Random Access Zipper - RAZ

Simple, Persistent Data Structure for Sequences

Editable List: Insert • Remove • Alter • Move • Goto

There are problems with all these sequence types

Slow access

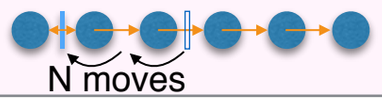
Zipper

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

type 'a zip =
  'a list * 'a list
```

A zipper is a cursor within a sequence. It consists of a pair of lists. The heads of the lists are the two elements nearest the cursor, and the lists extend away from it.

A Zipper is easy to edit by performing list insertion or deletion at the cursor. However, to move to a new location requires moves proportional to the distance.



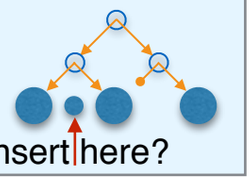
Non-intuitive edits

Tree

```
type 'a tree =
| Nil
| Leaf of 'a
| Bin of
    'a tree * 'a tree
```

A binary tree consists of 'branches' that hold two elements and 'leaves' that hold one. The structure spreads out like a tree. We can use it as a sequence if we only store data in the leaves.

A binary tree is easy to search through because it is so shallow. However, it must be reconfigured after edits to maintain this structure



Complex structure

Fingertree

```
type 'a node =
| Node2 of 'a * 'a
| Node3 of 'a * 'a * 'a
type 'a digit =
| One of 'a
| Two of 'a * 'a
| Three of 'a * 'a * 'a
| Four of 'a * 'a * 'a * 'a
type 'a finger =
| Nil
| Single of 'a
| Deep of
      'a digit
    * ('a node) finger
    * 'a digit
```

A Fingertree is a structure designed for sequences. Edits can be made at the head or tail of the sequence. Fingertrees can be split in two to access the inner items, and then appended back together.

Fingertrees have many cases in their structure. One is **recursive and nested**. This makes algorithms complex.

The RAZ overcomes these problems

Random Access Zipper

A RAZ is balanced by referring to stored values in the structure called '**levels**'. Subtrees are always of lower level than their containing nodes. We use random levels so that they will be appropriate for any situation.

```
type 'a tree =
| Nil
| Leaf of 'a
| Bin of lev * item_cnt
    * 'a tree * 'a tree
```

The RAZ can be in list form as well as tree form, so we store levels between elements, just like tree nodes are between subtrees.

```
type 'a list =
| Nil
| Cons of 'a * 'a list
| Level of lev * 'a list
| Tree of
      'a tree * 'a list
```

```
type 'a raz =
   'a list * 'a * 'a list
```

Tree + **List** + **Zipper**

`fold_up` exploits the independence of subtrees, producing results that could be cached because subtrees do not change if they are not edited.

O(n)

```
fold_up: ('a -> 'b) ->
   ('b -> 'b -> 'b) ->
      'a tree -> 'b
```

O(1)

```
move:    dir ->
  'a zip -> 'a zip
insert:  dir -> 'a ->
  'a zip -> 'a zip
remove:  dir ->
  'a zip -> 'a zip
```

O(log n)

```
focus:   pos ->
  'a tree -> 'a raz
unfocus: 'a raz -> 'a tree
```

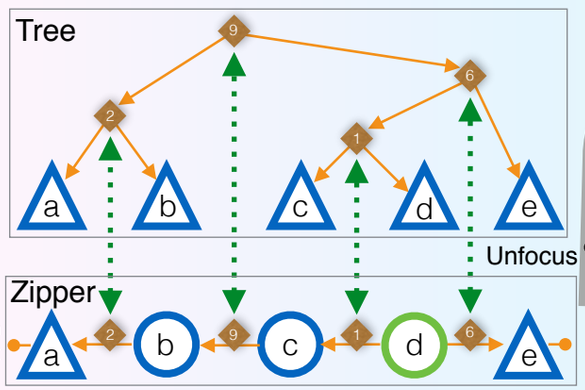
Usage example

```
raz                  a n d e
|> alter left b      a b d e
|> insert left c     a b c d e
|> insert right n    a b c d n e
|> remove right      a b c d e
|> unfocus           a b c d e
|> focus 1           a b c d e
|> insert left n     a n b c d e
```

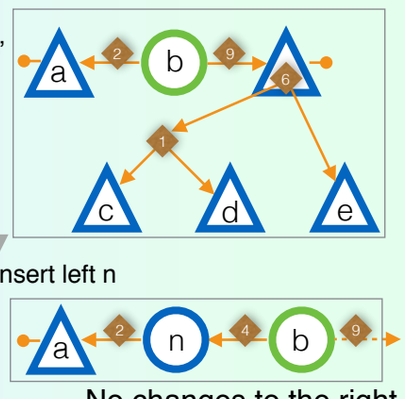
Simple edits are performed as expected. One element is the focus of the operation, shown above in bold and underlined. Focus and unfocus are used to move the cursor to another element.

The RAZ has two main forms

We **focus** the RAZ on a location to be edited. We also **unfocus** to use algorithms designed for trees. The correspondence of levels (brown diamonds) allows the RAZ to be converted between forms at will without loss of information.

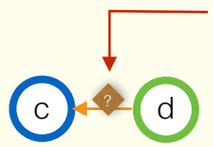


Focusing and unfocusing change the structure and editable state of the RAZ. Any subtrees that are not on the path of the focus are not affected. **This avoids extra work.**



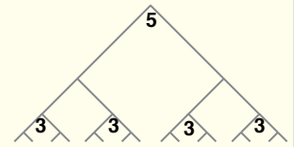
Some additional features of the RAZ

Probabilistic Balance



When we insert a new level, we choose a random number. This helps simplify the coding and use of the RAZ. When converting to a tree form, the relative levels are used to determine node heights.

A balanced Binary tree will have four times as many elements at height **3** than elements of height **5**. We select random numbers from this distribution to balance the RAZ.



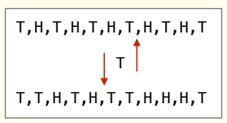
Random Balance Scales Well

Coin Flips	Ratio
T,H,T	0.500
H,H,T,T,T	0.666
H x4,T x5	0.800
H x999, T x1000	0.999

All these coin flip trials have one more tails than heads, but the ratio gets closer to `1.0` as more flips are made.

Random insertions don't break structure

Where could you insert a T into these sequences without ruining the pattern? Nowhere in the first and anywhere in the second. The RAZ is prepared for insertions anywhere.



Simple Code

The RAZ was implemented in OCaml in under **200** lines of code. Some design choices were made to amplify this simplicity. Programmers can implement their own modifications with confidence.

Focus is easy because the accumulator is the zipper form of the RAZ, but without the focused element.

```
let focus pos tree =
  let rec focus = fun pos tree (l, r) ->
  match tree with
  | Nil -> failwith "focus: internal Nil"
  | Leaf(elm) ->
    assert (pos == 0);
    (l,elm,r)
  | Bin(lv, _, branch_l, branch_r) ->
    let cnt = item_count branch_l in
    if pos < cnt
    then
      focus pos branch_l
      (l, Level(lv, Tree(branch_r, r)))
    else
      let new_pos = (pos - cnt) in
      focus new_pos branch_r
      (Level(lv, Tree(branch_l, l)), r)
in focus pos tree (Nil, Nil)
```

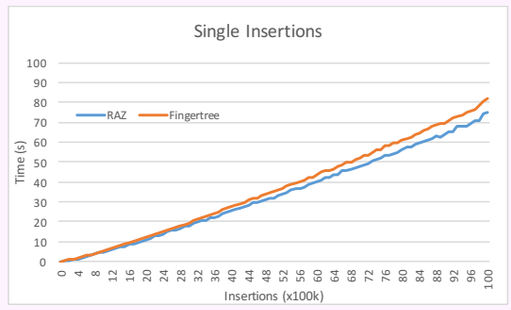
We insert whole subtrees with their level into the accumulator so that we don't have to rebalance later.

Below is the common pattern for all local edits. It's just a pattern match over inputs. And only one line is needed per match. There is even common logic that appears in each local edit.

```
let alter : dir -> 'a -> 'a raz -> 'a raz =
  let rec alter new side zip = match zip with
  | Nil -> failwith "alter: past end of seq"
  | Cons(_,rest) -> Cons(new,rest)
  | Level(lv,rest) -> Level(lv,alter new side rest)
  | Tree _ -> alter new side (trim side zip)
  in fun side elm (l,e,r) -> match side with
  | L -> (alter elm L l,e,r)
  | R -> (l,e,alter elm R r)
```

Fast Access

We built a sequence from scratch by inserting single elements into random places until we reached a target size. We plotted the median of three trials for each of 100 sizes. **The RAZ came out ahead by nearly 10%.**



With the RAZ, we have Accessibility, Editability, Simplicity, and Speed

by Kyle Headley

More info at kyleheadley.github.io

CUPLV
Programming Languages and Verification
University of Colorado Boulder