Lloyd Allison's Corecursive Queues: Why Continuations Matter

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

June 21, 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 $O(1)$ queues. In "Circular Programs and Self-Referential Structures," [1] Lloyd Allison uses *corecursion* 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 through the use of 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 demonstrates that *mapCont* cannot be expressed in terms of *callCC*, *return*, and ($\gg\!\!=$).

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 the 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, we corecursively define the Fibonacci trees instead.

```
data    Tree a b
     = Leaf    a
     | Branch b (Tree a b) (Tree a b)
        deriving (Eq, Show)
fib :: Int → Tree Int Int
fib n = fibs !! (n − 1)
   where
     fibs = Leaf 0 : Leaf 0 : zipWith3 Branch [1..] fibs (tail fibs)
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
```

Listing 1: Binary Trees

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 1. 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 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 in-
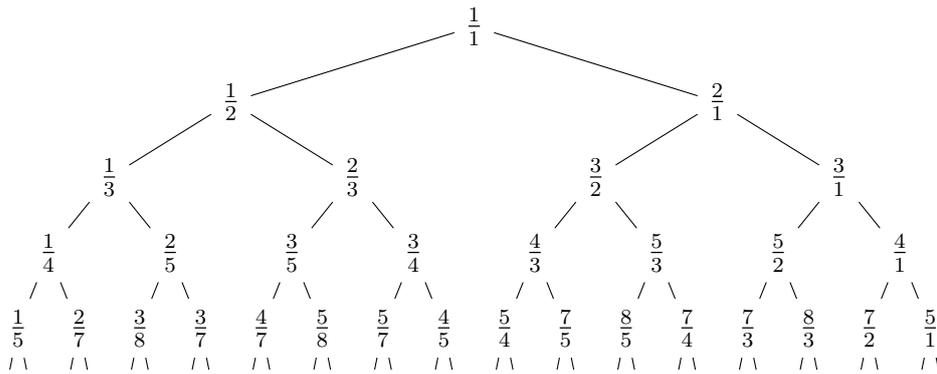
Figure 1: Five levels of the Stern-Brocot tree

finite, in the sense that every counterexample to this line of reasoning involves **partial data**. Common examples of partial data are infinite nonproductive loops, or the use of Haskell's *error* function, such as this definition of $\bot$:

$$\bot :: \forall\, a \circ a$$
$$\bot = error\ \texttt{"bottom"}$$

```
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 2: Useful functions

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 queues is given in Listing 3. A sample execution is given in Figure 2. This 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.

```
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                      head    =        [ ]
    explore   (n + 1) (Branch _ l r : head') = l : r : explore (n + 2) head'
    explore   (n + 1) (Leaf     _    : head') =        explore  n       head'
```

Listing 3: A Corecursive 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. GHC cannot currently allocate big ints inside a thunk, which is detrimental to performance. This issue is noticeable in certain micro-benchmarks, such as a complete traversal of a Fibonacci trees.

If one doesn't care about leaves or their contents, one might prefer a variant of *levelOrder* that does not enqueue the leaves. Listing 4 presents Lloyd Allison's original code, transliterated 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.
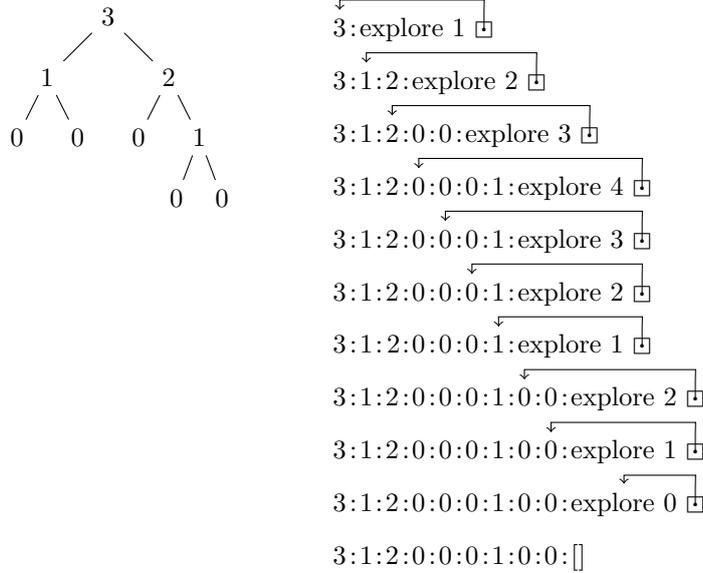
4

Figure 2: Calculating *levelOrder* (*fib* 5)

```
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 : head')
         = if (isBranch l)
            then if (isBranch r)
                  then   l : r : explore (n + 2) head'
                  else   l :     explore (n + 1) head'
            else  if (isBranch r)
                  then      r : explore (n + 1) head'
                  else          explore  n      head'
```

Listing 4: 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 ]
    enqs  []              n         head  =   deq    n      head
    enqs  (t : ts)        n         head
          | isBranch t                    =  t : enqs ts (n + 1) head
          | otherwise                     =     enqs ts  n       head
    deq                   0          _    =   []
    deq                   (n + 1) (t : head′) =   enqs ts′ n       head′
          where ts′ = childrenOf t
```

Listing 5: Removing the repeated branch

Because the code in Listing 5 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.

Although I didn't become aware of 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 felt that I immediately had a reasonable grasp of what was going on. In fact, I remember 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 over 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 functional-language interface is given by class *Queue* in Figure 6, and treats queues as first class values. Because Haskell is pure, all first class values are persistent. This gives

```
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 6: First-class versus monadic interfaces

us the freedom to enqueue an element, back up, enqueue a different element, and use both results in subsequent computations, as demonstrated by the *fork a b q =* (*enque a q, 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. 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 a monadic implementation.

In the examples so far, the logic that traverses the binary trees is commingled with the logic that defines the queue operations. Our goal is to separate these concerns. As the names suggest, *explore* in Listing 4 has no separation of concerns, while *enqs* and *deqs* in Listing 5 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, *enq1* and *enq2*, that differ only in their *continuation*. By parameterizing the continuation, we can refactor these into a single function. This process is demonstrated in Figure 7.

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.

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, and the logic is all but detangled. The last step should be easy, all we have to do is

```
levelOrder′3 t = q
  where
    q = enq1 [t] (0 :: Int) q
    enq2 [a, b] n head | isBranch a = a : (enq1 [b]) (n + 1) head
                       | otherwise  =   (enq1 [b])  n       head
    enq1 [a]    n head | isBranch a = a : (deq     ) (n + 1) head
                       | otherwise  =   (deq      )  n       head
    deq 0                        _       = [ ]
    deq (n + 1) (Branch _ l r :    head′) = enq2 [l, r] n head′
levelOrder′4 t = q
  where
    q = (enq t $ deq) (0 :: Int) q
    enq a k n q | isBranch a = a : k (n + 1) q
                | otherwise  =     k  n       q
    deq  0        _             = [ ]
    deq (n + 1 ) (Branch _ l r : q) = (enq l $ enq r $ deq) n q
```

Listing 7: Creating a reusable helper

pull the branch out of *enq* and define a separate helper function, recovering the original *explore* and fully reusable queue operations, as demonstrated in Listing 8.

In a sense, we have re-invented the traditional continuation passing style [6] 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* magically 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, in most cases there are reasons to prefer the latter. So we are ready to split Listing 8 into two parts: reusable corecursive queue operations versus the breadth-first tree traversal in Listings 9 and 10 respectively.

Of course, by using *childrenOf* analogous to *levelOrder′2*, and parameterizing it, we can produce breadth-first traversals of generic trees. Listing 11 shows two possible combinators, one that visits leaves and one that does not.

There are two stylistic points worth noting: 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 imple-

```
levelOrder' 5  t = q
   where
      q = (handle t (λ() → explore)) (0 :: Int) q
      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 + 1)  (e : q') =    k e   n        q'
```

Listing 8: A fully detangled traversal

```
type CorecQSt e = Int → [e] → [e]
newtype CorecQ e a
         = CorecQ {unCorecQ :: (a → CorecQSt e) → CorecQSt e}
instance Monad (CorecQ e) where
   return a = CorecQ (λk → k a)
   m ≫ f  = CorecQ (λk → unCorecQ m (λa → unCorecQ (f a) k))
instance MonadQueue e (CorecQ e) where
   enQ e = CorecQ (λk n q → e : (k () $! n + 1) q)
   deQ = CorecQ deq
      where
         deq k      0      q  = k Nothing 0 q
         deq k (n + 1) (e : q') = k (Just e) n q'
runCorecQ :: CorecQ element result → [element]
runCorecQ m = q
         where    q = unCorecQ m (λa n' q' → []) 0 q
```

Listing 9: A reusable implementation of Allison's queue

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

Listing 10: A binary tree traversal using generic queues

```
byLevel :: (MonadQueue a m) ⇒ (a → [a]) → [a] → m ()
byLevel childrenOf as = mapM_ enQ as ≫ explore
  where
    explore = deQ ≫= maybe (return ())
                    (λa → do
                            mapM_ enQ (childrenOf a)
                            explore               )
byLevel′ :: (MonadQueue a m) ⇒ (a → [a]) → [a] → m ()
byLevel′ childrenOf as = mapM_ handle as ≫ explore
  where
    handle a = when (hasChildren a) (enQ a)
    explore = deQ ≫= maybe (return ())
                    (λa → do
                            mapM_ handle (childrenOf a)
                            explore                  )
    hasChildren = ¬ ∘ null ∘ childrenOf
```

Listing 11: Traversals over generic trees

mented: 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 happens to be relatively small potatoes here. It is far more important that *enQ* and *deQ* get inlined. Because we are using typeclasses to abstract over the queue operations, inlining these operations is a clumsy thing to do in GHC. Indeed, ML-style functors are a far superior choice for this type of abstraction, from both the point of view of expression and implementation. Also, for some reason, the use of *childrenOf* in *byLevel*, even when inlined, represents a significant abstraction penalty in *byLevel*, whereas the usage of *childrenOf* in *levelOrder′2* does not.

Note the type of *runCorecQ*. A monadic interface enables one to declaratively specify side effects, but to be ultimately useful, these side effects must somehow be observed. In this case, the only aspect of the computation that can be observed is the list of elements enqueued. In particular, the final return value of *levelOrder″* cannot be observed. This limits the applicability of Listing 10

Wrapping the traditional two-stack queue in a state monad, or using local, explicitly mutable state as provided by *Control.Monad.ST* would be much more obvious implementations of the monadic interface. As we will see in the next section, these have the same semantics for *enQ* and *deQ*, but *observe* a completely different aspect of the computation.

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.

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.

Queues via explicit mutation

```
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 12: Queues via imperative linked lists

Of course, 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*.

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

Note the difference in type of *runSTQ* versus *runCorecQ*. These implementa-

tions observe different aspects of the computation. The function *byLevel* enqueues
the leaves of a tree while *byLevel′* does not; this difference can be observed using
runCorecQ, but from the point of view of *runSTQ*, these two functions are obser-
vationally equivalent; namely they both return () if the forest of trees is finite, and
diverge otherwise. This particular behavior is not particularly useful.

If one is interested in things other than thermal output and timing information,
a more conventional alternative would be to thread a value through *byLevel*. List-
ing 13 defines *foldrByLevel*, a function for computing right folds over breadth-first
traversals.

$$foldrByLevel \; :: (MonadQueue \; a \; m)$$
$$\Rightarrow (a \rightarrow [\,a\,]) \rightarrow (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [\,a\,] \rightarrow m \; b$$
$$foldrByLevel \; childrenOf \; f \; b \; as = mapM_- \; enQ \; as \gg explore$$
> **where**
> $$explore = deQ \ggg maybe \; (return \; b)$$
> $$(\lambda a \rightarrow \textbf{do}$$
> $$mapM_- \; enQ \; (childrenOf \; a)$$
> $$b \leftarrow explore$$
> $$return \; (f \; a \; b) \qquad\qquad)$$

$$prop\_foldrByLevel \; childrenOf \; f \; b \; as =$$
$$foldr \; f \; b \; (runCorecQ \; (byLevel \; childrenOf \; as))$$
$$\equiv \quad runSTQ \; (foldrByLevel \; childrenOf \; f \; b \; as)$$

Listing 13: Right folds over level-order 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 over
elements as they are dequeued, we could instead fold over elements before they are
enqueued. This computes the same fold, but now only nodes with branches need
to be put into the queue.

$$prop\_foldr\_ByLevel \; childrenOf \; f \; b \; as =$$
$$foldr \; f \quad b \; (runCorecQ \; (byLevel \quad childrenOf \; as))$$
$$\equiv \quad foldr \; f' \; b \; (runCorecQ \; (byLevel' \; childrenOf \; as))$$
> **where**
> $$f' \; a \; b = foldr \; f \; b \; (childrenOf \; a)$$

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

```
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

```
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
prop_foldrByLevel' childrenOf f b as =
        runSTQ (foldrByLevel  childrenOf f b as)
    ≡   runSTQ (foldrByLevel' childrenOf f b as)
```

Listing 15: Computing the same fold without enqueuing leaves

to this configuration: if we use *runCorecQ* to get the elements enqueued, then *byLevel* is equivalent to *foldrByLevel*, and similarly for the other two, as the return result does not matter. However, this equivalency is reversed for *runSTQ*, as the enqueued elements does not matter.

Two-Stack Queues and Monads

This section briefly reviews the traditional two stack queue; and explores three alternative monads that can be built around them. This section also discusses the consequences of monad tranformers; namely that transformers are not robust abstractions.

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 $O(n)$ enqueues. Alternately, one could store the queue in reverse, which makes enqueue an $O(1)$ operation, but then peeking at or removing the front element then becomes a $O(n)$ 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 do not destroy the queue as it existed. While *deque* is still $O(n)$ in the worst case, it works very well in practice because on average, *deque* is actually $O(1)$, 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 $O(1)$ 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 usually slower than either two-stack or corecursive queues, often significantly so.

It is well known that a monadic interface is able to enforce linear, non-persistent usage of data structures. By wrapping our two-stack queue in a state monad, and using only monadic enqueues and dequeues, we can guarantee amortized $O(1)$

```
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 16: Two-Stack Queues

operations.

```
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 17: First-class Queues inside *Control.Monad.State.Lazy*

Again, according to *runStateQ*, *byLevel* and *byLevel'* differ in run-time behavior, but are otherwise observationally equivalent. We can tweak this implementation to also keep track of the elements enqueued by using the Writer monad and the State transformer monad.

Writer monads 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.

Pretend for a moment that we don't know about *CorecQ* yet, and note the duplication of functionality. Essentially, the Writer monad creates something that's essentially the queue itself. Indeed, this was the other though that was running through my head when I first wrote something analogous to *levelOrder'2*; I started

```
newtype DebugQ q e a
    =       DebugQ (StateT (q e) (Writer [e]) a)
            deriving (Monad)
instance Queue q ⇒ MonadQueue e (DebugQ q e) where
    enQ e = DebugQ (tell [e] ≫ modify (enque e))
    deQ = DebugQ (StateT (return ∘ deque))
runDebugQ :: (Queue q) ⇒ DebugQ q element result → (result, [element])
runDebugQ (DebugQ m)
    = let ((result, final_queue), queue) = runWriter (runStateT m empty)
      in (result, queue)
```

Listing 18: Observing the list of elements enqueued

writing a traversal using first-class queues that output a list of nodes as they were visited, and although I was not using monads, I noticed this duplication and saw the opportunity to eliminate it using circular programming.

It should be noted that the *StateT* monad transformer cannot enforce linear usage of state for arbitrary monad parameters. For example, we can regain the ability to use state persistently by choosing the nondeterministic list monad. Monad transformers are not robust abstractions! This is a recurring theme in this paper, for which we will see two more examples.

The lazy state monad is notoriously inefficient on current implementations of Haskell; one is *much* better off using the strict state monad. The cps'ed 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. And in fact, we have used this particular idiom many times already in this paper, including inside *CorecQ* and *STQ*.

This state monad chooses to hide the type of the final result in a rank-2 type. The consequences of this choice is that we cannot implement some of the control operators such as *callCC* and *mapCont*, and that this monad cannot break out of the computation prematurely, analogous to *deq* in *levelOrder′5* in listing 8. The initial continuation that *runCpSt* passes to the computation is guaranteed to be called exactly once, if the computation terminates.

Another way to implement a continuation passing state monad is simply by using *Control.Monad.Cont.Cont* as the parameter to *StateT*. This choice exposes the result type, enabling the control operations proscribed in the last paragraph. Indeed, this is the only major difference between *CpSt s* and *StateT s (Cont r)*; in fact the *get* and *put* operations, when compiled *ghc − O2*, compile into almost the same machine code. The only difference is that the continuation $a → s → r$ is

```
newtype CpSt s a
  = CpSt { unCpSt :: ∀ r ∘ (a → s → r) → s → r }
instance Monad (CpSt s) where
  return a = CpSt (λk → k a)
  m ≫= f  = CpSt (λk → unCpSt m (λa → unCpSt (f a) k))
instance MonadState s (CpSt s) where
  get    = CpSt (λk s → k s  s )
  put s' = CpSt (λk _ → k () s')
runCpSt :: CpSt s a → s → (a, s)
runCpSt m s0 = unCpSt m (λa s → (a, s)) s0
```

Listing 19: A Continuation Passing State Monad

tupled and not curried. We use this observation to implement a new state monad
to encapsulate first-class queues, in listing 20

```
newtype CpStQ r e a = CpStQ (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 20:

However, there is a big caveat to this latest monad: not only have we added a
continuation semantics to the state semantics, we have *changed part of the existing
semantics from lazy to strict!* This variant does not return its result incrementally;
rather, nothing is returned until the computation terminates. This is easily ob-
served by the fact that *runStateQ (getUnion foldrByLevel sternBrocot)* returns
useful data, while *runCpStQ (getUnion foldrByLevel sternBrocot)* gets stuck in
an infinite nonproductive loop. This is in stark contrast to the use of *Writer* in
DebugQ, which left the state semantics undisturbed. Not only are monad trans-
formers not robust abstractions, they are not robust in any sense of the word!

Fortunately, *mapCont* comes to our rescue here. The lazy semantics can be
recovered by tweaking *foldrByLevel* to use this control operator to return pieces of
the final result incrementally instead of all at once. This would not have been an
option had we simply used the strict state monad, which is a great deal faster than

it's lazy cousin. This is demonstrated in listing 21, which is capable of computing folds over the Stern-Brocot tree.

```
class MonadMapCC a m | m → a where
  mapCC :: (a → a) → m b → m b
instance MonadMapCC r (CpStQ r e) where
  mapCC f (CpStQ m) = CpStQ (StateT (λs → mapCont f (runStateT m s)))
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 21: Modifying *foldrByLevel'* to handle the Stern-Brocot tree

Allison's queues in direct style

There are two approaches 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 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". 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*.

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

This is notable because I am confident that enQ cannot be expressed in terms of $callCC$, ($\ggg$), 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 this is so; but it turns out to be fairly trivial: $callCC$, ($\ggg$), and $return$ simply offer no way to add to introduce a 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 $mapCont$.

newtype $CpSt'\ s\ a$
$$= CpSt'\ \{\,runCpSt' :: \forall\ r \circ (\forall\ r' \circ a \rightarrow s \rightarrow r') \rightarrow s \rightarrow r\,\}$$

$$mapCpSt' :: (\forall\ a \circ a \rightarrow a) \rightarrow CpSt'\ s\ b \rightarrow CpSt'\ s\ b$$
$$mapCpSt'\ f\ m = CpSt'\ (\lambda k \rightarrow f\ (runCpSt'\ m\ k))$$

Now, thanks to free theorems, there are only three inhabitants of type $\forall\ a \circ a \rightarrow a$: id, $const\ \bot$ and $\bot$, none of which are useful. Of course, this monad itself isn't particularly useful itself: thanks to parametericity, one can't ever get results out of this monad; the only way to see what's going on inside is to cheat and use things such as $error$. But that isn't the point: because it admits ($\ggg$), $return$, and $callCC$, but not a $mapCont$, it shows that $mapCont$ can't be expressed in terms of the former operations.

Now, the type of our $CpSt$ idiom is $(a \rightarrow s \rightarrow r) \rightarrow s \rightarrow r$, which is isomorphic to $ContT\ r\ (Reader\ s)\ a$, so this would be a plausible place to start. Listing 22 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.

$$getReader \quad = lift\ ask$$
$$setReader \quad = modReader \circ const$$
$$modReader\ f = callCC\ (\lambda k \rightarrow local\ f\ (k\ ()))$$
$$stepReader\ f = \mathbf{do}$$
$$\quad st \leftarrow getReader$$
$$\quad \mathbf{let}\ (r, st') = f\ st$$
$$\quad setReader\ st'$$
$$\quad return\ r$$

Listing 22: State effects with readers and $callCC$

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.

```
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 23: 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′ m) = q
   where q = runReader (runContT m endpoint) (Len 0 q)
         endpoint _ = return []
```

Listing 24: Enqueue and dequeue in direct style

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* has many pitfalls associated with it. Detailing all the hazards would fill many, many pages. 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.

```
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 m = mfix (λq → run m end_point (Len 0 q))
  where
    end_point _ = return []
    run (CorecQT m) k st = runReader (runContT m k) st
```

Listing 25: A mostly broken monad transformer

The $mfix :: a \rightarrow m\ a$ used here is the topic of Levent Erkök's Ph.D. thesis, "Value Recursion in Monadic Computations". [11] The thesis argues that there is no $mfix$ on continuations. Note that this paper does not contradict this conjecture, we are using $mfix$ to define the run operation, not defining an $mfix$ for $CorecQT$.

This paper also is on the topic of value recursion, but by contrast Allison's queue *requires* the use of continuations. Therefore we appear to be talking about an alternate form of value recursion, and that it seems that CPS enables some varieties of value recursion while disabling others.

However, the lazy state monad has an $mfix$ operation, and we have already seen an implementation of the *MonadQueue* interface using *TwoStackQ*s and *Cont.Monad.State*. Therefore it appears as though there is a sensible semantics for an $mfix$ on the *MonadQueue* interface, even if the implementation happens to be corecursive in nature.

Whether or not a corecursive implementation can actually compute this semantics for $mfix$ is a very interesting question. Perhaps $CorecQ$ 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 $mfix$.

One might conjecture, as this paper tacitly assumes, that there is no $mfix$ over any monad implemented using continuations. This would imply that $CorecQ$ cannot be the underlying Monad of $CorecQT$, ruling out an intuitive way that one might 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 return results other than the queue itself from our corecursive implementation. Unfortunately, because the writer monad expects monoids, this approach isn't really suitable for preserving the result semantics of *STQ*, *StateQ*, *CpStQ* and friends.

Control.Monad.Writer e a is just a **newtype** alias for (a, e), so we will simply use lazy pairs and start over, instead of using our monad transformer directly. Also, now that we can return results, it makes sense to implement *mapCont* for this monad as well.

```
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 end_point (Len 0 q)
         end_point w = return ([], w)
         run m k st = runReader (runContT (unCorecQW m) k) st
```

Listing 26: Corecursive queues with return values

This code has a subtle performance trait; 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 the majority of functional programmers.

When applied to the running example of breadth first search, returning lazy pairs incurs either about 25

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, [13] 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.

Performance Measurements

| Description | Time (ms) | | -H500M | | Bytes allocated |
GHC 6.10.3	mean	σ	mean	σ	per *Branch*
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
GHC 6.8.3					
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

```
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 (λk → unQ m (λa → 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 (λk 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 (λ() → a) `seq` [])
```

Listing 27: Side channel thought experiment

This was tested on an Intel Core 2 Duo T9550, and GHC 6.10.3. The code that was used to produce these benchmarks is available on Hackage as control-monad-queue. 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 was computed. Timing information was gathered using *System.getCPUTime*, which on the test system had a resolution of 10 milliseconds.

Note that the code in this paper was not benchmarked directly for a variety of reasons. Each description is essentially equivalent to *levelOrder″* (Listing 10) 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 → [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 more time spent in the mutator.

Related Work

The Glasgow Haskell Compiler provides *Data.Sequence*, which is based on 2-3 finger trees. [14] 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 [15] 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

References

[1] Lloyd Allison. Circular programs and self-referential structures. **Software Practice and Experience**, 19(2) (Feb 1989).

[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).

[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).

[8] Wouter Swietstra. Why attribute grammars matter. **The Monad Reader**, 4 (Jul 2005).

[9] Chris Okasaki. **Purely Functional Data Structures**. Cambridge University Press (1998).

[10] Chris Okasaki. Simple and efficient purely functional queues and deques. **Journal of Functional Programming**, 5(4):pages 583–592 (Oct 1995).

[11] Levent Erkök. **Value Recursion in Monadic Computations**. Ph.D. thesis, OGI School of Engineering, OHSU, Portland, Oregon (2002).

[12] 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).

[13] André L. M. Santos. **Compilation by transfomation in non-strict functional languages**. Ph.D. thesis, University of Glasgow (Jul 1995).

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

[15] 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).