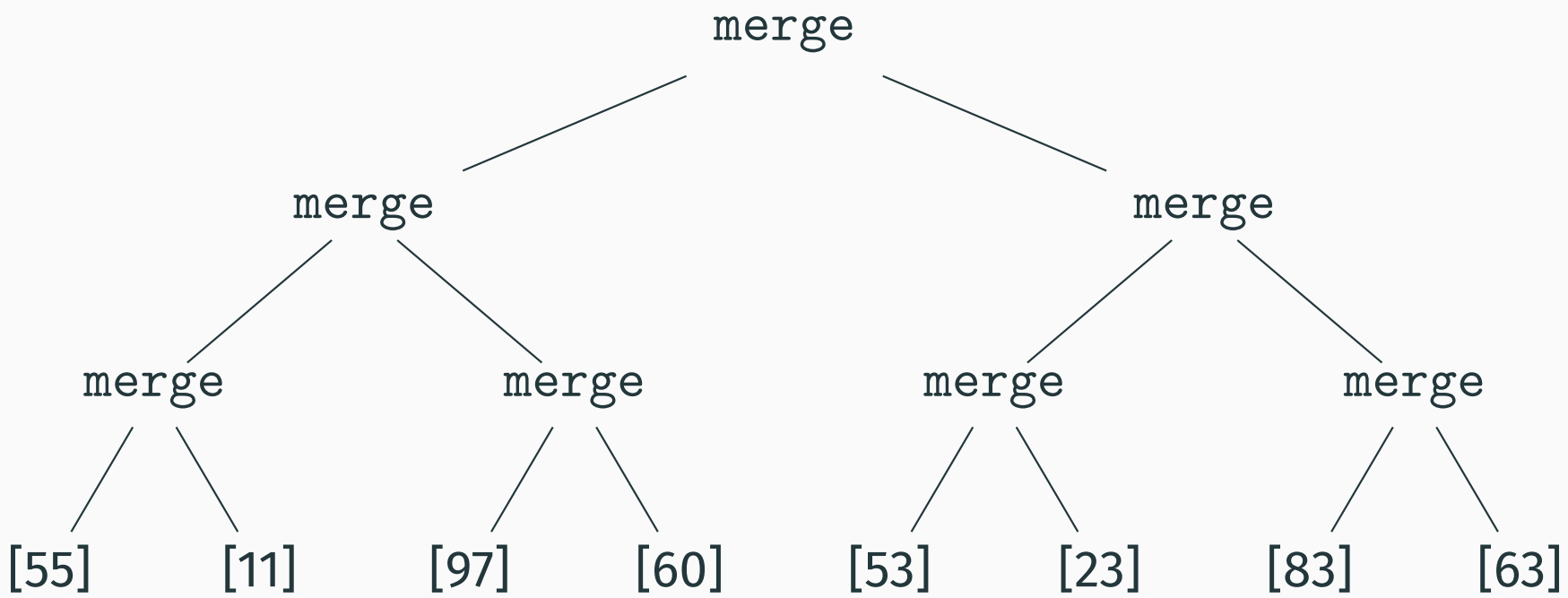
Lazy merging

```
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)
```

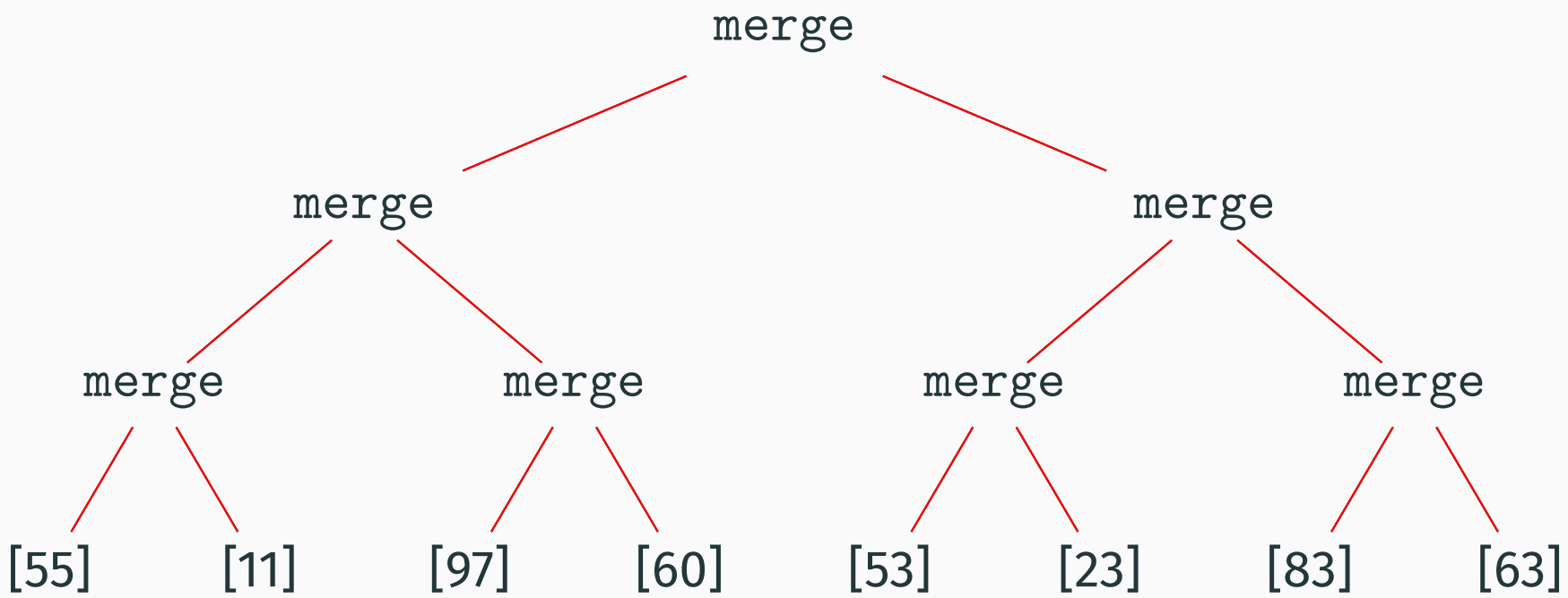
To obtain the first element of `merge s1 s2`, we must evaluate the first element of `s1` and the first element of `s2`.

To obtain the next element of `merge s1 s2`, we only need to evaluate the next element of one of the two inputs `s1` and `s2`. (The other next element has already been evaluated.)

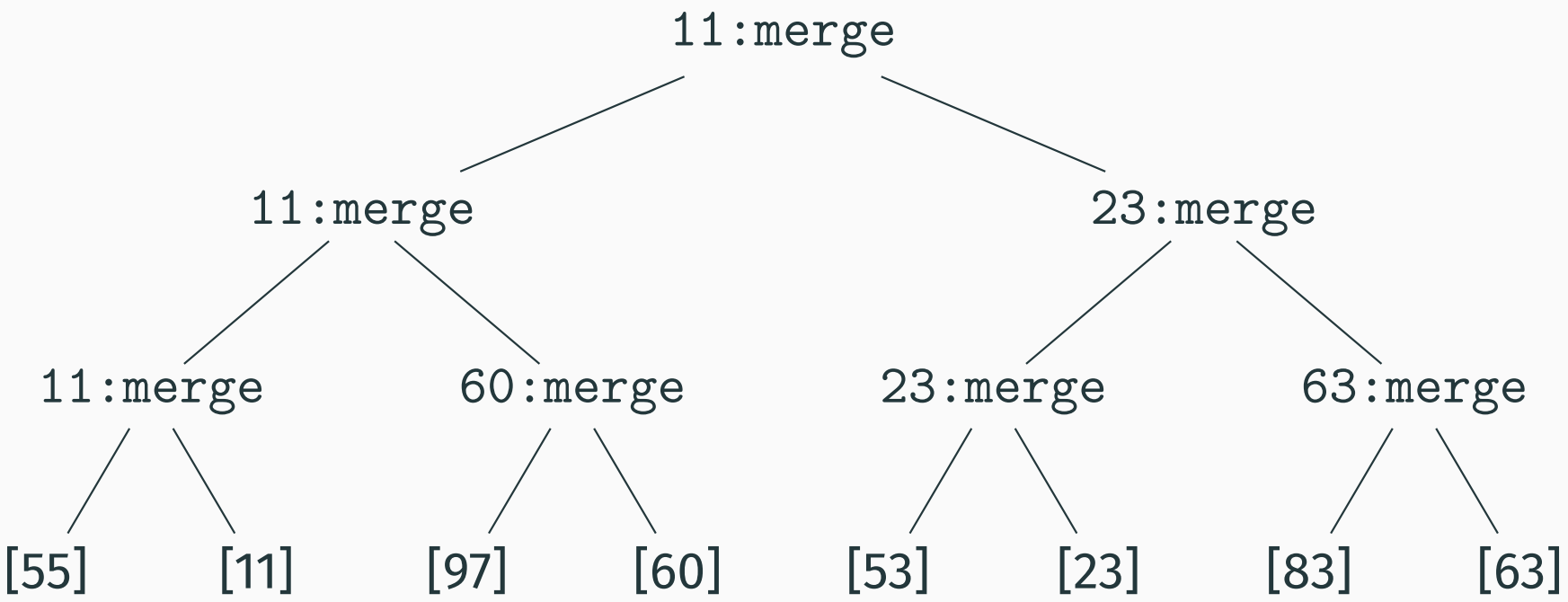
The tree of merges



The tree of merges

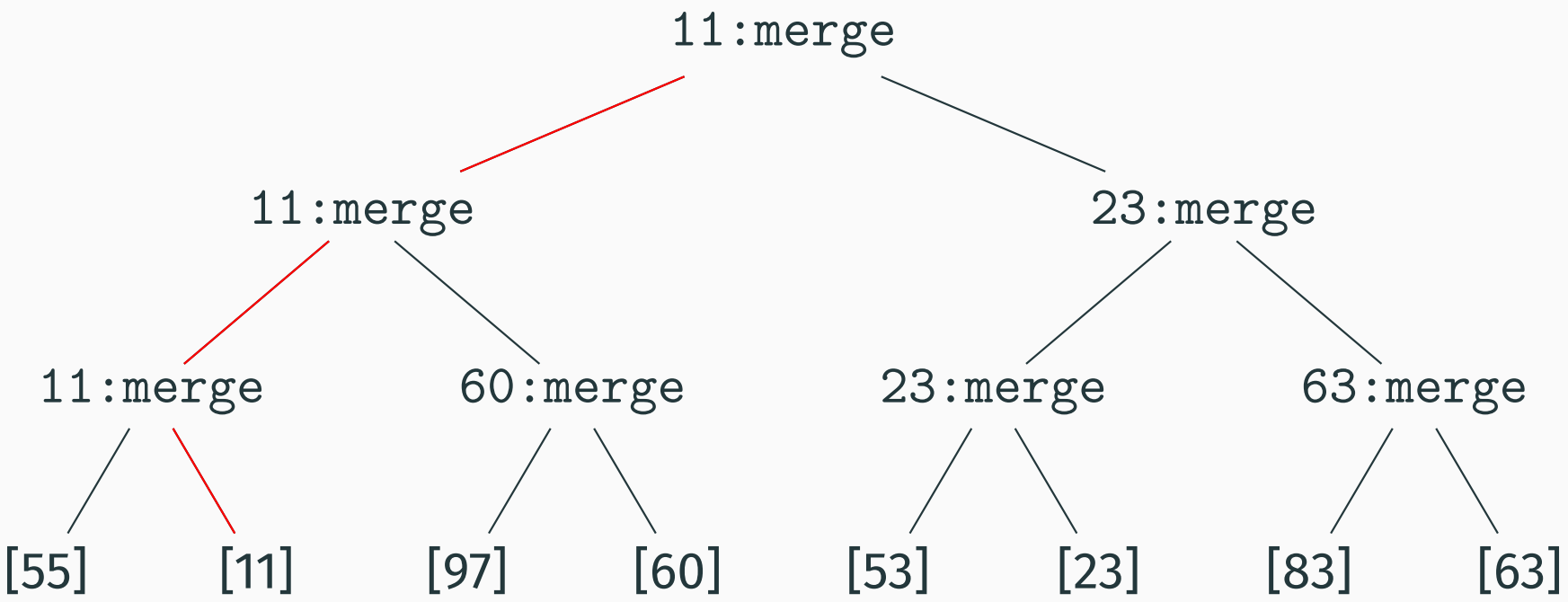


The tree of merges



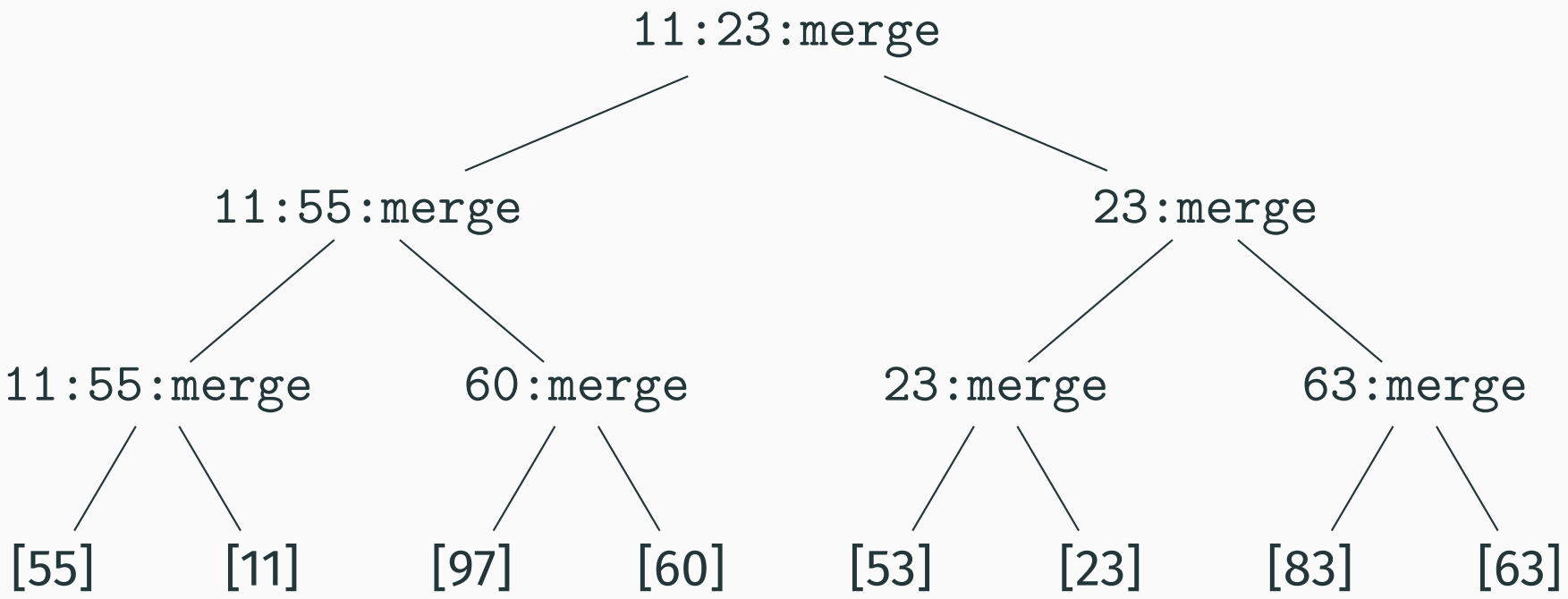
First result after evaluating all 7 `merge`.

The tree of merges



First result after evaluating all 7 `merge`.

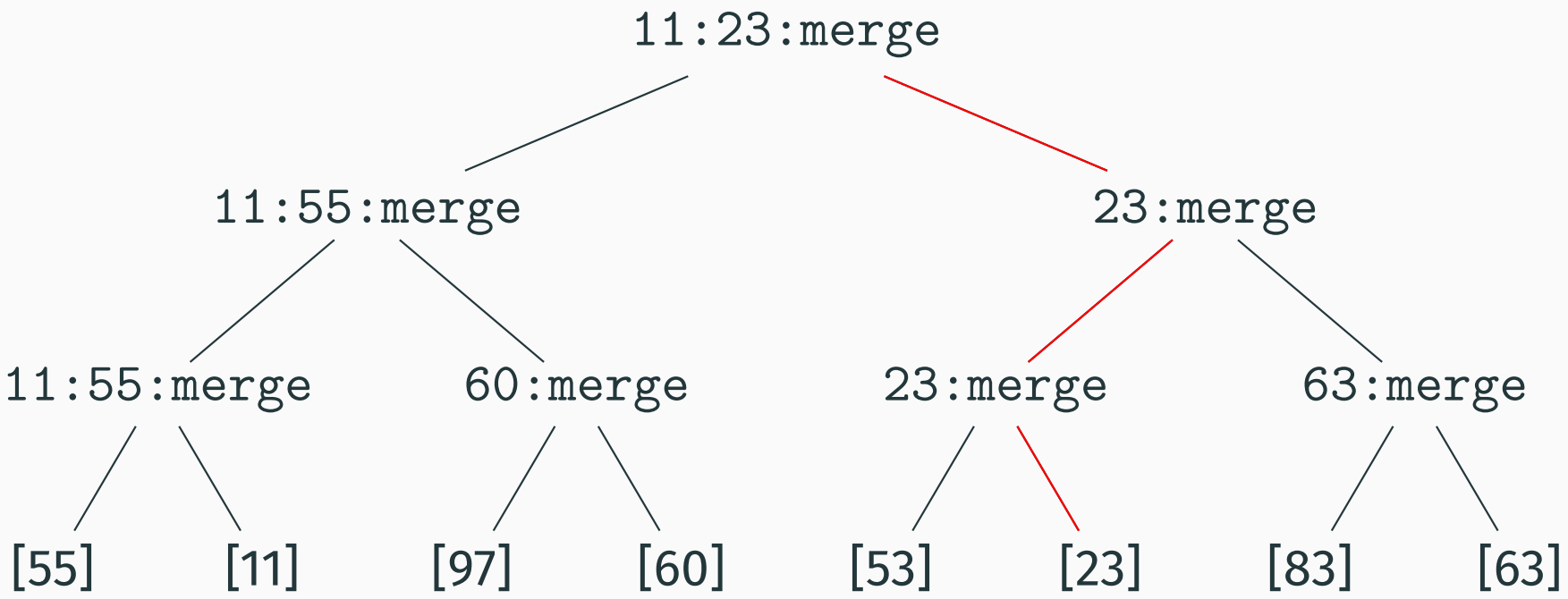
The tree of merges



First result after evaluating all 7 `merge`.

Second result after evaluating 3 `merge`.

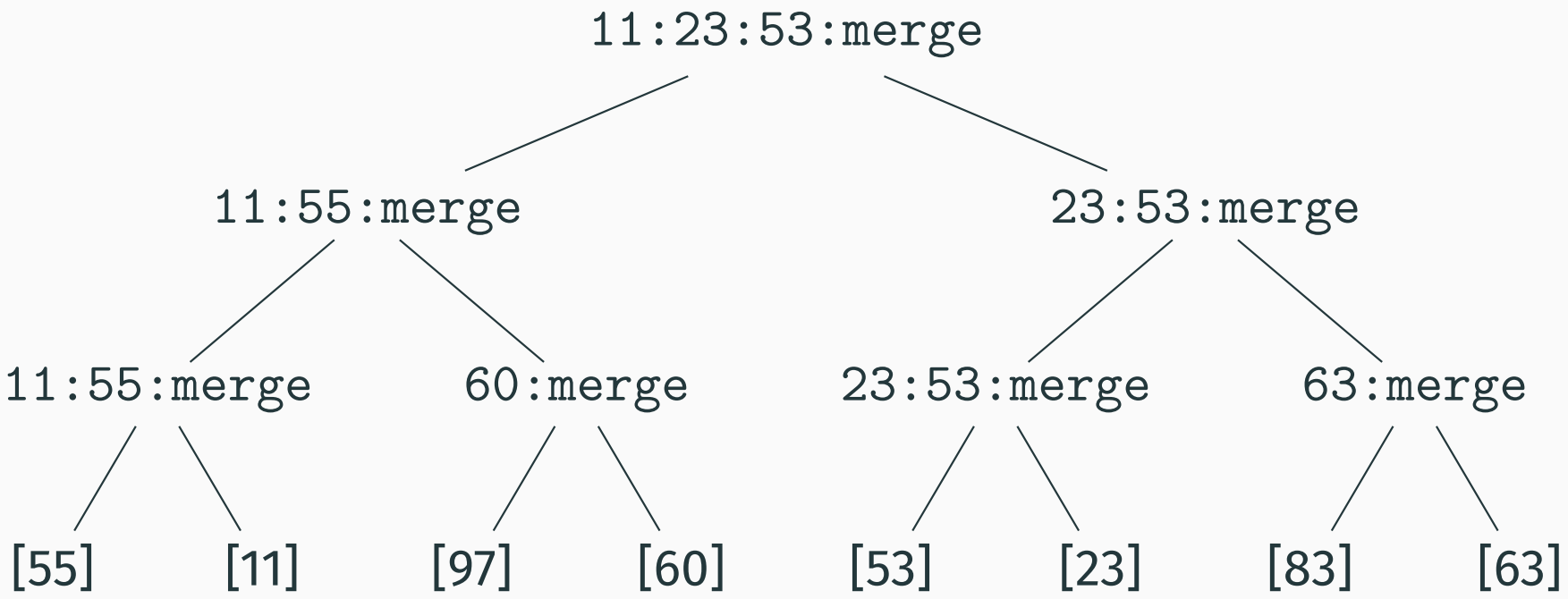
The tree of merges



First result after evaluating all 7 `merge`.

Second result after evaluating 3 `merge`.

The tree of merges



First result after evaluating all 7 `merge`.

Second result after evaluating 3 `merge`.

Third result after evaluating 3 `merge`.

Etc.

Concatenating or reversing lazy lists

```
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` is incremental: if each element of `s1` or `s2` is produced in time $\mathcal{O}(1)$, so is each element of `app s1 s2`.

`rev` is not incremental: the first element of `rev` ℓ is produced in time $\mathcal{O}(|\ell|)$, the others in time $\mathcal{O}(1)$.

Reconciling amortization and persistence

Reverse-then-concatenate lazy lists

Consider the lazy list $\texttt{app}\ s\ (\texttt{rev}\ \ell)$, where $\|s\| = \|\ell\| = n$.

The evaluation (in full) of $\texttt{rev}\ \ell$ triggers when we access the $n+1$-th element of this list (of length $2n$).

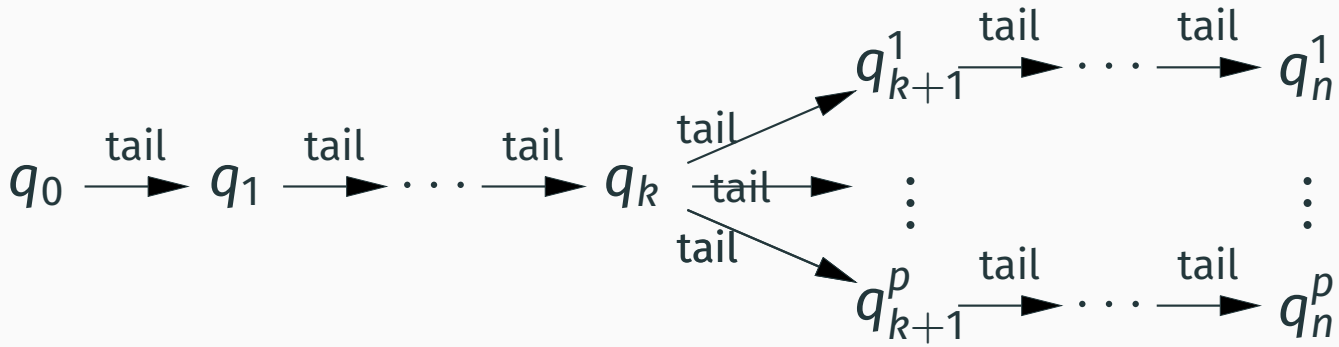
To evaluate (in full) the first i elements of this list takes time

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

In both cases, the time is $\leq 2i$.

We could therefore say that each element of the list $\texttt{app}\ s\ (\texttt{rev}\ \ell)$ evaluates in amortized constant time (2 units).

Lazy reverse-then-concatenate

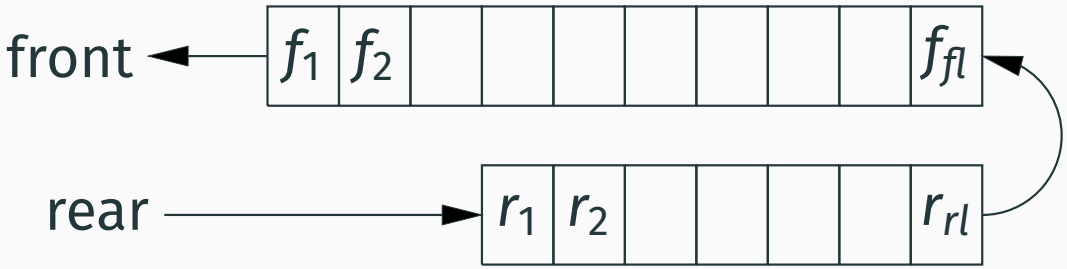


Let $q_0 = \texttt{app}\ s\ (\texttt{rev}\ \ell)$. Assume it is used non-linearly: after k `tail` we duplicate the list p times.

Thanks to laziness, `rev` ℓ is evaluated only once, and so are the $n - k$ remainingsteps of `app`. Therefore, the amortized cost of each `tail` remains constant.

(This would be false in call by name: `rev` ℓ would be evaluated p times. If $k = n - 1$ for example, the total cost would be $n + pn$ for $n + p$ `tail` operations, hence an amortized cost linear in p.)

The banker's queue (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)
```

A queue = a quadruple (fl, f, rl, r) where

- the front list, f, is lazy (head of f is first element out);
- the rear list, r, is strict (head of r is last element in);
- $fl = \|f\|$ and $rl = \|r\|$.

Invariant: $fl \geq rl$, therefore the queue is empty iff $fl = 0$.

The operations of the banker's queue

```
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)
```

Almost the same operations as for the usual non-lazy queue, except for the `check` function...

Normalization by rotation

The `check` function maintains the $|f| \geq |r|$ invariant, and makes sure the rear list is reversed "long enough in advance".

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

The lazy computation `app` f (`rev` r) is set up when $\|r\| = \|f\| + 1$.

As outlined previously, the cost $\|r\|$ it takes to evaluate `rev` r will be amortized over the future operations required to trigger this evaluation. The analysis is complicated by the fact that f could still contain non-evaluated `rev` r' coming from earlier rotations.

The 2.0 banker's method

No more accounts containing time credits, but only debts containing time debits.

To each suspension we associate a debt.

When evaluating `lazy` e, the initial debt must be $\geq$ actual evaluation cost of e.

The debt can be repaid (reduced) at any time by transferring time credits from the billed (amortized) cost.

$$\text{amortized time} \geq \text{actual time} + \text{repayments}$$

When the debt drops to zero, we can force the suspension and obtain its value, at no extra cost.

The debts of a lazy list

For a n-element list, we have n debts $d_1, \ldots, d_n \geq 0$.

The cumulative debt is $D(k) \stackrel{def}{=} \sum_{i=1}^{k} d_i$.

We can access (at no cost) the first k elements of the list if and only if $D(k) = 0$.

We can reduce a debt d_i by m, at an amortized cost of m:

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

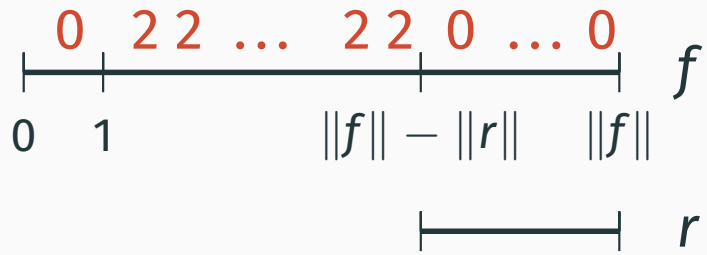
We can transfer a debt to an element farther to the left, at zero amortized cost:

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

The debts for the banker's queue

For a queue (f, r), the lazy list f has debts

- 0 for the first element;
- 2 for the next $\|f\| - \|r\| - 1$ elements;
- 0 for the last $\|r\|$ elements.

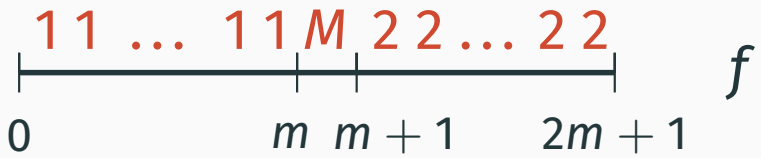


Since the first element of f has debt 0, it can be freely accessible at any time, notably by the `head` function.

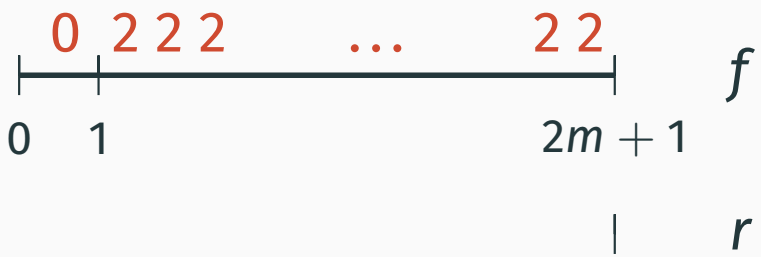
Analysis of the `check` operation

Rotation takes place when $\|f\| = m$ and $\|r\| = m + 1$.
All elements of f have 0 debt.

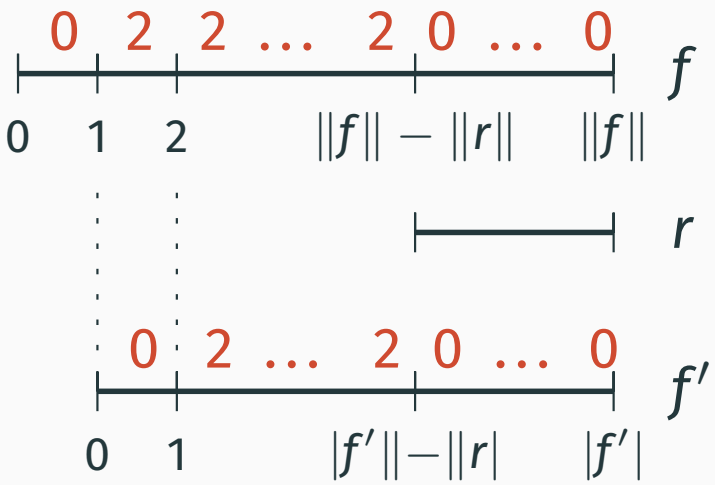
We create the list $\texttt{app}\ f\ (\texttt{rev}\ r)$ with debt 1 for each of the m `Cons` produced by `app`, debt $M = m + 1$ for the first element of `rev` r, and debit 2 (overpriced!) for the remaining elements.



We then spread $m - 1$ units from debt M over the beginning of the list, and we pay 1 to unblock the first element.



Analysis of the `tail` operation

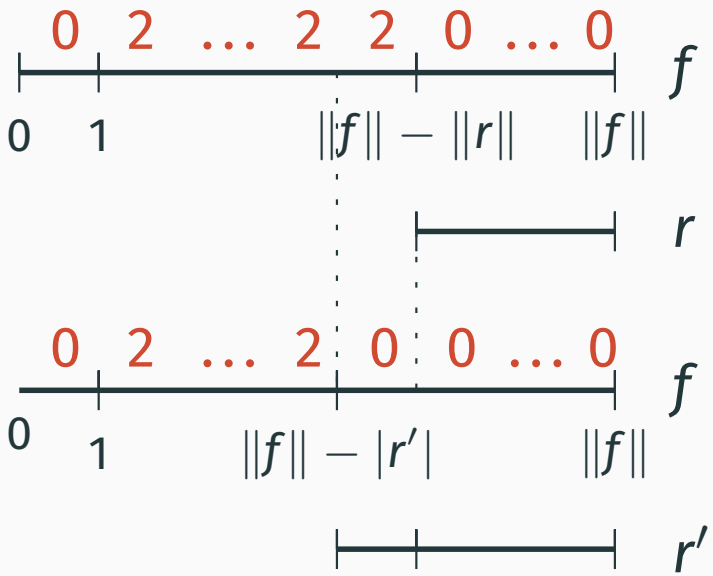


If `check` triggers no rotation, we go from f to f' by removing the first element of f. We need to unblock the second element of f, by paying 2.

If `check` triggers a rotation, we paid 1.

The actual time taken by `check` and `tail` is constant, hence `tail` runs in constant amortized time.

Analysis of the `add` operation



We go from r to r' by adding an element to r. We reduce the debt of the element in position $\|f\| - \|r\| - 1$ in f, from debt 2 to debt 0, by paying 2.

If `check` triggers a rotation, we need to pay 1 more.

The actual time taken by `check` and `add` is constant, hence `add` runs in constant amortized time.

Comparing the bankers

	Classic banker	Amortization banker
Maintains	cash reserves	debts
Reasons over	time credits	time debits
Amortizes over	past operations	future operations
Duplication?	credits not duplicable	residual debts duplicables
Uses?	single-threaded only	persistent

The 2.0 physicist's method

(A simplification of the 2.0 banker's method, appropriate when the persistent structure contains a single suspension.)

An anti-potential function Ψ : state of the structure $\to \mathbb{R}_+$

It's an upper bound of the sum of the debts from all the suspensions contained in the structure.

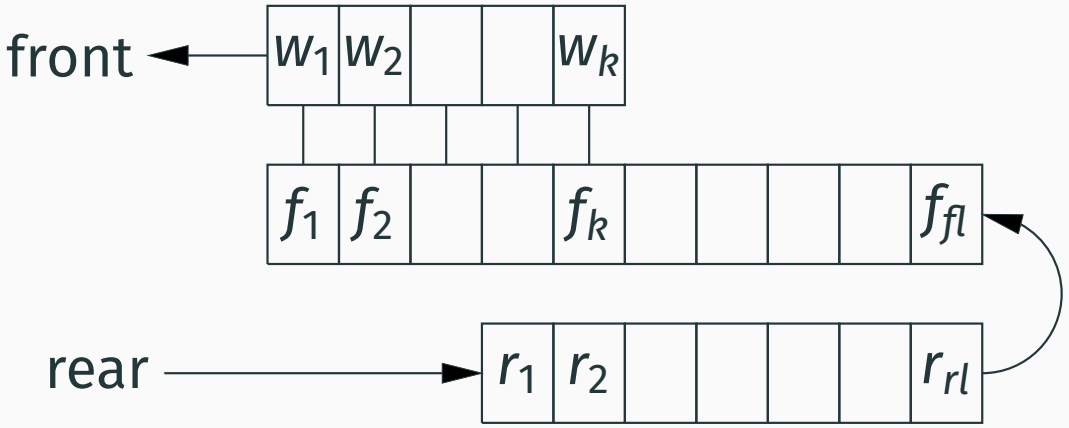
We can access the values of these suspensions iff $\Psi(s) = 0$.

Each operation must satisfy

$$\text{amortized time} \geq \text{strict time} - \Delta\Psi$$

Strict time is actual time + time to evaluate the newly-created suspensions, or in other words the actual time if we were using strict evaluation.

The physicist's queue (C. Okasaki, 1996)




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

A queue = a 5-tuple (w, fl, f, rl, r) with

- w: output list (an already-evaluated prefix of f)
- f: the front list, as a suspension of a strict list
- r: the rear list
- $fl = \|f\|$ et $rl = \|r\|$.

Invariants: $fl \geq rl$, and $w \neq [\,]$ unless the queue is empty.

The operations of the physicist's queue

```
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)
```

Similar to the banker's queue, except that `tail` extractions take place in parallel on the prefix `w` and on the suspended front list `f`.

Normalizations for the physicist's queue

First normalization: ensures that the prefix `w` is not empty if the queue is not empty.

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

Second normalization: maintains the $fl \geq rl$ invariant and makes sure the rear list is reversed "long enough in advance".

```
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, [])
```

Amortized analysis

Define the anti-potential by $\Psi(q) \stackrel{def}{=} \min(2\|w\|, \|f\| - \|r\|)$ so that $\Psi = 0$ in the two cases where we need to force a suspension.

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

If w is empty, $\Psi(q) = 0$, we can force `f`, the strict cost is 1 (`Lazy.force` is free), and $\Delta\Psi \geq 0$ $\rightarrow$ amortized cost 1.

Amortized analysis

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

```
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, [])
```

If $\|f\| < \|r\|$, we have $\|f\| = m$ and $\|r\| = m + 1$ for some m.

$\Psi(q) = 0$, therefore we can force `f`, and the strict cost is 1 plus that of evaluating `f' @ List.rev r`, namely $2m + 1$.

The anti-potential goes from 0 to $\min(2m, 2m + 1) = 2m$.

The amortized cost is, therefore, $(1 + 2m + 1) - (2m - 0) = 2$.

Amortized analysis

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

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

$\|r\|$ increases by 1, hence Ψ decreases by 1 or remains unchanged.
The strict cost is constant $\rightarrow$ constant amortized cost.

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

$\|w\|$ decreases by 1 and $\|f\|$ by 1 too, hence Ψ decreases by 1 or 2.
The strict cost is constant $\rightarrow$ constant amortized cost.

Eliminating amortization

“Real-time” applications

Some applications require bounds on the time taken by each operation:

- Hard real-time systems: control & command, robotics, …
- Soft real-time systems: audio, video, games, GUIs, …

Example: a GUI where

100 operations take 20ms each

is more pleasant to use than a GUI where

99 operations take 1ms each and 1 operation takes 1s

even though the latter completes the 100 operations faster than the former.

Eliminating amortization via scheduling

A general technique to transform an efficient amortized structure into an efficient real-time structure: scheduling.

Idea: instead of performing an expensive operation $\mathcal{O}(n)$ after n cheap operations $\mathcal{O}(1)$, we will

- incrementalize the expensive operation so that it can be done in n steps taking $\mathcal{O}(1)$ time each;
- schedule a step of the expensive operation during each of the n cheap operations.

Then, all operations take $\mathcal{O}(1)$ time in the worst-case, not just amortized $\mathcal{O}(1)$ time.

Incrementalize reverse-then-concatenate

How can we compute $\texttt{app}\ f\ (\texttt{rev}\ r)$ incrementally?

Let's generalize: how can we compute incrementally

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

The following equalities hold:

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

They show that the computation can be made incremental using a lazy list of results: each element is produced in $\mathcal{O}(1)$ time.

Scheduling the evaluation of a lazy list

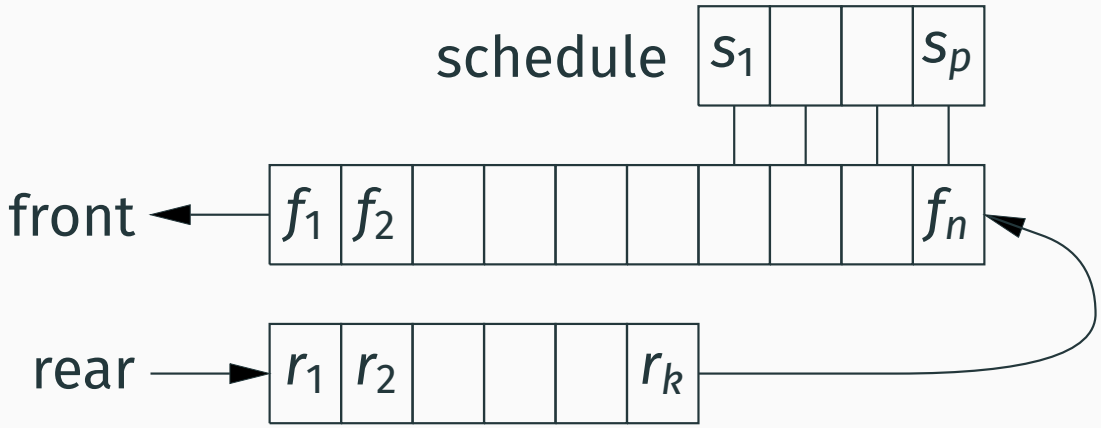
The amortized analysis for the banker's queue shows that it's not enough to let `tail` trigger the evaluations of the elements of `app` f (`rev` r). Every `add` as well as every `tail` must compute one element.

It's easy to schedule evaluations of the elements of a lazy list. Just consider pairs (f, s) where s (the schedule) is f initially, then later a suffix of f.

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

Every call to `exec` forces the computation of one list element, without changing the value of the list f.

Okasaki's real-time queue (1995)




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

A queue is a triple (f, r, s):

- the front list f: a lazy list;
- the rear list r: a strict list;
- the schedule s: a strict list, always a suffix of f;
- $\|f\| = \|r\| + \|s\|$, implying $\|f\| \geq \|r\|$.

Scheduling the rotations

If the schedule is empty, we set up a new rotation of f, r.

Otherwise, we compute one more element of 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')
```

All the suspensions built by `rotate` evaluate in constant time.

Hence, `exec` takes constant time.

The operations of the real-time queue

```
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)
```

All 3 operations run in constant time!

(It is possible to implement a purely functional, real-time queue without using lazy evaluation: the Hood-Melville queue. See T. Nipkow's seminar.)

Summary

On amortization

An approach that leads to data structures that are quite simple but remarkably efficient.

We saw this on the example of queues, but it's also true for structures based on balanced trees:

	$\mathcal{O}(\log n)$ worst-case	$\mathcal{O}(\log n)$ amortized
BST	AVL, red-black	splay trees
Heap	leftist heaps	skew heaps

For the amortized structures (right column), the trees carry no rebalancing information (no heights, no colors, etc). Every operation just tries to reduce imbalance locally.

On lazy evaluation

A very useful mechanism:

- to delay expensive computations until the cost is amortized by enough operations;
- to share the results of the computations, thus avoiding duplication in case of persistent, non-single-threaded use;
- to schedule a sequence of computation steps.

A conceptual question: are these lazy implementation still purely functional, or somewhat imperative?

An observation: reasoning (functional correctness & complexity) is much easier over lazy implementations than over imperative implementations.

References

References

The main support for this lecture:

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

The seminal paper:

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

A verification of the banker’s queue in separation logic:

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