Random Access Zipper

Kyle Headley*
University of Colorado Boulder†
kyle.headley@colorado.edu

Incremental computation (IC) is ubiquitous in modern software. A computation is incremental if repeating it with a changed input is faster than from-scratch recomputation. Most contemporary incremental functionality is ad-hoc, built into the efficient implementation of the system being designed. I have been working on language- and library-based incremental computation methods in order to increase the efficiency of code in a more systematic way.

One important part of the research is to have good functional data structures (side effects spoil memoization). General purpose IC is best applied to situations where small changes have little effect on the overall computation. Trees, for example, can be computed over recursively from leaf to root, so that the length of change propagation is logarithmic in the number of elements. Tree re-balancing is a problem, however, because a small change can effect a large part of the tree. Adding an item to the head of a list is also a small change. However, lists require linear time to seek to an inner element to change it, and change propagation also takes linear time.

As part of a recent IC project, one of my tasks was to distill the algorithmic essentials from a complex experimental code base. This code provided methods for working with lists in an effort to make searching more efficient. My simplification helped coalesce our ideas into a single data structure and methods for its manipulation. We dub this structure the "RAZ", which stands for Random Access Zipper.

A zipper is a functional data structure that represents a sequence of items and a cursor within it [2]. Its strength is that edits local to the cursor are performed in O(1) time, including insertion, removal, and movement of the cursor across a single item. Movement to arbitrary points in the zipper, however, requires a linear time search like an ordinary list.

To overcome this limitation, I wrote algorithms that build an "unfocused" tree out of the RAZ, with the sequence data in leaves, and another to refocus the tree into a "focused" RAZ with the cursor at the target point. To facilitate the transformations, the focused RAZ contains entire branches in unfocused form. These are partially deconstructed, or "trimmed", on demand as edits are made. The focus and unfocus operations take O(log n) time, and trimming is nearly constant time (logarithmic in the size of the closest subtree to the cursor), but the unfocused tree needs to be balanced.

The RAZ is balanced by including data in each tree node representing its level in the unfocused RAZ. Inspiration for the probabilistic levels was taken from [3], who devised a balancing method based on the data of the represented sequence. In the focused RAZ, level meta data appears between each item of the sequence and is handled behind the scenes by the editing operations. The levels are assigned randomly with a negative binomial distribution, that is, from a distribution similar to picking the height of a random node from a fully balanced tree. The levels define a unique tree structure with no need for re-balancing.

The inclusion of meta data in the tree nodes works well with Adapton, an incremental computation engine I worked on previously [1]. Adapton uses named locations when memoizing data structures. Instead of storing a level in a tree node, we store a unique name from which a level can be generated. These names can then be used as keys to retrieve memoized results of operations over entire subtrees, or sequences. Stats about the RAZ can be computed with memoized functions instead of stored in tree nodes. For example, the focus algorithm uses memoized element count of a subtree to search for a particular location in the RAZ.

There are three algorithms that exemplify the RAZ. They are provided below with explanation:

*ACM Student Member 1004593
†Work done as 1st year graduate, Spring '16

```
let unfocus (l, r) =
  let ltree = fold (fun a b ->
    merge (tree a) b) l Nil in
  let rtree = fold (fun a b ->
    merge b (tree a)) r Nil in
  merge ltree rtree

let rec merge t1 t2 =
  match t1, t2 with
  | Nil, _ -> t2
  | _, Nil -> t1
  | Leaf(_), Leaf(_) ->
    Bin(new_name(), t1, t2)
  | Leaf(_), Bin(n,l,r) ->
    Bin(n, merge t1 l, r)
  | Bin(n,l,r), Leaf(_) ->
    Bin(n, l, merge r t2)
  | Bin(n1,t1l,t1r), Bin(n2,t2l,t2r) ->
    if level n1 > level n2
    then Bin(n1, t1l, merge t1r t2)
    else Bin(n2, merge t1 t2l, t2r)
```

Unfocus (along with merge) folds over the two sides of the focused RAZ, builds trees from each item and merges all of them together. Merge works by checking the levels of both tree roots (leaves are considered lowest). The root with the highest level becomes the combined root, and the other tree is recursively merged into the branch at the appropriate side.

```
let rec focus_rec tree pos zip =
  match tree with
  | Nil -> zip
  | Leaf(l) ->
    if pos == 0
    then insert l zip R
    else insert l zip L
  | Bin(n, l, r) ->
    let lc = item_count_memo l in
    if pos < lc
    then focus_rec l pos
      (insert_tree n zip r Right)
    else focus_rec r (pos - lc)
      (insert_tree n zip l Left)
```

Focus takes an unfocused RAZ, a location, and an accumulator RAZ, and produces a focused RAZ. The path through the unfocused RAZ to the target location divides it into two, and items are inserted into the focused RAZ depending on which side of the path they are on. Subtrees that are not split are inserted into the focused RAZ unmodified.

```
let trim dir tl =
  let rec loop tlist =
  match tlist with
  | Nil | Cons(_,_) | Name(_,_) -> tlist
  | Tree(tree, rest) ->
    match tree with
    | Nil -> rest
    | Leaf(l) -> Cons(l,rest)
    | Bin(n,l,r) ->
      match dir with
      | Left ->
        loop (Tree(l,Name(n,Tree(r,rest))))
      | Right ->
        loop (Tree(r,Name(n,Tree(l,rest))))
  in loop tl
```

Trim is used when the RAZ cursor attempts to edit an included subtree. It breaks down the subtree as if it were focusing on the leaf closest to the cursor. The subtrees resulting from the trim are inserted back into the RAZ in order. The newly exposed item can then be edited in place.

Functions that edit the RAZ are not shown, but can be implemented as expected for a zipper, with one difference: along with regular functionality, insert adds a name, delete removes a name, and move moves through a name.

The RAZ and associated algorithms provide a way to edit a functional data sequence in logarithmic time regardless of the location or type of edits. Moreover, a RAZ can temporarily perform as a common zipper, with constant time edits, though the next unfocus call will take longer. A RAZ works well with incremental computation techniques, sharing necessary meta data, and providing a tree structure for ease of recomputation. It is a great structure for arbitrary incremental changes to a sequence of data.

References

[1] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. Incremental computation with names. In *OOPSLA*, 2015.

[2] Gérard Huet. The zipper. *Journal of Functional Programming*, 1997.

[3] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *POPL*, 1989.