Part 9

—

A Queue

CSCI 3110 Code

Summer 2015

1 Introduction

This code example is strictly optional (but rather informative) to read. I wanted to provide a purely functional queue that guarantees constant amortized time per operation no matter how we use the queue. This leads us into an area of algorithm design we did not touch on in class: amortized analysis of lazy, persistent data structures. The techniques required for such an analysis are really cool but beyond the scope of 3110. Most of what I discuss here, including the queue data structure itself, is taken directly from or based on Chris Okasaki's PhD thesis.

"Normally" (that is, when we want an imperative, ephemeral queue) we'd implement a queue as an array, a doubly-linked list or a singly-linked list with a tail pointer. None of these can be implemented purely functionally, at least not efficiently. The problem is that, since the data structures we create are immutable, every operation creates an entirely new data structure (which may share a significant portion of its representation with the data structure it was created from). Thus, there's nothing that prevents us from applying two or more operations to the same version of the data structure. In an imperative world, we call this a *persistent* data structure. In such a data structure, every update operation produces a new version and every operation is allowed to operate on any previous version of the data structure. This is in contrast to standard imperative data structures, whose operations usually update the data structures destructively: an update changes the representation of the data structure in place to produce the new version; nothing but the new version is accessible from this point on. This is known as an *ephemeral* data structure.

Okasaki models the manipulation of persistent data structures using logical futures and pasts. An operation o is in the logical future of another operation o' (and o' is in the logical past of o) if o's input depends on the output of o'. We also often say that o is in the logical future of any data structure produced by o'.

Since logical futures and pasts define a partial order on all the operations carried out by an algorithm, it should not come as a surprise that analyzing the total cost of a functional computation becomes non-trivial. Throw Haskell's lazy evaluation into the mix, and you have a real challenge on your hands. As

Okasaki argues, it's impossible to implement many functional data structures even with amortized costs of their operations without laziness. If we want worst-case bounds (something we won't explore here), then it's interesting to observe that a careful combination of strict and lazy evaluation is needed in many cases. Neither by itself can guarantee efficiency in the worst case.

2 An Unsuccessful Attempt at Implementing a Queue

The first attempt at implementing a queue functionally is as a pair of lists:

```
data Queue t = Queue {f, b :: [t]}
```

Then we would implement *enqueue* and *dequeue* as follows:

```
enqueue      :: Queue t → t → Queue t
enqueue q x = q {b = x : b q}

dequeue                 :: Queue t → Queue t
dequeue (Queue [ ] b) = dequeue (Queue (reverse b) [ ])
dequeue q               = q {f = tail (f q)}
```

It is not hard to prove that, if every operation in the algorithm has a single logical future, then the amortized cost per operation is constant. Multiple logical futures make this a poor implementation, however. For now, let's assume everything we do is evaluated strictly. Then consider a queue q produced using a sequence n *enqueue* operations: $q = \mathit{foldl\ enqueue\ emptyQueue\ xs}$, where xs contains n elements. This sequence of *enqueue* operations is easily seen to take $O(n)$ time, $O(1)$ per operation. Now let's follow this up with a *dequeue q* operations. No problem yet: We reverse the list b in $O(n)$ time and make the resulting list f. Since we have performed $n+1$ operations so far and their total cost is $O(n)$, we still have constant amortized cost per operation. But what if we performed n *dequeue q* operations. Each such operation would reverse b from scratch, at a cost of $\Theta(n)$. The total cost would therefore be $\Theta(n^2)$, but we have only $2n$ operations to pay for this cost. The amortized cost is no longer constant.

To see whether laziness helps (it doesn't with the queue implementation above), we need to dive a little deeper into how it's implemented. As already said, pretty much every operation creates a *thunk*, a piece of code that computes the result of the operation when it's needed, if it's needed. Once computed, it caches the result so it's not computed from scratch again. To make things concrete, you can think of a thunk as a record in memory consisting of a reference to a piece of code to compute the value of the thunk and a pointer to the result of this computation, initially null. When asking a thunk for its value, called "forcing the thunk", we first check its result pointer. If its not null, we simply return the result it points to. This takes constant time. If the pointer is null, we run the code referenced by the thunk and then update the pointer so it points to the result of the computation. Thus, next time we force the thunk, the pointer isn't null anymore and we simply return the result just computed at constant cost. Note that what I've just described is strictly imperative because we modify the result pointer of the thunk in place.

That shouldn't be surprising; we're talking about the implementation details of lazy evaluation here, and ultimately our computer only knows how to do imperative things.

Since accessing any value takes constant time and accessing a cached value through a thunk only requires us to reference one additional pointer, the cost of accessing the value of a thunk once it has been computed is only a (non-trivial) constant factor higher than accessing the value directly. Thus, we will think about evaluating a thunk as actually replacing the thunk with its value. Thus, we don't have to talk about thunks that have been forced already and thunks that haven't been forced yet. Instead, we can talk about values and thunks, which, because they still exist, haven't been forced yet.

One extra complication we may have to deal with is that some structures may be computed incrementally: they are defined by a thunk. When forcing the thunk, some part of the structure is created and includes references to more thunks representing the rest of the structure and to be evaluated when necessary. As an example, consider the list `[1..10]`. Initially, that's just a thunk. When we force this thunk, we create a pair (commonly called a cons-cell) consisting of the head of the list, 1, and a thunk representing the tail of the list, `[2..10]`. This thunk, when forced, and only then, gets replaced with a cons-cell consisting of 2 and a thunk for the list `[3..10]`, and so on. This makes figuring out the cost of lazy evaluation rather tricky. A first simple attempt at getting a handle on this is to pretend evaluation is strict or alternatively all thunks are forced immediately after they are created until no thunks remain. The resulting analysis provides a valid upper bound on the cost of a sequence of operations: if the result of a thunk produced by some operation o is subsequently needed by an operation o', it has no impact on the total cost of the sequence of operations whether we consider the cost of evaluating the thunk to be part of o's cost (strict evaluation) or of the cost of o' (lazy evaluation). If the value of a thunk is not needed at all, then all we do is overestimate the cost of the sequence of operations because we do charge o for the cost of this thunk. As we will see, there are situations where this overestimation becomes a problem for our amortized analysis because, well, we overestimate the cost, by more than a constant factor. Whenever the estimate is good enough, though, it helps simplify things substantially to assume strictness.

Whenever pretending evaluation is strict does not produce good enough upper bounds, we need to more carefully reason about when particular thunks are forced. To this end, we still consider the final data structure produced by forcing the thunk computing this data structure and recursively forcing all thunks this produces until no more thunks remain. We can model this recursive forcing of thunks as a graph structure: the children of a thunk are all parts of the data structure created by the thunk, as well as further thunks created by forcing this thunk. The children of a part of the data structure are the thunks it references. To illustrate this, two examples:

When defining the list `[1..10]`, the thunk representing the list has a single cons-cell as a child because that's all it creates. The children of this cons cell are 1 and a thunk representing the list `[2..10]`. This thunk once again has a cons-cell as child with children 2 and a thunk representing the list `[3..10]` and so on.

If, on the other hand, we look at a tail-recursive computation, say computing factorials

```
factorial n = fac 1 n
  where fac f 1 = f
        fac f x = fac (f ∗ x) (x − 1)
```

Then *factorial n* has a single thunk *fac* 1 *n* as a child. Its child is a thunk for *fac n* $(n-1)$, and so on, until we finally get a thunk for *fac* $(n!)$ 1 whose child is the value $n!$, without any further children. Now, observe that until we ask for the value of *factorial n*, none of these thunks is forced. When we need the value, *all* the thunks are forced. The same is true for any sequence of thunks that need to be forced to produce some part of a data structure. As long as we don't access this part of the data structure, we don't need to force any of the thunks; once we access this part of the data structure, we need to force all the thunks. Thus, for our analysis, we can collapse all thunks that need to be forced successively to produce a single part of the data structure into a single thunk whose cost is the total cost of the collapsed thunks. This leads a graph structure where every part of the data structure is produced by a unique thunk. This thunk needs to be forced the first time we try to access this part of the data structure and not before. This clear view of when a thunk needs to be forced will be helpful later in our analysis.

For now, back to our question whether laziness helps with our queue implementation above. The *dequeue q* operation now just creates a thunk for *reverse b*. that is forced only once we want to know the head of *reverse b*. *reverse* is implemented as:

```
reverse xs = rev [ ] xs
  where rev ys [ ]      = ys
        rev ys (x : xs) = rev (x : ys) xs
```

It is important to observe that *reverse b* collapses into a single thunk because the thunk for *reverse b* needs to be forced only once we need to know the head of *reverse b* and at this time, we need to force all thunks this creates recursively to produce the head of *reverse b*. For this reason, we call *reverse* a *monolithic* operation.

Here now is the problem. Let's say we implement a *front* operation that allows us to inspect the first element in the queue. We'll need such an operation eventually because otherwise the queue is useless!

```
front                 :: Queue t → Maybe t
front (Queue [ ] [ ]) = Nothing
front (Queue [ ] b)   = front (Queue (reverse b) [ ])
front (Queue f   _)   = Just (head f)
```

We can now either consider a sequence of $2n$ alternating *dequeue* and *front* operations on *q* or even just a sequence of *n front q* operations. For simplicity, we choose the latter: Each *front q* operation creates a thunk for *reverse b* and then immediately evaluates it because we actually want to know the first element in the list produced by *reverse b*. The reversal takes linear time. What's worse, the fact that every thunk gets forced only once doesn't help here because each *front q* operation creates a new thunk for *reverse b* and evaluates this thunk, so the reversal is really computed *n* times, for a total cost of $\Theta(n^2)$ for only $2n$ operations.

3 Queue Operations with Constant Amortized Cost

To avoid this problem, we should not wait to reverse *b* until we absolutely cannot wait any more (that is, until *f* is empty). The trick is to ensure that *f* is at least as long as *b*:

```
data Queue t = Queue {f, b        :: [t]
                     , lenF, lenB :: Int
                     }

enqueue q x = checkQueue (q {b = x : b q, lenB = lenB q + 1})

dequeue q = checkQueue (q {f = tail (f q), lenF = lenF q − 1})

checkQueue                         :: Queue t → Queue t
checkQueue q@(Queue f b lf lb) | lf ⩾ lb    = q
                               | otherwise = Queue (f ++ reverse b) [ ] (lf + lb) 0
```

Since this guarantees that the queue is empty if *f* is empty, the *front* operation becomes a little simpler:

```
front                  :: Queue t → Maybe t
front (Queue [ ] _ _ _) = Nothing
front q                = Just (head (f q))
```

Similar to stacks, we also implement convenience functions for building a queue from a list and vice versa, and for creating an empty queue:

```
queueFromList    :: [t] → Queue t
queueFromList xs = Queue xs [ ] (|xs|) 0

listFromQueue                  :: Queue t → [t]
listFromQueue (Queue f b _ _) = f ++ reverse b

emptyQueue :: Queue t
emptyQueue = queueFromList [ ]
```

So how does reversing *b* early help exactly? First the intuitive version: Let's assume we have a queue q_0 such that $|f| = |b| = n$ and, to keep the discussion simple, let's assume that *dequeue* is actually a combination of *dequeue* and *front*, which removes the first element from the queue *and* returns it. We do this to ensure that, when a *dequeue* operation produces a list consisting of only *reverse b*, the reversal actually needs to be forced, as it would have to be for *front*. *dequeue* q_0 produces a new queue q_1 whose front is f ++ *reverse b*. Let us ignore the cost of (++) completely for now. Then the cost of producing q_1 is constant because so far we've only created a thunk for *reverse b*. Now observe that there is no need to evaluate *reverse b* until we have removed all elements from f, which takes n *dequeue* operations. So consider a whole sequence $q_0, q_1, \ldots, q_{n+1}$, where $q_i = \mathit{dequeue}\ q_{i-1}$. Producing each queue q_i with $i \leq n$ takes constant time because we only have to remove the frontmost element from f. Producing q_{n+1} takes

$\Theta(n)$ time because we finally need to force the thunk for *reverse* b. So far that's fine because this means we have a cost of $\Theta(n)$ for a sequence of $n+1$ operations. Now let's consider what happens when we use some queue q_k as the starting point for multiple logical futures, exactly the case our original queue couldn't handle. Specifically, let's say we repeat the computation of $q_{k+1}, q_{k+2}, \ldots, q_{n+1}$ from q_k m times.

If $k > 0$, then we have $t = k + m(n+1-k)$ operations. Each of these operations has cost $O(1)$ except for the last operation in each of the m logical futures of q_k. For these final operations, observe that they all evaluate the same thunk for *reverse* b. Thus, the first of these operations actually spends $\Theta(n)$ time to evaluate *reverse* b ... and then the thunk is replaced with its result. All other operations just use this cached result. Thus, the total cost of the entire sequence of operations is $O(t+n) = O(t)$, again constant amortized cost per operation.

If $k = 0$, the picture is rather different. Since we produce q_1 from q_0 m times, we create m distinct thunks for *reverse* b. The final operation in each logical future forces one of these thunks, at a cost of $\Theta(n)$. Thus, the total cost of all t operations is $\Theta(t + mn)$. In this case, however, we have $t = m(n+1)$, so again the cost is bounded by $O(t)$.

It remains to formally prove that the amortized cost per operation is indeed $O(1)$. We use an adaptation of the accounting method for amortized analysis. The accounting method of ephemeral data structures places credits on different parts of the data structure when they are created and then uses these credits to pay for the cost of operations. This does not work that well with persistent data structures because persistence opens the door for us to try and use the same credit more than once to pay for the costs of operations. So, Okasaki went and turned the whole idea upside down: He places debits on parts of the data structure. Initially, the debits on each data structure part equal the cost of the thunk producing this part of the data structure. As long as there is no need to force a thunk, the thunk does not incur any cost. When forcing a thunk, we do incur its cost and we need to make sure that our analysis pays for this cost. We do this using two parts of the analysis: operations may pay debits on arbitrary parts of the data structure. The amortized cost of an operation then the number of debits paid by the operation plus what Okasaki calls the unshared cost of the operation—the cost of the operation assuming it does not force any thunks it did not create itself. To ensure these amortized costs pay for the costs of forcing thunks, we now require that our analysis ensures that no thunk is forced before the debits on the part of the data structure it produces have been paid in full. In other words, the amortized costs of the operations in the logical past of the operation that ends up forcing the thunk already account for the cost of forcing the thunk, so the sum of the amortized costs of the operations in any sequence is an upper bound on the actual cost of this sequence, including the cost of forcing all thunks that do get forced.

For our queue, we place debits on the nodes of the front list f. Let d_i be the number of debits of the ith node in f and let $D_i = \sum_{j=0}^{i} d_j$. We maintain the following debit invariant:

$$D_i \le \min(2i, |f| - |r|).$$

Since this invariant ensures that $D_0 = 0$, we can force the thunk producing the first cons cell of f, for example using *front*, whenever we want.

To enable the analysis of *queueFromList* and *listFromQueue*, we also assign debits to the nodes of any list not part of a queue. Or rather, we don't. We require that every node of such a list carries 0 debits, that is, its thunk can be evaluated at any time. Now let's look at the costs of the different operations. First the easy ones:

queueFromList simply installs the given list *xs* as the head of the queue. Since every element in the list has zero debits, this maintains the debit invariant. The actual cost of the operation is constant. So the amortized cost of *queueFromList* is also constant.

emptyQueue is just an application of *queueFromList* to the empty list. So this also has amortized constant cost.

listFromQueue reverses b and then concatenates f and *reverse* b. Since we want the resulting list to carry no debits, we have to pay for the costs of these two operations right away, and we have to pay off the debits of the elements in f. The cost of $f +\!\!+\ \mathit{reverse}\ b$ is $O(|f|+|b|)=O(n)$, where n is the number of elements in the queue as can be seen from the following implementation of $(+\!\!+)$ and the implementation of *reverse* given earlier:

```
[ ]      ++ ys = ys
xs       ++ [ ] = xs
(x : xs) ++ ys = x : (xs ++ ys)
```

By the debit invariant, the elements in f carry at most $|f|$ debits, so paying them off adds at most $|f| \in O(n)$ to the amortized cost of the operation. In summary, *listFromQueue* takes linear time.

Now let's look at *enqueue*, *dequeue*, and *front*. *front* is easy: Its unshared cost is constant because it only inspects the head of f. It also does not change the structure of q, so no debits need to be paid off to maintain the debit invariant. Thus, the amortized cost of *front* is constant.

For *enqueue* and *dequeue*, consider first the case when *checkQueue* does not reverse b. Then *enqueue* increases the length of b and thus decreases $|f|-|b|$ by one. Thus, for every node such that $D_i=|f|-|r|$ before the *enqueue* operation, the debit invariant is now violated. We restore the debit invariant by paying off the first debit in the queue, that is, we find the first node in f such that $d_j>0$ and decrease d_j by one. The amortized cost of the *enqueue* operation is thus constant: constant unshared cost plus one debit that is paid. Similarly, a *dequeue* operation that does not reverse r decreases the length of f by one and thus decreases $|f|-|r|$ by one. It also decreases the position of every node in f. Thus, any node in f may now exceed its bound on D_i by one or two. We fix this by paying off the first two debits in f. Again, the amortized cost is constant: constant unshared cost plus two debits paid off.

Finally, consider what happens when $|f|=|b|$ before the *enqueue* or *dequeue* operation and thus $m=|f|=|b|-1$ after the operation. In this case, the operation first behaves as above and thus incurs constant amortized cost, and then it replaces f with $f +\!\!+\ \mathit{reverse}\ b$. Observe that currently there are no debits on any nodes in f, by the debit invariant. To pay for the thunk created for *reverse* b, we place a debit of $|b|=m+1$ on the head of *reverse* b. To pay for the thunks created for the elements in f by $f +\!\!+\ \mathit{reverse}\ b$, we place one debit on every node in f. This ensures that the debit invariant is satisfied for every node in $f +\!\!+\ \mathit{reverse}\ b$ except the 0th node, which has cumulative debit $D_0=1$ and the first node of *reverse* b, which has cumulative debit $D_m = 2m + 1$. We restore the invariant by paying off the first debit in f again. Thus, also in this case, the amortized cost of *enqueue* and *dequeue* is constant because their unshared cost is constant and each pays off one debit.