Finger Trees Explained Anew, and Slightly Simplified (Functional Pearl)

Koen Claessen
koen@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Abstract

We explicitly motivate the subtle intricacies of Hinze and Paterson's Finger Tree datastructure, by step-wise refining a naive implementation. The result is a new explanation of how Finger Trees work and why they have the particular structure they have, and also a small simplification of the original implementation.

CCS Concepts: • **Theory of computation → Data structures design and analysis**; • **Software and its engineering → Functional languages**.

Keywords: datastructures, finger trees, amortized complexity, functional pearl

ACM Reference Format:
Koen Claessen. 2020. Finger Trees Explained Anew, and Slightly Simplified (Functional Pearl). In *Proceedings of the 13th ACM SIGPLAN International Haskell Symposium (Haskell '20), August 27, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3406088.3409026

1 Introduction

In 2006, Hinze and Paterson [3] introduced an astonishingly beautiful and versatile data structure that they dubbed "Finger Trees". Finger Trees are nowadays used as the sequence datatype *Seq* in the standard Haskell library *Data.Sequence*, providing a highly efficient implementation of sequences.

We can see the appeal of Finger Trees used as sequences in Fig. 1, which displays the time complexities of some of the basic sequence operations they support. For comparison, the time complexities of regular Haskell lists are also given in the table. Most impressive are the amortized complexities (where we compute the average complexity of an operation appearing in a sequence of n operations) – they strictly improve the time complexities over regular lists.

	regular lists	*Seq* worst-case	*Seq* amortized
head	$O(1)$	$O(1)$	$O(1)$
cons	$O(1)$	$O(\log n)$	$O(1)$
tail	$O(1)$	$O(\log n)$	$O(1)$
last	$O(n)$	$O(1)$	$O(1)$
snoc	$O(n)$	$O(\log n)$	$O(1)$
init	$O(n)$	$O(\log n)$	$O(1)$
$(+\!\!+)$	$O(n)$	$O(\log n)$	$O(\log n)$
length	$O(n)$	$O(1)$	$O(1)$
$(!!)$	$O(n)$	$O(\log n)$	$O(\log n)$

Figure 1. Time complexities of the operations

```
data Seq a   = Nil
             | Unit a
             | More (Some a) (Seq (Tuple a)) (Some a)
data Some a  = One a
             | Two a a
             | Three a a a
             | Four a a a a
data Tuple a = Pair a a
             | Triple a a a
```

Figure 2. Hinze and Paterson's datatype for *Seq*

As Hinze and Paterson noted in their paper, Finger Trees were not the first sequence data structure with these or similar worst-case and amortized time complexities. In the paper, after reviewing a few alternatives, they say "*[t]he biggest advantage of 2-3 finger trees is the simplicity of the data structure [...]*".

Finger Trees are certainly simpler than the alternative sequence datatypes mentioned in their paper. But Finger Trees are far from simple... Take a look at Fig. 2, which shows the original implementation that Hinze and Paterson used[1]. It involves an instance of non-regular recursion in the

[1]The original paper used different names for some of these types and constructor functions; we instead used the names that fit with the explanations in this paper.

Haskell '20, August 27, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8050-8/20/08…$15.00
https://doi.org/10.1145/3406088.3409026

Seq type (which is indeed needed), as well as two different helper types, one (*Some*) for sequences of length 1–4, and one (*Tuple*) for sequences of length 2–3.

A natural question is "why are these two different helper types with these particular length intervals needed?". Hinze and Paterson do not answer this question directly, although they say "*[w]e shall [...] allow[...] between one and four subtrees at each level*" and then claim that "*this [...] provides just enough slack to buffer deque operations efficiently.*"

The material in this paper was created in an attempt to confirm that their chosen implementation for Finger Trees was "just enough". The idea was that we would start with a very simple implementation, and make step-by-step incremental improvements as needed, until we would arrive at an implementation that satisfies all the time complexities mentioned in Fig. 1.

What we actually arrived at was something slightly less complicated! So, the first contribution of the paper is a slightly simpler implementation of Finger Trees. The second, more important contribution, is an explanation of why this implementation actually needs the datatypes it has. The third contribution is an alternative proof of the (sequential) amortized complexity analysis of the operations, which was done using an automated theorem prover. Our proof is simpler than Hinzes and Patersons, mostly because we prove a weaker result (although that claim is hard to make because their proof does not appear in their paper).

Before we start, let us establish some boundaries on the scope of this paper. This paper is not concerned with the implementation and analysis of the functions *length* and (!!) (and neither with other similar functions such as *take*, *drop* and *split*). These can be implemented in a very similar way to the original paper.

2 Try 0: Regular Lists

Let us start with implementing *Seq* using the same technique as regular lists in Haskell:

$$\begin{aligned}&\textbf{data}\ \mathit{Seq}\ a = \mathit{Nil}\\ &\qquad\qquad \mid \mathit{Cons}\ a\ (\mathit{Seq}\ a)\end{aligned}$$

All desired operations can be supported, but not with the desired complexity. For example the function *last* has to be recursive:

$$\begin{aligned}&\mathit{last} :: \mathit{Seq}\ a \to a\\ &\mathit{last}\ (\mathit{Cons}\ x\ \mathit{Nil}) = x\\ &\mathit{last}\ (\mathit{Cons}\ \_\ q) \quad = \mathit{last}\ q\end{aligned}$$

This leads to a time complexity of $O(n)$. The culprit is the fact that the last element of the sequence is hidden deep inside the data structure.

3 Try 1: Constant-Time Head and Last

If we look at the table with the complexities, we can see that the functions *head* and *last* both have time complexity $O(1)$. This suggests that the datatype for *Seq* should have the first and last elements readily accessible, i.e. not recursively nested inside the datastructure. This leads to our next try:

$$\begin{aligned}&\textbf{data}\ \mathit{Seq}\ a = \mathit{Nil}\\ &\qquad\qquad \mid \mathit{Unit}\ a\\ &\qquad\qquad \mid \mathit{More}\ a\ (\mathit{Seq}\ a)\ a\end{aligned}$$

We use two new constructors: *Unit* for singleton sequences, and *More* for sequences with at least two elements, directly exposing the first and last elements. The functions *head* and *last* can now be implemented as follows:

$$\begin{aligned}&\mathit{head} :: \mathit{Seq}\ a \to a\\ &\mathit{head}\ (\mathit{Unit}\ x) \quad\ = x\\ &\mathit{head}\ (\mathit{More}\ x\ \_\ \_) = x\\ &\mathit{last} :: \mathit{Seq}\ a \to a\\ &\mathit{last}\ (\mathit{Unit}\ y) \quad\ = y\\ &\mathit{last}\ (\mathit{More}\ \_\ \_\ y) = y\end{aligned}$$

They clearly have constant time complexity.

(We can see that the functions *cons* and *snoc* – and also *tail* and *init* – have the same time complexities. A reasonable choice is to implement *Seq* fully symmetrically w.r.t. the front and back, which is what we will do. In the remainder of the paper, we only show the implementations of *head*, *cons*, *tail*, (and (++)), and not *last*, *snoc*, and *init*, because the latter three are symmetrically implemented to the first three.)

What is wrong with the current implementation of *Seq*? The functions *cons* (and also *tail*) have to be recursive, e.g.:

$$\begin{aligned}&\mathit{cons} :: a \to \mathit{Seq}\ a \to \mathit{Seq}\ a\\ &\mathit{cons}\ x\ \mathit{Nil} \quad\ = \mathit{Unit}\ x\\ &\mathit{cons}\ x\ (\mathit{Unit}\ y) \quad\ = \mathit{More}\ x\ \mathit{Nil}\ y\\ &\mathit{cons}\ x\ (\mathit{More}\ y\ q\ z) = \mathit{More}\ x\ (\mathit{cons}\ y\ q)\ z\end{aligned}$$

This means that *cons* and *tail* have complexity $O(n)$. We need to come up with a solution for this.

4 Try 2: Logarithmic-Time Operations

As we can see in Fig. 1, all operations that do not have constant time complexity actually have logarithmic time complexity. How can this be achieved? The key idea that solves this was already introduced by Okasaki in 1998 [4] and called *implicit recursive slowdown*.

The intuition is this. Given a sequence type *Seq a*, we can make a new kind of sequence datatype as follows:

$$\textbf{type}\ \mathit{Seq2}\ a = \mathit{Seq}\ (a, a)$$

Elements in this new sequence datatype come in pairs, so to perform an operation on 2 elements in *Seq2*, one only has to

do perform one operation on *Seq*. So, *Seq2* can be thought of as twice as fast as *Seq*! Furthermore, if we manage to implement a sequence type *Seq a* where the recursive sequence type is of type *Seq2 a*, operations that have linear complexity in the recursion depth will have logarithmic complexity in the number of elements in the sequence! This is because each recursive subsequence has less than half the number of elements (pairs) as the original sequence has elements.

Here is an attempt at implementing this:

```
data Seq a = Nil
           | Unit a
           | More a (Seq (a, a)) a
```

However, our initial enthusiasm soon cools off when we try to implement the first non-trivial operation, *cons*:

```
cons :: a → Seq a → Seq a
cons x Nil              = Unit x
cons x (Unit y)         = More x Nil y
cons x (More y q z) =     -- impossible!
```

The last line is impossible to implement! The resulting sequence should have shape *More x . . z*, but we only have one *y* to add to the subsequence *q*, not two. (We can easily see that the idea is doomed from the beginning since sequences of size 3 are impossible to represent!)

5 Try 3: Fixing Cons and Tail

The problem is that, if *Seq* (*a*, *a*) is going to be the type of subsequences, we need *two* elements to add to it when we do add something. A straightforward way to fix this is to allow *More*-sequences to start and end with either one *or two* elements. In this way, single elements can always be paired up with another element before being added to the subsequence of pairs.

```
data Seq a    = Nil
              | Unit a
              | More (Some a) (Seq (a, a)) (Some a)
data Some a = One a
              | Two a a
```

Let us start by seeing how to implement *head*:

```
head :: Seq a → a
head (Unit x)                = x
head (More (One x)     _ _) = x
head (More (Two x _) _ _) = x
```

The function *cons* can now pair up elements before recursing:

```
cons :: a → Seq a → Seq a
cons x Nil                       = Unit x
cons x (Unit y)                  = More (One x) Nil (One y)
cons x (More (One y)     q u) = More (Two x y) q u
cons x (More (Two y z) q u) =
                                   More (One x) (cons (y, z) q) u
```

And *tail* does the opposite:

```
tail :: Seq a → Seq a
tail (Unit _)                = Nil
tail (More (Two _ x) q u) = More (One x) q u
tail (More (One _)     q u) = more0 q u
more0 :: Seq (a, a) → Some a → Seq a
more0 Nil (One y)     = Unit y
more0 Nil (Two y z)) = More (One y) Nil (One z)
more0 q    u          = More (Two x y) (tail q) u
                        where (x, y) = head q
```

Here, the helper function *more0* is used to take care of the special cases when the recursive subsequence is empty but the sequence itself is not.

The time complexities of *cons* and *tail* are $O(\log n)$, because the size of the subsequence they recurse on is (less than) half the size of their argument sequence.

(A side note: It is also possible to allow *zero or one* elements to appear at the start and end of *More*-sequences to fix the problem in this subsection. However, that would mean that *head* and *last* would not have constant time complexity anymore.)

So, we can support *head*, *cons*, and *tail* (and also *last*, *snoc*, and *init*) with their desired worst-case complexity. Unfortunately, the current design does not allow the implementation of (++) with sub-linear time complexity. Imagine computing a concatenation like this, for arbitrary subsequences *q1* and *q2*:

```
> More (One 1) q1 (One 7) ++More (Two 8 9) q2 (One 13)
```

The result should look something like:

```
More (One 1) (q1 ... 7, 8, 9 ... q2) (One 13)
```

The problem is that we want to add *three* elements 7, 8, 9 in between two subsequences *q1* and *q2*, that only consist of pairs. There is no way to do this other than to destruct all pairs in at least one of those subsequences.

6 Try 4: Preparing for Append

The problem mentioned in the previous section can be fixed by allowing the tuples in the subsequences to represent an odd number as well as an even number of elements. Since we want the worst-case time complexity of many of our operations to be logarithmic, the number of elements per tuple should be at least 2. A natural choice is to allow tuples of size 2 or 3 in the subsequences:

```
data Seq a   = Nil
             | Unit a
             | More (Some a) (Seq (Tuple a)) (Some a)
data Some a = One a
             | Two a a
data Tuple a = Pair a a
             | Triple a a a
```

(We leave the sizes of the sequences in the *Some* datatype untouched, for now.) The functions *head* and *last* are unchanged, and *cons* only needs a small change (use the *Pair* constructor instead of $(-,-)$):

```
cons :: a → Seq a → Seq a
cons x Nil                    = Unit x
cons x (Unit y)               = More (One x) Nil (One y)
cons x (More (One y)   q u) = More (Two x y) q u
cons x (More (Two y z) q u) =
                              More (One x) (cons (Pair y z) q) u
```

The function *tail* is a little bit more intricate. What should happen when we compute:

```
> tail (More (One 1) (cons (Triple 2 3 4) q) (One 13))
```

We cannot just lift out the *Triple* 2 3 4 from the subsequence (using *head* and *tail* recursively) and place those three elements at the start, because we only have space for one or two elements. The only possibility is to turn the *Triple* into a *Pair* and only lift out the first component 2:

```
More (One 2) (cons (Pair 3 4) q) (One 13)
```

To implement this, the function *tail* does not have to change, but its helper function *more0* does:

```
more0 :: Seq (Tuple a) → Some a → Seq a
more0 Nil (One y)     = Unit y
more0 Nil (Two y z)) = More (One y) Nil (One z)
more0 q   u           =
  case head q of
    Pair x y      → More (Two x y) (tail q) u
    Triple x _ _ → More (One x) (map1 chop q) u
                   where chop (Triple _ y z) = Pair y z

map1 :: (a → a) → Seq a → Seq a
map1 f (Unit x)                = Unit (f x)
map1 f (More (One x)   q u) = More (One (f x))   q u
map1 f (More (Two x y) q u) = More (Two (f x) y) q u
```

We also introduce a new helper function *map1* that applies a function only to the first element of a sequence. Note that *map1* is not recursive, so it has constant time complexity. The function *tail* still has logarithmic time complexity, because there is only one recursive call, on the subsequence.

7 Try 4 Continued: Implementing Append

How about $(+\!\!+)$? Just like Hinze and Paterson, we are going to introduce a number of helper functions. It is going to be necessary to convert elements in the *Some* datatype to lists:

```
toList :: Some a → [a] 1..2
toList (One x)   = [x]
toList (Two x y) = [x, y]
```

We are also going to convert small lists of elements into lists of tuples:

```
toTuples :: [a] 2..6 → [Tuple a] 1..2
toTuples [ ]          = [ ]
toTuples [x, y]       = [Pair x y]
toTuples [x, y, z, w] = [Pair x y, Pair z w]
toTuples (x : y : z : xs) = Triple x y z : toTuples xs
```

We have carefully annotated the list types with the lengths that we are going to use the function on[2]. These functions have constant time complexity, as long as we guarantee not to violate the size requirements.

When implementing $(+\!\!+)$, it turns out to be useful to generalize and take an additional small list of elements as an extra argument, that should be inserted in between the two argument sequences. The generalized function is called *glue*, and the implementation of $(+\!\!+)$ becomes trivial using *glue*:

```
(++) :: Seq a → Seq a → Seq a
q1 ++ q2 = glue q1 [ ] q2
```

The implementation of *glue* looks as follows:

```
glue :: Seq a → [a] 0..2 → Seq a → Seq a
glue Nil       as q2       = foldr cons q2 as
glue q1        as Nil      = foldl snoc q1 as
glue (Unit x)  as q2       = foldr cons q2 (x : as)
glue q1        as (Unit y) = foldl snoc q1 (as ++L [y])
glue (More u1 q1 v1) as (More u2 q2 v2) =
  More u1
       (glue q1
             (toTuples (toList v1 ++L as ++L toList u2)
             q2)) v2
```

(Here, $+\!\!+_L$ stands for the append function on regular lists.) The first 4 cases are base cases, and use the already established *cons* and *snoc*, together with *foldr* and *foldl* to construct the result sequence. The time complexity is logarithmic, because the lists involved have bounded size. The last case calls *glue* recursively on *q1* and *q2*, combining the end *v1* of the first sequence, the middle elements *as*, and the start *u2* of the second sequence, into the new middle elements. The function *toTuples* converts a longer list of elements into a shorter list of tuples.

[2] *toTuples* is actually defined for all list lengths, except for size 1.

The specified list sizes are respected: The argument to *toTuples* combines three lists of sizes $^{1..2}$, $^{0..2}$, and $^{1..2}$, resulting in a list of possible sizes $2..6$, which is exactly what *toTuples* requires. It in turn produces a list of possible sizes $^{1..2}$, which is what *glue* accepts. The function $(+\!\!+)$ is the only function that will call *glue* with an empty list. All size annotations on the lists in this section are the smallest possible ones.

The result is a sequence datatype implementation where all the worst-case complexities from Fig. 1 are satisfied!

8 Try 5: Amortized Constant-Time Cons and Tail

The current implementation of *Seq* supports all operations, but the amortized complexity of *cons* and *tail* is not yet constant. Consider a sequence of the following shape:

```
More (One _) (More (One (Pair _ _))
                   (More (One (Pair _ _))
                         (More (One (Pair _ _))...)...)...) ...
```

Applying *tail* to this sequence will recurse all the way down, so it will take logarithmic time. The resulting sequence will have the shape:

```
More (Two _ _) (More (Two _ _)
                     (More (Two _ _)
                           (More (Two _ _)...)...)...) ...
```

Applying *cons* to this sequence will recurse all the way down, so it will take logarithmic time. The resulting sequence will have the same shape as the sequence we started with.

The problem is that for each possible shape that the start of a sequence can have, there is an operation (*cons* or *tail*) that may recurse in that case. However, if we look at the code, we see that *cons* and *tail* do not need to recurse in every case, in which case they take constant time.

Looking closer, *tail* recurses when the start of a sequence has minimal length (*One*), and *cons* recurses when the start of a sequence has maximal length (*Two*). What if starts (and ends) of sequences could have *three* different lengths (i.e. *One*, *Two*, and *Three*), so that there is a case where neither *tail* nor *cons* needs to recurse?

We can implement this idea as follows:

```
data Seq a   = Nil
             | Unit a
             | More (Some a) (Seq (Tuple a)) (Some a)
data Some a = One a
             | Two a a
             | Three a a a
data Tuple a = Pair a a
             | Triple a a a
```

This is our last try – the data structure design is finished! What is left is to implement the operations.

The function *head* gets one extra case:

```
head :: Seq a → a
head (Unit x)                  = x
head (More (One x)       _ _)  = x
head (More (Two x _)     _ _)  = x
head (More (Three x _ _) _ _)  = x
```

And so does *cons*:

```
cons :: a → Seq a → Seq a
cons x Nil                     = Unit x
cons x (Unit y)                = More (One x) Nil (One y)
cons x (More (One y)   q u)    = More (Two x y) q u
cons x (More (Two y z) q u)    = More (Three x y z) q u
cons x (More (Three y z w) q u) =
  More (Two x y) (cons (Pair z w) q) u
```

The last two lines are new. The last line is the recursive case, and it seems that we have a choice to either leave *Two x y* at the top and push *Pair z w* recursively (as we do here), or to leave *One x* at the top and push *Triple y z w* recursively. But in order to have a constant amortized time complexity, it is very important that we choose the first! The *Two* case will not trigger a recursive step in either *cons* or *tail* in a later call, and thus it is important to introduce it when we perform a recursive call right now. We shall see more details on this in the next section.

The function *tail* has to be reimplemented too. Mostly, this involves just adding extra cases to the existing functions. Here is *tail*:

```
tail :: Seq a → Seq a
tail (Unit _)                  = Nil
tail (More (Three _ x y) q u)  = More (Two x y) q u
tail (More (Two _ x)     q u)  = More (One x) q u
tail (More (One _)       q u)  = more0 q u
```

The helper function *more0* also gets an extra case:

```
more0 :: Seq (a, a) → Some a → Seq a
more0 Nil (One y)          = Unit y
more0 Nil (Two y z))       = More (One y) Nil (One z)
more0 Nil (Three y z w))   = More (One y) Nil (Two z w)
more0 q   u                =
  case head q of
    Pair x y     → More (Two x y) (tail q) u
    Triple x _ _ → More (One x) (map1 chop q) u
                   where chop (Triple _ y z) = Pair y z
```

We can see that *tail* also leaves a *Two* as the start of the sequence when it performs a recursive call. It may be tempting to remove the use of *chop* here, because we have room for three elements at the start, but that would destroy the constant amortized complexity! When the first element of

the subsequence is a *Triple*, we prefer not to perform any recursion at all.

For completeness, here is *map1*:

```
map1 :: (a → a) → Seq a → Seq a
map1 f (Unit x)                 = Unit (f x)
map1 f (More (One x)       q u) = More (One (f x))     q u
map1 f (More (Two x y)     q u) = More (Two (f x) y)   q u
map1 f (More (Three x y z) q u) =
                                  More (Three (f x) y z) q u
```

And finally, we have to adapt (⧺), which was implemented using some helper functions. Looking at the code, we can see that the only helper function that actually pattern matched on *Some* is *toList*:

```
toList :: Some a → [a] 1..3
toList (One x)       = [x]
toList (Two x y)     = [x, y]
toList (Three x y z) = [x, y, z]
```

The implementations of the other functions (*toTuples*, (⧺), and *glue*) do not have to be changed! But the list sizes annotations of *toTuples* and *glue* do of course change. The largest size of the "middle element list" of *glue* now becomes 3. This in turn means that the largest argument to *toTuples* now has 9 elements.

```
toTuples :: [a] 2..9 → [Tuple a] 1..3
glue     :: Seq a → [a] 0..3 → Seq a → Seq a
```

The function *toTuples* produces a list of maximum size 3, which is accepted by *glue* as an argument.

We have sucessfully adapted the implementations of our operations to the change of adding *Three* to the *Some* type. Do we now have the desired amortized complexities?

9 Amortized Complexity Analysis

In this section, we detail a simplified amortized complexity analysis. Here, we formally state what our proof obligations are. We have discharged these with an automated theorem prover.

We start by focusing on the functions *cons* and *tail*.

There exist various methods for showing amortized complexity of operations on a datatype. Okasaki discusses several such methods in his book [4]; most notably the *Banker's method*, which was chosen by Hinze and Paterson in their original paper (which sadly does not contain details of their proof), and the *Physicist's method*, which we will use here. The reason we choose the Physicist's method here is that it is generally considered simpler, and seems more amenable to formal verification.

In the Physicist's method, we start by constructing a function *pot* that assigns a natural number (the "potential") to every element of the datatype. The idea is that the places in the data structure that may cause certain operations to cost too much time will contribute to increasing the potential. Operations may use the potential that exists in a data structure as virtual time, but only if the operation itself contributes to decreasing the potential.

Given any choice of potential function, we can define the amortized time that a operation takes as follows:

$$\mathit{amortized\ time} \equiv \mathit{actual\ time} + \mathit{increase\ in\ potential}$$

In our case, we want to find a potential function that leads to the amortized time of *cons* and *tail* being constant.

So what should *pot* be? We already saw that occurrences of *One* and *Three* are problematic for *tail* and *cons*, respectively. What if we just count the number of occurrences of these "dangerous' constructors? We made sure to transform occurrences of *One* and *Three* into occurrences of *Two* whenever a recursive call was made, so that looks promising.

```
pot :: Seq a → Nat
pot Nil           = 0
pot (Unit _)      = 0
pot (More u q v) = dang u + pot q + dang v
  where
    dang (One _)     = 1
    dang (Two _ _)   = 0
    dang (Three _ _ _) = 1
```

How can we show that *cons* and *tail* have amortized constant time complexity with respect to this potential function? Let us first formalize the actual time that *cons* and *tail* take for a given argument. We define the function *consT* to compute the number of computation steps needed for *cons* to compute its result. We approximate the number of steps as 1 for any case that takes constant time, and furthermore add the number of recursive calls:

```
consT :: a → Seq a → Nat
consT _ Nil                        = 1
consT _ (Unit _)                   = 1
consT _ (More (One _)       _ _)   = 1
consT _ (More (Two _ _)     _ _)   = 1
consT _ (More (Three _ z w) q _)   = 1 + consT (Pair z w) q
```

We created the function definition for *consT* by copying the definition of *cons* and replacing the right-hand sides by the appropriate times. We can do the same for *tail* using a function *tailT*:

```
tailT :: Seq a → Nat
tailT (Unit _)                  = 1
tailT (More (Three _ _ _) _ _)  = 1
tailT (More (Two _ _)     _ _)  = 1
tailT (More (One _)       q u) = more0T q u
  where
```

```
more0T Nil _      = 1
more0T q   _      =
  case head q of
    Pair _ _      → 1 + tailT q
    Triple _ _ _ → 1
```

What is left to prove are the following two properties. They express that "actual time + increase in potential is less than a constant".

```
prop_AmortizedCons x q =
  consT x q + pot (cons x q) − pot q ⩽ 3
prop_AmortizedTail q =
  not (null q) ⟹
    tailT q + pot (tail q) − pot q ⩽ 2
```

How did we come up with these constants 3 and 2? Well, both properties can be QuickChecked [1], and we picked the lowest constants for which the properties succeeded. At further inspection of the counter examples, we can see that *cons* increases the potential from 0 to 2 when adding an element to a sequence of length 1, forcing the 3 in the property.

The above properties can both be proved by structural induction on the sequence q in a straightforward way. However, there are quite a large number of cases to be considered, which are easy to forget in a proof by hand. We used HIP [2], a translator for Haskell programs to first-order logic, and then used an automated theorem prover (in our case E [5]) in order to fully automatically discharge the proof obligations required in the induction proofs.

Append. A few words about (⧺) in the context of amortized complexity. Even though the worst-case and the amortized case are the same for (⧺), we need to include it in the analysis nonetheless. If we didn't, a danger may be that (⧺) increases the potential more than its time complexity allows! Fortunately, this is not the case. As the value of *pot* itself is in $O(\log n)$ (because its value is constant in the recursion depth), the amortized time of $O(\log n)$ for (⧺) will always be able to pay for the possible worst-case increase of potential.

Sequential vs. persistent amortized complexity. Hinze and Paterson make an even stronger claim in their paper: the amortized times of the operations do not only hold for a *sequential* usage of the datastructure (i.e. every result of an operation is only used once as the argument to another operation, as shown here), but also for a *persistent* usage (i.e. every result can be used an unlimited amount of times as an argument to any other operation). The proof for this claim needs a lot more detail than the proof we provide here, and in particular needs to involve an argument about laziness. We believe the same argument holds for the simpler version of finger trees presented here, but just like Hinze and Paterson, we do not provide a proof here either. Finding a simpler proof for the persistent amortized complexity is future work.

10 Discussion and Conclusion

We started this work with the motivation of understanding the detailed choices behind Hinze and Paterson's Finger Trees. From an initial naive implementation of sequences, we used step-by-step refinements to end up with a data structure that is in fact slightly simpler than Finger Trees! On the way, we developed a detailed explanation of Finger Trees and why they work, including a more detailed and complete proof of its amortized time complexity.

One less constructor. The constructor *Four* turns out to be not needed in our implementation. It seems that our use of *chop* in the function *tail* eliminated the need for *Four*. This certainly leads to simpler code, because many functions in the library (not only the ones mentioned here) have one less case to consider.

Are there other advantages? Our expectation is that there is no difference in run-time behavior, although we have not done any serious experiments to shine light on the matter. In fact, we believe that an implementation that allows larger *Tuple*s will behave faster in practice, because then there is less administration related to the recursion. Larger *Tuple*s means larger *Some*s. The detailed development in this paper opens up for new designs of Finger Trees that can generalize in three dimensions: (1) the size of the smallest sequences that are not represented using *More* (right now 0..1), (2) the size of sequences in *Some* (right now 1..3), and (3) the size of sequences in *Tuple* (right now 2..3). These choices all depend on each other. For example, if we aim for large tuples, we could choose (1) for small sequences 0..15, (2) for *Some* 8..24, and (3) for *Tuple* sequences of sizes 16..31.

Explanation. Hinze and Paterson start from 2-3 trees, mention that they have nice properties, and then add a "finger" (a direct reference to the first and last elements) to be able to implement *head* and *last* efficiently. This paper starts with adding a "finger" to a naive sequence type and then refines the result. These two explanations are complementary and both give insights that the other does not.

Symmetric treatment. Just like Hinze and Paterson, we chose to present the data structure in a one-sided way. The operations *last*, *snoc*, and *init* are indeed completely symmetrical. One worry the reader may have is about the amortized complexity analysis. What happens when the operations *cons* and *tail* are freely mixed with *snoc* and *init*? The analysis still holds, because all 4 operations use the same identical potential function *pot*.

Production code. The code in this paper is not meant to be production code. In a real implementation, we'd have to be more careful with memory usage. In one place particular, where *head*, *tail*, and *chop* are used in the implementation of *tail*, a potential space leak exists (that can be remedied by strictness annotations). The actual implementation in

Data.Sequence is very optimized and uses lots of strictness annotations.

Another spot that can be optimized is the use of the (short) helper lists in the implementation of *glue*. In fact, they can be eliminated completely by specializing those functions, resulting in a lot of nested cases. (This is also done in *Data.Sequence*.) For the sake of explanation, this was not done here. Implementing *glue* without these helper lists and without excessive case analyses is still future work.

QuickCheck. QuickCheck was used during the development of the various data structures that made it in the paper (as well as many many more versions that didn't!). Both functional correctness properties (linking the behavior of the operations to their corresponding list functions) as well as the amortized complexity properties proved extremely useful. Any kind of simplification that could not be made immediately results in a counter example.

Also, playing around with amortized complexity (different potential functions, different choices in the operations) was a breeze because the QuickCheck properties were there. For a while we thought we had developed an even simpler version of Finger Trees that had $O(\log n)$ amortized complexity for $(+\!\!+)$ but $O(n)$ worst-case! This turned out to be wrong, so we had to rethink.

The functions *consT* and *tailT* were implemented by hand, which introduces a risk that the versions that are used for testing and proving do not correspond to the versions in the implementation. An automated translation (e.g. going from *cons* to *consT*) would have been nice, but this remains future work.

Acknowledgments

Thanks to the anonymous referees for their valuable comments on an earlier draft of this paper. Thanks to John Hughes for an insightful discussion on amortized complexity. This research was partly funded by the Swedish Science Council (VR, "SyTeC", 2016-06204) and the Swedish Foundation for Strategic Research (SSF, "Octopi", RIT17-0023).

References

[1] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[2] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *International Conference on Automated Deduction*, pages 392–406. Springer, 2013.

[3] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006.

[4] C. Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[5] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In Pascal Fontaine, editor, *Proc. of the 27th CADE, Natal, Brasil*, number 11716 in LNAI, pages 495–507. Springer, 2019.