Random-access lists, nested data types and numeral systems

Balázs Kőműves

Falkstenen AB

Leipzig, 2016 September 14

Singly linked lists

Lists are the functional programmer's favourite[1] data structure.

- very simple
- *persistent*
- $O(1)$ `cons`
- BUT, $O(k)$ access to the k-th element :(
- $O(n)$ length
- 3 extra words per element (with GHC)
- etc...

[1] maybe debatable :)

Random access lists

We can do better:

- still relatively simple implementation
- average / amortized / worst-case[2] $O(1)$ `cons`
- $O(\log(k))$ access to the k-th element
- $O(\log(n))$ length
- possibly more compact in-memory representation
- etc...

So we can achieve a strictly better list-replacement! (modulo constant factors, of course)

[2] depending on implementation details

Credits

No originality is claimed here.

Credits / History:

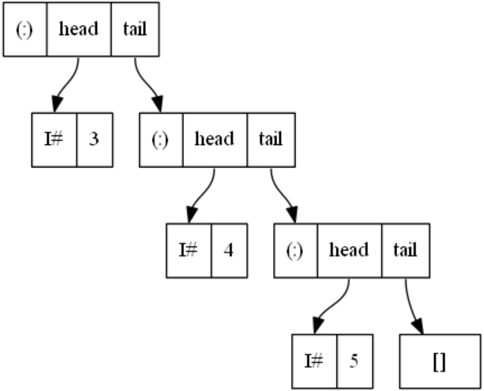
- (Skip lists: William Pugh, 1990)
- Purely Functional Random-Access Lists: Chris Okasaki, 1995
- (Skip trees: Xavier Messeguer, 1997)
- Finger trees: Ralf Hinze and Ross Paterson, 2006
- The nested data type trick I learned from Péter Diviánszky

Implementation:

`http://hackage.haskell.org/package/nested-sequence`

Lists in memory

This is how a list is represented in the computer (using GHC):



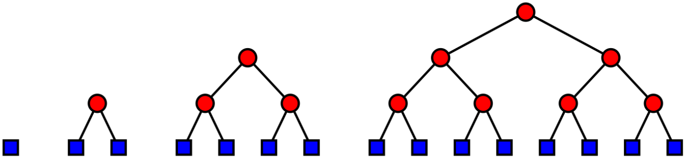
`[3,4,5] :: [Int]`

Leaf binary random-access lists

Consider a list of length 13. Decimal 13 is in binary 1 1 0 1, as $13 = 8 + 4 + 1$. The idea is that will group the elements of the list according to digits of the binary expansion:

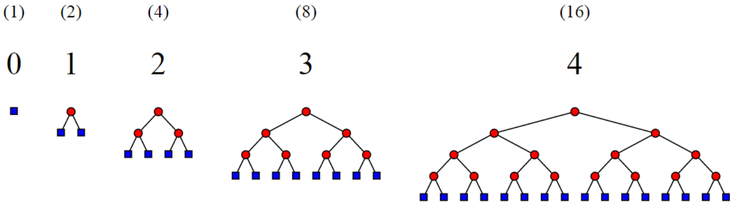
$$[\ \underbrace{a_1}_{1}\ \mid\ \underbrace{\square\ \square}_{(2)}\ \mid\ \underbrace{a_2\ \ a_3\ \ a_4\ \ a_5}_{4}\ \mid\ \underbrace{a_6\ \ a_7\ \ a_8\ \ a_9\ \ a_{10}\ \ a_{11}\ \ a_{12}\ \ a_{13}}_{8}\ ]$$

And then store the corresponding elements in complete binary trees. So the data structure is basically a list of larger and larger binary trees, with data stored on the leaves:



$$[\ a_1 \qquad \square\ \square \qquad a_2\ \ a_3\ \ a_4\ \ a_5 \qquad a_6\ \ a_7\ \ a_8\ \ a_9\ \ a_{10}\ \ a_{11}\ \ a_{12}\ \ a_{13}\ ]$$

Leaf binary random-access lists, II




```
data BinTree a = Leaf a
               | Node (BinTree a) (BinTree a)

type RAL a = [Maybe (BinTree a)]

cons :: a -> RAL a -> RAL a
cons x = go (Leaf x) where
  go s []         = [Just s]
  go s (mb:rest) = case mb of
    Nothing -> Just s   : rest                       -- no carry
    Just t  -> Nothing : go (Node s t) rest          -- carry
```

Dictionary

Set	container
$\mathbb{N}$	sequence type `List a`
increment	`cons`
decrement	`tail`
addition	`append`
linked list	unary number system
random-access list	(skew) binary number system

Classic vs. nested binary trees

The usual binary tree[3] definition in Haskell:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

Issues:

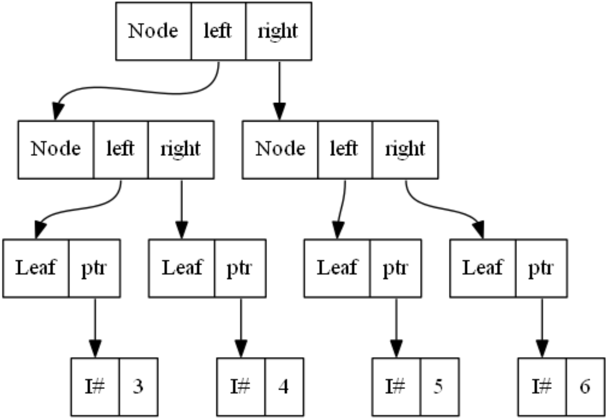
- minor: Cannot guarantee the shape
  (we want *complete* binary trees here)
- major: There is an extra indirection at the leaves.
  This costs *two extra words* per element!
  (that's 16 bytes on a 64-bit machine)

Ugly solution for the latter:

```
data Ugly a = Singleton a
            | Cherry    a a
            | Node (Ugly a) (Ugly a)
```

[3] with data only on the leaves

Naive binary trees



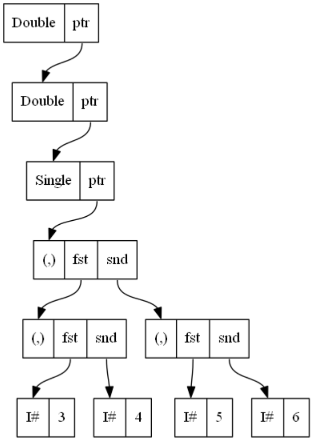
$3 \cdot (2^d - 1) + 2 \cdot 2^d$ words for $n = 2^d$ elements, that is, 5 words per element, even worse than lists!

Nested complete binary trees

We can encode complete binary trees also as a *nested data type*:

```
data Tree' a
  = Single a
  | Double (Tree' (a,a))

example = Double
        $ Double
        $ Single ((3,4),(5,6))
```

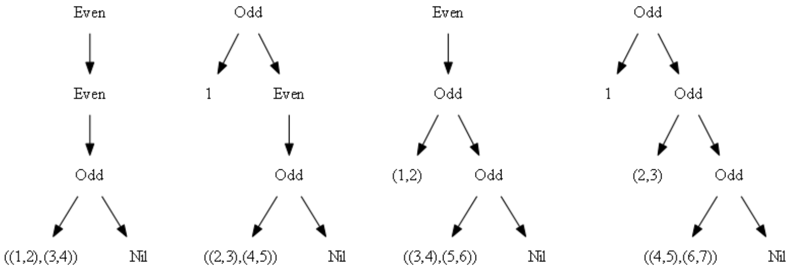


Memory footprint:

$3(n-1) + 2\log(n) + 2$ words

Nested leaf binary random-access lists

```
data Seq a = Nil
           | Even   (Seq (a,a))
           | Odd  a (Seq (a,a))
```



Random access-lists of length 4, 5, 6 and 7

Basic operations

```
data Seq a = Nil
           | Even   (Seq (a,a))
           | Odd  a (Seq (a,a))
```

```
cons :: a -> Seq a -> Seq a
cons x seq = case seq of
  Nil       -> Odd x Nil
  Even   ys -> Odd x ys
  Odd y ys  -> Even $ cons (x,y) ys
```

```
cons :: (a,a) -> Seq (a,a) -> Seq (a,a)
```

```
lookup :: Int -> Seq a -> a
lookup !k seq =
  case seq of
    Even   ys -> cont k ys
    Odd y ys  -> if k==0 then y else cont (k-1) ys
  where
    cont k xs = if even k then x else y where
      (x,y) = lookup (div k 2) xs
```

Running time analysis

Both `cons` and `lookup` are clearly worst-case $O(\log(n))$.

However, in practice they are much better!

Consider the *average* running time of `cons`. Half of the cases the list will have even length $\rightarrow$ we stop after 1 step. Half of the remaining cases will have a length of the form $4n + 1$ $\rightarrow$ we stop after 2 steps. Half of the remaining cases will have a length $8n + 3$...

$$\text{avg. } \texttt{cons} \text{ time} \;=\; \frac{1}{2}\cdot 1 + \frac{1}{4}\cdot 2 + \frac{1}{8}\cdot 3 + \ldots \;<\; \sum_{i=1}^{\infty} \frac{i}{2^i} \;=\; 2$$

`lookup k` should be on average $O(\log(k))$

(What about amortized running time? Tricky to analyse in the lazy purely functional setting, I *think* the same results may be also true for amortized cost...)

Nested leaf n-ary random-access lists

For the n-ary version, we proceed exactly the same way. Consider for example the quaternary ($n = 4$) version:

```
data Seq4 a
  = Nil
  | Zero         (Seq (a,a,a,a))    -- digit 0
  | One    a     (Seq (a,a,a,a))    -- digit 1
  | Two    a a   (Seq (a,a,a,a))    -- digit 2
  | Three a a a (Seq (a,a,a,a))    -- digit 3

cons :: a -> Seq4 a -> Seq4 a
cons x seq = case seq of
  Nil                 -> One   x     Nil
  Zero           rest -> One   x     rest
  One    a       rest -> Two   x a   rest
  Two    a b     rest -> Three x a b rest
  Three a b c rest -> Zero $ cons (x,a,b,c) rest
```

Skew number systems

In the skew n-ary number system, we allow one more digit apart from $0, 1, \ldots, n-1$. We will call this digit n. However, it is allowed to appear at most once, and it must be the first (least significant) non-zero digit.

Example (skew-binary): 1 0 0 1 0 1 1 $\underline{2}$ 0 0 0 0

Incrementation algorithm:

- if there is an n digit, set it to zero and increment the next digit
- otherwise just increment the least significant digit

At most one carry operation! $\rightarrow$ possible to implement in constant time $\rightarrow$
$\rightarrow$ this translates to *worst-case* $O(1)$ `cons`.

Skew n-ary random-access lists

How many skew numbers are with (at most) k digits?

$$f(k) := \text{number of } k\text{-digit skew } n\text{-ary numbers}$$

$$f(k) = n \cdot f(k-1) + 1 = \sum_{i=0}^{k} n^k$$

It follows (convince yourself) that:

$$[\ a_k \ \ a_{k-1} \ \ \ldots \ \ a_1 \ \ a_0\ ] \qquad \longmapsto \qquad \sum_{i=0}^{k} a_k \cdot f(k) \in \mathbb{N}$$

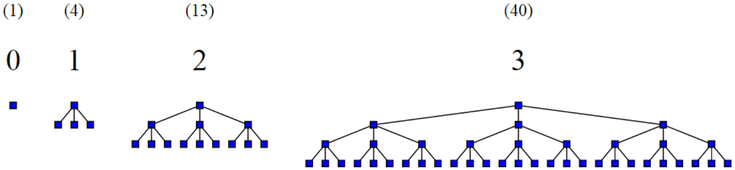
Observation: $f(k)$ equals to the number of "full" (data on both the nodes and the leaves) n-ary trees with depth k!

Thus we will store data on *both* the nodes and the leaves. It's magic!

Skew n-ary random-access lists, II.

Observation: $f(k)$ equals to the number of “full” (data on both the nodes and the leaves) n-ary trees with depth k!

Thus we will store data on *both* the nodes and the leaves (this also reduces memory consumption, by the way):



$$
\begin{aligned}
1 + \quad 1 + \quad 1 + 1 &= 4 \\
4 + \quad 4 + \quad 4 + 1 &= 13 \\
13 + 13 + 13 + 1 &= 40
\end{aligned}
$$

Problem: for a truly $O(1)$ `cons` implementation, we have to “jump over” the zero digits. For *nested* trees, this becomes somewhat tricky. Should be easy with dependent types, but how to convince GHC to accept our program?

Memory footprint

Comparison of the (average) memory footprint (with GHC) of some similar data structures, in extra words per element:

`Data.List`	3
`Data.RandomAccessList`	3
`Data.Sequence`	2.5
`Data.Vector`	1

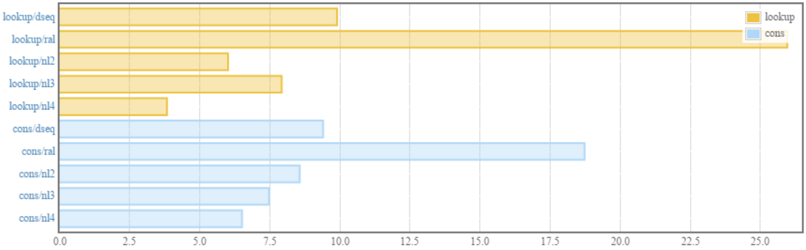
Random-access lists:

	leaf		skew	
	naive	clever	naive	clever
binary	5	3	3	2
ternary	4	2	3	1.666
quaternary	3.666	1.666	3	1.5
$n \to \infty$	3	1	3	1
n-ary	$2 + \frac{n+1}{n-1}$	$\frac{n+1}{n-1}$	3	$\frac{n+2}{n}$

Speed comparison

Libraries compared: `Data.Sequence` (finger tree), `Data.RandomAccessList`, and nested leaf- binary/ternary/quaternary

Lookup & cons:



Update: