Lloyd Allison's Corecursive Queues: Why Continuations Matter

by Leon P Smith ⟨leon@melding-monads.com⟩

July 29, 2009

Abstract

In a purely functional setting, real-time queues are traditionally thought to be much harder to implement than either real-time stacks or amortized $\mathcal{O}(1)$ queues. In "Circular Programs and Self-Referential Structures," [1] Lloyd Allison uses **corecursion** and **self-reference** to implement a queue by defining a lazy list in terms of itself. This provides a simple, efficient, and attractive implementation of real-time queues.

While Allison's queue is general, in the sense it is straightforward to adapt his technique to a new algorithm, a problem has been the lack of a reusable library implementation. This paper solves this problem by structuring the corecursion using a monadic interface and continuations.

Because Allison's queue is not fully persistent, it cannot be a first class value. Rather, it is encoded inside particular algorithms written in an extended continuation passing style. In direct style, this extension corresponds to *mapCont*, a control operator found in *Control.Monad.Cont*, part of the Monad Template Library for Haskell. [2] This paper argues that *mapCont* cannot be expressed in terms of *callCC*, *return*, and $(\gg=)$.

The essence of this paper is *mfixish*, a novel fixpoint operator for continuations. It cooperates with *mapCont* to create value recursion. Although this paper tends to avoid explicit use of *mfixish*, it can be used to introduce the self-reference inherent in corecursive queues.

Introduction

Richard Bird is well known for popularizing "circular programming," [3] which in modern terminology is included under the term "corecursion." [4] One of the best known examples defines an infinite list of Fibonacci numbers. However, as this paper is about queues, our running example is breadth-first traversals of binary trees. Thus, for our first example in Figure 1, we corecursively define the Fibonacci trees instead.

```
data Tree a b
   = Leaf     a
   | Branch b (Tree a b) (Tree a b)
     deriving (Eq, Show)

labelDisj :: (a → c) → (b → c) → Tree a b → c
labelDisj leaf branch (Leaf     a    ) = leaf     a
labelDisj leaf branch (Branch b _ _) = branch b

childrenOf :: Tree a b → [Tree a b]
childrenOf (Leaf     _    ) = [ ]
childrenOf (Branch _ l r) = [l, r]
```

Listing 1: Binary Trees and useful helper functions

The indexing was chosen so that the number of leaves in the nth Fibonacci tree is equal to the nth Fibonacci number. The branches are labeled with the the depth of the tree. This definition uses self-reference and sharing to efficiently represent each additional tree with a constant amount of extra space. Of course, fully traversing such a tree would take an exponential amount of time.

The second example defines the Stern-Brocot tree, shown in Figure 2. Despite that this definition does not employ self reference, this is a corecursive definition because it is infinite and thus requires lazy evaluation. The Stern-Brocot tree is interesting because every positive rational number is generated in reduced form at exactly one branch. Not only does this prove that the rationals are countable, it can be computed more efficiently than the standard Cantor diagonalization.

These examples were chosen such that any two subtrees in this family are equal if and only if their labels are equal. This is true even for the Fibonacci trees: even though labels are repeated, the subtrees are still equal. This property can be exploited to efficiently and accurately test whether two breadth-first traversals might be equivalent.

Having separate types for the labels of branches and leaves enables one to better exploit Haskell's type system. An example is the polymorphic leaf type

```
fib :: Int → Tree Int Int
fib n = fibs !! (n − 1)
  where
    fibs = Leaf 0 : Leaf 0 : zipWith3 Branch [1 ..] fibs (tail fibs)
```

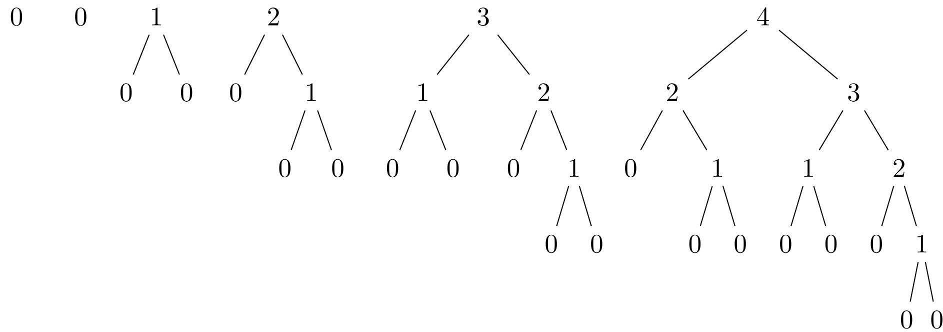

Figure 1: Fibonacci Trees

```
sternBrocot :: Tree a (Ratio Integer)
sternBrocot = loop 0 1 1 0
  where loop a b x y
     = Branch (m % n) (loop a b m n) (loop m n x y)
       where m = a + x
             n = b + y
```

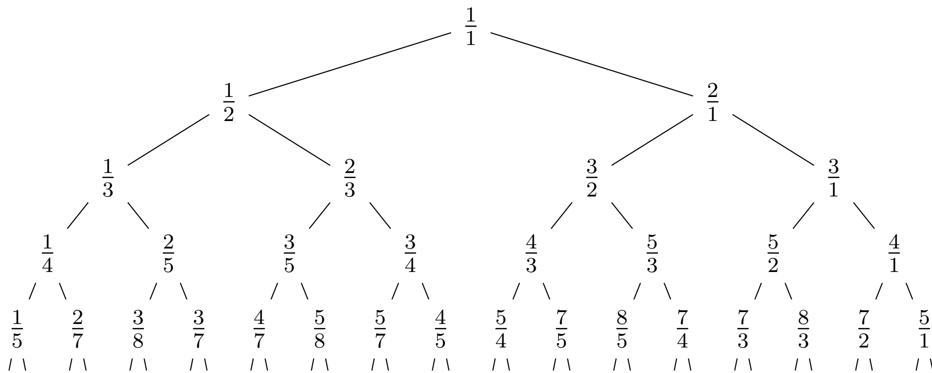

Figure 2: The Stern-Brocot Tree

for *sternBrocot*. Along the lines of Philip Wadler's classic paper "Theorems for Free", [5] this almost proves that the Stern-Brocot tree is both complete and infinite, in the sense that every counterexample to this line of reasoning involves **partial data**. Common examples of partial data are **infinite nonproductive loops** and the use of Haskell's *error* function, such as this definition of $\bot$:

$\bot :: \forall\ a.a$
$\bot = error$ `"bottom"`

However, not every corecursive definition produces a conceptually infinite data structure. Lloyd Allison's queue is a good example: it is self-referential, and thus depends on lazy evaluation. Allison's queues can and often do produce a finite object.

A simple breadth-first traversal using Allison's queue is given along with a sample execution in Figure 3. The execution abuses notation slightly, as necessary for readability: the elements of the queue are trees, not labels. However, as the labels are unique for the examples given, this does not lead to ambiguity.

A corecursive queue is represented by a single lazy list. The end of the queue is represented by a thunk, which can produce the next element on demand. This thunk contains a pointer back to the first element in the queue and the number of elements currently in the queue. When an element is enqueued, call-by-need evaluation **implicitly mutates** the end of the list.

The Fibonacci example uses a lazy list sort of like a queue. New elements are "enqueued" when they are produced by *zipWith*, which occurs in sync with elements being "dequeued" when they are consumed by pattern matching inside *zipWith*.

Of course, most queue-based algorithms don't have this level of synchronization. During a level-order traversal of a Fibonacci tree, the queue will grow and shrink frequently. We must be careful not to run off the end of the queue and pattern match against elements that aren't there. The easiest approach is to track the number of elements in the queue.

Pattern matching creates demand for computation, thus pattern matching on the empty queue causes this thunk to reenter itself, creating an infinite nonproductive loop. In effect, in order to compute the answer, the answer must have already been computed. Explicitly tracking length breaks this cycle. By knowing via other means that the queue is empty, we can avoid pattern matching and continue or terminate the queue as needed, as illustrated in the last step of the sample execution.

Note that the type of the counter is explicitly given. Otherwise, Haskell would typically default to arbitrary precision integers. On GHC, this leads to a noticable, though modest, slowdown and increase in memory allocation in certain microbenchmarks, such as a complete traversal of a Fibonacci trees.

If one doesn't care about leaves or their contents, one might prefer a variant

```
levelOrder :: Tree a b  → [Tree a b]
levelOrder    tree      = queue
  where
    queue = tree : explore 1 queue

    explore :: Int →  [Tree a b]             → [Tree a b]
    explore   0                       q    =       []
    explore   (n + 1) (Branch _ l r : q′) = l : r : explore (n + 2) q′
    explore   (n + 1) (Leaf   _     : q′) =         explore  n      q′
```

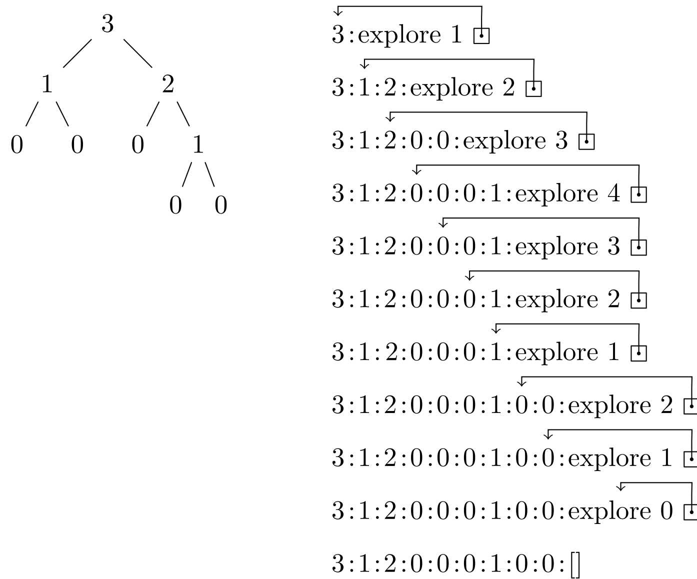

Figure 3: A Corecursive Queue, and a trace of *levelOrder* (*fib* 5)

of *levelOrder* that does not enqueue the leaves. Listing 2 presents Lloyd Allison's original code, translated from Lazy ML to Haskell. It contains repeated code in the inner conditional branches. This approach is not unreasonable in this specific case, however, it does not scale. The easiest way to eliminate the redundant branch is introduce a helper function that processes a list of trees to possibly enqueue.

Because the code in Listing 3 makes use of *childrenOf*, it easily generalizes to arbitrary trees. This code is very similar to a breath-first graph searching algorithm I wrote in December 2000, simplified to trees. This is notable because it was the day after I first read Richard Bird's classic paper, which is also prominently cited by Allison.

```
isBranch = labelDisj (const False) (const True)
levelOrder' ::  Tree a b → [Tree a b]
levelOrder'     tree     = queue
  where
    queue | isBranch tree = tree : explore 1 queue
          | otherwise     =        explore 0 queue
    explore :: Int → [Tree a b] → [Tree a b]
    explore  0        _
      = []
    explore (n + 1) (Branch _ l r : q')
      = if (isBranch l)
        then if (isBranch r)
             then  l : r : explore (n + 2) q'
             else  l :     explore (n + 1) q'
        else if (isBranch r)
             then      r : explore (n + 1) q'
             else          explore  n      q'
```

Listing 2: Avoiding leaves

```
levelOrder'2 :: Tree a b → [Tree a b]
levelOrder'2    tree     =  queue
  where
    queue = enqs [tree] 0 queue
    enqs :: [Tree a b] → Int →  [Tree a b] → [Tree a b]
    deq  ::              Int →  [Tree a b] → [Tree a b]
    enqs  []             n      q          =    deq      n      q
    enqs  (t : ts)       n      q
        | isBranch t                       = t : enqs ts (n + 1) q
        | otherwise                        =    enqs ts  n      q
    deq                  0      _          =    []
    deq                 (n + 1) (t : q')   =    enqs ts' n      q'
          where ts' = childrenOf t
```

Listing 3: Removing the repeated conditional branch

Although I didn't become aware of Lloyd Allison's work until six years later, at the time I had guessed that somebody had come up with it before. It is a testament to Bird's writing that he has inspired at least two people, and probably more, to independently arrive at the same idea.

The code was rather difficult to write the first time, but I immediately felt that I had a reasonable grasp of what was going on. In fact, I remember part of the thought process behind my endeavor: I was trying to implement a queue using naive list concatenation, employing Richard Bird's technique to eliminate a quadratic number of passes.

Reusable Corecursive Queues

We have now written three corecursive queues. A natural question to ask is how to implement a reusable library for this technique, so that we don't have to start from scratch every time we would like to use it. This subsection informally derives such an implementation.

What kind of interface would this library have? The traditional interface treats queues as first class values, as given by *Queue* in Figure 4. Because Haskell is pure, all first class values are persistent. This gives us the freedom to enqueue an element, back up, enqueue a different element, and use both results in subsequent computations, as demonstrated by the $\mathit{fork}\ a\ b\ q = (\mathit{enque}\ a\ q, \mathit{enque}\ b\ q)$.

Due to implicit mutation, Allison's queues cannot be used persistently. This implies that they cannot be first class values in a pure language! Thus looking for an implementation of *enque* and *deque* is futile, as any interface must enforce linearity upon enqueue. Monadic interfaces offer a well-known solution to this problem, so it seems plausible that we could find an implementation of a monadic interface.

In the examples so far, the logic that traverses the binary trees is entangled with the logic that defines the queue operations. Our goal is to separate these concerns.

```
class Queue q where
  empty :: q e
  enque :: e → q e → q e
  deque :: q e → (Maybe e, q e)
class Monad m ⇒ MonadQueue e m | m → e where
  enQ :: e → m ()
  deQ :: m (Maybe e)
```

Listing 4: First-class versus monadic interfaces

As the names suggest, *explore* in Listing 2 has no separation of concerns, while *enqs* and *deqs* in Listing 3 isolate the basic operations into two mutually recursive functions.

If we had an implementation of *enQ*, it would be easy to write a helper function that takes a single tree, and enqueues it iff it is a *Branch*. Perhaps if we had this helper, it would be easier to factor out the queue operations. We will work backwards from this guess, and forwards from $levelOrder'2$ to write this single-element filter.

Currently, *enqs* loops over candidates to possibly enqueue. By inlining *childrenOf* and then unrolling *enqs*, we get two helper functions that differ only in their **continuation**. By parameterizing the continuation, we can refactor these into a single function. Continuations are **required** because every queue operation must return the resulting queue from the rest of the computation. Thus every enqueue or dequeue must be aware of what operation comes next. This process is demonstrated in Listings 5 and 6.

The resulting definition of $levelOrder'4$ is easily the best means of expression thus far: unlike $levelOrder'$ and $levelOrder'3$ it does not repeat any logic, and unlike $levelOrder'2$, the queue operations are expressed much more directly. All that is left to do is to parameterize the continuation of *deq*, and to remove the conditional branch from *enq*. Once this is completed, we are left with a fully disentangled traversal in Listing 7.

In a sense, we have re-invented the traditional continuation passing style [6] (CPS) with a slight twist, namely, that the tail of a list is considered to be a tail call. This is not a problem relative to the current understanding of the CPS transform, where any part of the program that is guaranteed to terminate, such as lazy cons [7], may optionally be left untouched by the transform.

Now, the only difference between *deq* and the monadic *deQ* is that the *deq* breaks out of it's loop and leaves the computation, whereas *deQ* returns *Nothing* in this case. Although the breaking out of the computation is superficially pleasing in this particular case, there are good reasons to prefer the latter in most cases. So we are ready to split Listing 7 into two parts: reusable corecursive queue operations versus the breadth-first tree traversal in Listings 8 and 9 respectively.

I first wrote something like *CorecQ* in August 2005, although it took me several years to understand my own code. Of course, this raises the question of how one can write code that one doesn't understand.

The answer is simple: type theory. Following the reasoning at the beginning of this subsection, I had been growing rather suspicious that corecursive queues could be abstracted via a monadic interface. After reading Wouter Swierstra's "Why Attribute Grammars Matter" [8] and coming to opinion that the examples contained therein are kind of lame, I was motivated to produce better, more compelling examples. Corecursive queues naturally came to mind.

```
levelOrder'3 tree = queue
  where queue = enq1 [tree] (0 :: Int) queue
        enq2 [a, b] n q | isBranch a = a : (enq1 [b]) (n + 1) q
                        | otherwise  =     (enq1 [b])  n      q
        enq1 [a]    n q | isBranch a = a : (deq     ) (n + 1) q
                        | otherwise  =     (deq     )  n      q
        deq 0                 _  = []
        deq n (Branch _ l r : q') = enq2 [l, r] (n - 1) q'
```

Listing 5: Inlining *childrenOf* and unrolling *enqs*

```
levelOrder'4 tree = queue
  where queue = (enq tree $ deq) (0 :: Int) queue
        enq a k n q | isBranch a = a : k (n + 1) q
                    | otherwise  =     k  n      q
        deq 0                    _ = []
        deq n (Branch _ l r : q) = (enq l $ enq r $ deq) (n - 1) q
```

Listing 6: Parameterizing the continuations of *enq1* and *enq2*

```
levelOrder'5 tree = queue
  where queue = (handle tree $ (λ() → explore)) (0 :: Int) queue
        handle t | isBranch t = enq t
                 | otherwise  = ret ()
        explore = deq      $ (λ(Branch _ l r) →
                  handle l $ (λ() →
                  handle r $ (λ() →
                  explore    )))
        enq e    k n     q  = e : k () (n + 1) q
        ret  a   k n     q  =     k a   n      q
        deq      k 0     q  =     []
        deq      k n (e : q') =   k e  (n - 1) q'
```

Listing 7: A fully disentangled traversal in explicit CPS

```
newtype CorecQ e a
  = CorecQ (Cont (Int → [e] → [e]) a) deriving (Monad)
mkCorecQ x = (CorecQ (Cont x))
unCorecQ (CorecQ (Cont x)) = x
runCorecQ :: CorecQ element result → [element]
runCorecQ m = queue
  where queue = unCorecQ m (λa n′ q′ → [ ]) 0 queue
instance MonadQueue e (CorecQ e) where
  enQ e = mkCorecQ enq
    where enq k n q = e : k () (n + 1) q
  deQ = mkCorecQ deq
    where deq k 0 q = k Nothing 0 q
          deq k n (e : q′) = k (Just e) (n − 1) q′
```

Listing 8: A reusable implementation of Allison's queue

```
levelOrder′′ :: MonadQueue (Tree a b) q ⇒ Tree a b → q ()
levelOrder′′ t = handle t >>= (λ() → explore)
  where
    handle t | isBranch t = enQ t
             | otherwise  = return ()
    explore = deQ      >>= maybe (return ()) (λ(Branch _ l r) →
              handle l >>= (λ() →
              handle r >>= (λ() →
              explore       )))
```

Listing 9: Traversing a binary tree using generic queues

For ten hours I struggled, lost and confused, starting over several times. Eventually I came to realize that the state monad was not a suitable vehicle for Allison's technique, and started thinking towards continuations. Soon enough the types worked out and everything felt "right." I was rather confident that it would work before I tried it. And as if by magic, it worked.

The problem had to be simplified before I got anything working at all. In the first three failed attempts, I tried to implement a full-blown *CorecQW*. However, fourth and first successful implementation could not even track the length of the queue internally, rather, it was the client's responsibility to avoid the infinite nonproductive loop.

Observing Monadic Computations

Monadic computations must somehow be **observed**, otherwise they are useless. Note the type of $runCorecQ :: CorecQ\ e\ a \to [e]$. The only observable aspect of the *CorecQ* monad is the list of enqueued elements. In particular, the final return value a cannot be observed.

Listing 10 gives two generic combinators for level-order traversals. They parameterize *childrenOf*, allowing arbitrary trees to be traversed. In fact, one need not even explicitly construct a tree: the Fibonacci trees could be traversed using this definition of *fibChildren*, for example.

$$
\begin{aligned}
&fibChildren\ 0 = [\,] \\
&fibChildren\ 1 = [0,0] \\
&fibChildren\ n = [n-2, n-1]
\end{aligned}
$$

They also return a generic *MonadQueue*, allowing not only corecursive implementations, but also alternative implementations. For example, more obvious implementations of *MonadQueue* include wrapping the traditional two-stack queue in a state monad, or using explicitly mutable state as provided by *Control.Monad.ST*. As we will see in the next section, although these have the same semantics for *enQ* and *deQ*, they observe only the final return value.

Thus, these combinators produce level-order traversals when observed by *runCorecQ*, but are not particularly useful when observed with implementations that observe only return values.

There are two points worth noting about the definition of *byLevel* and *byLevel′*: the top level definitions are not recursive, and the recursive *explore* does not pass *childrenOf* to itself repeatedly. This combination plays nice with the Glasgow Haskell Compiler as current implemented: inlining *byLevel* opens up the possibility of inlining *childrenOf* as well, or at least eliminating an indirect jump.

While this idiom can be worthwhile; it is far more important performance-wise that *enQ* and *deQ* be inlined. Because we are using typeclasses to abstract over

$$
\begin{array}{l}
\mathit{byLevel}, \mathit{byLevel}' :: (\mathit{MonadQueue}\ a\ m) \Rightarrow (a \rightarrow [\,a\,]) \rightarrow [\,a\,] \rightarrow m\ () \\
\mathit{byLevel}\ \mathit{childrenOf}\ \mathit{as} = \mathit{mapM_}\ \mathit{enQ}\ \mathit{as} \gg \mathit{explore} \\
\quad \textbf{where} \\
\qquad \mathit{explore} = \mathit{deQ} \ggg \mathit{maybe}\ (\mathit{return}\ ()) \\
\qquad\qquad\qquad\qquad (\lambda a \rightarrow \textbf{do} \\
\qquad\qquad\qquad\qquad\qquad \mathit{mapM_}\ \mathit{enQ}\ (\mathit{childrenOf}\ a) \\
\qquad\qquad\qquad\qquad\qquad \mathit{explore}\ ) \\
\mathit{byLevel}'\ \mathit{childrenOf}\ \mathit{as} = \mathit{mapM_}\ \mathit{handle}\ \mathit{as} \gg \mathit{explore} \\
\quad \textbf{where} \\
\qquad \mathit{handle}\ a = \mathit{when}\ (\mathit{hasChildren}\ a)\ (\mathit{enQ}\ a) \\
\qquad \mathit{explore} = \mathit{deQ} \ggg \mathit{maybe}\ (\mathit{return}\ ()) \\
\qquad\qquad\qquad\qquad (\lambda a \rightarrow \textbf{do} \\
\qquad\qquad\qquad\qquad\qquad \mathit{mapM_}\ \mathit{handle}\ (\mathit{childrenOf}\ a) \\
\qquad\qquad\qquad\qquad\qquad \mathit{explore}\ ) \\
\qquad \mathit{hasChildren} = \neg.\mathit{null}.\mathit{childrenOf}
\end{array}
$$

Listing 10: Generic traversals over generic trees

the queue operations, inlining these operations is a clumsy thing to do in GHC. Indeed, ML-style functors are a superior choice for this type of abstraction.

Queues via explicit mutation

Being able to efficiently implement real-time queues using monads is not particularly newsworthy, as a knowledgeable programmer could always make use of *Control.Monad.ST*, which provides genuinely mutable state. However, the point is that the corecursive implementation based on implicit mutation is **shorter and safer** than an imperative linked list based on explicit mutation. Moreover, *CorecQ* is **faster** than *STQ* on current versions of GHC.

In some cases, arrays offer worthwhile constant-factor performance advantages over linked lists. Thus, barring other concerns such as concurrency, the only explicitly mutable implementations of queues truly worth considering in Haskell are those that employ mutable arrays.

Note the type $\mathit{runSTQ} :: \mathit{STQ}\ e\ a \rightarrow a$. This observes the return result but does not observe anything about the queue elements. With this in mind, even though *byLevel* visits leaves and *byLevel′* does not, these combinators are observationally equivalent. They both return () if the forest is a finite number of finite trees, and diverge otherwise.

```
data List       st a = Null | Cons a ! (ListPtr st a)
type ListPtr st a = STRef st (List st a)

type        STQSt st r e = ListPtr st e → ListPtr st e → ST st r
newtype STQ e a
   = STQ { unSTQ :: ∀ r st.((a → STQSt st r e) → STQSt st r e)}

instance Monad (STQ e) where
   return a = STQ (λk → k a)
   m >>= f  = STQ (λk → unSTQ m (λa → unSTQ (f a) k))

instance MonadQueue e (STQ e) where
   enQ e = STQ $ λk a z → do
                        z' ← newSTRef Null
                        writeSTRef z (Cons e z')
                        k () a z'
   deQ   = STQ $ λk a z → do
                        list ← readSTRef a
                        case list of
                          Null         → k Nothing a  z
                          (Cons e a') → k (Just e) a' z

runSTQ :: STQ element result → result
runSTQ m = runST $ do
   ref ← newSTRef Null
   unSTQ m (λr _a _z → return r) ref ref
```

Listing 11: Queues via Imperative Linked Lists

If one is interested in things other than thermal output and timing information, a more conventional alternative to changing the monad would be to thread a value through *byLevel*. Listing 12 defines *foldrByLevel*, a function for computing right folds over breadth-first traversals.

Listing 14 demonstrates how we can use *foldrByLevel* to concisely define various level order traversals over a forest of binary trees. For example, we can visit the labels of only leaves, or only branches, or both, and these properties are reflected in the resulting types.

There is no need for *foldrByLevel* to enqueue leaf nodes. Instead of folding elements after they are dequeued, we could instead fold elements before they are enqueued. Listing 13 computes the same fold, even though it enqueues branches only.

An orthogonal change allows *foldrByLevel'* to be more lazy. If the initial forest

```
foldrByLevel :: (MonadQueue a m)
             ⇒ (a → [a]) → (a → b → b) → b → [a] → m b
foldrByLevel childrenOf f b as = mapM_ enQ as ≫ explore
  where
    explore = deQ ≫= maybe (return b)
                          (λa → do
                                 mapM_ enQ (childrenOf a)
                                 b ← explore
                                 return (f a b)              )

prop_foldrByLevel childrenOf f b as =
        foldr f b (runCorecQ (byLevel childrenOf as))
    ≡   runSTQ (foldrByLevel childrenOf f b as)
```

Listing 12: Right folds over level-order traversals

```
foldrByLevel′ :: (MonadQueue a m)
              ⇒ (a → [a]) → (a → b → b) → b → [a] → m b
foldrByLevel′ childrenOf f b as = handleMany as
  where
    handleMany []       = explore
    handleMany (a : as) = do
      when (hasChildren a) (enQ a)
      b ← handleMany as
      return (f a b)
    explore = deQ ≫= maybe (return b) (handleMany.childrenOf)
    hasChildren = ¬.null.childrenOf
```

Listing 13: A slightly lazier fold that does not enqueue leaves

```
cid = const id
getUnion    f = f childrenOf (labelDisj (:)  (:) ) []
getLeaves   f = f childrenOf (labelDisj (:)  cid) []
getBranches f = f childrenOf (labelDisj cid (:) ) []
runSTQ.getUnion    foldrByLevel :: [Tree a a] → [a]
runSTQ.getLeaves   foldrByLevel :: [Tree a b] → [a]
runSTQ.getBranches foldrByLevel :: [Tree a b] → [b]
```

Listing 14: Handy functions and example use cases

of trees is infinite, *foldrByLevel* will get stuck in a nonproductive loop. This is because Listing 12 enqueues the entire forest before doing any folding. The enhanced version interleaves folding with enqueue operations. Thus *foldrByLevel′* is a true generalization of *foldr*, whereas the original is not.

Of course, this property depends on the semantics of the monad: the final result must be observed in a sufficiently lazy fashion. This is not true of *runSTQ* even if the lazier variant of *ST* is used, and so *foldrByLevel* is semantically equivalent to *foldrByLevel′* relative to this monad. This equivalency will be relaxed relative to *StateQ* in the next section.

We now have four generic combinators and two ways to run each, for a total of eight possibilities. There is a pleasing combination of symmetry and anti-symmetry to this configuration: if we use *runCorecQ* to observe the elements enqueued, then *byLevel* is equivalent to *foldrByLevel*, which differ from *byLevel′* and *foldrByLevel′*. However, if we use *runSTQ* to observe the result, then *byLevel* is equivalent to *byLevel′*, which differ from *foldrByLevel* and *foldrByLevel′*.

Monad Transformers

This section briefly reviews the traditional two-stack queue, and explores how they can be encapsulated inside a variety of different monads. The first wraps the two-stack queue in a state monad, thus guaranteeing amortized $\mathcal{O}(1)$ performance. The second implementation employs a state transformer and a writer monad to observe both the elements enqueued and the final result. The last introduces the continuation passing state monad, which makes explicit an idiom already used in *CorecQ* and *STQ* of the previous sections.

We explore the guarantees that various types of monads provide, and demonstrate how most of these guarantees are lost when the monad transformers are used. We argue that monad transformers are not robust abstractions, culminating in the rather fragile corecursive queue transformer of the next section.

Two-Stack Queues

Purely functional stacks are easy because the simplest solution works well. Due to sharing, persistent linked lists make reasonably efficient stacks. When pushing an element onto the stack, all that is necessary is to allocate and initialize a single new *cons* cell. Removing an element is a simple pointer dereference, and involves no allocation.

However, the same naive usage of lists leads to quadratic behavior. Concatenating a single element onto the end of the list involves copying the entire list, leading to $\mathcal{O}(n)$ enqueues. Alternately, one could store the queue in reverse, which makes

```
data TwoStackQ e = TwoStackQ [e] [e]
instance Queue TwoStackQ where
  empty = TwoStackQ [] []
  enque z (TwoStackQ [] [])        = TwoStackQ [z]      []
  enque z (TwoStackQ (a : as) zs) = TwoStackQ (a : as) (z : zs)
  deque   (TwoStackQ [] [])        = (Nothing, TwoStackQ []  [])
  deque   (TwoStackQ (a : as) zs)
         | null as                  = (Just a  , TwoStackQ as' [])
         | otherwise                = (Just a  , TwoStackQ as  zs)
           where as' = reverse zs
```

Listing 15: Two-Stack Queues

enqueue an $\mathcal{O}(1)$ operation, but then peeking at or removing the front element then becomes a $\mathcal{O}(n)$ operation.

Traditionally, purely functional queues combine these approaches. [9] The queue is represented by two stacks: the front stack and the back stack. The front stack holds the beginning of the queue, and the back stack holds the remainder of the queue in reverse. To enqueue something, push it on the back stack. To dequeue something, pull it off the front stack. If the front stack is subsequently empty, reverse the back stack onto the front.

Of course, because two-stack queues are first-class values in Haskell, they are automatically persistent. Unlike an imperative language, operations on the queue preserve older versions of the queue. While *deque* is still $\mathcal{O}(n)$ in the worst case, it works very well in practice because on average, *deque* is actually $\mathcal{O}(1)$, provided that the queue is not used persistently. Under this assumption, every element is moved at most once from the back to the front.

There are implementations that guarantee $\mathcal{O}(1)$ worst-case operations, even with persistent usage, such as Chris Okasaki's incremental reversals of lazy lists. [10] However this solution has a relatively high constant factor, in practice is often slower than other options, sometimes significantly so.

The State Monad

Monadic interfaces can enforce linear, non-persistent usage of data structures, but do not necessarily do so. The *StateQ* monad guarantees linearity by wrapping the queue in a *State* monad and hiding *get* and *put* operations, ensuring amortized $\mathcal{O}(1)$ operations. However, the state transformer monad, *StateT*, cannot make this guarantee! The ability to use state persistently can be recovered by choosing

```
newtype StateQ e a = StateQ (State (TwoStackQ e) a) deriving (Monad)
instance MonadQueue e (StateQ e) where
  enQ = StateQ.modify.enque
  deQ = StateQ (State deque)
runStateQ :: StateQ element result → result
runStateQ (StateQ m) = let (result, finalQ) = runState m empty
                       in  result
```

Listing 16: First-class Queues inside *Control.Monad.State.Lazy*

the nondeterministic list monad and lifting *MonadPlus* operations, for example.

The Writer Monad

Editorial Note This section is deeply flawed. In particular, the standard implementation of the Writer monad does not ensure the efficient use of list concatination. A continuation-based implementation of the Writer monad, on the other hand, does. This confusion probably arose because the corecursive queue monad essentially uses a continuation-based writer monad to add elements to the queue. For example, compare the use of *mapCont* in the implementation of *enQ* for *CorecQ′* in Listing 23 and the *MonadWriter* instance for *Cont* in [11]. **End Note**

So far, we have only been able to observe either the enqueued elements or the final result, but not both. We can employ the state transformer in conjunction with the Writer monad to observe both aspects of the computation. The amortized $\mathcal{O}(1)$ guarantee is unaffected by the use of *Writer*.

Writer monads partially enforce the efficient use of list concatenation. In the MTL, *Writer* $[e]$ a is just a newtype isomorphism for $([e], a)$. It provides a function *tell* that takes a list and concatenates the remainder of the computation onto the end of the list. This naturally associates to the right, and thus avoids quadratic behavior. Of course, *tell* accepts arbitrary length lists, and one could inefficiently produce a long argument to *tell*.

Note the duplication of functionality present in *WriterQ*. Essentially, the Writer monad re-creates the queue in a parallel data structure. This observation was another motivation behind the creation of my first corecursive queue, analogous to Listing 3. I started writing a graph traversal using first-class queues, producing a list of nodes as they were visited. Although I was not using monads, I noticed the duplication and saw an opportunity to eliminate it using circular programming.

The Continuation Passing State Monad

Less well known than the regular state monad is the continuation passing state monad, as shown in Listing 18. This paper has already used this idiom twice. It has been used to track the length and head of the queue inside *CorecQ*, and to track the references to the start and end of the mutable list inside *STQ*.

The lazy state monad is notoriously inefficient on current implementations of Haskell; one is much better off using the strict state monad. The continuation-passing state monad is even faster. Compared to the lazy state monad, the biggest advantage is that we aren't returning many pairs of lazy tuples, at the cost of sacrificing some laziness.

In the case of *CpSt*, a rank-2 type is used to hide the final result. While this has little effect on the generated code, it has profound consequences. Specifically, we cannot implement the control operators *callCC* and *mapCont*, nor can we break out of the computation early. However we can implement an *mfix* operator! This last observation is due to Matt Morrow and will be demonstrated in Listing 26. As long as the computation terminates, the final continuation that *runCpSt* initially passes to the computation is guaranteed to be called exactly once.

A nearly identical monad can be obtained by passing *Cont* as a parameter to *StateT*. Other than the fact that the final result type is exposed, the effect is the same as implementing *CpSt* in Listing 18. In fact, when compiled `ghc -O2`, the given operations compile into almost the same code. The only significant, though minor, difference is that the continuation $a \to s \to r$ is tupled and not curried.

Listing 19 gives an implementation of the queue interface in terms of *StateT* and *Cont*. However, *StateT* is **lazy** but *CpStQ* is **strict**! Not only have we added a continuation semantics to *StateT*, we have also changed part of the existing

```haskell
newtype WriterQ e a
  = WriterQ (StateT (TwoStackQ e) (Writer [e]) a)
    deriving (Monad)

instance MonadQueue e (WriterQ e) where
  enQ e = WriterQ (tell [e] >> modify (enque e))
  deQ   = WriterQ (StateT (return.deque))

runWriterQ :: WriterQ element result -> (result, [element])
runWriterQ (WriterQ m)
  = let ((result, final_queue), queue) = runWriter (runStateT m empty)
    in  (result, queue)
```

Listing 17: Observing the list of elements enqueued

```
newtype CpSt st a
  = CpSt {unCpSt :: ∀ res.(a → st → res) → st → res}
instance Monad (CpSt st) where
  return a = CpSt (λk → k a)
  m >>= f  = CpSt (λk → unCpSt m (λa → unCpSt (f a) k))
instance MonadState st (CpSt st) where
  get      = CpSt (λk st → k st st )
  put st'  = CpSt (λk _ → k () st')
runCpSt :: CpSt st a → st → (a, st)
runCpSt m st = unCpSt m (λa st' → (a, st')) st
```

Listing 18: A Continuation Passing State Monad

```
newtype CpStQ r e a
  = CpStQ {unCpStQ :: StateT (TwoStackQ e) (Cont r) a}
    deriving (Monad)
instance MonadQueue e (CpStQ r e) where
  enQ = CpStQ.modify.enque
  deQ = CpStQ (StateT (return.deque))
runCpStQ :: CpStQ r e r → r
runCpStQ (CpStQ m) = runCont (runStateT m empty) (λ(r, q) → r)
```

Listing 19: Queues via another continuation passing state monad

```
class MonadMapCC a m | m → a where
  mapCC :: (a → a) → m b → m b

instance MonadMapCC r (CpStQ r e) where
  mapCC f = CpStQ.mapStateT (mapCont f).unCpStQ

foldrByLevel'' :: (MonadQueue   a m
                  , MonadMapCC b m)
               ⇒ (a → [a]) → (a → b → b) → b → [a] → m b
foldrByLevel'' childrenOf f b as = handleMany as
  where
    handleMany []       = explore
    handleMany (a : as) = do
      when (hasChildren a) (enQ a)
      mapCC (f a) (handleMany as)
    explore = deQ >>= maybe (return b) (handleMany.childrenOf)
    hasChildren = ¬.null.childrenOf
```

Listing 20: Restoring laziness to *foldrByLevel′* using *mapCont*

semantics. This is in contrast to the use of *Writer* in *WriterQ*, which left the state semantics undisturbed. Not only are monad transformers not robust abstractions, they are not robust in any sense of the word!

More precisely, *StateQ* returns its result incrementally while *CpStQ* doesn't return anything until the entire computation terminates. This is easily observed on the Stern-Brocot tree: *runStateQ* (*getBranches foldrByLevel* [*sternBrocot*]) returns useful data, while *runCpStQ* (*getBranches foldrByLevel* [*sternBrocot*]) gets stuck in an infinite nonproductive loop.

Fortunately, incremental results can be recovered through the use of *mapCont*, as illustrated in Listing 19. By tweaking *foldrByLevel* to use this control operator, we can traverse the Stern-Brocot tree. This would not be possible had we used the strict state monad, or hidden the final result type.

Allison's Queues in Direct Style

There are two styles for programming with continuations. The first is by explicitly by writing functions in the continuation passing style. (CPS) In this style, all calls to non-primitive functions are tail calls, and functions have an extra continuation parameter, which this paper has called k. By contrast, the direct style does not manipulate continuations explicitly, but rather uses them implicitly or via control

```
getReader :: MonadReader a m ⇒ m a
getReader     = ask

setReader :: (MonadReader a m, MonadCont m) ⇒ a → m ()
setReader     = modReader.const

modReader :: (MonadReader a m, MonadCont m) ⇒ (a → a) → m ()
modReader f = callCC (λk → local f (k ()))

stepReader :: (MonadReader a m, MonadCont m) ⇒ (a → (b, a)) → m b
stepReader f = do
  st ← getReader
  let (r, st′) = f st
  setReader st′
  return r
```

Listing 21: State effects with readers and *callCC*

```
data Len a = Len !Int a

deQ_len (Len 0             q  ) = (Nothing, Len 0 q )
deQ_len (Len (n + 1) (e : q′)) = (Just e   , Len n q′)

inc_len (Len n head) = Len (n + 1) head
```

Listing 22: Utility functions for tracking length

```
newtype CorecQ′ e a
   = CorecQ′ {unCorecQ′ :: ContT [e] (Reader (Len [e])) a}
     deriving (Monad)

instance MonadQueue e (CorecQ′ e) where
  enQ e = CorecQ′ (mapContT (liftM (e:)) (modReader inc_len))
  deQ   = CorecQ′ (stepReader deQ_len)

runCorecQ′ :: CorecQ′ e a → [e]
runCorecQ′ (CorecQ′ m) = q
  where q = runReader (runContT m endpoint) (Len 0 q)
        endpoint _ = return []
```

Listing 23: Enqueue and dequeue in direct style

operators such as *callCC*, or *shift* and *reset*.

This paper uses an extended CPS that allows the tail of a lazy cons to be considered a "tail call", even though it is not a proper tail call. In fact, the first four *levelOrder* variants are already written in this extended CPS, albeit in a static form that does not parameterize the continuations. They move towards a more direct style, with only *enQ* and *deQ* written in an explicit, parameterized CPS.

Completing this process by writing *enQ* and *deQ* in direct style as well is a natural theoretical endeavor. Of course, *enQ* uses the lazy cons extension to CPS, and in direct style this corresponds to *mapCont*.

$$mapCont :: (r \to r) \to Cont\ r\ a \to Cont\ r\ a$$
$$mapCont\ f\ m = Cont\ (\lambda k \to f\ (runCont\ m\ k))$$

The type of our *CpSt* idiom is $(a \to s \to r) \to s \to r$, which is isomorphic to $ContT\ r\ (Reader\ s)\ a$, so this would be a plausible place to start. Listing 21 demonstrates a way to implement state operation in terms of continuations and readers. The function *ask* is defined in *Control.Monad.Reader* and retrieves the value from the reader, while *local* takes a function and a monad, and modifies the value in the reader during the execution of it's second argument. The reader maintains it's original value otherwise.

By using *callCC* to grab the entire remainder of the computation, we can use *local* to mutate the reader. With the addition of a few helper functions to manage the counter, we are set up for concise definitions of *enQ* and *deQ* in direct style.

Independence of *mapCont*

The use of *mapCont* is notable because I am confident that *enQ* cannot be expressed in terms of *callCC*, $(\gg=)$, and *return*. The MTL's continuations are partially delimited, as seems necessary for the general utility of *mapCont*. However, the analogous conjecture in terms of *shift* and *reset* is certainly not true.

It may not be obvious why *mapCont* is independent of the rest, but it turns out to be fairly trivial: *callCC*, $(\gg=)$, and *return* simply offer no way to add to the control context by introducing something that is not a proper tail call. More formally, we can modify the type of *CpSt*, which uses higher-ranked types to hide the result type, into a form that admits *callCC*, but prohibits a useful form of *mapCont*.

Note that the code in Listing 24 is purely theoretical, not useful, because computations inside of $CpSt'$ cannot be observed without cheating. The definitions are the same as the corresponding definitions of *Cont*. Since the use of typeclasses are not essential, these definitions have meaning independent of their types.

The types of *callCC*, *return*, and $(\gg=)$ are unchanged, however the definition of *mapCont* gives a type error without providing an explicit higher-ranked type. Thanks to free theorems, there are only three inhabitants of type $\forall\ a.a \to a$: *id*,

```
newtype CpSt' s a
  = CpSt' {runCpSt' :: ∀ r.(∀ r'.a → s → r') → s → r}
instance Monad (CpSt' s) where
  return a = CpSt' (λk → k a)
  m >>= f  = CpSt' (λk → runCpSt' m (λa → runCpSt' (f a) k))
instance MonadCont (CpSt' s) where
  callCC f = CpSt' (λk → runCpSt' (f (λa → CpSt' (\_ → k a))) k)
mapCpSt' :: (∀ a.a → a) → CpSt' s b → CpSt' s b
mapCpSt' f m = CpSt' (λk → f (runCpSt' m k))
```

Listing 24: A "proof" of the independence of *mapCont*

const ⊥ and ⊥, none of which are useful. This argument may not be complete, but I believe it can be the basis for a formal proof of independence.

Corecursive Queue Transformers

Now that *CorecQ* is in the direct style, it is somewhat easier to come up with a plausible monad transformer. Unfortunately, *runCorecQT* is mostly broken. For example, as noted previously, the corecursive queue implementation makes use of implicit mutation, and thus depends on enforced linearity. The non-deterministic list monad enables us to regain non-linear, persistent use. Not surprisingly, the list monad is incompatible with this transformer.

Those interested should experiment with which monads work and which don't. Of particular interest is the *IO* monad. Getting a simple variant of Unix's *tail* command to work properly around this transformer is an interesting exercise that presents some difficulty.

The $mfix :: a \rightarrow m\ a$ used here is the topic of Levent Erkök's Ph.D. thesis, "Value Recursion in Monadic Computations". [12] The thesis argues that there is no *mfix* on continuations. Note that this transformer does not contradict this conjecture, as we are using *mfix* to define the run operation, not defining an *mfix* for *CorecQT*.

Value recursion is also a primary topic of this paper, however, our application requires the use of continuations. Thus it would appear that we are discussing an alternate form of value recursion, and that CPS enables some varieties of value recursion while disabling others.

The *StateQ* monad is implemented via *State*, which has an *mfix* operator. Intuitively, it would seem as though this *mfix* semantics makes sense for any *MonadQueue* implementation. Whether or not a corecursive implementation can actually compute this semantics for *mfix* is a very interesting question. Perhaps

```
newtype CorecQT e m a
  = CorecQT (ContT (m [e]) (Reader (Len [e])) a)
    deriving (Monad)

instance Monad m ⇒ MonadQueue e (CorecQT e m) where
  enQ z = CorecQT (mapContT (liftM (liftM (z:))) (modReader inc_len))
  deQ   = CorecQT (stepReader deQ_len)

runCorecQT :: (MonadFix m) ⇒ CorecQT e m a → m [e]
runCorecQT m = mfix (λq → run m end_point (Len 0 q))
  where
    end_point _ = return (return [])
    run (CorecQT m) k st = runReader (runContT m k) st
```

Listing 25: A rather fragile queue transformer

CorecQ would be a good avenue for research regarding Remark 5.2.1 on page 61 of Erkök's thesis, speculating on the existence of special cases when continuations happen to have an *mfix*.

Matt Morrow has observed that by hiding the result type of a continuation via higher-ranked types, a *mfix* operator can in fact be implemented. Of course, this technique also prohibits implementations of *callCC* and *mapCont*. An *mfix* for *CpSt* is given in Listing 26. This does not directly contradict Erkök's conjecture, because the type of *CpSt* differs from *Cont*. Currently, this observation is a bit of a mystery, so this paper will not attempt to expound further.

Also included in Listing 26 is the alternate fixpoint operator *mfixish* that is at the heart of this paper. It does not have the same type as *mfix*; so neither does it contradict Erkök's conjecture.

```
instance MonadFix (CpSt st) where
  mfix f = CpSt (λk st → let (a, st′) = unCpSt (f a) (, ) st in k a st′)

mfixish   :: (r → Cont r a) → Cont r a
mfixish   f = Cont   (λk → fix   (λr → runCont   (f r) k))

mfixishT :: (MonadFix m) ⇒ (r → ContT r m a) → ContT r m a
mfixishT f = ContT (λk → mfix (λr → runContT (f r) k))
```

Listing 26: Fixpoints for Value Recursion on Continuations

One might assume, as this paper tacitly does, that there is no *mfix* over a monad implemented using continuations with an exposed return type. This would imply

that *CorecQ* cannot be used in conjunction with *CorecQT*, ruling out an way that one might intuitively try to implement multiple queues.

Returning Results from Corecursive Queues

Thankfully, *Control.Monad.Writer* is compatible with the queue transformer of the last section. This enables us to observe results other than the queue itself. The benefit is that we can now usefully execute *foldrByLevel* and its variants using corecursive queues.

Unfortunately, because the writer monad expects monoids, this approach isn't really suitable for preserving the result semantics of *STQ* and other implementations given in this paper. The type *Writer e a* is just a newtype alias for (a, e), so instead of using our monad transformer directly, we will simply use lazy pairs and start over.

Because *CorecQW* can return results other than the queue, it makes sense to implement *mapCont* and *mfixish* for this monad. For a demonstration of *mfixishQW*, we implement Chris Okasaki's breadth-first renumbering algorithm [13] in Listing 28.

```
newtype CorecQW w e a
   = CorecQW {unCorecQW :: ContT ([e], w) (Reader (Len [e])) a}
     deriving (Monad)
instance MonadQueue e (CorecQW w e) where
   enQ e = CorecQW (mapContT (liftM ((e:) *** id)) (modReader inc_len))
   deQ   = CorecQW (stepReader deQ_len)
instance MonadMapCC w (CorecQW w e) where
   mapCC f = CorecQW.mapContT (liftM (id *** f)).unCorecQW
runCorecQW :: CorecQW w e w → ([e], w)
runCorecQW m = (q, w)
   where (q, w) = run m (λw → return ([], w)) (Len 0 q)
                run m k st = runReader (runContT (unCorecQW m) k) st
mfixishQW :: (w → CorecQW w e a) → CorecQW w e a
mfixishQW f = CorecQW (mfixishT (λ~(q', w) → unCorecQW (f w)))
```

Listing 27: Corecursive queues with return values

Chris Okasaki's algorithm uses two queues and a stack to relabel a tree with increasing integers. Our implementation makes use of two seperate, corecursive queues in place of the first-class queues used in the original paper. In both Chris

Okasaki's original implementation and ours, the stack is represented implicitly using the program stack. As the stack guards the second queue from falling off the end and entering a nonproductive loop, there is no need to track the length of this second queue explicitly.

Although our rendering of Chris Okasaki's solution is **implemented** using corecursion; the function itself is **not** corecursive. It cannot renumber the Stern-Brocot tree, for example. Instead it gets stuck in an infinite nonproductive loop. For a truly corecursive implementation of breadth-first renumbering, we recall Jones and Gibbons' solution [13][14] in Listing 29.

```
renum :: Integral int => Tree a b -> Tree int int
renum t = last q2
  where
    (_, q2) = runCorecQW (mfixishQW (\q2 -> trav 0 q2 t >> return []))
    trav n q t@(Leaf _)
      = do
          q' <- mtrav (n + 1) q
          mapCC ((Leaf n):) (return q')
    trav n q t@(Branch _ l r)
      = do
          enQ l >> enQ r
          mtrav (n + 1) q >>= \(r' : l' : q') ->
            mapCC ((Branch n l' r'):) (return q')
    mtrav n q = deQ >>= maybe (return q) (trav n q)
```

Listing 28: Chris Okasaki's Breadth-First Renumbering Algorithm

Performance of CorecQW

CorecQW exhibits a subtle performance discrepancy; due to the fact that we are returning lazy pairs, there are two paths of execution through the computation. Which path is followed depends on whether the consumer is demanding elements of the queue, or part of the result. This concept is fairly well known among logic programmers, but may be suprising to many functional programmers.

When applied to the running example of breadth first search, returning lazy pairs incurs either about 25% or 63% abstraction penalty compared to the original *CorecQ*, even if the extra result is (). Eager programmers accustomed to quality implementations of ML and Scheme are used to returning multiple values with little or no undue overhead. To be fair, GHC performs similar optimizations on

```
lazyRenum :: Integral int ⇒ Tree a b → Tree int int
lazyRenum t = t'
  where
    (ns, t') = loop (0 : ns, t)
    loop (n : ns, Leaf     _    ) = (n + 1 : ns  , Leaf     n      )
    loop (n : ns, Branch _ l r) = (n + 1 : ns'', Branch n l' r')
      where
        (ns' , l') = loop (ns , l)
        (ns'', r') = loop (ns', r)
```

Listing 29: Jones and Gibbons' Corecursive Renumbering Algorithm

strict pairs [15], and neither Scheme nor ML offer lazy tuples nativly. Supporting lazy tuples efficiently is a significantly harder problem.

As a thought experiment, I attempted to implement my own value return mechanism, by starting with the original *CorecQ* and using *unsafePerformIO* and *IORefs* to open up a "side channel." In the process, I broke the full laziness optimization, [16] which must be turned off in order for this code to terminate. It was instructive, as I'm suspicious I ended up creating something similar to what GHC is already doing.

The basic idea is that if we demand the result, we enter a thunk which forces a small bit of queue computation, and then re-reads itself. This process repeats until the queue computation terminates: then the thunk gets replaced with a concrete value which gets returned the next time the thunk re-reads itself. By enabling the trace output and running this code, you can see it in action.

The downside to this naive approach is that it exhibits an **inversion of demand**. The queue should be smart enough to realize that if a result is demanded, then it should demand it's own computation until a result (or part thereof) is returned, saving a number of indirect jumps.

Let me emphasize I am not advocating this style of programming, nor the use of this code! In fact, GHC's native tuples are faster! This code is simply to demonstrate the two code paths, and as such will produce different output depending on whether or not one demands the result.

This experiment appears to be a constant factor slower than GHC's tuples. It exhibits the same performance dissimilarity between the two code paths. It appears to work in the presence of *callCC*, but only implements *mapCont* for the queue, not the result. Thus an incremental *foldrByLevel* is not possible with this monad.

```haskell
type QSt r e = IORef r -> IORef [e] -> Int -> [e] -> [e]
newtype Q r e a = Q { unQ :: ((a -> QSt r e) -> QSt r e) }
instance Monad (Q r e) where
    return a = Q ($a)
    m >>= f  = Q (\k -> unQ m (\a -> unQ (f a) k))
unsafeRead  ref   = unsafePerformIO (readIORef  ref  )
unsafeWrite ref a = unsafePerformIO (writeIORef ref a)
unsafeNew   a     = unsafePerformIO (newIORef   a    )
instance Show e => MonadQueue e (Q r e) where
  enQ x = Q (\k r e !n xs -> let xs' = (k () r e $! n + 1) xs
                             in  trace ("enQ $ " ++ show x)
                                   (unsafeWrite e xs' `seq` (x : xs')))
  deQ = Q delta
    where
      delta k r e    0     xs     = k Nothing r e 0 xs
      delta k r e (n + 1) (x : xs) = trace ("deQ " ++ show x)
                                         (k (Just x) r e n xs)
runQ m = (trace "reading return value" `seq` unsafeRead r (), queue)
  where
    r = unsafeNew init
    init () = unsafePerformIO $ do
      trace "forcing computation\n" (return ())
      xs <- readIORef e
      force xs
      trace "reading return value\n" (return ())
      f <- readIORef r
      return (f ())
    e = unsafeNew queue
    queue = unQ m breakK r e 0 queue
force []      = return ()
force (_ : _) = return ()
breakK a r e n xs = trace ("setting return value: " ++ show a)
                      (unsafeWrite r (\() -> a) `seq` [])
```

Listing 30: Side channel thought experiment

Performance Measurements

Description	Time		-H500M		Bytes
	mean	σ	mean	σ	
levelOrder'	446	5	172	15	44.0
CorecQ	555	5	619	4	133.5
CorecQW _	696	5	1128	6	213.6
CorecQW ()	907	56	2235	11	213.6
Side Channel _	959	3	1171	7	228.7
Side Channel ()	1500	56	2171	7	276.4
STQ	1140	8	1087	14	371.2
TwoStack	1158	4	778	10	185.8
Okasaki	1553	7	1574	12	209.0
Data.Sequence	962	5	1308	5	348.1

Figure 4: Performance using GHC 6.10.3

Description	Time		-H500M		Bytes
	mean	σ	mean	σ	
levelOrder'	461	2	173	15	44.1
CorecQ	458	4	267	13	67.5
CorecQW _	526	5	713	5	141.2
CorecQW ()	781	62	1775	62	141.3

Figure 5: Performance using GHC 6.8.3

This was tested on an Intel Core 2 Duo T9550, and both GHC 6.10.3 and 6.8.3. The code that was used to produce these benchmarks is available on hackage as control-monad-queue. [17] The results of the tests can be found in Figures and 5.

Each of the variants in the table were run on the 34th fibonacci tree, which has 5.7 million branches. The functions were run 20 times, and the first few trials were discarded. The remaining trials were averaged, and the standard deviation σ was computed. Timing information, presented in milliseconds, was gathered using *System.getCPUTime*, which on the test system had a resolution of 10 milliseconds. The final column of the two tables gives the average number of bytes allocated for every *Branch*.

Note that the code in this paper was not benchmarked directly for a variety of reasons. Each description is essentially equivalent to *levelOrder″* (Listing 9) run with the appropriate monad. This means that the bottom four variants don't return anything useful. While this isn't fair for implementing a drop-in replacement for $levelOrder' :: Tree\ a\ b \to [\,Tree\ a\ b\,]$, it is more fair for comparing the relative performance of the queues themselves.

The tests were also attempted using the `-Hsize` option to set a suggested heap size and reduce the frequency of garbage collection; this did indeed reduce the percentage of time spent in the garbage collector, but this was usually more than offset in increased time spent in the mutator.

Related Work

The Glasgow Haskell Compiler provides *Data.Sequence*, which is based on 2-3 finger trees. [18] This offers amortized, asymptotically efficient operations to many kinds of operations on persistent sequences, and is much more general data structure than a queue.

Chris Okasaki [10] implements first-class real-time queues, even under persistent usage. It is interesting that this solution also makes essential use of laziness, and is based around the incremental reversal of lazy lists.

Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan [19] have a clever way of implementing a queue using delimited continuations. This employs the dynamic extent of *control* and *prompt*, as opposed to the static extent of *shift* and *reset*. This solution does not employ the use of circular programming.

Conclusions

For whatever reason, Lloyd Allison's queue is not widely appreciated within the modern functional programming community. This deserves to change, as corecursive queues are both academically interesting and practical. They are not as general as other queues, but when they fit a problem, they are an excellent choice. Thus they occupy an interesting place in the functional programmer's toolbox.

Acknowledgements

I'd like to thank Amr Sabry and Olivier Danvy for particularly useful comments, Matt Hellige for a fun discussion that lead to the *unsafePerformIO* thought experiment, Matt Morrow for the insight that *mapCont* could not be implemented on *CpSt*, Andres Löh for some assistance with LaTeX, Stefan Ljungstrand for

some criticism, and others including Dan Friedman, Will Byrd, Aziz Ghuloum, Ron Garcia, Roshan James, Michel Salim, and Michael Adams, who enthusiastically endured my often inept attempts at explaining this work before I really understood it.

I'd also like to acknowledge the giants whose shoulders made this work possible, including Richard Bird, Philip Wadler, Daniel Friedman and David Wise, the designers and implementors of Haskell and the Monad Template Library, and of course, Robin Milner and J. Roger Hindley.

References

[1] Lloyd Allison. Circular programs and self-referential structures. **Software Practice and Experience**, 19(2) (Feb 1989).
http://www.csse.monash.edu.au/~lloyd/tildeFP/1989SPE/.

[2] Andy Gill et al. The monad template library.
http://hackage.haskell.org/package/mtl.

[3] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. **Acta Informatica**, 21(3):pages 239–250 (Oct 1984).

[4] Kees Doets and Jan van Eijck. **The Haskell Road to Logic, Maths, and Programming**. King's College Publications (2004).

[5] Philip Wadler. Theorems for free! In **FPCA '89: Proceedings of the fourth international conference on functional programming languages and computer architecture**, pages 347–359. ACM, New York, NY, USA (1989).
http://homepages.inf.ed.ac.uk/wadler/topics/parametricity.html.

[6] Daniel P Friedman, Mitchell Wand, and Christopher T Haynes. **Essentials of Programming Languages**. MIT Press, 2 edition (2001).

[7] Daniel P Friedman and David S Wise. Cons should not evaluate it's arguments. **Automata, Languages, and Programming**, pages 257–284 (1976).
http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR44.

[8] Wouter Swietstra. Why attribute grammars matter. **The Monad Reader**, 4 (Jul 2005). http://www.haskell.org/sitewiki/images/8/85/TMR-Issue13.pdf.

[9] Chris Okasaki. **Purely Functional Data Structures**. Cambridge University Press (1998). http://www.eecs.usma.edu/webs/people/okasaki/pubs.html#cup98.

[10] Chris Okasaki. Simple and efficient purely functional queues and deques. **Journal of Functional Programming**, 5(4):pages 583–592 (Oct 1995).
http://www.eecs.usma.edu/webs/people/okasaki/pubs.html#jfp95.

[11] Leon P Smith. Writer monads via continuations. http://blog.melding-monads.com/2010/03/14/writer-monads-via-continuations/.

[12] Levent Erkök. **Value Recursion in Monadic Computations**. Ph.D. thesis, OGI School of Engineering, OHSU, Portland, Oregon (2002).
http://leventerkok.googlepages.com/erkok-thesis.pdf.

[13] Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In **ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming**, pages 131–136. ACM, New York, NY, USA (2000).
http://www.eecs.usma.edu/webs/people/okasaki/pubs.html#icfp00.

[14] Geraint Jones and Jeremy Gibbons. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Technical report, Dept of Computer Science, University of Auckland (1993).
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.4052.

[15] Clem Baker-Finch, Kevin Glynn, and Simon Peyton-Jones. Constructed product result analysis for haskell. **Journal of Functional Programming**, 14(2):pages 211–245 (Mar 2004).
http://research.microsoft.com/en-us/um/people/simonpj/Papers/cpr/.

[16] André L. M. Santos. **Compilation by transfomation in non-strict functional languages**. Ph.D. thesis, University of Glasgow (Jul 1995).
http://www.di.ufpe.br/~alms/ps/thesis.ps.gz.

[17] Leon P Smith. control-monad-queue.
http://hackage.haskell.org/package/control-monad-queue.

[18] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. **Journal of Functional Programming**, 16(2):pages 197–217 (2006).
http://www.soi.city.ac.uk/~ross/papers/FingerTree.pdf.

[19] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. **Science of Computer Programming**, 60(3):pages 274–297 (2006). http://www.brics.dk/RS/05/36/.