Purely Functional Data Structures

Advanced Data Structures
Final Work

Arnau Abella
Universitat Politècnica de Catalunya

June 3, 2020

1 Summary of the paper

The chosen paper, in fact a book, is *Purely Functional Data Structures* [Oka98].

The book has around 300 pages, 11 chapters and covers a wide variety of contents which, most of them, can't be given without previous content. For this reason, this work will focus only in chapter 5,6 and 7, which are, in my opinion, the most relevant introductory chapters. The list topics presented in these chapters are the following:

- Chapter 5. Fundamentals of Amortization
- Chapter 6. Amortization and Persistence via Lazy Evaluation
- Chapter 7. Eliminating Amortization

I'd love to talk about all different data structures mentioned in the book such as Binomial Heaps, Splay Heaps, Pairing Heaps, Bottom-Up Merge with Sharing, Hood-Melville Real-Time Queues, Binary Random-Access Lists, Skew Binomial Heaps, etc., and also about more advanced topics like *lazy rebuilding*, *numerical representations*, *data-strutural bootstrapping*, *implicit recursive slowdown*, etc. but, for today, we will work those 3 previously mentioned chapters and focus on a very simple and educative data structure, a *Queue*.

1.1 Fundamentals of Amortization

Given a sequence of operations, we may wish to know the running time of the entire sequence, but not care about the running time of any individual operation. Given a sequence of n operations, we wish to bound the total running time of the sequence by $O(n)$.

To prove an amortized bound, one defines the amortized cost of each operation and then proves that, for any sequence of operations, the total amortized cost of the operations is an upper on the total actual cost, i.e.,

$$\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i \tag{1}$$

where a_i is the amortized cost of operation i, t_i is the actual cost of operation i, and m is the total number of operations.

The key to proving amortized bounds is to show that expensive operations occur only when the accumulated savings are sufficient to cover the remaining cost.

Tarjan [Tar85] describes two techniques for analyzing amortized data structures: the *banker's method* and the *physicist's method*.

In the *banker's method*, the accumulated savings are represented as *credits* that are associated with individual locations in the data structure. The amortized cost of any operation is defined to be the actual cost of the operation plus the credits allocated by the operation minus the credits spent by the operation, i.e.

$$a_i = t_i + c_i - \bar{c_i} \tag{2}$$

where c_i is the number of credits allocated by the operation i and $\bar{c_i}$ is the number of credits spent by operation i. Every credit must be allocated before it is spent, and no credit may be spent more than once. Therefore, $\sum c_i \geq \sum \bar{c_i}$, which in turn guarantees that $\sum a_i \geq \sum t_i$. Proofs using the banker's method typically define a *credit invariant* that regulates the distribution of credits in such a way that, whenever an expensive operation might occur, sufficient credits have been allocated in the right locations to cover its cost.

In the *physicist's method*, one describes a function Φ that maps each object d to a real number called the *potential* of d. The function Φ is tipically chosen

so that the potential is initially zero and is always non-negative. Then, the potential represents a lower bound on the accumulated savings.

Let d_i be the output of operation i and the input of operation $i+1$. Then, the amortized cost of operation i is defined to be the actual cost plus the change in potential between d_{i-1} and d_i, i.e.,

$$a_i = t_i + \Phi(d_i) - \Phi(d_{i-1}) \tag{3}$$

The accumulated actual cost of the sequence of operations are

$$\begin{aligned}\sum_{i=1}^{j} t_i &= \sum_{i=1}^{j} (a_i + \Phi(d_{i-1}) - \Phi(d_i)) \\ &= \sum_{i=1}^{j} a_i + \sum_{i=1}^{j} (\Phi(d_{i-1}) - \Phi(d_i)) \\ &= \sum_{i=1}^{j} a_i + \Phi(d_0) - \Phi(d_j)\end{aligned}$$

1.1.1 Queues

We next illustrate the banker's and pysicist's methods by analyzing a simple functional implementation of the FIFO queue abstraction (see Listing 1).

Both `snoc` and `head` run in $O(1)$ worst-case time, but `tail` takes $O(n)$ time in the worst-case. However, we can show that `snoc` and `tail` both take $O(1)$ amortized time using either the banker's method or the physicist's method.

Using the banker's method, we maintain a credit invariant that every element in the rear list is associated with a single credit. Every `snoc` into a non-empty queue takes one actual step, and allocates a credit to the new element of the rear list, for an amortized cost of two. Every `tail` that does not reverse the rear list takes one actual step and neither allocates nor spends any credits, for an amortized cost of one. Finally, every `tail` that does reverse the rear list takes $m+1$ actual steps, where m is the length of the rear list, and spends the m credits contained by that list, for an amortized cost of $m+1-m=1$.

```haskell
{-# LANGUAGE Strict #-}
module Chapter5.BatchedQueue where

import           StrictList (List(..))
import qualified StrictList as SL

data Queue a = Queue (List a) (List a)

empty :: Queue a
empty = Queue Nil Nil

isEmpty :: Queue a -> Bool
isEmpty (Queue Nil _) = True
isEmpty (Queue _ _) = False

checkf :: Queue a -> Queue a
checkf (Queue Nil r) = Queue (SL.reverse r) Nil
checkf q = q

snoc :: a -> Queue a -> Queue a
snoc x (Queue f r) = checkf (Queue f (Cons x r))

head :: Queue a -> a
head (Queue Nil _) = error "empty"
head (Queue (Cons x _) _) = x

tail :: Queue a -> Queue a
tail (Queue Nil _) = error "empty"
tail (Queue (Cons _ f) r) = checkf (Queue f r)
```

Listing 1: Functional Queue

Using the physicist's method, we define the potential function Φ to be the length of the rear list. Then every `snoc` into a non-empty queue takes one actual step and increases the potential by one, for an amortized cost of two. Every `tail` that does not reverse the rear list takes one actual step and leaves the potential unchanged, for an amortized cost of one. Finally, every `tail` that does reverse the rear list takes $m + 1$ actual steps and sets the new rear list to `[]`, decreasing the potential by m, for an amortized cost of $m + 1 - m = 1$.

1.2 Amortization and Persistence via Lazy Evaluation

The amortized bounds break in the presence of persistence. In this chapter, we demonstrate how lazy evaluation can mediate the conflict between amortization and persistence, and adapt both the banker's and physicit's methods to account for lazy evaluation. We then illustrate the use of these new methods on the well-known *Queue*.

1.2.1 Execution Traces and Logical Time

Traditional methods of amortization break in the presence of persistence because they assume a unique future, in which the accumulated savings will be spent at most once. However, with persistence, multiple logical futures might all try to spend the same savings.

We model logical time with *execution traces*, which give an abstract view of the history of computation. An *execution trace* is a directed graph whose nodes represent operations of interest, usually just update operations on the data type in question. An edge from v to v' indicates that the operation v' uses some result of operation v. The *logical history* of operation v, denote $\hat{v}$, is the set of all operation on which the result of v depends (including v itself). In other words, $\hat{v}$ is the set of all nodes w such that there exists a path from w to v. A *logical future* of node v is any path from v to a terminal node. If there is more than one such path, then the node v has multiple logical futures.

Execution traces generalize the notion of *version graphs* [DSST89], which are often used to model the histories of persistent data structures.

1.2.2 Reconciling Amortization and Persistence

In this section, we show how the banker's and physicist's methods can be repaired by replacing the notion of accumulated savings with accumulated debt, where debt measure the cost of unevaluated lazy computations.

We must finde a way to guarantee that if the first application of f to x is expensive, then subsequent applications of f to x will not be. Without side-effects, this is impossible under *call-by-value* (i.e., strict evaluation) or *call-by-name* (i.e., lazy evaluation without memoization). Therefore, amortization cannot be usefully combined with persistence in languages supporting only these evaluation orders.

But now consider *call-by-need* (i.e., lazy evaluation with memoization). If x contains some suspended component that is needed by f, then the first

application of f to x forces the (potentially expensive) evaluation of that component and memoizes the result. Subsequent operations may then access the memoized result directly. This is exactly the desired behavior!

1.2.3 A Framework for Analyzing Lazy Data Structures

Historically, the most common technique for analyzing lazy programs has been to pretend that they are actually strict. We next describe a basic framework to support such analysis. In the remainder of this chapter, we adapt the banker's and physicist's methods to this framework, yielding both the first techniques for analyzing persistent amortized data structures and the first practical techniques for analysing non-trivial lazy programs.

We classify the costs of any given operation into several categories. The *unshared cost* of an operation is the actual time it would take to execute the operation under the assumption that every suspension in the system at the beginning of the operation has already been forced and memoized. The *shared cost* of an operation is the time that it would take to execte every suspension created but not evaluated by the operation. The *complete cost* of an operation is the sum of its shared and unshared costs. Note that the complete cost is what the actual cost of the operation would be if lazy evaluation were replaced with strict evaluation.

Realized costs are the shared costs for suspensions that are executed during the overall computation. *Unrealized costs* are the shared costs for suspensions that are never executed. The *total actual cost* of a sequence of operations is the sum of the unshared costs and the realized shared costs.

We account for shared costs using the notion of *accumulated debt*. Initially, the debt is zero, but every time a suspension is created, we increase the accumulated debt by the shared cost of the suspension (and any nested suspensions). Each operation then pays off a portion of the accumulated debt. The *amortized cost* of an operation is the unshared cost of the operation plus the amount of accumulated debt paid off by the operation. We are not allowed to force a suspensions until the debt associated with the suspension is entirely paid off.

We avoid the problem of reasoning about multiple logical futures by reasoning each logical future *as if it were the only one*. From the point of view of the operation that creates a suspension, any logical future that forces the suspension must itself pay for the suspension. Using this method, we sometimes pays off a debt more than once, thereby overestimating the total time required for a particular computation, but this does no harm and is a small

price to pay for the simplicity of the resulting analyses.

1.2.4 The Banker's Method

We adapt the banker's method to account for accumulated debt rather than accumulated savings by replacing credits with debits. Each debit represents a constant amount of suspended work. When we initially suspend a given computation, we create a number of debits proportional to its shared cost and associate each debit with a location in the object. If the computation is *monolithic* (i.e., once begun, it runs to completition), then all debits are usually assigned to the root of the result. On the other hand, if the computation is *incremental* (i.e., decomposable into fragments that may be executed independently), then the debits may be distributed among the roots of the partial results.

The amortized cost of an operation is the unshared cost of the operation plus the number of debits discharged by the operation. Note that the number of debits created by an operation is *not* included in its amortized cost. To prove an amortized bound, we must show that, whenever we access a location (possibly triggering the execution of a suspension), all debits associated with that location have already been discharged. Debits leftover at the end of the computation correspond to unrealized shared costs, and are irrelevant to the total actual cost.

Incremental functions play an important role in the banker's method because they allow debits to be dispersed to different locations in a data structure. Then, each location can be accessed as soon as its debits are discharged, without waiting for the debits at other locations to be discharged. In practice, this means that the initial partial result of an incremental computation can be paid for very quickly, and that subsequent partial results may be paid for as they are needed. Monolithic functions, on the other hand, are much less flexible.

The proof of this method is omitted for brevity.

1.2.5 Banker's Queues

We next develop an efficient persistent implementation of queues, and prove that every operation runs in $O(1)$ amortied time using the banker's method.

Now, waiting until the front list becomes empty to reverse the rear list does not leave sufficient time to pay for the reverse. Instead, we periodically *rotate* the queue by moving all the elements of the rear stream to the end of the

front stream. When should we rotate the queue ? Recall that `reverse` is a monolithic function. We must therefore set up the computation far enough in advance to be able to discharge all its debits by the time its result is needed. The `reverse` computation takes $|r|$ steps, so we allocate $|r|$ debits to account for its cost. The earliest the reverse suspensions could be forced is after $|f|$ applications of `tail`, so if we rotate the queue when $|r| \approx |f|$ and discharge one debit per operation, then we will have paid for the reverse by the time it is executed. In fact, we rotate the queue whenever r becomes one longer than f, thereby maintaining the invariant that $|f| \geq |r|$. Incidentally, this guarantees that f is empty only if r is also empty.

The complete code for this implementation appears in Listing 2.

To understand how this implementation deals efficiently with persistence, consider the following scenario. Let q_0 be some queue whose front and rear streams are both of length m, and let $q_i = tail(q_{i-1})$, for $0 < i \leq m + 1$. The queue is rotated during the first application of `tail`, and the reverse suspension is created by the rotation is forced during the last application of `tail`. This reversal takes m steps, and its cost is amortized over the sequence $q_1 \ldots q_m$.

Now, choose some branch point k, and repeat the calculation from q_k to q_{m+1}. Do this d times. How often is the reverse executed? It depends on whether the branch point k is before or after the rotation. Suppose k is after the rotation. In fact, suppose $k = m$ so that each of the repeated branches is a single `tail`. Each of these branches forces the reverse suspension, but they each force the *same* suspension, so the reverse is executed only once. Memoization is crucal here — without memoization, the reverse would be re-executed each time, for a total cost of $m(d + 1)$ steps, with only $m + 1 + d$ operations over which to amortize this cost. Memoization gives us an amortized cost of only $O(1)$ per operation.

It is possible to re-execute the reverse however. Simply take $k = 0$. Then the first `tail` of each branch repeats the rotation and creates a new reverse suspension. This new suspenson is forced in the last tail of each branch, executing the reverse. Because these are different suspensions, memoization does not help at all. The total cost of all the reversals is $m \cdot d$, but now we have $(m + 1)(d + 1)$ operations over which to amortize this cost, again yielding an amortized cost of $O(1)$ per operation.

```haskell
module Chapter6.BankersQueue where

data Queue a =
  Queue {-# UNPACK #-} !Int -- Length of f
        [a]                 -- f
        {-# UNPACK #-} !Int -- Length of r
        [a]                 -- r

empty :: Queue a
empty = Queue 0 [] 0 []

isEmpty :: Queue a -> Bool
isEmpty (Queue lenf _ _ _) = lenf == 0

check :: Queue a -> Queue a
check q@(Queue lenf f lenr r)
  | lenr <= lenf = q
  | otherwise    = Queue (lenf + lenr) (f ++ reverse r) 0 []

snoc :: a -> Queue a -> Queue a
snoc x (Queue lenf f lenr r) = check (Queue lenf f (lenr + 1)
↪  (x:r))

head :: Queue a -> a
head (Queue _ [] _ _) = error "empty"
head (Queue _ (x:_) _ _) = x

tail :: Queue a -> Queue a
tail (Queue _ [] _ _) = error "empty"
tail (Queue lenf (_:f') lenr r) = check (Queue (lenf - 1) f'
↪  lenr r)
```

Listing 2: Banker's Queue

By inspection, the unshared cost of every queue operation is $O(1)$. Therefore, to show that the amortized cost of every queue operation is $O(1)$, we must prove that discharging $O(1)$ debits per operation suffices to pay off every suspension before it is forced. In fact, only `snoc` and `tail` discharge any debits.

Let $d(i)$ be the number of debits on the ith node of the front stream and let $D(i) = \sum_{j=0}^{i} d(j)$ be the cumulative number of debits on all nodes up to and including the ith node. We maintain the following *debit invariant*:

$$D(i) \leq min(2i, |f| - |r|)$$

The $2i$ term guarantees that all debits on the first node of the front stream have been discharged (since $d(0) = D(0) \leq 2 \cdot 0 = 0$). The $|f| - |r|$ term guarantees that all debits in the entire queue have been discharged whenever the streams are of equal length, which happens just before the next rotation.

Theorem 1.1. `snoc` *and* `tail` *maintain the debit invariant by discharging one and two debits, respectively.*

Proof. Purely Functional Data Structures, pg.66. □

1.2.6 The Physicist's Method

Like the banker's method, the physicist's method can also be adapted to work with accumulated debt rather than accumulated savings. In the traditional physicist's method, one describes a potential function Φ that represents a lower bound on the accumulated savings. To work with debt instead of savings, we replace Φ with a function Ψ that maps each object to a potential representing an upper bound on the accumulated debt. Roughly speaking, the amortized cost of an operation is then the complete cost of the operation (i.e., the shared and unshared cost) minus the change in potential. Recall that an easy way to calculate the complete cost of an operation is to pretend that all computation is strict.

This method is explored further in the book using as an example *Binomial Heaps*, *Physicist's Queues*, *Bottom-Up Mergesort with Sharing* and *Lazy Pairing Heaps* which is omitted in this document for brevity.

The complete code for *Phisicist's Queues* appears in Listing 3.

```haskell
module Chapter6.PhysicistQueue where

import           StrictList (List(..))
import qualified StrictList as SL

import           Util.Suspension

data Queue a =
  Queue (List a)                -- w
        {-# UNPACK #-} !Int -- Length of f
        (Susp (List a))         -- f
        {-# UNPACK #-} !Int -- Length of r
        (List a)                -- r

empty :: Queue a
empty = Queue Nil 0 (S Nil) 0 Nil

isEmpty :: Queue a -> Bool
isEmpty (Queue _ lenf _ _ _) = lenf == 0

checkw :: Queue a -> Queue a
checkw (Queue Nil lenf f lenr r) = Queue (force f) lenf f lenr
↪  r
checkw q = q

check :: Queue a -> Queue a
check q@(Queue _ lenf f lenr r)
  | lenr <= lenf = checkw q
  | otherwise = let f' = force f
                in checkw (Queue f' (lenf + lenr) (S (f' <>
                ↪  SL.reverse r)) 0 Nil)

snoc :: a -> Queue a -> Queue a
snoc x (Queue w lenf f lenr r) = check (Queue w lenf f (lenr +
↪  1) (Cons x r))

head :: Queue a -> a
head (Queue Nil _ _ _ _) = error "empty"
head (Queue (Cons x _) _ _ _ _) = x

tail :: Queue a -> Queue a
tail (Queue Nil _ _ _ _) = error "empty"
tail (Queue (Cons _ w') lenf f lenr r) = check (Queue w' (lenf
↪  - 1) (S (SL.tail (force f))) lenr r)
```


Listing 3: Physicist's Queue

1.3 Eliminating Amortization

In some application areas, such as *real-time systems* or *interactive systems*, it is important to bound the running times of individual operations, rather than the sequences of operations. In these situations, a worst-case data structure will often be preferable to an amortized data structure, even if the amortized data structure is simpler and faster overall.

1.3.1 Scheduling

Amortized and worst-case data structures differ mainly in when the computations charged to a given operation occur. In a worst-case data structure, all computations charged to an operation occur during the operation. In an amortized data structure, some computations charged to an operation may actually occur during later operations.

In a lazy amortized data structure, any operation might take longer than the stated bounds. However, this only occurs when the operation forces a suspension that has been paid off, but that takes a long time to execute. To achieve worst-case bounds, we must guarantee that every suspension executes in no more than the allotted time.

Define the *intrinsic cost* of a suspension to be the amount of time it takes to force the suspension under the assumption that all other suspensions on which it depends have already been forced and memoized, and therefore each take only $O(1)$ time to execute.

The first step in converting an amortized data structure to a worst-case data structure is to *reduce the intrinsic cost of every suspension to less than the desired bounds*. Usually, this involves rewriting expensive monolithic functions to make them incremental, either by changing the underlying algorithms slightly or by switching from a representation that supports only monolithic functions, such as suspended lists, to one that supports incremental functions as well, such as streams.

Even if every suspension has a small intrinsic cost, some suspensions might still take longer than the allotted time to execute. This happens when one suspension depends on another suspension, which in turn depends on a third, and so on. If none of the suspensions have been executed previously, then forcing the first suspension results in a cascade of forces.

The second step in converting an amortized data structure to a worst-case data structure is to *avoid cascading forces by arranging that, whenever we force a suspension, any other suspensions on which it depends have already*

been forced and memoized. Then, no suspension takes longer than its intrinsic cost to execute. We accomplish this by systematically *scheduling* the execution of each suspension so that each is ready by the time we need it. The trick is to regard paying off debt as a literal activity, and to force each suspension as it is paid for.

We extends every object with an extra component, called the *schedule*, that, at least conceptually, contains a pointer to every unevaluated suspension in the object. Some of the suspensions in the schedule may have already been evaluated in a different logical future, but forcing these suspensions a second time does no harm since it can only make an algorithm run faster than expected, not slower. Every operation, in addition to whatever other manipulations it performs on an object, forces the first few suspensions in the schedule. The exact number of suspensions forced is governed by the amortized analysis; typically, every suspension takes $O(1)$ time to execute, so we force a number of suspensions proportional to the amortized cost of the operation. Depending on the data structure, maintaining the schedule can be non-trivial. For this technique to apply, adding a new suspension to the schedule, or retrieving the next suspension to be forced, cannot require more time than the desired worst-case bounds.

1.3.2 Real-Time Queue

As an example of this technique, we convert the amortized banker's queues of section 1.2.5 to worst-case queues. Queues such as these that support all operations in $O(1)$ worst-case time are called *real-time queues* [HM81].

In the original data structure, queues are rotated using `++` and `reverse`. Since reverse is monolithic, our first task is finding a way to perform rotations incrementally. This can be done by executing one step of the reverse for every step of the `++`. We define a function `rotate` such that

```
rotate xs ys a ≡ xs ++ reverse ys ++ a
```

Then

```
rotate f r [] ≡ f ++ reverse r
```

The extra argument, a, is called an *accumulating parameter* and is used to accumulate the partial results of reversing ys. It is initially empty.

Rotations occur when $|r| = |f| + 1$, so initially $|ys| = |xs| + 1$. This relationship is preserved throughout the rotation, so when xs is empty, ys contains a single element. The base case is therefore

```
rotate [] (y : []) a
  ≡ [] ++ reverse (y : []) ++ a
  ≡ y : a
```

In the recursive case,

```
rotate (x:xs) (y:ys) a
  ≡ (x:xs) ++ reverse (y:ys) ++ a
  ≡ x : (xs ++ reverse (y:ys) ++ a)
  ≡ x : (xs ++ reverse ys ++ (y:a))
  ≡ x : (rotate xs ys (y:a))
```

Puttng these cases together, we get

```
rotate' :: [a] -> List a -> [a] -> [a]
rotate'     [] (Cons y  _) a = y : a
rotate' (x:xs) (Cons y ys) a = x : rotate' xs ys (y:a)
```

Note that the intrinsic cost of every suspension created by rotate is $O(1)$.

Next, we add a schedule to the datatype. The original datatype was

```
data Queue a = Queue Int [a] Int [a]
```

We extend this type with a new field s of type `[a]` that represents a *schedule* for forcing the nodes of f. Besides adding s, we make two further changes to the datatype. First, to emphasize the fact that the nodes of r need not be scheduled, we change r from a stream to a list. Second, we eliminate the length fields. We no longer need the length fields to determine when r becomes longer than f — instead, we can obtain this information from the schedule. The new datatype is thus

```
data Queue a = Queue [a] (List a) [a]
```

With this representation, the major queue functions are simply

```
snoc :: a -> Queue a -> Queue a
snoc x (Queue f r s) = let r' = Cons x r in exec f r' s

head :: Queue a -> a
head (Queue (x:_) _ _) = x
```

```
tail :: Queue a -> Queue a
tail (Queue (_:f) r s) = exec f r s
```

The helper function `exec` executes the next suspension in the schedule and maintains the invariant that $|s| = |f| - |r|$ (which incidentally guarantees that $|f| \geq |r|$ since $|s|$ cannot be negative). `snoc` increases $|r|$ by one and `tail` decreases $|f|$ by one, so when `exec` is called, $|s| = |f| - |r| + 1$. If s is non-empty, then we restore the invariant simply by taking the tail of s. If s is empty, then r is one longer than f, so we rotate the queue. In either case, the very act of pattern matching against s to determine whether or not it is empty forces and memoizes the next suspension in the schedule.

```
exec :: [a] -> List a -> [a] -> Queue a
exec f r (x:s) = x `seq` Queue f r s
exec f r [] = let f' = rotate f r in Queue f' Nil f'
```

The complete code for this implementation appears in Listing 4

By inspection, every queue operation does only $O(1)$ work outside of forcing suspensions, and no operation forces more than three suspensions. Hence, to show that all queues operations run in $O(1)$ worst-case time, we must prove that no suspension takes more than $O(1)$ time to execute.

Only three forms of suspensions are created by the various queues functions.

- `Nil` is created by `empty` and `exec`. This suspension is trivial and therefore executes in $O(1)$.
- `y:a` is creted in both lines of `rotate` and is also trivial.
- `x:rotate xs ys (y:a)` is created in the second line of `rotate`. This suspensions allocates a `Cons`, builds a new suspension, and makes a recursive call to `rotate`, which pattern matches agaist the first node in xs and immediately creates another suspension. Of these actions, only the force inherent in the pattern match has even the possibility of taking more than $O(1)$ time. But note that xs is a suffix of the front stream that existed just before the previous rotation. The treatment of the schedule s guarantees that *every* node in that stream was forced and memoized prior to the rotation, so forcing this node again takes only $O(1)$ time.

Since every suspension executes in $O(1)$ time, every queue operation runs in $O(1)$ worst-case time.

```haskell
{-# LANGUAGE DeriveAnyClass #-}
{-# LANGUAGE DeriveGeneric  #-}
{-# LANGUAGE LambdaCase     #-}
{-# OPTIONS_GHC -fno-warn-orphans #-}
module Chapter7.RealTimeQueue where

import           Control.DeepSeq
import           GHC.Generics
import           StrictList      (List (..))

data Queue a =
    Queue [a]       -- f
          (List a)  -- r
          [a]       -- Schedule
  deriving (Eq, Generic, Generic1, NFData, NFData1)

empty :: Queue a
empty = Queue [] Nil []

isEmpty :: Queue a -> Bool
isEmpty = \case
  Queue [] _ _ -> True
  _            -> False

-- | Incremental f ++ reverse r
rotate :: [a] -> List a -> [a]
rotate f1 r1 = rotate' f1 r1 []
  where
    rotate' :: [a] -> List a -> [a] -> [a]
    rotate' [] (Cons y _) a      = y:a
    rotate' (x:xs) (Cons y ys) a = x : rotate' xs ys (y:a)
    rotate' _ _ _                = error "Inconceivable by invariant"

exec :: [a] -> List a -> [a] -> Queue a
exec f r (x:s) = x `seq` Queue f r s
exec f r [] =
  let f' = rotate f r
   in Queue f' Nil f'

snoc :: a -> Queue a -> Queue a
snoc x (Queue f r s) =
  let r' = Cons x r
   in exec f r' s

head :: Queue a -> a
head (Queue [] _ _)    = error "empty"
head (Queue (x:_) _ _) = x

tail :: Queue a -> Queue a
tail (Queue [] _ _)    = error "empty"
tail (Queue (_:f) r s) = exec f r s
```



```haskell
-- | Orphan instances
instance NFData a => NFData (List a)
instance NFData1 List
```

Listing 4: Real-Time Queue

2 The importance of Okasaki's work

Historically, the most common technique for analyzing lazy programs has been to pretend that they are actually strict. C. Okasaki yields both the first techniques for analyzing persistent amortized data structures and the first practical techniques for analyzing non-trivial lazy programs.

Purely Functional Data Structures also present new designing techniques of functional data structures such as *lazy rebuild*, a variant of *global rebuilding* [Ove83], *numerical represetation* for functional data structures, *data-structural bootstrapping* for functional data structures, and a framework called *implicit recursive slowdown*, a variant of *recursive slowdown*, that is based on lazy binary numbers instead of segmented binary numbers.

3 Experiment & Results

In order to empirically proof that the implementation from Real-Time Queues of section 1.3.2 runs in worst-case $O(1)$ time, we are going to implement the following experiment design:

1. Build a *Real-Time Queue* with a random permutation of operations (to prevent speculation of the performance) of size $n = \{1000, 10000, 100000\}$.
2. For each queue, analyze the performance of a sequence of `snoc/tail` operations.
3. Evaluate if all operations are executed in constant time, independently of the size or the order of operations.

Benchmarking a lazy programming language is an arduous task and it is easy to end up not measuring the right thing. For this reason, the implemenation of the experiment uses the following frameworks:

- *Criterion*: provides both a framework for executing and analysing benchmarks and a set of driver functions that makes it easy to build and run benchmarks, and to analyse their results.
- *QuickCheck*: is a library for random testing of program properties.

The complete code for this experiment appears in Listing 5.

```haskell
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TupleSections       #-}
{-# LANGUAGE TypeApplications    #-}
module Main where

import qualified Chapter7.RealTimeQueue as RTQ
import qualified Data.Foldable          as Foldable
import qualified Data.List              as List
import           Gauge
import           Prelude                hiding (tail)
import           Test.QuickCheck

-- Recall that gauge do not have a pretty html output.

main :: IO ()
main = defaultMain
  [ bgroup "Real-Time Queues"
      [ runBench 1000
      , runBench 10000
      , runBench 100000
      ]
  ]

runBench :: Int -> Benchmark
runBench size =
  env (generate $ genQueue @Int size) $ \q ->
    bgroup (show size)
      [ bench "snoc" $ whnf (applyN 100 (RTQ.snoc 0)) q
      --, bench "head" £ whnf RTQ.head q
      , bench "tail" $ whnf (applyN 100 RTQ.tail) q
      ]

-- | Returns a Queue of the given size.
--
-- FIXME: generating queues of size > 1000000 is slow.
genQueue
  :: Arbitrary a
  => Int
  -> Gen (RTQ.Queue a)
genQueue n
  | n < 0     = error "Size must be positive"
  | n == 0    = return RTQ.empty
  | otherwise = go 0 RTQ.empty
  where
    go m q
      | n == m    = return q
      | m == 0    = snoc q >>= go 1
      | otherwise = do
          (q', m') <- frequency [ (4, (,m + 1) <$> snoc q)
                                , (1, (,m - 1) <$> tail q)
                                ]
          go m' q'

    snoc q' = (`RTQ.snoc` q') <$> arbitrary

    tail q' = pure $ RTQ.tail q'

applyN :: Int -> (a -> a) -> a -> a
applyN n f = Foldable.foldr (.) id (List.replicate n f)
```

Listing 5: Benchmark implementation

```
benchmarked Real-Time Queues/1000/snoc
time                 500.5 ns   (487.8 ns .. 515.7 ns)
                     0.992 R²   (0.986 R² .. 0.997 R²)
mean                 516.0 ns   (507.8 ns .. 523.4 ns)
std dev              26.35 ns   (20.62 ns .. 34.16 ns)
variance introduced by outliers: 30% (moderately inflated)

benchmarked Real-Time Queues/1000/tail
time                 558.8 ns   (545.5 ns .. 578.7 ns)
                     0.993 R²   (0.990 R² .. 0.997 R²)
mean                 565.9 ns   (558.7 ns .. 573.6 ns)
std dev              23.98 ns   (19.88 ns .. 28.63 ns)
variance introduced by outliers: 23% (moderately inflated)

benchmarked Real-Time Queues/10000/snoc
time                 544.3 ns   (517.7 ns .. 587.2 ns)
                     0.967 R²   (0.934 R² .. 0.995 R²)
mean                 543.4 ns   (526.7 ns .. 571.6 ns)
std dev              71.97 ns   (38.76 ns .. 118.5 ns)
variance introduced by outliers: 75% (severely inflated)

benchmarked Real-Time Queues/10000/tail
time                 554.8 ns   (552.4 ns .. 558.6 ns)
                     0.999 R²   (0.999 R² .. 1.000 R²)
mean                 555.6 ns   (553.7 ns .. 558.1 ns)
std dev              7.229 ns   (4.559 ns .. 12.58 ns)

benchmarked Real-Time Queues/100000/snoc
time                 485.2 ns   (454.3 ns .. 520.2 ns)
                     0.969 R²   (0.951 R² .. 0.984 R²)
mean                 559.4 ns   (541.2 ns .. 595.7 ns)
std dev              70.19 ns   (33.82 ns .. 122.9 ns)
variance introduced by outliers: 65% (severely inflated)

benchmarked Real-Time Queues/100000/tail
time                 553.6 ns   (546.0 ns .. 561.6 ns)
                     0.999 R²   (0.997 R² .. 1.000 R²)
mean                 561.1 ns   (558.3 ns .. 568.6 ns)
std dev              11.32 ns   (3.197 ns .. 23.47 ns)
```

Figure 1: Benchmarks results

The results of the experiment appear in Figure 1. The *standard deviation* is low which is a good indicator that the values are near the mean of the set. The *real* execution time of these operations is inside the *confidence interval*. As the range is narrow, we can assume that the values are significant. Furthemore, the *coefficient of determination* (R^2) is near 1 which means that the observed outcomes replicate well the model with respect to the total variation.

The results display similar *mean* execution time for all operations independently of the size of the queue and the order of the operations. Applying a *Student's t-test* we can determine if all the operations do follow the same execution time.

From the previous results, we can conclude that the functional *Real-Time Queues* from section 1.3.2 has a $O(1)$ worst-case running time.

4 Conclusion

Reading *Purely Functional Data Structures* has been very educative. The book is well structured and easy to follow. It has helped me to get a deeper understanding on how to design and implement purely functional data structures. And, I also learnt *Standard M.L*, while reading the book, which is interesting from a programming language designer point of view.

Finally, it has helped me to get a better understanding of *lazy evaluation* which, in conjunction with *equational reasoning* and *denotational semantics*, are still not very explored techniques.

References

[DSST89] James R. Driscoll, Neil Sarnak, Daniel D. K. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, February 1989.

[HM81] Robert Hood and Robert Melville. Real-time queue operations in pure lisp. *Information Processing Letters*, 13:50–53, November 1981.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[Ove83] Mark H. Ovemars. The design of dynamic data structures. *LNCS*, 156, 1983.

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, April 1985.