The Random Access Zipper: Simple, Purely-Functional Sequences
(Research Paper)

Kyle Headley[†] and Matthew A. Hammer

University of Colorado Boulder
kyle.headley@colorado.edu, matthew.hammer@colorado.edu

[†] Student author (also, main author)

Abstract. We introduce the Random Access Zipper, or RAZ, a simple-to-code, purely-functional data structure for editable sequences. The RAZ combines the structure of a zipper with that of a tree: Like a zipper, edits can be done in constant time, and in batches. Like a tree, the edit location can be moved to another part of the sequence in log time. While existing data structures provide these time-bounds, none do so with the same simplicity and brevity of code as the RAZ. Simplicity provides the opportunity for more programmers to extend the structure to their own needs, and we provide some suggestions for how to do so.

1 Introduction

One of the most useful tools a functional programmer knows of is the singly-linked list. It is not just used directly from a library, but the principles of its design are built into programmer-defined data structures. The simplicity of the design of a list, with two cases and a couple of data values, helps facilitate this extensibility. Lists are not the best choice for all situations, though, because the programmer can only edit the head of a list. To get around this, researchers have developed many other data structures representing sequences. These structures perform well, allowing edits in constant-time and moving the edit location in log-time. Unfortunately, they lack the simplicity and extensibility of a singly-linked list.

In this paper, we introduce the "random access zipper" or RAZ, a general-purpose data structure for a sequence that performs well and has a simple design. We achieve this by using a type of tree structure designed for incremental computation [11], providing balance without a rigid form. We also incorporate the design of a zipper [9], which provides a "cursor" defining an edit location that can be moved through a sequence. The code of the resultant union is simple, requiring few special cases, generally only a pattern-match on input, and only two non-trivial helper functions. We present it in section 4

The core functionality of the RAZ is to represent a sequence as two complimentary forms. The main, editable form presents a zipper-like API to the user, with the cursor at a particular element. The second, intermediate form presents a binary tree (with sequence data in the leaves) to the user for simple recursive traversal, or for use in our "append" function to concatenate two sequences. To switch forms, the user can "focus" a tree onto an element, producing a zipper with cursor at that element. The user can also "unfocus" any zipper to produce a tree. To access parts of the zipper, a user can call the local "move" function to move the cursor to an adjacent element, or use the composition of unfocus and focus to move the cursor to any other element. We provide more detailed examples of these actions in section 2.

Our implementation of the RAZ requires under 200 lines of OCaml. This code includes ten main functions, each with at most a case select over the inputs, a recursive call, and one non-trivial helper function. In contrast, a finger tree [8], another representation of a sequence with similar performance, requires approximately 800 lines in the OCaml "batteries included" repo to provide similar functionality. The main functions of this code are grouped into five to ten sub functions, each of which have a case select on the inputs, recursive calls, and sometimes calls to other main functions.

This simplicity does not mean poor performance. The binary tree form of the RAZ allows focusing on a specific leaf in time logarithmic in the number of elements. Local insertions are exactly like linked list insertion, taking constant time. Other edits are a bit more complicated, but take amortized constant time. Unfocusing, or building a single tree from the RAZ, is our most demanding operation. It performs work that other data structures do during local editing. This can take constant time for a new sequence and log time for a sequence unmodified since the prior focus. We present an empirical performance evaluation in section 5.

In addition to the sections mentioned above, we describe some of the design choices we made that increase code simplicity in section 3; we present some suggestions for extending the RAZ in section 6; and we discuss other similar data structures in section 7.

2 Overview

Here we present some examples demonstrating the structure of the RAZ along with the operations that modify it globally, and a few local edits. We briefly contrast this with a finger tree structure, to provide some context.

The RAZ makes use of probabilistic balance when in tree form. That is, a random number (with an appropriate distribution) is assigned to each binary node in the tree. These numbers are used to determine which of a pair of nodes will be parent and which will be child, based on magnitude. The positioning of these numbers in the RAZ is elided from the following examples for clarity, but elaborated on in section 4. Our initial examples show a RAZ with perfect balance for clarity. In general, this will not be the case, but let us assume that initial numbers were chosen favorably.

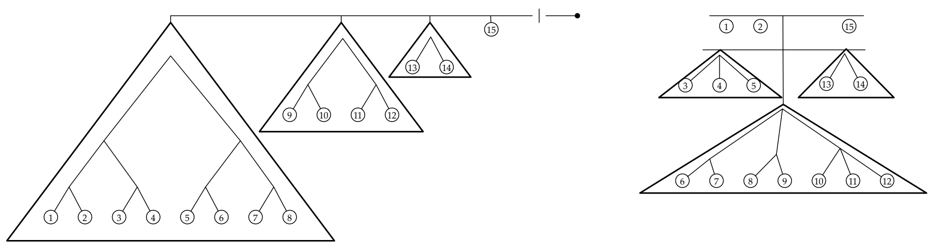

Fig. 1. A RAZ (left) and finger tree (right) representing the same sequence

Figure 1 depicts both a RAZ and a finger tree containing 15 elements, numbered sequentially. Elements are shown as circles, internal trees have a triangle around them, with diagonal lines denoting internal tree structure. Horizontal and vertical lines elide a simple structure for access to elements: a list in the RAZ case and an set of data types for the finger tree. Both these data structures provide access to the upper right element, labeled “15”. We elide the current item and right side of the RAZ, as it is not important for this example.

A finger tree has one core branch that leads deep into its structure, ending at a single subtree. At each other node of the core branch, there are two additional branches called “fingers”. The subtrees attached to a finger are 2-3 trees of the same height as their depth along the core. Each finger can contain up to four subtrees, though here we show only one. Similarly the top-level fingers can contain up to four exposed elements. Of note in all the following examples is that both the zipper form of the RAZ and the finger tree can represent the same sequence in multiple ways. However, each subtree of the RAZ can only be structured in one way (dependent on previously chosen random numbers).

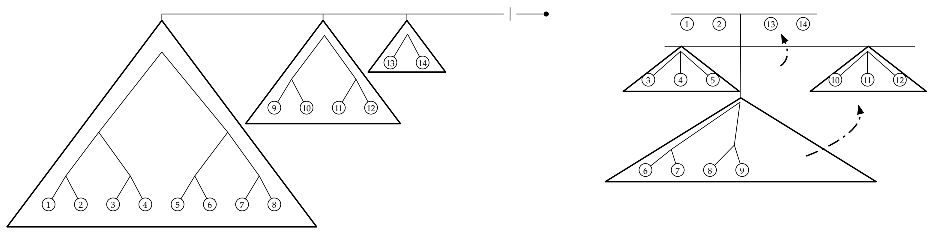

Fig. 2. Removing an element from a RAZ (left) and a finger tree (right), with structure maintenance on the finger tree

One major difference between the finger tree and RAZ is when they need to adjust their structure to maintain invariants. Figure 2 shows the result of deleting element 15 in both our example structures. They both expose this element, but the RAZ requires no maintenance at this point, while the finger tree does, since there are no elements left in the top finger. This is done by promoting a tree from the next deeper finger. In this case, the finger tree must promote another tree from even deeper. These promotions are indicated by the arrows in the figure.

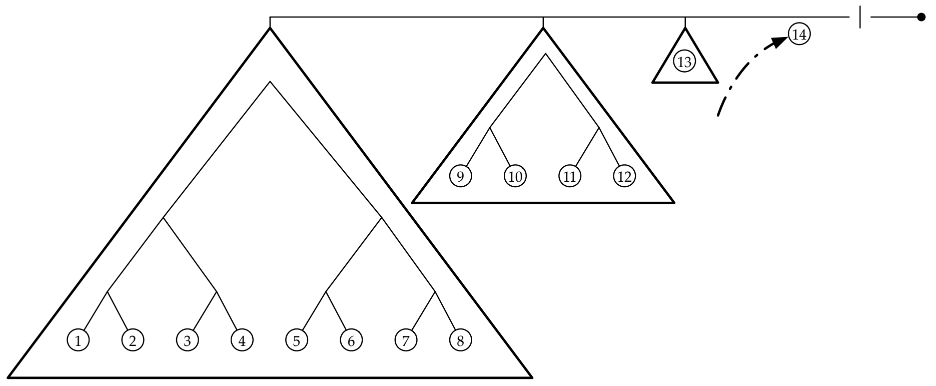

Fig. 3. Preparing a RAZ for another deletion

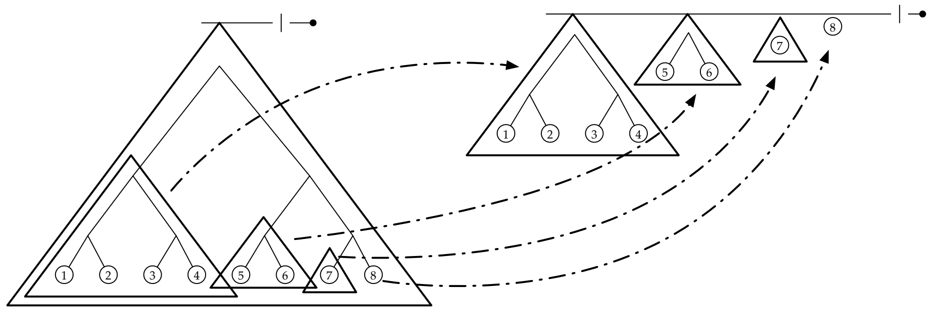

Fig. 4. Trimming a large subtree of a RAZ

The RAZ is not exempt from structural changes. If we tried to delete another element, it would first need to deconstruct a subtree to expose an element, as seen in figure 3. We call this "trimming" the tree. In this case very little work needed to be done, but if we tried to delete all the elements, when we reached the final subtree, the trimming would look like figure 4. As the arrows indicate, the subtree is broken down into multiple parts, each becoming a subtree on the main list.

The next three figures demonstrate the append and focus operations on the RAZ. Following our example, if we had moved the cursor across the RAZ rather than deleting items, our main append step may look like figure 5. (We skipped the step where we build up the right tree, it is the reverse of the trim shown

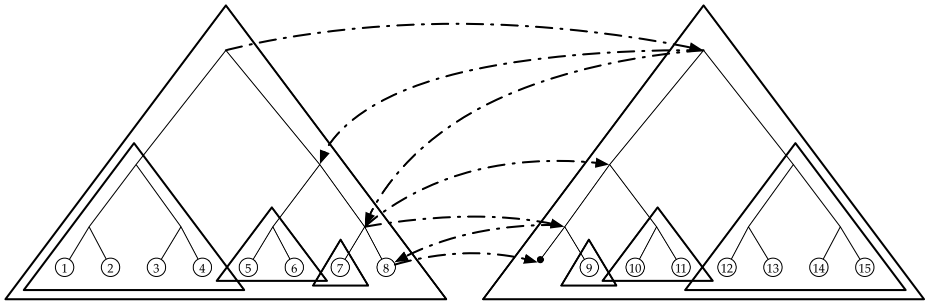

Fig. 5. Comparing subtrees during an append

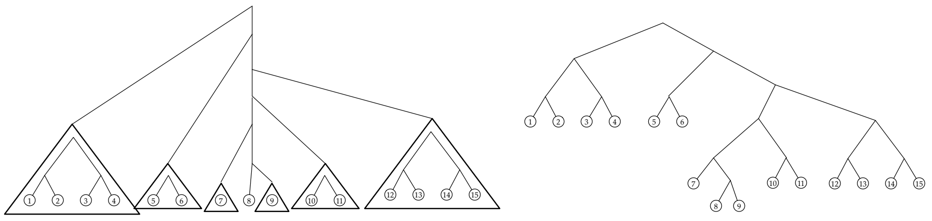

Fig. 6. The result of an append, visually depicting the algorithm (left), and the traditional tree view(right)

above.) To append a left and right tree, we recursively descend their right and left sides, respectively. We make several comparisons, as the arrows indicate. This operation is better visualized on the right side of figure 6, where the central line shows the sorted heights of the nodes. It works like the merge step of a mergesort, placing higher values first, and comparing the next values. A more traditional view of the appended sequence is shown on the right side of figure 6.

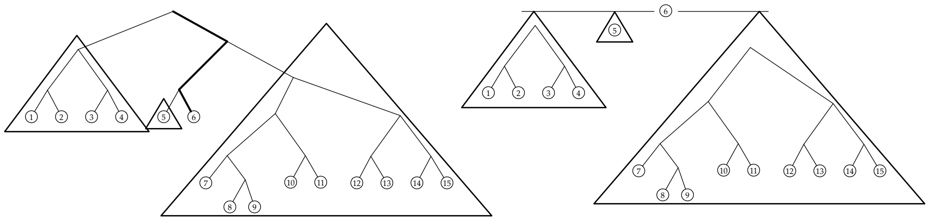

Fig. 7. Focusing into a RAZ, the path through the tree form (left), and the resultant zipper form (right)

Next we focus the previously created tree form of the RAZ onto element 6. To do this, we traverse the path through the tree nodes to reach 6, as shown in the left side of figure 7. (To facilitate this, we store in each node the total number of elements below it.) For each path we don't take, we store the entire subtree in an accumulator list, one for each side of the tree. This pair of accumulators becomes the zipper form of the RAZ when we reach the target leaf and make it the current item. The right side of figure 7 shows the results, and we can see that the original tree was split into subtrees along the path to the target by the triangles around each.

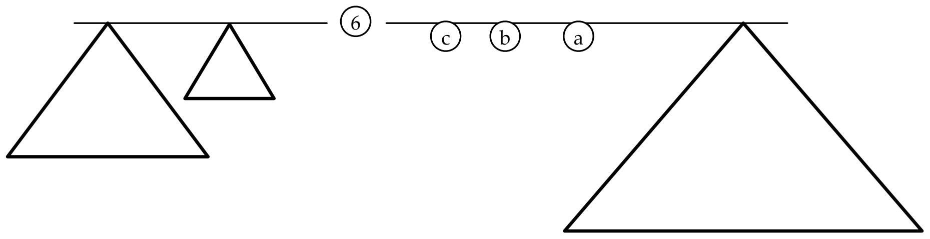

Fig. 8. Inserting elements into a RAZ doesn't change any subtrees

Finally, figure 8 shows the results of inserting "a", "b", and "c" in order into the right side of the zipper created above. Insertion never requires restructuring, so there is no change in any subtree of the RAZ. This is indicated by eliding the tree structure withing the triangles. Removing these elements again could be done without needing to trim subtrees to access them. This can make batch editing of the RAZ very efficient in some cases.

3 Design

We made a number of design decisions for the RAZ, mainly opting for simplicity of code. One decision was to use a probabilistically balanced tree rather than the strict structure of many other tree types. Every time an element is inserted into the RAZ, we also include meta data from a negative binomial distribution. This type of distribution provides numbers as if chosen from the heights of nodes of a perfectly balanced binary tree. The unfocus operation creates a tree based on this meta data. This eliminates the need to rebalance a tree when data changes, which eliminates some of the most complex code that tree structures require. Using random numbers for balance means that the RAZ may not be well balanced in any local area, but it will be balanced globally, which is the most important for performance at scale.

An additional benefit of providing a tree with specific heights is that of a stable form. The structure of the RAZ does not depend on the order of operations, but on the internal meta data. This means that when data is changed, the tree of the RAZ is changed minimally, specifically, the path from the changed data to the root will be different, but the rest of the tree will remain the same. This is of great benefit for incremental computation, which is most effective for small changes. The RAZ was inspired by [11], where they present a number of structures for incremental computation. Those designs use a hash of the data to determine heights, which presents a problem when a sequence contains a subsequence of identical values. The RAZ sidesteps this issue with explicit values for tree heights.

Another design decision we made was to provide a current element at the cursor position, rather than leaving the cursor between elements. Doing this provided about a 25% reduction in code, removing some complex parts that dealt with our tree-height meta data. Because height is located between elements in the focused RAZ, leaving the cursor between elements broke the symmetry. The RAZ could have the height to the left or right of the cursor. Even when making a global choice of one side, each local editing action required two forms, one for left edits and one for right edits. By singling out a current item, we have heights on both sides of the cursor. This can make local edits a bit unintuitive, because we focus on one element but modify elements on either side. An elegant solution for the ends of the RAZ is to construct a sequence with a null element at each end. Focusing on a null element and inserting a new element to the left modifies the sequence as expected of a list structure, and the null element stays at the end.

4 Implementation

The RAZ is implemented in about 170 lines of OCaml. It includes five local editing operations, three global modification operations, two non-trivial helpers, and some trivial ones. We explain the code below, showing most of it, with type signatures for all elided code.

```
type lev = int
type dir = L | R
type 'a tree =
  | Nil | Leaf of 'a
  | Bin of lev * int * 'a tree * 'a tree
type 'a tlist =
  | Nil | Cons of 'a * 'a tlist
  | Level of lev * 'a tlist
  | Tree of 'a tree * 'a tlist
type 'a zip = ('a tlist * 'a * 'a tlist)
```

Our types include representatives for the two forms of the RAZ, tree and zipper, and a tree-list used to build the two sides of the zipper. We see that the tree-list includes trees and the zipper includes a current element. The tree also includes `int` data in the bin nodes, which is used to store the count of elements in its branches. Most importantly, we see level data stored in both the bin nodes of the tree and in the tree-list. This is the meta data we use to keep track of the relative heights of bin nodes. It is randomly generated and inserted inserted into the tree-list when inserting a new element, and stored to maintain the balance of trees probabilistically. More specific information in described with the code below.

```
let rnd_level : unit → lev
let item_count : 'a tree → cnt

let singleton e = (Nil,e,Nil)
let empty n = (Level(rnd_level(),Cons(n,Nil)),n,Nil)
```

The code above provides some helper functions. `rnd_level` produces a random number from a negative binomial distribution. That is, it selects a height from one of the nodes of a perfectly balanced (infinite) tree. `item_count` accesses the data stored in a bin node, and provides defaults for the other cases of a tree. `singleton` is presented because the RAZ is focused on an element, and needs an initialization. An alternative is `empty`, which takes a value considered null, to be used as endpoints of the RAZ. This is useful for determining bounds in our simplistic model. We start focused on the rightmost element, allowing initial inserts left to create the sequence.

```
let trim : dir → 'a tlist → 'a tlist =
  fun d tl → match tl with
  | Nil | Cons _ | Level _ → tl
  | Tree(t, rest) →
  let rec trim = fun h1 t1 → match h1 with
  | Nil → failwith "poorly formed tree"
  | Leaf(elm) → Cons(elm,t1)
  | Bin(lv,_,l,r) → match d with
    | L → trim r (Level(lv,Tree(l,t1)))
    | R → trim l (Level(lv,Tree(r,t1)))
  in trim t rest
```

`Trim` breaks down a subtree in a tree-list, exposing an element for editing. The first few lines check that the head is a tree, returning if not. The bin nodes are broken down, pushing the larger side into the current tree-list and recursing on the side to be further broken down. Levels are also added to the tree-list, such that each level will be between the subtrees that it was originally between as part of a bin node. By maintaining this order, we can easily reconstruct the trimmed tree later on, shown below in the `grow` code.

```
let insert d ne (l,e,r) = match d with
  | L → (Level(rnd_level(),Cons(ne,l)),e,r)
  | R → (l,e,Cons(ne,Level(rnd_level(),r)))

let remove : dir → 'a zip → 'a zip =
  let rec remove d s = match s with
  | Nil → failwith "remove past end of seq"
  | Cons(_,rest) → rest
  | Level(lv,rest) → remove d rest
  | Tree _ → remove d (trim d s)
  in fun d (l,e,r) → match d with
  | L → (remove L l,e,r)
  | R → (l,e,remove R r)

let view_c : 'a zip → 'a
let view : dir → 'a zip → 'a
let alter_c : 'a → 'a zip → 'a zip
let alter : dir → 'a → 'a zip → 'a zip
let move : dir → 'a zip → 'a zip
```

The next group of code is for local edits. Inserting into a tree-list is straight forward, like inserting into a list, except that a new level must be generated for each new element. We do that here with a call to get a random value. `Remove` has a main branch over the direction parameter, which just applies a function to the proper side of the RAZ. This function has to deal with a few situations. One is that the first item in either side of the RAZ should be a level, followed by a tree or element. We abstract over that layout, assuming zero or more levels, simplifying the code to a simple recursive call. Similarly, we make a recursive call on the tree case after trimming to expose the next element. The case when an element is present is simple here, ignoring it by returning the rest of the tree-list. The code for the rest of the local edits is structured the same way as `remove`. `View` returns the first element to one side, while `view_c` returns the current element. `alter` and `move` could both be accomplished by appropriate use of insert and remove, but we include them so that they work without affecting the surrounding levels.

```
let focus : 'a tree → int → 'a zip =
  fun t p →
  let c = item_count t in
  if p >= c || p < 0 then failwith "out of bounds"
  let rec focus = fun t p (l,r) →
  match t with
  | Nil → failwith "internal Nil"
  | Leaf(elm) → assert (p == 0); (l,elm,r)
  | Bin(lv, _, bl, br) →
    let c = item_count bl in
    if p < c then focus bl p (l,Level(lv,Tree(br,r)))
    else focus br (p - c) (Level(lv,Tree(bl,l)),r)
  in focus t p (Nil,Nil)

let rec append : 'a tree → 'a tree → 'a tree =
  fun t1 t2 →
  let tot = (item_count t1)+(item_count t2) in
  match (t1, t2) with
  | Nil, _ → t2 | _, Nil → t1
  | Leaf(_), Leaf(_) →
    Bin(rnd_level(), tot, t1, t2)
  | Leaf(_), Bin(lv,_,l,r) →
    Bin(lv, tot, append t1 l, r)
  | Bin(lv,_,l,r), Leaf(_) →
    Bin(lv, tot, l, append r t2)
  | Bin(lv1,_,t1l,t1r), Bin(lv2,_,t2l,t2r) →
    if lv1 >= lv2
    then Bin(lv1, tot, t1l, append t1r t2)
  else Bin(lv2, tot, append t1 t2l, t2r)
```

`Focus` and `append` work as described in the section 2. The main recursive loop of `focus` ends when it reaches the target leaf, making it the current item of the constructed RAZ. Otherwise, when encountering bin nodes, it chooses which branch to descend based on the item count, and adds the other to an accumulator based on which side of the bin node it came from. `Append` is similar to the merge step of mergesort. It compares two trees by level, returning the higher one with a recursive call to append the lower one to the appropriate subtree. When encountering nil trees we return the other. Leaves are treated as having the lowest level possible, with a leaf-leaf case requiring a new level between them. This will not happen when unfocusing a RAZ, since we keep all items in order. However, if the user wants to merge arbitrary trees, then this level will be the one between the two.

```
let head_as_tree : 'a tlist → 'a tree
let tail : 'a tlist → 'a tlist

let grow : dir → 'a tlist → 'a tree =
  fun d t →
  let rec grow = fun h1 t1 →
    match t1 with Nil → h1 | _ →
    let h2 = head_as_tree t1 in
    match d with
    | L → grow (append h2 h1) (tail t1)
    | R → grow (append h1 h2) (tail t1)
  in grow (head_as_tree t) (tail t)

let unfocus : 'a zip → 'a tree =
  fun (l,e,r) → append (grow L l)
    (append (Leaf(e)) (grow R r))
```

The final code is all used for `unfocus`. `Head_as_tree` returns the head of a tree-list, constructing a tree out of it if is not already one. `Tail` returns the rest of the tree-list. The `unfocus` function itself just makes calls to `grow` and `append`, which create trees from tree-lists and combine them into a single tree. `Grow` is the opposite of `trim`, building up subtrees one branch at a time. It makes calls to `append`, however, since it reverses the trimming (or focusing) in order, subtrees will be separated by levels, constructed by `head_as_tree` as a bin node with two nil branches. These branches will be appended in a single step. `Grow` works by keeping an accumulator tree, and folding the elements of the input tree-list into it with appends.

5 Evaluation

In a future revision, this section of the paper will evaluate the RAZ on a benchmark of randomized edits. In particular, we hope to empirically demonstrate the time bounds argued in the paper. In addition, we also plan to compare the performance of the RAZ against the 2-3 Finger tree on the same benchmarks. Finally, *by employing hash-consing and measuring space usage, we plan to demonstrate that the RAZ is suitable for hash-consing (e.g., for $O(1)$-time equality comparisons), while the 2-3 Finger tree enjoys less structural sharing, and is thus less sutiable for settings that use hash-consing.*

We welcome input from the reviewers about this evaluation, which is currently underway.

6 Enhancements

One benefit of an extremely simple data structure is that the programmer can modify it to suit their specific purpose. The unfocused form of the RAZ is a binary tree we have annotated with size info in order to search for an element at a particular location. By using other types of annotations, the RAZ can be used for many other purposes. For example, by annotating each subtree with the highest priority of elements within, we have a priority tree, which can access the next item in log time, while still adding items in constant time. The paper on finger trees [8] has additional suggestions of this kind.

As was mentioned in section 3, there were two design choices that prevented some potential use cases for the RAZ. Further enhancements could provide this functionality. The ends of the RAZ could point to a central element so that it works as a deque. This would require difficult refocusing if the cursor position moved past the central point. Or, the RAZ's cursor could be repositioned between two sequence elements instead of at a current value as in the code above. This would streamline some types of local editing. There would be a couple of changes to the code to allow this, one being that the focus call would need to push the final element into one side of the accumulator. The choice of side would determine how the local edit code must change. Each of those functions would require two cases, one for when there is level meta data present and one where there is only data. An invariant for which side the level appears on must be maintained to avoid degradation of the RAZ structure.

There are a number of enhancements dealing with our explicit storage of level data in the RAZ. For example, this extra space could be reclaimed by generating levels based on a hash of a nearby element. This was the design of probabilistic data structures of [11], and provides a canonical tree, one that is structured the same way regardless of edit history. There are two main disadvantages with this approach. One is that duplicate elements will have duplicate levels, which will interfere with the balance of the unfocused RAZ. Another is that the RAZ is unfocused from the cursor outwards, meaning that elements on one side are in reverse order from the other side. Maintaining a canonical tree in this context would require additional code complexity. Though if a canonical tree is not needed than this is not a problem.

Alternately, the level data could be set with programmer defined values rather than randomized ones. This allows direct control over the balance of the unfocused RAZ. For example, if it was known beforehand that there would be no insertions or deletions, a perfect balance could be set. Another use for programmer-defined balance is to use levels as priority controls. High levels mean faster access, so elements could be inserted along with a high priority if the programmer expects to focus on them often. As above, care would need to be taken to match the priority to the element. Reaching it from different directions in the unfocus step could negate the benefit.

7 Related Work and Alternative Approaches

We review related work on representing purely-functional sequences that undergo small (constant-sized) edits, supplementing the discussions from earlier sections. We also discuss hypothetical approaches based on (purely-functional) search trees, pointing out their differences and short-comings for representing sequences.

The "Chunky Decomposition Scheme". The tree structure of the RAZ is inspired by the so-called "chunky decomposition scheme" of sequences, from Pugh and Teiltelbaum's 1989 POPL paper on purely-functional incremental computing [11]. Similar to skip lists [10], this decomposition scheme hashes the sequence's elements to (uniquely) determine a probabilistically-balanced tree structure. The RAZ enhances this structure with a focal point, local edits at the focus, and a mechanism to overcome its inapplicability to sequences of repeated (non-unique) elements. In sum, the RAZ admits efficient random access for (purely-functional) *sequence editing*, to which the '89 paper alludes, but not does not address.

Finger trees. As introduced in Section 1, a finger tree represents a sequence and provides operations for a double-ended queue (aka, deque) that push and pop elements to and from its two ends, respectively. The 2-3 finger tree supports these operations in amortized constant time. Structurally, it consists of nodes with three branches: a left branch with 1–4 elements, a center for recursive nodes, and a right branch with 1–4 elements. Each center node consists of a *complete* 2-3 tree. This construction's shallow left and right (non-center) branches admit efficient, amortized constant-time access to the deque's ends. This construction also provides efficient (log-time) split and append operations, for exposing elements in the center of the tree, or merging two trees into a single 2-3 tree. The split operation is comparable to the focus operation of the RAZ; the append operation is comparable to that of the RAZ.

While similar in asymptotic costs, in settings that demand *canonical forms* and/or which employ *hash consing*, the 2-3 finger tree and RAZ are significantly different; this can impact the asymptotics of comparing sequences for equality. In the presence of hash-consing, structural identity coincides with physical identity, allowing for $O(1)$-time equality checks of arbitrarily-long sequences. As a result of their approach, 2-3 finger trees are history dependent. This fact makes them unsuitable for settings such as memoization-based incremental computing [11, 7, 6].

Balanced representations of Sets. Researchers have proposed many approaches for representing sets as balanced search trees, many of which are amenable to purely-functional representations (e.g., Treaps [2], Splay Trees [12], AVL Trees [1], and Red-Black Trees [3]). Additionally, skip lists [10] provide a structure that is tree-like, and which is closely related to the probabilistic approach of the RAZ. However, search trees (and skip lists) are *not* designed to represent sequences, but rather sets (or finite mappings).

Structures for sequences and sets are fundamentally different. Structures for sequences admit operations that alter the presence, absence and ordering of elements, and they permit elements to be duplicated in the sequence (e.g., a list of n repeated characters is different from the singleton list of one such character). By contrast, structures for sets (and finite maps) admit operations that alter the presence or absence of elements in the structure, but not *the order* of the elements in the structure—rather, this ordering is defined by the element's type, and is not represented by the set. Indeed, the set representation uses this (fixed) element ordering to efficiently search for elements. Moreover, set structures typically do not distinguish between the sets with duplicated elements—e.g., `add(elm, set)` and `add(elm, add(elm, set))` are the same set, whereas a sequence structure would clearly distinguish these cases.

Encoding sequences with sets. In spite of these differences between sets and sequences, one can *encode* a sequence using a finite map, similar to how one can represent a mutable array with an immutable mapping from natural numbers to the array's content; however, like an array, editing this sequence by element insertion and removal is generally problematic, since each insertion or removal (naively) requires an $O(n)$-time re-indexing of the mapping. Overcoming this efficiency problem in turn requires employing so-called *order maintenance data structures*, which admit (amortized) $O(1)$-time insertion, removal and comparison operations for a (mutable) total order [5, 4]. Given such a structure, the elements of this order could be used eschew the aforementioned re-indexing problem that arises from the naive encoding of a sequence with a finite map. Alas, existing order maintenance data structures are *not* purely-functional, so additional accommodations are needed in settings that require persistent data structures. By contrast, the RAZ is simple, efficient and purely-functional.

8 Conclusion

We present the Random Access Zipper (RAZ), a novel data structure for representing a sequence. We show its simplicity by providing most of the code, which contains a minimal number of cases and helper functions. We describe some of the design decisions that increase simplicity. We evaluate the RAZ, demonstrating that it provides time bounds on par with far more complex data structures. Finally, we suggest multiple ways to enhance the RAZ to suit additional use cases.

References

1. M AdelsonVelskii and Evgenii Mikhailovich Landis. An algorithm for the organization of information. Technical report, DTIC Document, 1963.
2. Cecilia R. Aragon and Raimund Seidel. Randomized search trees. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 540–545, 1989.
3. Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.
4. Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, pages 152–164, 2002.
5. Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 365–372, 1987.
6. Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 748–766, 2015.
7. Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. Adapton: composable, demand-driven incremental computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 18, 2014.
8. Ralf Hinze and Ross Paterson. Finger trees: A simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006.
9. Gérard Huet. The zipper. *Journal of Functional Programming*, 1997.
10. William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings*, pages 437–449, 1989.
11. William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *POPL*, 1989.
12. Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 235–245, 1983.