Persistent data structures, fifth lecture

Numerical representations and non-regular data types

Xavier Leroy

2023-04-07

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

Numerical representations

Data structures and numerical representations

To better understand or to design a data structure, it can be helpful to reduce it to a number.

Typically: a collection $\rightarrow$ the number of elements.

Operations on the structure correspond to arithmetic operations:

insertion	$\rightarrow$	increment
deletion	$\rightarrow$	decrement
merge (disjoint union)	$\rightarrow$	addition

The concrete representation of the data structure corresponds to a particular way to write the number, for instance:

singly-linked list $\rightarrow$ Peano numbers

Lists and Peano numbers

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

```
type num =
  | Zero
  | Succ of num
```

Constant-time operations:

- cons ($\ell \to$ `Cons`(x, ℓ))
- tail (`Cons`$(x, \ell) \to \ell$)
- increment ($n \to$ `Succ` n)
- decrement (`Succ` $n \to n$)

Linear-time operations:

- concatenation (ℓ_1 `@` ℓ_2)
- n-th element (`List.nth` ℓ n)
- addition ($n_1 + n_2$)
- comparison ($> n$)

Binary numbers

Number	Representation	Number	Representation
0:		8:	**0001**
1:	**1**	9:	**1001**
2:	**01**	10:	**0101**
3:	**11**	11:	**1101**
4:	**001**	12:	**0011**
5:	**101**	13:	**1011**
6:	**011**	14:	**0111**
7:	**111**	15:	**1111**

Little-endian representation (least significant bit first):
a list of digits $d_0, d_1, \ldots, d_{p-1}$ with $d_i \in \{\mathbf{0}, \mathbf{1}\}$.

This list denotes the integer number $\sum_{i=0}^{p-1} d_i \cdot 2^i$.

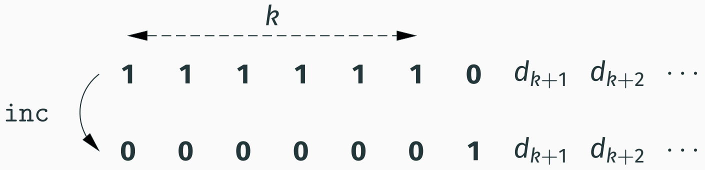
Representation and basic operations

```
type digit = Zero | One
type num = digit list

let rec inc = function
  | [] -> [One]
  | Zero :: n -> One :: n
  | One :: n -> Zero :: inc n

let rec dec = function
  | [] -> raise Error
  | [One] -> []
  | One :: n -> Zero :: n
  | Zero :: n -> One :: dec n
```

Algorithmic complexity of increment

```
let rec inc = function
  | [] -> [One]
  | Zero :: n -> One :: n
  | One :: n -> Zero :: inc n
```

`inc` takes time proportional to $k + 1$,
where k is the number of **1** that precede the first **0**:



If n is the number denoted by the list, we have $n \geq 2^k - 1$.

Therefore, `inc` runs in worst-case time $\mathcal{O}(\log n)$.

Amortized analysis of increment

```
let rec inc = function
  | [] -> [One]
  | Zero :: n -> One :: n
  | One :: n -> Zero :: inc n
```

We say that a digit is dangerous if it can trigger a carry that needs to be propagated, and not dangerous if there is never a carry.

For `inc`, **1** is dangerous, **0** is not dangerous.

Take $\Phi(n) =$ number of dangerous digits in the list n.

If k is the number of **1** preceding the first **0**,

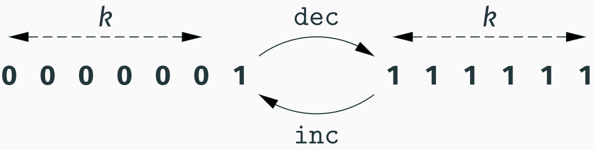
- `inc` takes actual time $k + 1$
- $\Delta\Phi = 1 - k$ (since one **1** appears and k **1** become **0**)

Therefore, `inc` runs in constant amortized time.

Amortized analysis of increment and decrement

A similar analysis shows that `dec` runs in constant amortized time. (Taking **0** as the dangerous digit.)

Yet, a sequence of n `inc` and `dec` can take time $n \log n$ ...



We perform $n = 2^k$ `inc` operations, going from 0 to 2^k, then n sequences `dec; inc`, each taking time $2k$

$\rightarrow 3n$ operations in time $2n \log n$.

Why is this possible? We used different potentials Φ to analyze `inc` and `dec`!

A number system

To each position i, we associate

a weight $w_i \in \mathbb{N}+$;

a set of allowed digits $D_i \subseteq \mathbb{N}$.

The sequence $d_0, d_1, \ldots$ with $d_i \in D_i$ denotes the number $n = \sum_{i=0} d_i w_i$.

Examples of number systems:

- Binary (base 2) numbers: $D_i = \{\mathbf{0}, \mathbf{1}\}$ and $w_i = 2^i$.
- Decimal (base 10) numbers: $D_i = \{\mathbf{0}, \ldots, \mathbf{9}\}$ and $w_i = 10^i$.
- Days, hours, minutes, seconds:
  $D_0 = D_1 = \{\mathbf{0}, \ldots, \mathbf{59}\}$, $D_2 = \{\mathbf{0}, \ldots, \mathbf{23}\}$, $D_3 = \mathbb{N}$
  $w_0 = 1$, $w_1 = 60$, $w_2 = 60 \times 60$, $w_3 = 60 \times 60 \times 24$.
- Redundant binary numbers: $D_i = \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$ and $w_i = 2^i$.

Redundant binary numbers

Using three digits **0**, **1** and **2**.

A given number can have multiple representations.

0:	
1:	**1**
2:	**01, 2**
3:	**11**
4:	**001, 02, 21**
5:	**101, 12**
6:	**011, 201, 22**
7:	**111**
8:	**0001, 002, 021, 211**
9:	**1001, 102, 121**
10:	**0101, 012, 2001, 202, 221**
11:	**1101, 112**
12:	**0011, 0201, 022, 2101, 212**
13:	**1011, 1201, 122**
14:	**0111, 2011, 2201, 222**
15:	**1111**
16:	**00001, 0002, 0021, 0211, 2111**
17:	**10001, 1002, 1021, 1211**

Increment and decrement over the redundant representation

```
let rec inc = function
  | [] -> [One]
  | Zero :: n -> One :: n
  | One :: n -> Two :: n
  | Two :: n -> One :: inc n
```

The last case is justified by $(2 + 2n) + 1 = 1 + 2(n + 1)$.

```
let rec dec = function
  | [] -> raise Error
  | [One] -> []
  | Two :: n -> One :: n
  | One :: n -> Zero :: n
  | Zero :: n -> One :: dec n
```

The last case is justified by $(0 + 2n) - 1 = 1 + 2(n - 1)$.

Increment and decrement over the redundant representation

Decrement is not the inverse of increment!

Number	Increments ↓	Decrements ↑
1	**1**	**1**
2	**2**	**01**
3	**11**	**11**
4	**21**	**001**
5	**12**	**101**
6	**22**	**011**
7	**111**	**111**
8	**211**	**0001**
9	**121**	**1001**
10	**221**	**0101**
11	**112**	**1101**
12	**212**	**0011**
13	**122**	**1011**
14	**222**	**0111**
15	**1111**	**1111**

Amortized analysis

```
let rec inc = function ... | Two :: n -> One :: inc n
let rec dec = function ... | Zero :: n -> One :: dec n
```

We classify **0** and **2** as dangerous digits. Only **1** is not dangerous.

Take $\Phi(n) =$ number of dangerous digits in the list n.

Each time `inc` or `dec` calls itself recursively,
Φ decreases by 1 (a **2** or a **0** becomes a **1**).

Therefore, `inc` and `dec` run in constant amortized time, even if we interleave calls to `inc` and `dec`.

Amortization and persistence

As in the 3rd lecture, this amortized complexity extends to persistent uses of numbers, provided we use lazy lists (streams) of digits instead of lists of digits.

```
type digit = Zero | One | Two
type num = digit stream
let rec inc n =
  lazy (match Lazy.force n with
        | Nil -> Cons(One, lazy Nil)
        | Cons(Zero, n) -> Cons(One, n)
        | Cons(One, n) -> Cons(Two, n)
        | Cons(Two, n) -> Cons(One, inc n))
```

To show the $\mathcal{O}(1)$ amortized time bound, we use the 2.0 banker's method, putting two time debits on each **1** digit and one debit on **0** and **2**.

A problem with the zero digit

We can have arbitrarily-many zero digits at the end of a number: $\mathbf{1} = \mathbf{10} = \mathbf{1000000000000000000000}$.

This does not change the complexity of `inc` and `dec`, but makes comparison against zero arbitrarily slow.

```
let rec iszero = function
  | [] -> true
  | One :: _ -> false
  | Zero :: n -> iszero n
```

The time taken by `iszero n` is not bounded by a function of the number denoted by the list…

Solution 1: ensure that a list of digits never ends in **0**. (Complicates the computations a bit.)

Solution 2: represent numbers without using zero digits!

Zero-less binary representation

For example, using the digits **1**, **2**, **3**.

0		9	**121, 311, 33**
1	**1**	10	**221**
2	**2**	11	**112, 131, 321**
3	**11, 3**	12	**212, 231**
4	**21**	13	**122, 312, 331**
5	**12, 31**	14	**222**
6	**22**	15	**1111, 113, 132, 322**
7	**111, 13, 32**	16	**2111, 213, 232**
8	**211, 23**	17	**1211, 123, 3111, 313, 332**

Zero-less operations

```
type digit = One | Two | Three
type num = digit list

let iszero = function [] -> true | _ -> false

let rec inc = function
  | [] -> [One]
  | One :: n -> Two :: n
  | Two :: n -> Three :: n
  | Three :: n -> Two :: inc n  (* (3 + 2n) + 1 = 2 + 2(n+1)  *)

let rec dec = function
  | [] -> raise Error
  | [One] -> []
  | Three :: n -> Two :: n
  | Two :: n -> One :: n
  | One :: n -> Two :: dec n    (* (1 + 2n) - 1 = 2 + 2(n-1)  *)
```

Sparse representation

Instead of the dense positional representation

number = list of digits

we can use a sparse representation

number = list of (nonzero digit, weight) pairs

(in strictly increasing order of weights)

or, if the only digits are **0** and **1**,

number = list of weights (strictly increasing)

Example: 13 is **1, 4, 8** in sparse repr. and **1011** in dense repr.

Increment and decrement in sparse binary representation

```
type num = int list (* powers of 2, in strictly increasing order *)

let iszero = function [] -> true | _ -> false

let rec carry c n =
  match n with
  | [] -> [c]
  | w :: n’ -> if c < w then c :: n else carry (2 * c) n’

let rec borrow c n =
  match n with
  | [] -> raise Error
  | w :: n’ -> if c = w then n’ else c :: borrow (2 * c) n

let inc n = carry 1 n
let dec n = borrow 1 n
```

Data structures inspired by number systems

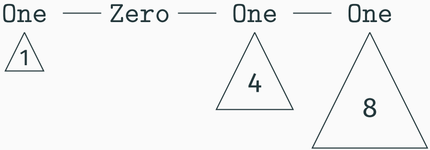
From a number system to a data structure

General idea:

A structure = a list of digits

A digit d with rank i = d subtrees of w_i elements each.

Example: in binary ($w_i = 2^i$), using digits **0** and **1**, a 13-element structure will have the following shape.



Which trees correspond to weights?

For a binary representation, we need trees of size 2^i.

To “propagate carries” during insertion ($\approx$ increment), we need a simple way to combine two trees of size 2^i into a tree of size 2^{i+1}.

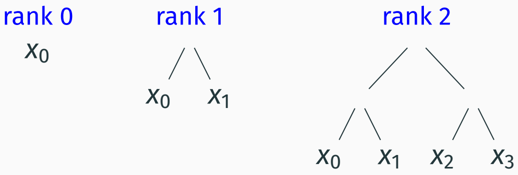
Two examples used in the following:

- Perfect binary trees with values at leaves (used for random-access lists).
- Binomial trees (used for priority queues).

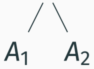
Perfect binary trees (PBT) with values at leaves

PBT of rank $0 =$ a single value x.

PBT of rank $i+1 =$ two PBTs of rank i, joined by a node.

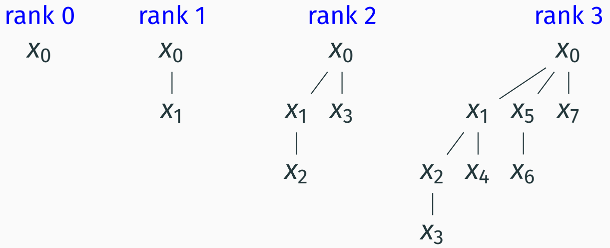


A good match for implementing indexed sequences:
accessing the j-th value x_j takes time $i = \log n$ (binary search).

To combine A_1 and A_2 of rank i, just form  with rank $i+1$.

Binomial tree of rank i =

a value x and i binomial trees of ranks $i-1, \dots, 1, 0$.

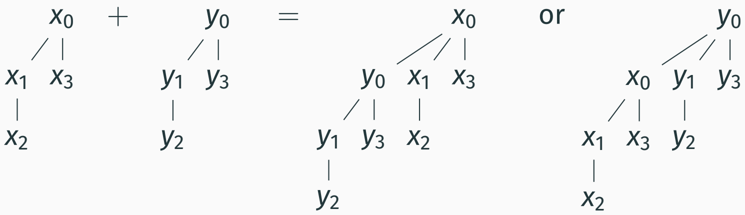


A binomial tree of rank i has 2^i elements.

It has $\binom{i}{d}$ elements at depth d.

Binomial trees

To combine two binomial trees of rank *i*,
add one of them as the first subtree of the other.



A good match for implementing heaps
(for each subtree, the smallest element is at the root).

Random-access list

The operations of a singly-linked list:

`cons`, `head`, `tail`, `isempty`

plus direct (“random”) access to the *i*-th element of the list:

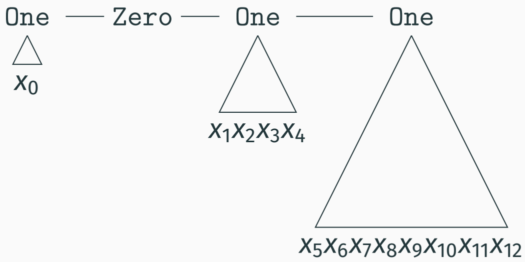
`get` $i\ \ell$, `set` $i\ v\ \ell$

Complexity objective: $\mathcal{O}(1)$ for `head`, $\mathcal{O}(1)$ amortized for `tail` and `cons`, $\mathcal{O}(\log n)$ for `get` and `set`.

A random-access list patterned after binary numbers

The representation is structured like binary numbers, using **0** and **1** as digits, and perfect binary trees with values at leaves as weights.

Example: the 13-element list $[x_0, \ldots, x_{12}]$.



Remark: for n elements, we have $\mathcal{O}(\log n)$ trees.

Insertion in the list: the `cons` operation

```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
type 'a digit = Zero | One of 'a tree
type 'a seq = 'a digit list

let rec cons_tree t r =
  match r with
  | [] -> [One t]
  | Zero :: r -> One t :: r
  | One t' :: r -> Zero :: cons_tree (Node(t, t')) r

let cons x r = cons_tree (Leaf x) r
```

`cons` follows the same pattern as incrementing a binary number.

The `head` and `tail` operations

```
let rec uncons_tree = function
  | [] -> raise Empty
  | [One t] -> (t, [])
  | One t :: r -> (t, Zero :: r)
  | Zero :: r ->
      let (Node(t1, t2), r’) = uncons_tree r in
      (t1, One t2 :: r’)

let head r =
  let (Leaf x, _) = uncons_tree r in x

let tail r =
  let (_, r’) = uncons_tree r in r’
```

`uncons_tree` follows the same pattern as decrementing a binary number, but returns the first tree as an extra result.

Random access: the `get` operation

```
let rec get_tree i t w =
  match t with
  | Leaf x -> assert (i = 0 && w = 1); x
  | Node(t1, t2) ->
      let w = w / 2 in
      if i < w then get_tree i t1 w else get_tree (i - w) t2 w

let rec get_rec i r w =
  match r with
  | [] -> raise Out_of_bounds
  | Zero :: r' -> get_rec i r' (w * 2)
  | One t :: r' ->
      if i < w then get_tree i t w
               else get_rec (i - w) r' (w * 2)

let get i r = get_rec i r 1
```

Complexity analysis

Same analysis as for binary numbers:

Operation	Digits **0, 1**	
`head`	$\mathcal{O}(\log n)$ ✘	
`cons`, `tail`	$\mathcal{O}(\log n)$ ✘ (*)	
`get`, `set`	$\mathcal{O}(\log n)$ ✔	

(*) A sequence of n `cons` takes time $\mathcal{O}(n)$, as well as a sequence of n `tail`, but not a sequence of n `cons`-then-`tail`.

Complexity analysis

Same analysis as for binary numbers:

Operation	Digits **0, 1**	Digits **1, 2, 3**
`head`	$\mathcal{O}(\log n)$ ✘	$\mathcal{O}(1)$ ✔
`cons`, `tail`	$\mathcal{O}(\log n)$ ✘ (*)	$\mathcal{O}(1)$ amortized ✔
`get`, `set`	$\mathcal{O}(\log n)$ ✔	$\mathcal{O}(\log n)$ ✔

(*) A sequence of n `cons` takes time $\mathcal{O}(n)$, as well as a sequence of n `tail`, but not a sequence of n `cons`-then-`tail`.

We switch to a representation using three digits **1, 2, 3**:

- zero-less representation $\rightarrow$ `head` in $\mathcal{O}(1)$ worst-case;
- redundant representation $\rightarrow$ `cons`, `tail` in $\mathcal{O}(1)$ amortized.

Redundant and zero-less: the `cons` operation

```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
type 'a digit =
  | One of 'a tree
  | Two of 'a tree * 'a tree
  | Three of 'a tree * 'a tree * 'a tree
type 'a seq = 'a digit list

let rec cons_tree t r =
 match r with
  | [] -> [One t]
  | One t1 :: r -> Two(t, t1) :: r
  | Two(t1, t2) :: r -> Three(t, t1, t2) :: r
  | Three(t1, t2, t3) :: r ->
     Two(t, t1) :: cons_tree (Node(t2, t3)) r

let cons x r = cons_tree (Leaf x) r
```

Redundant and zero-less: the `head` and `tail` operations

```
let head = function
  | [] -> raise Empty
  | One(Leaf x) :: _ -> x
  | Two(Leaf x, _) :: _ -> x
  | Three(Leaf x, _, _) -> x
  | _ -> assert false

let rec uncons_tree = function
  | [] -> raise Empty
  | [One t] -> (t, [])
  | Three(t1, t2, t3) :: r -> (t1, Two(t2, t3) :: r)
  | Two(t1, t2) :: r -> (t1, One t2 :: r)
  | One t :: r ->
     let (Node(t1, t2), r') = uncons_tree r in
     (t, Two(t1, t2) :: r')

let tail r =
  let (_, r') = uncons_tree r in r'
```

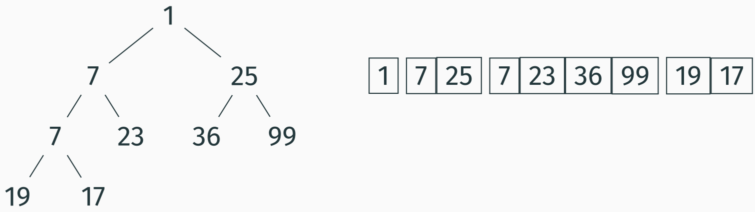
Priority queues

A multiset of elements, with operations

- `insert` x h : add element x
- `find_min` h : return the smallest element of h
  (more generally: the element with highest priority)
- `remove_min` h : remove the smallest element of h
- `merge` h_1 h_2 : return the union of h_1 and h_2.

Applications: scheduling; graph algorithms (shortest paths); sorting (the famous *heapsort* algorithm).

Heaps



A tree carrying values at nodes.

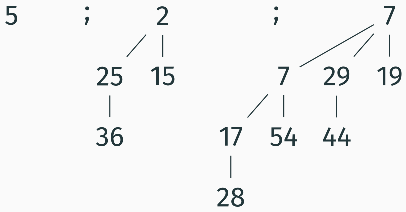
Values increase along every branch.

Consequently, the smallest value is always at the root.

Binomial heaps (J. Vuillemin, 1978)

A sparse binary representation of the number of elements in the priority queue, using binomial trees of rank i for weights 2^i.

Example: a priority queue containing 13 elements.



The list is ordered by strictly increasing ranks of binomial trees.

Each tree is ordered like a heap.

An implementation of binomial trees

```
type 'a tree = { rank: int; value: 'a; children: 'a tree list }

let link t1 t2 =
  assert (t1.rank = t2.rank);
  if t1.value <= t2.value then
    { t1 with rank = t1.rank + 1; children = t2 :: t1.children }
  else
    { t2 with rank = t2.rank + 1; children = t1 :: t2.children }
```

Combining two trees (using the `link` function) preserves the heap invariant.

Insertion

```
type 'a heap = 'a tree list

let rec insert_tree t h =
  match h with
  | [] -> [t]
  | t' :: h' ->
      if t.rank < t'.rank
      then t :: h
      else insert_tree (link t t') h'

let insert x h =
  insert_tree { rank = 0; value = x; children = [] } h
```

Same pattern as incrementing a sparse binary number.

Merging two binomial heaps

```
let rec merge h1 h2 =
  match h1, h2 with
  | [], _ -> h2
  | _, [] -> h1
  | t1 :: h1', t2 :: h2' ->
      if t1.rank < t2.rank then t1 :: merge h1' h2
      else if t2.rank < t1.rank then t2 :: merge h1 h2'
      else insert_tree (link t1 t2) (merge h1' h2')
```

Same pattern as adding two sparse binary numbers.

Extracting the smallest element

```
let rec extract_min = function
  | [] -> raise Empty
  | [t] -> (t, [])
  | t :: h ->
      let (t’, h’) = extract_min h in
      if t.value <= t’.value then (t, h) else (t’, t :: h’)

let find_min h =
  let (t, _) = extract_min h in t.value

let remove_min h =
  let (t, h’) = extract_min h in
  merge (List.rev t.children) h’
```

If `t` is a well-formed binomial tree,

`List.rev t.children` is a well-formed binomial heap!

Complexity analysis

For a n-element heap, its representation is a list of at most $\log n$ binomial trees

$\rightarrow$ all operations run in worst-case time $\mathcal{O}(\log n)$.

The `insert` operation runs in $\mathcal{O}(1)$ amortized time, like increment of a binary number.

(Potential Φ = length of the list = number of **1** bits in the binary representation of n.)

Note: we cannot have `insert`, `find_min` and `remove_min` in $\mathcal{O}(1)$ amortized time. Otherwise, we could sort in linear time!

Non-regular data types

Algebraic types: regular or not

An algebraic type with one or several type parameter is regular if all recursive occurrences of the type use the same type parameters.

```
type 'a list = Nil | Cons of 'a * 'a list
```

Algebraic types: regular or not

An algebraic type with one or several type parameter is regular if all recursive occurrences of the type use the same type parameters.

```
type 'a list = Nil | Cons of 'a * 'a list
```

It is non regular or nested if recursive occurrences use "bigger" type parameters, for example `'a * 'a` instead of `'a`.

```
type 'a nest = Nil | Cons of 'a * ('a * 'a) nest
```

Example of a value of type `int nest`:

```
Cons(1, Cons((2,3), Cons(((4,5),(6,7)), Nil))).
```

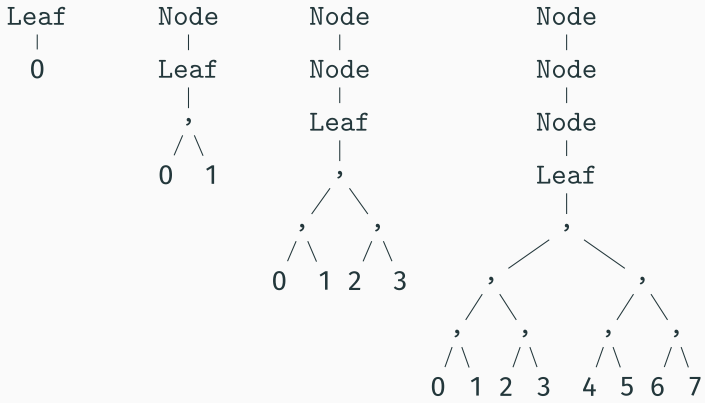
A non-regular type: perfect binary trees with values at leaves

```
type 'a ptree = Leaf of 'a | Node of ('a * 'a) ptree
```

A non-regular type: perfect binary trees with values at leaves

```
type 'a ptree = Leaf of 'a | Node of ('a * 'a) ptree
```

Some values of type `int ptree`:



Operations on perfect binary trees

```
let rec size : 'a. 'a ptree -> int = function
  | Leaf x -> 1
  | Node t -> 2 * size t

let rec leftmost : 'a. 'a ptree -> 'a = function
  | Leaf x -> x
  | Node t -> fst (leftmost t)

let rec rightmost : 'a. 'a ptree -> 'a = function
  | Leaf x -> x
  | Node t -> snd (rightmost t)
```

Note: we must annotate functions with their polymorphic types ($\forall\alpha, \alpha\ \texttt{ptree} \to \ldots$) because this is polymorphic recursion, for which type inference is undecidable in general.

Comparison with the usual, regular type of binary trees

```
type 'a tree =
  | Leaf of 'a
  | Node of 'a tree * 'a tree

let rec size = function
  | Leaf x -> 1
  | Node(t1, t2) ->
      size t1 + size t2

let rec leftmost = function
  | Leaf x -> x
  | Node(t1, t2) -> leftmost t1
```

```
type 'a ptree =
  | Leaf of 'a
  | Node of ('a * 'a) ptree

let rec size : ... = function
  | Leaf x -> 1
  | Node t -> 2 * size t

let rec leftmost : ... = function
  | Leaf x -> x
  | Node t -> fst (leftmost t)
```

A random-access list

Instead of a regular list of digits, each digit being a perfect binary tree, let's use a nest-like list with non-regular recursion (`'a` becomes `'a * 'a`).

```
type 'a digit = Zero | One of 'a
type 'a seq = Nil | Cons of 'a digit * ('a * 'a) seq
```

Examples of sequences with 1 to 6 elements: (`::` is infix `Cons`)

```
One 1 :: Nil
Zero  :: One(2,1) :: Nil
One 3 :: One(2,1) :: Nil
Zero  :: Zero     :: One((4,3),(2,1)) :: Nil
One 5 :: Zero     :: One((4,3),(2,1)) :: Nil
Zero  :: One(6,5) :: One((4,3),(2,1)) :: Nil
```

The cons and uncons operations

```
let rec cons : 'a. 'a -> 'a seq -> 'a seq = fun x s ->
  match s with
  | Nil -> Cons(One x, Nil)
  | Cons(Zero, s) -> Cons(One x, s)
  | Cons(One y, s) -> Cons(Zero, cons (x, y) s)

let rec uncons : 'a. 'a seq -> 'a * 'a seq = function
  | Nil -> raise Empty
  | Cons(One x, s) -> (x, Cons(Zero, s))
  | Cons(Zero, s) ->
      let ((x, y), t) = uncons s in (x, Cons(One y, t))
```

Random access: for reading

```
let rec get : 'a. int -> 'a seq -> 'a = fun i s ->
  match s with
  | Nil -> raise Out_of_bounds
  | Cons(Zero, s) -> get2 i s
  | Cons(One x, s) -> if i = 0 then x else get2 (i - 1) s

and get2 : 'a. int -> ('a * 'a) seq -> 'a = fun i s ->
  let (x0, x1) = get (i / 2) s in
  if i mod 2 = 0 then x0 else x1
```

Random access: for writing and modification

To "cross the recursion", we need to generalize writing at index *i* to modification of the value at *i* by any function `f: 'a -> 'a`.

```
let rec update : 'a. int -> ('a -> 'a) -> 'a seq -> 'a seq
= fun i f s ->
  match s with
  | Nil -> raise Out_of_bounds
  | Cons(Zero, s) -> Cons(Zero, update2 i f s)
  | Cons(One x, s) ->
      if i = 0 then Cons(One(f x), s)
               else Cons(One x, update2 (i - 1) f s)
and update2 : 'a. int -> ('a -> 'a) -> ('a * 'a) seq -> ('a * 'a) seq
= fun i f s2 ->
  let f2 (x0, x1) = if i mod 2 = 0 then (f x0, x1) else (x0, f x1) in
  update (i / 2) f2 s2

let set : 'a. int -> 'a -> 'a seq -> 'a seq = fun i v s ->
  update i (fun _ -> v) s
```

Finger trees

Finger trees (Hinze et Paterson, 2006)

A purely-functional data structure for sequences of elements, with many efficient operations:

- Lookup, insertion, deletion at both ends in amortized time $\mathcal{O}(1)$, worst-case time $\mathcal{O}(\log n)$. (dequeue)
- Concatenation of two sequences in time $\mathcal{O}(\log n)$. (rope)
- After annotation with a monoid (see next lecture):
  direct access to the i-th element in time $\mathcal{O}(\log n)$; (functional array)
  direct access to the smallest in time $\mathcal{O}(\log n)$. (priority queue)

Finger trees combine the two techniques described in this lecture: numerical representations and non-regular data types.

Step-by-step construction of finger trees (Claessen, 2020)

Think of a list-like structure with direct access to the first and to the last element:

```
type 'a seq =
  | Nil
  | Unit of 'a
  | More of 'a * 'a seq * 'a
```

Operations `head` and `last` take constant time, but `cons` and `add` take linear time:

```
let rec cons x = function
  | Nil -> Unit x
  | Unit y -> More(x, Nil, y)
  | More(y, s, z) -> More(x, cons y s, z)
```

Using a non-regular data type

Sub-sequences (`s` in `More(x, s, y)`) would be much shorter if they contained `'a * 'a` pairs instead of mere `'a` elements.

```
type 'a seq =
  | Nil
  | Unit of 'a
  | More of 'a * ('a * 'a) seq * 'a
```

Problem: we're unable to represent a sequence of length 3...

More generally, representable sequences have lengths $L = \{0, 1\} \cup \{2 + 2\ell \mid \ell \in L\} = \{0, 1, 2, 4, 6, 10, 14, 22, \ldots\}$.

Using digits

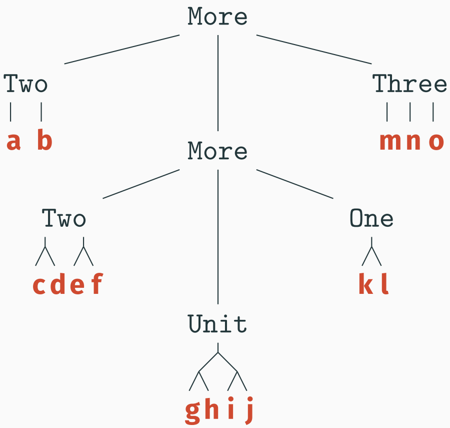
To be able to represent all lengths, let's put a digit (= a small number of elements) on both sides of the sub-sequence.

```
type 'a digit =
  | One of 'a | Two of 'a * 'a | Three of 'a * 'a * 'a

type 'a seq =
  | Nil
  | Unit of 'a
  | More of 'a digit * ('a * 'a) seq * 'a digit
```

We recognize a binary number system, zero-less and with redundant digits.

$\rightarrow$ Operations similar to increment and decrement (`cons`, `tail`, `add`, `take`) will run in $\mathcal{O}(1)$ amortized time.

An example of a finger tree



Each of the left and right fringes looks like a random-access list.

The `cons` operation

```
let rec cons : 'a. 'a -> 'a seq -> 'a seq = fun x t ->
  match t with
  | Nil -> Unit x
  | Unit y -> More(One x, Nil, One y)
  | More(One y, s, z) -> More(Two(x, y), s, z)
  | More(Two(y1, y2), s, z) -> More(Three(x, y1, y2), s, z)
  | More(Three(y1, y2, y3), s, z) ->
      More(Two(x, y1), cons (y2, y3) s, z)
```

Exercise: define `add` (insertion at end of sequence), in a completely symmetric manner.

The head and tail operations

```
let rec uncons : 'a. 'a seq -> 'a * 'a seq = fun t ->
  match t with
  | Nil -> raise Empty
  | Unit y -> (y, Nil)
  | More(Three(y1, y2, y3), s, z) -> (y1, More(Two(y2, y3), s, z))
  | More(Two(y1, y2), s, z) -> (y1, More(One y2, s, z))
  | More(One y, Nil, One z) -> (y, Unit z)
  | More(One y, Nil, Two(z1, z2)) -> (y, More(One z1, Nil, One z2))
  | More(One y, Nil, Three(z1, z2, z3)) ->
      (y, More(One z1, Nil, Two(z2, z3)))
  | More(One y, s, z) ->
      let ((y1, y2), s') = uncons s in (y, More(Two(y1, y2), s', z))

let head s = fst (uncons s)
let tail s = snd (uncons s)
```

Concatenating two sequences

The base cases are easy:

$$\texttt{concat}\,\texttt{Nil}\,s = s \qquad \texttt{concat}\,s\,\texttt{Nil} = s$$

$$\texttt{concat}\,(\texttt{Unit}\ \ x)\,s = \texttt{cons}\,x\,s \qquad \texttt{concat}\,s\,(\texttt{Unit}\ \ x) = \texttt{add}\,x\,s$$

The recursive case is problematic:

$$\texttt{concat}\,(\texttt{More}(x_1, s_1, y_1))\,(\texttt{More}(x_2, s_2, y_2)) = \texttt{More}(x_1, ??, y_2)$$

The sequence written ?? must be the concatenation of s_1, the elements of digit y_1, the elements of digit x_2, and s_2.

Concatenating two sequences

Let's generalize concatenation to a `glue` function

```
glue : 'a seq -> 'a list -> 'a seq -> 'a seq
```

`glue` s_1 ℓ s_2 is a sequence containing the elements of s_1 followed by the (short) list of elements ℓ, followed by the elements of s_2.

Obviously, we have `concat` $s_1\ s_2 =$ `glue` $s_1\ [\,]\ s_2$.

The recursive case for `glue` is of the following shape:

$$\begin{aligned} &\texttt{glue}\ (\texttt{More}(x_1, s_1, y_1))\ \ell\ (\texttt{More}(x_2, s_2, y_2)) \\ &\quad = \ \texttt{More}(x_1, \texttt{glue}\ s_1\ (\texttt{elements}\ y_1\ \texttt{@}\ \ell\ \texttt{@}\ \texttt{elements}\ x_1)\ s_2, y_2) \end{aligned}$$

where `@` is the usual concatenation over lists,
and `elements: 'a digit -> 'a list`.

Gluing two sequences and a list

```
glue : 'a seq -> 'a list -> 'a seq -> 'a seq
```

$$\texttt{glue}\,(\texttt{More}(x_1, s_1, y_1))\,\ell\,(\texttt{More}(x_2, s_2, y_2)) = \texttt{More}(x_1, \texttt{glue}\,s_1\,\ell'\,s_2, y_2)$$

where $\ell' = \texttt{elements}\,y_1\;\texttt{@}\;\ell\;\texttt{@}\;\texttt{elements}\,x_2$.

Type error! ℓ' is a list of elements (type `'a list`) while the recursive call to `glue` expects a list of pairs of elements (type `('a * 'a) list`).

Design error! The length of ℓ' can be odd. In this case, we cannot concatenate it with s_1 and s_2, which are sequences of pairs of elements.

A more flexible non-regular recursion

Let's use sub-sequences that contain not just pairs `'a * 'a` but also triples `'a * 'a * 'a`.

```
type 'a node = Pair of 'a * 'a | Triple of 'a * 'a * 'a
type 'a seq =
  | Nil
  | Unit of 'a
  | More of 'a digit * 'a node seq * 'a digit
```

This is reminiscent of the 2-3 trees from the 2nd lecture: perfect trees with nodes of degree 2 or 3.

The `cons`, `uncons`, `add`, `unadd` operations extend easily (exercice).

Gluing two sequences and a list

$$\texttt{glue}\,(\texttt{More}(x_1, s_1, y_1))\,\ell\,(\texttt{More}(x_2, s_2, y_2)) = \texttt{More}(x_1, \texttt{glue}\,s_1\,\ell'\,s_2, y_2)$$

where $\ell' = \texttt{to\_nodes}\,(\texttt{elements}\,y_1 \texttt{ @ } \ell \texttt{ @ } \texttt{elements}\,x_2)$.

`to_nodes` takes a list of elements of length $\neq 1$ and turns it into a list of `Pair` and `Triple` nodes.

```
let rec to_nodes = function
  | [] -> []
  | [x] -> assert false
  | [x1; x2] -> [Pair(x1, x2)]
  | [x1; x2; x3; x4] -> [Pair(x1, x2); Pair(x3, x4)]
  | x1 :: x2 :: x3 :: xs -> Triple(x1, x2, x3) :: to_nodes xs
```

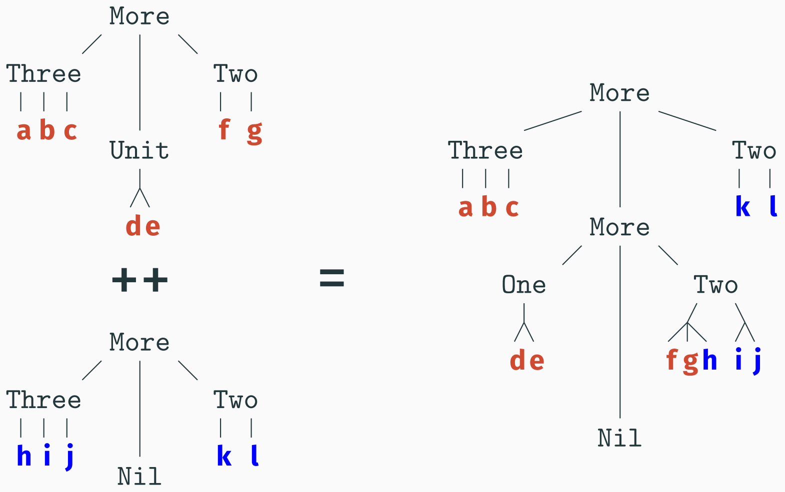
If ℓ has 0 to 3 elements, the argument of `to_nodes` has 2 to 9 elements, and ℓ' has 1 to 3 elements.

The complete code for gluing and concatenation

```
let elements = function
  | One x -> [x]
  | Two(x1, x2) -> [x1; x2]
  | Three(x1, x2, x3) -> [x1; x2; x3]

let rec glue: 'a. 'a seq -> 'a list -> 'a seq -> 'a seq = fun s1 a s2 ->
  match s1, s2 with
  | Nil, _ -> List.fold_right cons a s2
  | _, Nil -> List.fold_left (Fun.flip add) s1 a
  | Unit x1, _ -> List.fold_right cons (x1 :: a) s2
  | _, Unit x2 -> List.fold_left (Fun.flip add) s1 (a @ [x2])
  | More(x1, s1, y1), More(x2, s2, y2) ->
     More(x1, glue s1 (to_nodes (elements y1 @ a @ elements x2)) s2, y2)

let concat s1 s2 = glue s1 [] s2
```

Running time is $\mathcal{O}(\min(\log n_1, \log n_2))$.

An example of concatenation



Summary

Summary

Number systems are "design patterns" for list-like data structures that are efficient because the sizes of list elements increase exponentially.

(This is called *implicit recursive slowdown* in Okasaki's book.)

Using non-regular data types, we can reflect invariants over sizes in the types, and be guided by types while writing the code.

Finger trees are versatile, efficient, and relatively simple, but better performance can be obtained with more complex structures.

(Kaplan & Tarjan 1996, 1999: all operations in $\mathcal{O}(1)$ worst-case; see also Arthur Charguéraud's seminar talk.)

Going further: data structural bootstrapping

(A. Buchsbaum, PhD Princeton, 1995.)

A set of techniques to build efficient data structures from simpler structures, either less efficient or with limited functionalities.

In this lecture, we saw one kind of boostrapping:

- From fixed-size containers (pairs, digits)
  to arbitrary-size containers (sequences).

Okasaki (chap. 10) shows other examples:

- Adding missing operations
  (e.g. queues $\to$ lists with fast concatenation).
- Reducing the complexity of some operations
  (e.g. heap with $\mathcal{O}(\log n)$ merge $\to$ heap with $\mathcal{O}(1)$ merge).

References

References

The main support for this lecture:

- Chris Okasaki, *Purely Functional Data Structures*, chapters 9 and 11.

The original article on finger trees:

- Ralf Hinze et Ross Paterson, *Finger trees: a simple general-purpose data structure*, J. Funct. Program. 16(2), 2006.

A more accessible presentation:

- Koen Claessen, *Finger trees explained anew, and slightly simplified*, Haskell symposium 2020.