Performance Analysis of Zippers

Vít Šefl

Faculty of Mathematics and Physics
Charles University, Prague, Czech Republic
sefl@ksvi.mff.cuni.cz

Abstract. The zipper is a powerful technique of representing purely functional data structures in a way that allows fast access to a specific element. It is often used in cases where the imperative data structures would use a mutable pointer. However, the efficiency of zippers as a replacement for mutable pointers is not sufficiently explored. We attempt to address this issue by comparing the performance of zippers and mutable pointers in two common scenarios and three different languages: C++, C♯, and Haskell.

1 Introduction

Some programming techniques make use of the ability to keep a pointer to internal parts of a data structure. Such a pointer is usually called a *finger* [9]. As an example, a finger can be used to track the most recently used node in a tree. Tree operations can then start from the finger instead of starting from the root of the tree, which can lead to a speedup if the program frequently operates on values that are stored near each other.

However, fingers lose most of their utility when applied to purely functional data structures. Operations that make use of fingers frequently require the structure to contain pointers to parent nodes or require mutability. Pointers to parent nodes create loops which hugely complicate update operations.

The *zipper* [5] is a technique of representing purely functional data structure in a way that allows direct access to a selected location. Different data structures have different zipper representations: we, therefore, distinguish between list zippers, tree zippers, etc. Zippers differ from fingers in a crucial way. Unlike a finger, a zipper contains the data structure. A finger can be removed, and the structure it was pointing to remains intact while removing a zipper removes the structure it contains. As a consequence, while two fingers give direct access to two locations, two zippers do not.

Despite these differences, there is a variety of tasks that can be solved by both approaches. Our goal was to compare the effectiveness of these two techniques. We chose two tasks where the ability to directly access a location inside a data structure and perform local updates is beneficial: traversing a tree in an arbitrary way and building a tree from a sorted sequence. Each task was implemented in Haskell, C++, and C♯, using the programming style common to that language.

Note that we compared the performance difference between these techniques, rather than performance across programming languages.

This work is organized as follows. In the next section, we discuss zipper representations. The third section looks at single location zippers in detail. The testing methodology, as well as the programming tasks themselves, are presented in the fourth section. Finally, the fifth section details our findings.

The source code used for performance testing is available online. [1]

2 Related Work

Huet's original zipper technique [5] relies on manually analyzing the data type and then defining the corresponding zipper structure. Listing 1 shows an example of such a zipper.

Listing 1. List and its zipper

```
data List a = Nil | Cons a (List a)

data ListZipper a = ListZipper
  { before :: List a
  , focus  :: a
  , after  :: List a
  }
```

This approach becomes problematic when working with heterogeneous data structures (a structure containing elements of multiple types), or when working with many different zipper representations.

For heterogeneous collections, Huet's zipper can be used to represent only the positions of one type of elements, which is quite limiting. Adams [2] shows how to build a zipper for heterogeneous collections by using generic programming techniques based on the ideas of Lämmel and Peyton Jones [7]. Another benefit of this approach is that new data structures do not need a custom implementation of the zipper structure, which reduces the boilerplate that is usually present when dealing with zippers.

Instead of using an explicit data structure, the zipper can be represented as a suspended traversal of the original structure. Kiselyov [6] uses delimited continuations to implement suspended computation to great effect. Applications include creating a zipper for any type that is a member of Haskell's `Traversable` type class, zipping two data structures for side-by-side comparison and various operations on zippers capable of representing multiple positions.

Another way of dealing with the boilerplate code is to automate the generation of auxiliary data structures. For each regular algebraic data type, the type of one-hole contexts can be obtained by differentiating the original type, not unlike differentiation in calculus [8,1]. A zipper is obtained by combining an element of the given type and the one-hole context. As a result, the zipper does

[1] `https://github.com/vituscze/performance-zippers`

not need to be defined for each data structure separately [4]. We explore this technique in more detail in the following section.

Ramsey and Dias [11] use zippers to represent control flow graphs in a low-level optimizing compiler. The compiler is written in OCaml, giving the opportunity to use an imperative approach based on mutable pointers as well as a purely functional approach based on zippers. As part of their analysis, the authors also include performance comparison. Zippers are shown to perform slightly better than mutable pointers.

3 Zipper

Huet's zipper is based on the idea of pointer reversal. Reversing all pointers along the path from the root of the structure to a selected element called a *focus* creates a structure that is rooted at the focus. This reversal has multiple advantages. Direct access to the focus allows its modification in constant time. Even in a purely functional setting where in-place modifications are not available, creating a copy of the focused node may be used instead. The rest of the structure stays intact and can be shared.

Similarly, accessing the parent and children of the focus can be done in constant time, which can be used to efficiently move the focus around the structure. Moving the focus is accomplished by reversing the pointers.

Huet shows how to represent this kind of pointer reversal as a purely functional structure. The nodes on the path from the root to the focus are stored in a list. Each element of the list must contain the values and substructures that are not descended into as well as the direction taken when moving towards the focus. The list is reversed, ensuring the parent of the focus is in the head position (instead of the root of the structure).

Listing 2. Binary tree and its zipper

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

data PathChoice a
  = NodeL a (Tree a) -- Focus is in the left subtree
  | NodeR (Tree a) a -- Focus is in the right subtree

data Context = Context
  (Tree a)        -- Left subtree of the focus
  (Tree a)        -- Right subtree of the focus
  [PathChoice a] -- Path to the root

data Zipper a = Zipper a (Context a)
```

Listing 2 defines a binary tree and its zipper. Listing 3 shows how to move the focus of this zipper to the parent node.

Listing 3. Focus movement

```haskell
up :: Zipper a -> Maybe (Zipper a)
up (Zipper _ (Context _ _ [])) = Nothing
up (Zipper x (Context l r (NodeL p pr:ps))) = Just $
  Zipper p (Context (Node l x r) pr ps)
up (Zipper x (Context l r (NodeR pl p:ps))) = Just $
  Zipper p (Context pl (Node l x r) ps)
```

However, since the zipper structure depends on the original data structure, these types and operations need to be defined for each structure separately. One way to solve this problem is to automate this process by using data type differentiation [8,1]. We give a brief overview of this technique here.

An *algebraic data type* is a data type defined as a combination of products (tuples) and sums (variants), potentially in a recursive way. Algebraic data types that do not change the parameters in recursive occurrences are known as *regular types*. For these types, the derivative is defined as follows.

$$\begin{aligned}
\partial_x(0) &= 0 && \text{(empty type)}\\
\partial_x(1) &= 0 && \text{(unit type)}\\
\partial_x(y) &= 0 && \text{(type variable)}\\
\partial_x(x) &= 1 && \text{(type variable)}\\
\partial_x(F+G) &= \partial_x(F) + \partial_x(G) && \text{(sum type)}\\
\partial_x(F\times G) &= \partial_x(F)\times G + F\times\partial_x(G) && \text{(product type)}\\
\partial_x(\mu y.F) &= [\mu y.F/y]\partial_x(F)\times \text{List }([\mu y.F/y]\partial_y(F)) && \text{(least fixed point)}
\end{aligned}$$

The expression $[y/x]t$ denotes a capture-avoiding substitution. The variables can be introduced as parameters of the entire type (such as a in List a) or by the least fixed point operation, which is used to define recursive types. The resulting derivative is a type of *one-hole contexts*. A one-hole context is a structure that uniquely describes one position within the original data structure. Zipper then consists of a one-hole context together with an element of the original structure.

For example, a binary tree is a regular algebraic data type, and its zipper can be obtained by computing the derivative.

$$\begin{aligned}
\partial_a(\text{Tree } a) &= \partial_a(\mu x.1 + x\times a\times x)\\
&= [\text{Tree } a/x]\partial_a(1 + x\times a\times x)\times \text{List }([\text{Tree } a/x]\partial_x(1 + x\times a\times x))\\
&= [\text{Tree } a/x](x\times x)\times \text{List }([\text{Tree } a/x](a\times x + x\times a))\\
&= \text{Tree } a\times \text{Tree } a\times \text{List }(a\times \text{Tree } a + \text{Tree } a\times a)
\end{aligned}$$

This derivative matches the definition of the tree context given in Listing 2.

The zippers used for performance testing in this work were based on algebraic data type differentiation. The resulting zipper representation was manually adjusted to provide better control over its strictness properties.

4 Performance Testing

To compare the performance of zippers and fingers, we implemented tree traversal and tree insertion in three different programming languages. The solutions based on zippers were implemented in Haskell. The solutions based on fingers were implemented in C++ and C♯. We included two imperative languages, one with manual memory management and the other with garbage collection, to check how the memory management model affected the relative performance. Unless specified otherwise, when discussing the imperative solutions, we are talking about the C++ solution.

The tasks were chosen to test the performance under two different memory allocation requirements. Tree traversal can avoid memory allocation altogether, while tree insertion cannot. Both tasks were tailored to finger-based solution, which was done to better represent the common use case of finger- and zipper-based approaches. In the following, we use the term *cursor* to refer to either a zipper or a finger.

4.1 Tree Traversal

The first task focuses on tree traversal. We are given a binary tree and a vector describing positions within the tree together with replacement values. The goal is to replace the specified elements of the original tree with the given values.

For cursor-based solutions, the input vector contains instructions that specify the movement of the cursor relative to its previous location. These movement instructions are interspersed with the replacement instructions. The element under the cursor is replaced with the given value whenever such instruction is encountered. As an example, replacing the left child of the root with 10 and right child with 20 would be represented as `fromList [Mov L, Set 10, Mov U, Mov R, Set 20]`.

We compared this approach to a solution where the replacement operation always starts at the root of the tree. The input vector describes the positions relative to the root of the tree. When a replacement value is encountered, the specified element is replaced, and the position is reset back to the root of the tree. The vector corresponding to the previous example would be `fromList [Mov L, Set 10, Mov R, Set 20]`. We do not allow `Mov U` as it is not necessary to describe a position.

This input format was chosen for better control over the spatial locality of the positions, which allowed us to observe how the cursor-based solutions behave depending on the average distance between positions. This task also allowed us to compare the performance of imperative solutions when memory allocation is not a factor.

Listing 4 specifies the desired behavior of both solutions. For simplicity, the specification does not handle incorrect inputs (such as positions outside the tree).

Listing 4. Tree traversal specification

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
data Dir = L | R | U

-- Replace an element at position determined by a list
-- of left/right directions.
```

```
replace :: a -> [Dir] -> Tree a -> Tree a
replace v []     (Node l _ r) = Node l v r
replace v (L:ds) (Node l x r) = Node (replace v ds l) x r
replace v (R:ds) (Node l x r) = Node l x (replace v ds r)
replace _ _      t            = t

data Cmd a = Mov Dir | Set a

-- Specifies the behavior of cursor-based solutions.
cursor :: Tree a -> Vector (Cmd a) -> Tree a
cursor tree = fst . foldl step (tree, [])
  where
    step (t, ds) (Mov U) = (t, tail ds)
    step (t, ds) (Mov d) = (t, d:ds)
    step (t, ds) (Set v) = (replace v (reverse ds) t, ds)

-- Specifies the behavior of root-based solutions.
root :: Tree a -> Vector (Cmd a) -> Tree a
root tree = fst . foldl step (tree, [])
  where
    step (t, ds) (Mov d) = (t, d:ds)
    step (t, ds) (Set v) = (replace v (reverse ds) t, [])
```

4.1.1 Imperative Solution

Listing 5 defines the structures used to represent the binary tree. Member functions are omitted for brevity.

Listing 5. Imperative binary tree (memory layout)

```
struct node_t {
  node_t* parent;
  node_t* left;
  node_t* right;
  int64_t value;
};

struct tree_t {
  node_t* root;
  node_t* finger;
};
```

Movement instructions are represented by integer constants to simplify the code. The input vector is processed by iterating over all its elements, applying the corresponding finger operation at each step. We evaluated the imperative solutions on a perfect binary tree of a specified depth.

4.1.2 Functional Solution

The functional solution is more involved. Since the task is meant for a cursor-based approach, the zipper lends itself to this problem naturally. However, the root-based solution presents a few problems that have to be addressed.

The tree and zipper definitions shown in Listing 6 follow the definitions from Listing 2, with the exception that each data type contains strictness annotations. Fields annotated with `!` are evaluated whenever the enclosing structure is, which ensures that the entire tree is fully evaluated at all times.

Listing 6. Binary tree and its zipper (with strictness annotations)

```
data Tree = Node !Tree !Int64 !Tree | Leaf

data Path
  = PathLeft  !Int64 !Tree  !Path
  | PathRight !Tree  !Int64 !Path
  | Nil

data Zipper = Zipper !Tree !Int64 !Tree !Path
```

As a consequence, the standard list type is replaced with a custom type. GHC is also instructed to unbox the integer fields, which is done to ensure that the cost of operating on boxed values does not have any impact on the performance. Unboxed vectors from the vector package are used to represent the input vector.

The zipper comes with operations that replace the focused element and move the focus left, right, and up. Processing the input vector is implemented as a strict left fold. The zipper is the accumulator value, and in each step, we apply zipper operation that corresponds to the element of the vector.

When starting from the root, replacing an element of the tree can be done easily with a recursive function that reads the vector in each recursive call and descends into the correct subtree. The problem is propagating the information about how many elements of the input vector were consumed so that the next operation can start from the correct position.

State Monad Solution To make sure the root-based solution is efficient, we compared a few ways of dealing with this issue. The obvious solution is to use a state monad. Note that laziness in the state is unwanted, and the strict monad version is about twice as fast. Analyzing GHC's core language [10], the monadic code was optimized away, and most parameters were unboxed. The only value that was not unboxed was the state returned by the replacement operation. Replacing the standard state monad with a handwritten one that uses unboxed integer did not improve the performance in a statistically significant way, however.

ST Monad Solution Another way of passing the state is to use the imperative `ST` monad. The standard implementation of `STRef` is limited to boxed types, which hugely degraded the performance. The standard references had to be replaced with unboxed references from the unboxed-ref package.

findIndices Solution Instead of propagating the new position via various versions of the state monad, the modification operation can be given hints on where to start. These hints can be provided by an auxiliary vector containing the positions where each descent starts. We can create this vector by using the `findIndices` function from the vector package.

findIndex Solution This approach has a few issues. The input vector has to be traversed twice, and the auxiliary vector has to be stored in the memory. We can avoid the memory allocation by computing the hints as needed, instead of all at once, by using the `findIndex` function. To measure the impact of this double traversal, we also implemented a function where the vector of hints is a part of its input. The vector is precomputed, and its time requirements were not included in the comparison.

Much like the imperative solution, all functional solutions were evaluated on a perfect binary tree of a specified depth.

4.2 Tree Insertion

The second task focuses on tree building. Building a search tree can be done much more efficiently when the input sequence is sorted. The search for a new insertion point can be skipped since it will always be the leftmost or the rightmost node (depending on the order of the input sequence). This node can be tracked with a finger that is updated each time a new element is inserted. The same can be done with a zipper, although the standard tree insert operation cannot be reused.

To test a zipper for a different structure, we chose 2-3 trees [3] for this task. The structure is redundant: all data is kept in the leaf nodes, and internal nodes contain the minimum of their right subtree (and of the middle subtree, whenever applicable). The task is then to build a redundant 2-3 tree from a descending sequence of the given length. The standard solutions start from the root of the tree when looking for the insertion point. The cursor-based solutions start in the leftmost node and perform no additional search.

4.2.1 Imperative Solution Listing 7 defines the structures used to represent the 2-3 tree. Member functions are omitted for brevity.

Listing 7. Imperative 2-3 tree (memory layout)

```
struct node_t {
  std::array<int64_t, 2> values;
  std::array<node_t*, 3> children;
  node_t* parent;
  bool is_two_node;
};

struct tree_t {
  node_t* root;
  node_t* last_inserted;
};
```

Tree insertion follows the standard algorithm. We obtain the insertion point and attempt to insert the value into the corresponding leaf node. When the leaf node is full, we allocate a new node and redistribute all the values from the original node. After this split, we are left with a two-node and a three-node. We take the middle value and the right node and attempt to insert them into the parent node. We repeat this process until no split occurs or the root is reached. Note that splitting an inner node results in two two-nodes because the middle value does not need to be duplicated.

The split operation puts the inserted value into a two-node when inserting values in descending order. As a result, leaf nodes are only split every second insertion. The implementation could be improved slightly to also provide similar benefit for insertion in ascending order.

We also tried the following variations of the tree operations: non-recursive destructor, split operation that allocates the left node, and recursive root-based insertion. The impact on the performance was either detrimental or statistically insignificant.

We repeatedly insert values into the tree in descending order and measure the time taken. In the case of C++ solution, this measurement also includes the time spent on deallocation, giving a fairer comparison to the languages with garbage collection.

4.2.2 Functional Solution Listing 8 shows a definition of 2-3 trees with strictness annotations.

Listing 8. Functional 2-3 tree

```
data Tree
  = Leaf
  | Node2 !Tree !Int64 !Tree
  | Node3 !Tree !Int64 !Tree !Int64 !Tree
```

To insert a value into the tree, we recursively insert it into the correct subtree. The result of this insertion is either one subtree or two subtrees and a value. The first case is handled by replacing the corresponding subtree; the second case indicates that a split occurred and is handled similarly to the imperative solution.

To obtain a zipper, we compute the derivative of a parametrized version of the 2-3 tree type.

$$\begin{aligned}
F &= 1 + ax^2 + a^2x^3 \\
\partial_a(F) &= x^2 + 2ax^3 \\
\partial_x(F) &= 2ax + 3a^2x^2 \\
\partial_a(\text{Tree } a) &= \partial_a(\mu x.F) \\
&= [\text{Tree } a/x]\partial_a(F) \times \text{List } ([\text{Tree } a/x]\partial_x(F)) \\
&= ((\text{Tree } a)^2 + 2a(\text{Tree } a)^3) \times \text{List } (2a(\text{Tree } a) + 3a^2(\text{Tree } a)^2)
\end{aligned}$$

If the focus is in a two-node, then there is only one choice for the element, and the context is given by the two subtrees. This case is represented by (Tree $a)^2$. If the focus is in a three-node, there are two choices for the element (left or right).

The context is given by the three subtrees and the value of the element that was not chosen, or $2a(\text{Tree } a)^3$.

The path also distinguishes between two-nodes and three-nodes. In the case of a two-node, there are two choices for the focus position (left or right subtree). The context is given by the value of the element and the other subtree. This case is represented by $2a(\text{Tree } a)$. In the case of a three-node, there are three choices for the focus (left, middle, or right subtree) and the context is given by the values of two elements and the other two subtrees, resulting in the final term $3a^2(\text{Tree } a)^2$.

Since the insertion algorithm only needs to know the position of the last insertion and not the particular element, we simplify the zipper by removing this choice point. The type variable is replaced with `Int64` and the list type is replaced with a custom strict list. Listing 9 shows the resulting type.

Listing 9. 2-3 tree zipper

```
data Nonempty
  = Nonempty2 !Tree !Int64 !Tree
  | Nonempty3 !Tree !Int64 !Tree !Int64 !Tree

data PathChoice
  = Path2L !Int64 !Tree
  | Path2R !Tree !Int64
  | Path3L !Int64 !Tree !Int64 !Tree
  | Path3M !Tree !Int64 !Int64 !Tree
  | Path3R !Tree !Int64 !Tree !Int64

data Path = Nil | Cons !PathChoice !Path
data Zipper = Zipper !Nonempty !Path
```

Inserting a value by using a zipper more closely resembles the imperative solution. The key difference is that instead of pointers to parent nodes, the zipper contains a list of choices along the path from the root to the focus. Instead of descending into the tree, the zipper-based insertion needs to descend into this list.

When a node splits and we attempt to add the value and one of the freshly split nodes to the parent node, we also need to include information about the position of the split node in relation to the value. This position is necessary to reconstruct the extra information contained in the zipper. The imperative solution assumes the split node is always to the right.

Much like the imperative solution, we repeatedly insert values into the tree in descending order and measure time taken.

5 Results

All experiments were performed on Intel Core i7-4750HQ processor with 24 GB of main memory under Windows 10 operating system. Each program was compiled with the highest available level of compiler optimizations, and in the case of GHC, LLVM backend was used for code generation. Garbage collectors were allowed to only run in a single thread. Each solution was executed with an increasing number of iterations until a time limit of three minutes was reached. The measured times were normalized to one iteration. Mean execution time, as well as standard deviation, were computed. Error bars represent one standard deviation. The raw measurements are available online. [2]

5.1 Tree Traversal

The input files were generated by randomly picking 1,000,000 elements out of a perfect binary tree with 20 levels and outputting the path between them. We evaluated the tree traversal in four scenarios which were obtained by biasing the random generator towards particular areas of the tree: no bias, bottom bias, right bias, and bottom-right bias. One input file was generated for each scenario to ensure any performance differences were not due to different input data.

This comparison only includes results of the fastest variant of the functional root-based solution: the approach based on `findIndex`. Its precomputed version is only marginally faster, showing that the double traversal has a low impact on the performance. The state and `ST` solutions are much slower. Interestingly, the `ST` solution is slightly slower than the purely functional state solution. Full comparison of these variants can be found in Figure 6.

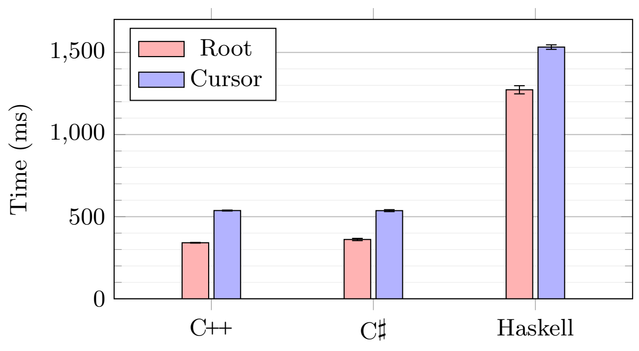

Fig. 1. Tree Traversal Performance (no bias)

When the spatial locality is low (Figures 1 and 2), the root-based solutions show a clear advantage over the cursor-based solutions. The relative gains of the root-based approach are in the range of 50% to 60% for the imperative solutions and around 20% for the functional solution.

When the spatial locality is high (Figure 3 and Figure 4), the cursor-based solutions take over. In the case of the right bias, C++ reaches 150% speedup, C♯

[2] `https://github.com/vituscze/performance-zippers/blob/master/data.csv`

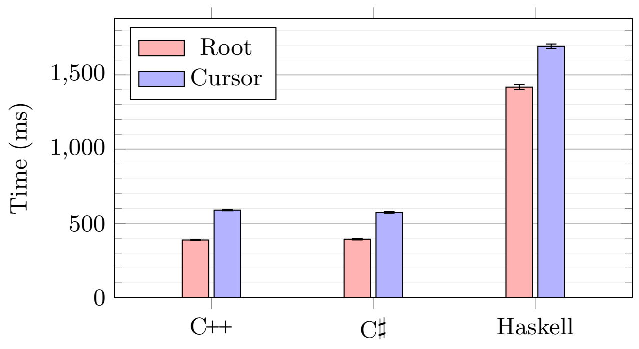

Fig. 2. Tree Traversal Performance (bottom bias)

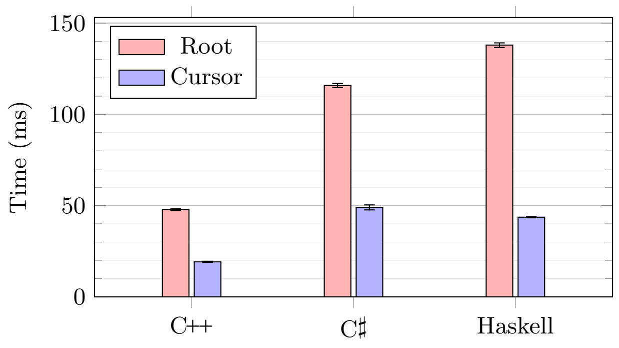

Fig. 3. Tree Traversal Performance (right bias)

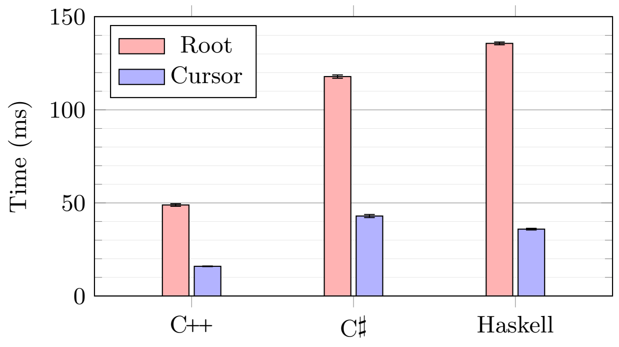

Fig. 4. Tree Traversal Performance (bottom-right bias)

135% and Haskell 220%. Bottom-right bias increases this gap even more. C++ reaches 205% speedup, C♯ 175% and Haskell 280%.

Notice that the root-based solutions also show a considerable performance boost when the input data has high spatial locality. This boost is a consequence of cache-friendly memory access pattern. In all scenarios, the zipper-based solution exhibits smaller performance losses (low spatial locality) and higher performance gains (high spatial locality) when compared to the finger-based solutions.

5.2 Tree Insertion

Evaluating insertion into a 2-3 tree was done by repeatedly constructing a tree containing 10,000,000 elements. The ordered sequence was not part of the input. Instead, the elements of this sequence were generated on the fly and inserted into the tree directly, without any auxiliary structure. As mentioned earlier, this task compared fingers and zippers in an environment where memory allocation is necessary. For this reason, the C++ solution also evaluated the time it took to deallocate the structure, giving a better comparison with C♯ and Haskell.

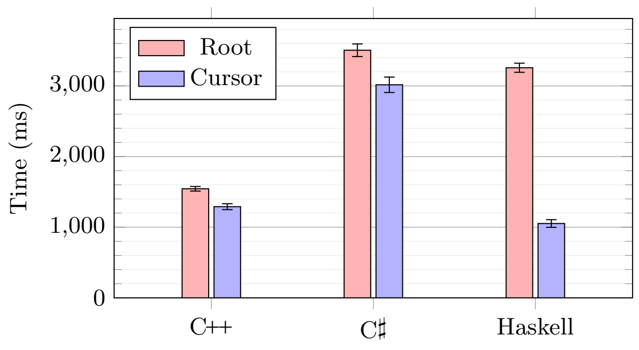

Fig. 5. 2-3 Tree Insertion Performance

The results are shown in Figure 5. All three solutions show a preference for cursor-based approaches. In C++ and C♯, the finger-based insertion is roughly 20% faster than the root-based insertion. In Haskell, the zipper-based insertion is 210% faster.

Note that both the root-based and finger-based insertion allocate $\mathcal{O}(1)$ nodes (amortized) per insertion in imperative languages. The root-based functional solution needs to copy the path from the root to the insertion point, leading to $\mathcal{O}(\log n)$ new nodes per insertion. The zipper-based insertion, therefore, not only avoids the cost of finding the insertion point but also leads to significantly reduced allocation count.

Comparing the C++ and C♯ results did not point to memory management as a major factor. Reducing the size of the tree (by performing fewer insertions) showed that the gap between C++ and C♯ decreased slightly, which hints to a minor performance benefit from using garbage collection.

The C++ solution could be further optimized by using a memory pool instead of the standard `new` and `delete` operators. However, we did not want to deviate from the standard memory management models. In a similar vein, we decided against fine-tuning the garbage collector parameters for the Haskell and C♯ solutions.

6 Conclusion

While zippers lack the flexibility and ease of use of mutable pointers, they are nevertheless a powerful tool when working with purely functional data structures. However, it was unclear whether zippers offer the same performance benefit as the imperative approach.

We compared fingers and zippers in two scenarios: arbitrary tree traversal and tree insertion. The first test measured the effectiveness of zippers when its imperative counterpart does not have to allocate memory. This test focused on fast access to a selected element as well as the ability to move the focus. The second test considered the case where both the imperative and functional approaches need to allocate memory. It focused on the pointer reversal aspect of zippers.

We provided evidence that when zippers are used in a functional setting, they offer higher performance gains compared to mutable pointers used in an imperative setting. More importantly, zippers provide this gain without undermining the benefits of purely functional data structures. We hope that this work encourages functional programmers to use zippers before reaching for imperative techniques when optimizing their code.

References

1. Abbott, M., Altenkirch, T., McBride, C., Ghani, N.: ∂ for data: Differentiating data structures. Fundam. Inf. 65(1-2), 1–28 (Aug 2004)
2. Adams, M.D.: Scrap your zippers: A generic zipper for heterogeneous types. In: Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming. pp. 13–24. WGP '10, ACM, New York, NY, USA (2010)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press, 3rd edn. (2009)
4. Hinze, R., Jeuring, J., Löh, A.: Type-indexed data types. Sci. Comput. Program. 51(1-2), 117–151 (May 2004)
5. Huet, G.: The zipper. J. Funct. Program. 7(5), 549–554 (Sep 1997)
6. Kiselyov, O.: Generic zipper: the context of a traversal. `http://okmij.org/ftp/continuations/zipper.html` (2015)
7. Lämmel, R., Peyton Jones, S.: Scrap your boilerplate: A practical design pattern for generic programming. In: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation. pp. 26–37. TLDI '03, ACM, New York, NY, USA (2003)
8. McBride, C.: The derivative of a regular type is its type of one-hole contexts (extended abstract). `http://strictlypositive.org/diff.pdf` (2001)
9. Mehta, D., Sahni, S.: Handbook Of Data Structures And Applications, chap. 11. Finger Search. Chapman & Hall/CRC (2004)
10. Peyton Jones, S., Santos, A.: A transformation-based optimiser for Haskell. Sci. Comput. Program. 32(1-3), 3–47 (Sep 1998)
11. Ramsey, N., Dias, J.: An applicative control-flow graph based on Huet's zipper. Electron. Notes Theor. Comput. Sci. 148(2), 105–126 (Mar 2006)

A Additional Charts

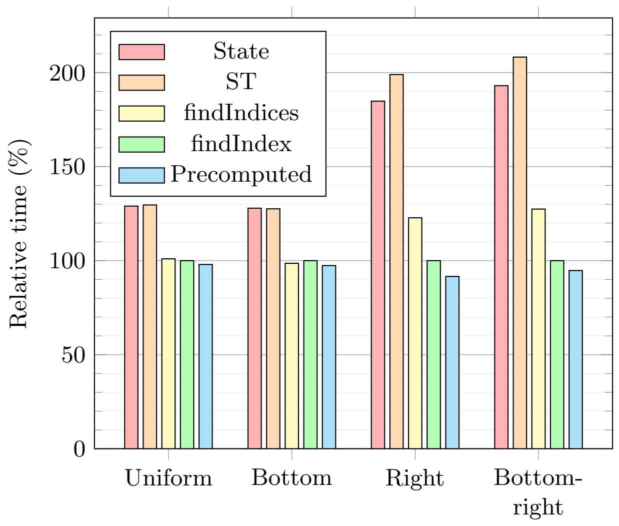

Fig. 6. Tree Traversal Performance (Haskell)