Functional Data Structures

Chris Okasaki*

School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, Pennsylvania, USA 15213
(e-mail: cokasaki@cs.cmu.edu)

1 Introduction

Efficient data structures have been studied extensively for over thirty years. Nowadays, when a C programmer needs an efficient data structure for a particular problem, he or she can often simply look one up in any of a number of good textbooks or handbooks. However, the same cannot be said for the ML or Haskell programmer. Although some imperative data structures can be adapted quite easily to a functional setting, most cannot.

Why should functional data structures be any more difficult to design and implement than imperative ones? There are two basic problems. First, from the point of view of designing and implementing efficient data structures, functional programming's stricture against destructive updates (assignments) is a staggering handicap, tantamount to confiscating a master chef's knives. Like knives, destructive updates can be dangerous when misused, but tremendously effective when used properly. Certainly the functional programmer expects to reap substantial benefits by giving up destructive updates, but we must not be blind to the potential costs of this tradeoff.

The second difficulty is that functional data structures are expected to be more flexible than their imperative cousins. In particular, when we update an imperative data structure we typically accept that the old version of the data structure will no longer be available, but, when we update a functional data structure, we expect that both the old and new versions of the data structure will be available for further processing. A data structure that supports multiple versions is called *persistent*, while a data structure that allows only a single version at a time is called *ephemeral* [7]. Functional programming languages have the curious property that *all* data structures are automatically persistent. Imperative data structures are typically ephemeral, but when a persistent data structure is required, imperative programmers are not surprised if the persistent data structure is more complicated and perhaps even asymptotically less efficient than its ephemeral counterparts.

Exercise 1. List five situations in which persistence might be useful, even for an imperative programmer. ◇

* Research supported by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050.

In spite of these difficulties, however, researchers have developed numerous functional data structures that are asymptotically as efficient as the best imperative solutions for the same problems. In this tutorial, we will explore efficient functional implementations of several common abstractions, including FIFO queues, catenable lists, and mergeable heaps (priority queues). The emphasis will always be on developing data structures that achieve a good compromise between simplicity and efficiency in practice.

Notation We will present all source code in Standard ML [20], extended with the following primitives for lazy evaluation:

```
type α susp
val delay : (unit → α) → α susp
val force  : α susp → α
```

These primitives are actually supported by several implementations of Standard ML, including Standard ML of New Jersey. As a notational convenience, we will write *delay* (*fn* () ⇒ e) as \$$e$, where the scope of \$ extends as far to the right as possible.

We will also assume the presence of the small streams library shown in Figure 1. This library is far from complete, containing only those stream operations we will actually use in this paper. Note that the *cons* operation supplied by this library is strict, not lazy. In fact, the only lazy operations in this library are $+\!\!+$ (infix append) and *reverse*.

2 FIFO Queues

Stacks and queues are usually the first two data structures studied by beginning computer science students. The typical imperative implementation of (unbounded) stacks as linked lists translates very naturally to a functional setting. However, the typical imperative implementation of (unbounded) queues as linked lists does not because it uses destructive updates at the end of the list. Thus, queues are perhaps the simplest example of a data structure whose implementation in a functional setting is substantially different from its implementation in an imperative setting. For this reason, functional queues have been widely studied [11, 9, 3, 23, 24].

A minimal signature for queues appears in Figure 2. The three main operations are *snoc (q, x)*, which adds an element x to the rear of queue q; *head (q)*, which extracts the first element of q; and *tail (q)*, which deletes the first element of q. The signature also includes a value *empty* representing the empty queue, and a predicate *isEmpty*. To be practical, a queue library should contain many more utility functions, but these can all be defined in terms of the above primitives.

```
signature STREAM =
sig
   type α Stream
   exception EMPTY
   val empty   : α Stream
   val isEmpty : α Stream → bool
   val cons    : α × α Stream → α Stream          (* strict cons *)
   val head    : α Stream → α          (* raises EMPTY if stream is empty *)
   val tail    : α Stream → α Stream   (* raises EMPTY if stream is empty *)
   val ++      : α Stream × α Stream → α Stream  (* infix append *)
   val reverse : α Stream → α Stream
end

structure S : STREAM =
struct
   datatype α StreamCell = Nil | Cons of α × α Stream
   withtype α Stream = α StreamCell susp
   exception EMPTY
   val empty = $Nil
   fun isEmpty s = case force s of Nil ⇒ true | _ ⇒ false
   fun cons (x, s) = $Cons (x, s)
   fun head s = case force s of Nil ⇒ raise EMPTY | Cons (x, s) ⇒ x
   fun tail s = case force s of Nil ⇒ raise EMPTY | Cons (x, s) ⇒ s
   fun s ++ t = $case force s of
                    Nil ⇒ force t
                  | Cons (x, s) ⇒ Cons (x, s ++ t)
   fun reverse s = let fun rev (Nil, t) = force t
                         | rev (Cons (x, s), t) = rev (force s, cons (x, t))
                   in $rev (force s, empty) end
end
```

Fig. 1. A small streams library.

```
signature QUEUE =
sig
   type α Queue
   exception EMPTY
   val empty   : α Queue
   val isEmpty : α Queue → bool
   val snoc    : α Queue × α → α Queue
   val head    : α Queue → α         (* raises EMPTY if queue is empty *)
   val tail    : α Queue → α Queue   (* raises EMPTY if queue is empty *)
end
```

Fig. 2. Signature for queues.

2.1 A Partial Solution

The most common implementation of queues [9, 3] is as a pair of lists, one representing the front portion of the queue and the other representing the rear portion of the queue in reverse order.

```
datatype α Queue = Queue of α list × α list
```

In this representation, the first element of the queue is the head of the front list and the last element of the queue is the head of the rear list. These locations can be accessed very quickly.

```
fun snoc (Queue (f, r), x) = Queue (f, x :: r)
fun head (Queue (x :: f, r)) = x
fun tail (Queue (x :: f, r)) = Queue (f, r))
```

However, what happens when we attempt to take the head or tail of a queue whose front list is empty? If the rear list is also empty, then the queue is empty, so we raise an exception. Otherwise, the first element of the queue is the *last* element of the rear list. In this case, we reverse the rear list, install the result as the new front list, and try again.

```
fun head (Queue ([], [])) = raise EMPTY
  | head (Queue ([], r)) = head (Queue (rev r, []))
  | head (Queue (x :: f, r)) = x
fun tail (Queue ([], [])) = raise EMPTY
  | tail (Queue ([], r)) = tail (Queue (rev r, []))
  | tail (Queue (x :: f, r)) = Queue (f, r)
```

Note that the recursive calls in the second clauses of both *head* and *tail* will always fall through to the third clauses because *rev r* is guaranteed to be non-empty. Hence, we could easily optimize these calls by inlining the appropriate code from the third clauses. However, we won't bother because there is a more disturbing inefficiency. It is very common to ask for both the head and tail of the same queue. But, as it stands, this might result in reversing the same rear list twice! To prevent this, we will maintain the invariant that the front list is empty only if the entire queue is empty. Then the *head* operation at least will never need to reverse the rear list. The *snoc* operation must also be modified to obey the invariant.

```
fun snoc (Queue ([], _), x) = Queue ([x], [])
  | snoc (Queue (f, r), x) = Queue (f, x :: r)
fun head (Queue ([], _)) = raise EMPTY
  | head (Queue (x :: f, r)) = x
fun tail (Queue ([], _)) = raise EMPTY
  | tail (Queue ([x], r)) = Queue (rev r, [])
  | tail (Queue (x :: f, r)) = Queue (f, r)
```

Note the use of wildcards here. We know by the invariant that if the front list is empty, then so is the rear list, so we avoid the redundant check by using a wildcard.

A somewhat cleaner way to write this code is in terms of a *pseudo-constructor* (also called a *smart constructor* [1]) that enforces the invariant. This pseudo-constructor, called *queue*, takes the place of the real constructor *Queue* but verifies that the front list is not empty.

```
fun queue ([], r) = Queue (rev r, [])
  | queue (f, r) = Queue (f, r)
```

Now, we replace *Queue* with *queue* in the definitions of *snoc* and *tail*.

```
fun snoc (Queue (f, r), x) = queue (f, x :: r)
fun tail (Queue ([], _)) = raise EMPTY
  | tail (Queue (x :: f, r)) = queue (f, r))
```

Note that the real constructor *Queue* is still used for pattern matching. The complete source code for this implementation is given in Figure 3.

```
structure Queue0 : QUEUE =
struct
  datatype α Queue = Queue of α list × α list
       (* Invariant: null f implies null r *)

  exception EMPTY

  val empty = Queue ([], [])
  fun isEmpty (Queue (f, r)) = null f

  fun queue ([], r) = Queue (rev r, [])
    | queue (f, r) = Queue (f, r)

  fun snoc (Queue (f, r), x) = queue (f, x :: r)
  fun head (Queue ([], _)) = raise EMPTY
    | head (Queue (x :: f, r)) = x
  fun tail (Queue ([], _)) = raise EMPTY
    | tail (Queue (x :: f, r)) = queue (f, r)
end
```

Fig. 3. A common, but not always efficient, implementation of queues.

This implementation is easy to analyze using traditional techniques of amortization [27]. The basic idea is to save a credit with every *snoc* operation. Then, every queue has a number of credits equal to the length of the rear list and these credits can be used to pay for reversing the rear list when the front list becomes empty. By this argument, we see that every queue operation requires only $O(1)$ amortized time.

Unfortunately, traditional analysis techniques for amortization were developed in an imperative setting and rely on a hidden assumption that all data structures are ephemeral, that is, that sequences of operations on data structures are single-threaded [25]. However, this assumption is routinely violated in a functional setting, where all data structures are automatically persistent. In the next section, we modify this implementation of queues to support persistence efficiently and introduce a proof technique for proving amortized bounds for persistent data structures.

Exercise 2. Show that the above implementation of queues takes greater than $O(1)$ amortized time per operation by constructing a (non-single-threaded) sequence of n operations that requires greater than $O(n)$ time to execute. $\diamond$

2.2 Efficient Persistent Queues

The above queues are very efficient as long as they are used in a single-threaded fashion. Many applications obey this restriction, but for a general-purpose queue library, we would prefer an implementation that is efficient even when used persistently.

As discussed in [22, 24], lazy evaluation is the key to integrating amortization and persistence. The memoization implicit in lazy evaluation allows the same work to be shared between different threads instead of being repeated for each thread. (Here the term "threads" refers not to concurrency, but to different paths of data flow.) The first step in modifying queues to deal with persistence is thus to use streams instead of lists.[1] To simplify later operations, we also explicitly track the lengths of the two streams.

```
datatype α Queue = Queue of α S.Stream × int × α S.Stream × int
```

Note that a pleasant side effect of maintaining this length information is that we can trivially support a constant-time *size* operation.

Next, we strengthen the invariant to guarantee that the front stream is always at least as long as the rear stream. We call this the *balance invariant*. As a special case, the balance invariant implies that if the front stream is empty, then so is the rear stream. When the rear stream becomes one longer than the front stream, we perform a *rotation* by reversing the rear stream and appending it to the front stream. The major queue operations are now given by

```
fun snoc (Queue (f, lenf, r, lenr), x) = queue (f, lenf, S.cons (x, r), lenr+1)
fun head (Queue (f, lenf, r, lenr)) =
        if lenf > 0 then S.head f else raise EMPTY
fun tail (Queue (f, lenf, r, lenr)) =
        if lenf > 0 then queue (S.tail f, lenf−1, r, lenr) else raise EMPTY
```

where the pseudo-constructor *queue* is defined as

[1] Actually, only the front list must be changed to a stream. The rear list could remain a list, but for simplicity, we will use streams for both.

```
fun queue (f, lenf, r, lenr) =
        if lenr ≤ lenf then Queue (f, lenf, r, lenr)
        else Queue (S.++ (f, S.reverse r), lenf+lenr, S.empty, 0)
```

The complete code for this implementation appears in Figure 4.

```
structure Queue1 : QUEUE =
struct
   datatype α Queue = Queue of α S.Stream × int × α S.Stream × int
           (* Invariant: |f| ≥ |r| *)

   exception EMPTY

   val empty = Queue (S.empty, 0, S.empty, 0)
   fun isEmpty (Queue (f, lenf, r, lenr)) = (lenf = 0)

   fun queue (f, lenf, r, lenr) =
           if lenr ≤ lenf then Queue (f, lenf, r, lenr)
           else Queue (S.++ (f, S.reverse r), lenf+lenr, S.empty, 0)

   fun snoc (Queue (f, lenf, r, lenr), x) = queue (f, lenf, S.cons (x, r), lenr+1)
   fun head (Queue (f, lenf, r, lenr)) =
           if lenf > 0 then S.head f else raise EMPTY
   fun tail (Queue (f, lenf, r, lenr)) =
           if lenf > 0 then queue (S.tail f, lenf–1, r, lenr) else raise EMPTY
end
```

Fig. 4. Efficient persistent queues.

Exercise 3. Verify that this implementation of queues takes only $O(n)$ time to execute your sequence of operations from Exercise 2. ◇

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

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

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

This informal argument suggests that these queues require only $O(1)$ amortized time per operation even when used persistently. We can formalize this proof using a *debit argument*. For every suspension, we allocate enough debits to cover the cost of forcing the suspension. Then, we discharge $O(1)$ debits with every operation. We prove that we never force a suspension before we have discharged all of its debits.

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

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

The $2i$ term guarantees that all debits on the first node of the front stream have been discharged (since $d_0 = D_0 \leq 2 \cdot 0 = 0$), so this node may be forced at will (for instance, by a *head* operation). The $|f| - |r|$ term guarantees that all debits in the entire queue have been discharged whenever the streams are of equal length (i.e., just before the next rotation).

Now, every *snoc* operation that does not cause a rotation simply adds a new element to the rear stream, increasing $|r|$ by one and decreasing $|f| - |r|$ by one. This will cause the invariant to be violated at any node for which D_i was previously equal to $|f| - |r|$. We can restore the invariant by discharging the first debit in the queue, which decreases every subsequent cumulative debit total by one. Similarly, every *tail* operation that does not cause a rotation simply removes an element from the front stream. This decreases $|f|$ by one (and hence $|f| - |r|$ by one), but, more importantly, it decreases the index i of every remaining node by one, which in turn decreases $2i$ by two. Discharging the first two debits in the queue restores the invariant.

Finally, consider a *snoc* or *tail* that causes a rotation. Just before the rotation, we are guaranteed that all debits in the queue have been discharged, so, after the rotation, the only debits are those generated by the rotation itself. If $|f| = m$ and $|r| = m+1$ at the time of the rotation, then there will be m debits for the append and $m+1$ debits for the reverse. The append function is *incremental* (i.e., does only one step at a time and delays the rest) so we place one debit on each of the first m nodes. On the other hand, the reverse function is *monolithic* (i.e., once begun, it runs to completion) so we place $m+1$ debits on node m, the first node of the reversed stream. Thus, the debits are distributed such that

$$d_i = \begin{cases} 1 & \text{if } i < m \\ m+1 & \text{if } i = m \\ 0 & \text{if } i > m \end{cases} \qquad \text{and} \qquad D_i = \begin{cases} i+1 & \text{if } i < m \\ 2m+1 & \text{if } i \geq m \end{cases}$$

This distribution violates the invariant at both node 0 and node m, but discharging the debit on the first node restores the invariant. □

This proof uses a debit argument in the style of [22, 24] instead of a traditional credit argument [27]. Debit arguments are more suitable for analyzing persistent data structures because, although a single credit cannot be spent more than once, it does no harm to discharge the same debit more than once. Debit arguments allow you to reason about the running time of each thread individually, without worrying about inter-thread dependencies. The intuition is that different threads amortize the cost of lazy operations either over separate sequences (as when $k = 0$ in the earlier discussion), or over overlapping sequences (as when $1 \leq k \leq m$ in the earlier discussion). In the case of separate sequences, each debit is discharged only once, but in the case of overlapping sequences, each debit may be discharged more than once. The key is that whenever we force a suspension, we know that all of its debits have been discharged *at least* once. Memoization guarantees that work is not duplicated in separate threads unless it has also been paid for separately in the two threads.

Exercise 4 (Output-restricted Deques). Extend the queues in Figure 4 with a *cons* operation that adds an element to the front of a queue instead of the rear. (A data structure that allows elements to be inserted at both the front and the rear, but removed only from the front, is called an *output-restricted deque*). How does this new operation interact with the balance invariant? With the debit invariant? ◇

Exercise 5 (Min-Queues). A *min-list* is a list data structure that additionally supports a *findMin* operation that returns the minimum element in the list. (This differs from a priority queue in that there is no *deleteMin* operation.) Min-lists can be implemented by maintaining a secondary list of rightward minima. In other words, a min-list containing the elements $x_1 \cdots x_n$ would consist of the list $[x_1, \ldots, x_n]$ and a secondary list $[y_1, \ldots, y_n]$, where $y_i = \text{Min}_{j=i}^{n} x_j$. Note that, for $i < n$,

$$\begin{aligned} y_i &= \text{Min}_{j=i}^{n} x_j \\ &= \min(x_i, \text{Min}_{j=i+1}^{n} x_j) \\ &= \min(x_i, y_{i+1}) \end{aligned}$$

Therefore, *cons* can be implemented as

```
fun cons (x, ([ ], [ ])) = ([x],[x])
  | cons (x, (xs, ys)) = (x :: xs, min (x, hd ys) :: ys)
```

(a) Use these ideas to implement min-streams, in the style of Figure 1. You may assume that the elements are integers. Be careful to make the ++ (append) and *reverse* operations as lazy as possible.
(b) Use min-streams to implement min-queues, in the style of Figure 4. Each operation should take only $O(1)$ amortized time. ◇

Exercise 6 (Deques). A *double-ended queue* (also called a *deque*) supports insertion and removal of elements at both ends. We can adapt the implementation of queues in Figure 4 to support deques by making it symmetric. Currently, there are two sources of asymmetry. First, the balance invariant prevents the rear stream from getting too long with respect to the front stream. For deques, we should instead prevent either stream from getting too long with respect to the other. In particular, for some constant $c > 1$, we should maintain the following invariant:

$$|f| \leq c|r| + 1 \ \wedge \ |r| \leq c|f| + 1$$

(The "+1" in each of these constraints allows for deques of size 1.) Second, rotations move all the elements in the queue to the front stream, but for deques, rotations should instead divide the elements equally between the front and rear streams.

(a) Use these ideas to implement deques, in the style of Figure 4. Be careful to make rotations as lazy as possible. You may need to extend the streams library with a few extra operations.
(b) Use a debit argument to prove that your deques require only $O(1)$ amortized time per operation. ◇

Exercise 7 (Worst-case Queues [23]). To get queues that run in $O(1)$ worst-case time instead of $O(1)$ amortized time, we systematically schedule the execution of each suspension. We arrange that each suspension takes only $O(1)$ time to execute, and force one suspension per *snoc* or *tail* operation.

(a) Write a *rotation* function that is entirely incremental by doing one step of the reverse for every step of the append. You may need to extend the streams library with a few extra operations.
(b) Add a new field of type α *Stream* to each queue that points to the first stream node that has not yet been forced. Rewrite the pseudo-constructor *queue* to force the node and advance the pointer to the next node.
(c) Show that the new stream field always has length $|f| - |r|$. Take advantage of this fact to eliminate the two length fields from the representation. ◇

2.3 Eliminating Unnecessary Overheads

The above implementation of queues is asymptotically optimal — you can't ask for better bounds than $O(1)$ time per operation. However, in practice, it tends to be fairly slow. There are at least two reasons for this. First, lazy evaluation is slower than strict evaluation, because of the need to create and memoize suspensions. Compilers for lazy languages recognize this fact and use strictness analysis to turn lazy evaluation into strict evaluation whenever possible. However, when lazy evaluation serves an algorithmic purpose, as it does here, it will never be eliminated by strictness analysis. But even if we need lazy evaluation, maybe we don't need so much of it.

Second, this implementation uses appends in a rather inefficient way. The append operation takes time proportional to the size of the left list (or stream). Hence, for maximum efficiency, it should always be called in right-associative contexts. For example, on lists, executing $(xs \mathbin{@} (ys \mathbin{@} zs))$ takes time proportional to $|xs| + |ys|$, whereas executing $((xs \mathbin{@} ys) \mathbin{@} zs)$ takes time proportional to $2|xs| + |ys|$. If we take a snapshot of the front stream at any given moment, it always has the form

$$(\cdots((f \mathbin{+\!\!+} \mathit{reverse}\ r_1) \mathbin{+\!\!+} \mathit{reverse}\ r_2) \mathbin{+\!\!+} \cdots) \mathbin{+\!\!+} \mathit{reverse}\ r_k$$

In general, using appends in left-associative contexts like this results in potentially quadratic behavior. Fortunately, in this case, the latter streams are much longer than the earlier streams, so the total number of append steps is still linear. But even if this use of append does not threaten the asymptotic bounds of our implementation, it does significantly increase the constant factor.

Exercise 8. This exercise is to determine the overhead associated with the inefficient use of appends.

(a) Calculate the number of append steps executed while building and then consuming a queue of size n (i.e., n calls to *snoc* followed by n calls to *tail*).

(b) Repeat this calculation assuming that rotations are performed, not when $|r| = |f| + 1$, but when $|r| = c|f| + 1$, for some constant c. ◇

To make our implementation faster in practice, we will address both of these issues. First, we will replace streams with a combination of ordinary lists and a short stream of suspended lists. This drastically reduces the number of suspensions from one per element to two per suspended list. This would not help if we still touched a suspension every operation, but we will also arrange to touch suspensions only occasionally. Second, we will use appends only on short streams.

Recall that the front stream of a queue has the form

$$(\cdots((f \mathbin{+\!\!+} \mathit{reverse}\ r_1) \mathbin{+\!\!+} \mathit{reverse}\ r_2) \mathbin{+\!\!+} \cdots) \mathbin{+\!\!+} \mathit{reverse}\ r_k$$

Writing the rear stream as r, we can decompose the queue into three parts: f, r, and the collection $m = \{\mathit{reverse}\ r_1, \ldots, \mathit{reverse}\ r_k\}$. Previously, f, r, and each $\mathit{reverse}\ r_i$ was a stream, but now we can represent f and r as ordinary lists and each $\mathit{reverse}\ r_i$ as a suspended list. This eliminates the vast majority of suspensions, and avoids almost all of the overheads associated with lazy evaluation. But how should we represent the collection m? As we will see, this collection is accessed in FIFO order, so it is tempting to represent it as a queue. However, this collection will always be small, so it will be simpler—and just as efficient—to represent it as a stream. In doing so, we reintroduce some lazy evaluation, but the overheads of this will be negligible. The new representation is thus

```
datatype α Queue =
        Queue of α list × α list susp S.Stream × int × α list × int
```

The second integer tracks the length of r, and the first integer tracks the combined lengths of f and all the suspended lists in m.

The old balance invariant served two purposes. It kept r from getting too long and it kept f from becoming empty. In this new representation we deal with these two issues separately. First, we guarantee that f is never empty unless the entire queue is empty. If f is empty and m is non-empty, then we remove the first suspended list from m, force it, and install the result as the new f. Second, we guarantee that $|r| \leq |f| + \sum_{s \in m} |s|$. When r becomes too long, we add $\$rev\ r$ to the end of m. These two invariants are enforced by the pseudo-constructor *queue*, which in turn calls a second pseudo-constructor *queue'*.

```
fun queue ([], m, lenfm, r, lenr) =
        if S.isEmpty m then Queue (r, S.empty, lenr, [], 0)   (* |r| ≤ 1 *)
        else queue' (force (S.head m), S.tail m, lenfm, r, lenr)
   | queue q = queue' q
and queue' (q as (f, m, lenfm, r, lenr)) =
        if lenr ≤ lenfm then Queue q
        else Queue (f, msnoc (m, $rev r), lenfm+lenr, [], 0)
```

In the second line of *queue*, we install r directly as the new f without reversing it. We are justified in doing this because we know that r contains at most a single element, so $rev\ r = r$.

The *msnoc* operation called by *queue'* is simply *snoc* on streams, defined by

```
fun msnoc (m, s) = S.++ (m, S.cons (s, S.empty))
```

Although in general it is quite slow to implement *snoc* in terms of append, in this case, we know that m is always short, so the inefficiency is tolerable.

Now, we can define the major operations on queues as follows:

```
fun snoc (Queue (f, m, lenfm, r, lenr), x) = queue (f, m, lenfm, x :: r, lenr+1)
fun head (Queue ([], _, _, _, _)) = raise EMPTY
   | head (Queue (x :: f, m, lenfm, r, lenr)) = x
fun tail (Queue ([], _, _, _, _)) = raise EMPTY
   | tail (Queue (x :: f, m, lenfm, r, lenr)) = queue (f, m, lenfm−1, r, lenr)
```

The complete code for this implementation is shown in Figure 5.

Exercise 9. Prove that for any queue of size $n \geq 1$, $|m| \leq \lfloor \log_2 n \rfloor$. ◇

Exercise 10. Repeat Exercise 8 for this new implementation. ◇

```
structure Queue2 : QUEUE =
struct
  datatype α Queue =
        Queue of α list × α list susp S.Stream × int × α list × int

  exception EMPTY

  val empty = Queue ([], S.empty, 0, [], 0)
  fun isEmpty (Queue (f, m, lenfm, r, lenr)) = null f

  fun msnoc (m, s) = S.++ (m, S.cons (s, S.empty))
  fun queue ([], m, lenfm, r, lenr) =
        if S.isEmpty m then Queue (r, S.empty, lenr, [], 0)   (* |r| ≤ 1 *)
        else queue' (force (S.head m), S.tail m, lenfm, r, lenr)
    | queue q = queue' q
  and queue' (q as (f, m, lenfm, r, lenr)) =
        if lenr ≤ lenfm then Queue q
        else Queue (f, msnoc (m, $rev r), lenfm+lenr, [], 0)

  fun snoc (Queue (f, m, lenfm, r, lenr), x) = queue (f, m, lenfm, x :: r, lenr+1)
  fun head (Queue ([], _, _, _, _)) = raise EMPTY
    | head (Queue (x :: f, m, lenfm, r, lenr)) = x
  fun tail (Queue ([], _, _, _, _)) = raise EMPTY
    | tail (Queue (x :: f, m, lenfm, r, lenr)) = queue (f, m, lenfm−1, r, lenr)
end
```

Fig. 5. A faster implementation of queues.

Exercise 11. Use a debit argument to show that this implementation takes only $O(1)$ amortized time per operation. (Hint: Keep track of the debits on stream nodes and the debits on suspended lists separately. Allow at most one debit per stream node, and require that all suspended lists except the last be fully paid off.) ◇

2.4 Bibliographic Notes

Hood and Melville [11] and Gries [9, pages 250–251] first proposed the implementation of queues in Figure 3. Burton [3] proposed a similar implementation, but without the restriction that the first list be non-empty whenever the queue is non-empty. (Burton combines *head* and *tail* into a single operation, so he does not need this restriction to support *head* efficiently.) Hoogerwoord [12] later proposed a similar implementation of deques.

Hood and Melville [11] also gave a rather complicated implementation of queues supporting all operations in $O(1)$ worst-case time. Hood [10] and Chuang and Goldberg [5] later extended this implementation to handle the double-ended case. Okasaki [23] showed how to use lazy evaluation to simplify these implementations, while still retaining the worst-case bounds (see Exercise 7). The implementation in Figure 4 is a simplification of this approach, first appearing in [24]

Kaplan and Tarjan [16] proposed yet another implementation of constant-time functional deques, based on an entirely different technique known as *recursive slowdown*.

3 Catenable Lists

Appending lists is an extremely common operation, at least conceptually. Unfortunately, appending lists can be slow. For instance, we discussed in the previous section how left-associative appends can result in quadratic behavior. To prevent this, programmers often transform their programs to remove as many appends as possible (for instance, by using accumulating parameters). But what if it were possible to design a list data structure that supported append (also known as catenation or concatenation) in constant time, without sacrificing the existing constant-time operations? Such a data structure is called a *catenable list*. In particular, we want a implementation of catenable lists that supports every operation in Figure 6 in constant time. Note that *cons* has been eliminated in favor of a *unit* operation that creates a singleton list, because *cons* (x, xs) can be simulated by *unit* x ++ xs. Similarly, *snoc* (xs, x) can be simulated by xs ++ *unit* x. Since this data structure supports insertions at either end, but removals only from the front, it might more accurately be called a catenable output-restricted deque.

```
signature CATENABLE =
sig
   type α Cat                                    (* catenable lists *)

   exception EMPTY

   val empty    : α Cat
   val isEmpty  : α Cat → bool

   val unit     : α → α Cat                      (* create a singleton list *)
   val ++       : α Cat × α Cat → α Cat          (* infix append *)

   val head     : α Cat → α                      (* raises EMPTY if list is empty *)
   val tail     : α Cat → α Cat                  (* raises EMPTY if list is empty *)
end
```

Fig. 6. Signature for catenable lists.

We first consider a simple solution that makes append fast, at the cost of slowing down *head* and *tail*. We then transform the data structure to recover efficient *head* and *tail* operations.

3.1 A Partial Solution

The simplest way to make append fast is to make it a constructor, as in the following datatype:

datatype α Cat = Empty | Unit **of** α | App **of** α Cat $\times$ α Cat

This is the type of *binary leaf trees*, with elements stored at the leaves from left to right.

Now, *unit* and $+\!\!+$ simply call the appropriate constructors. The *head* operation might be written naively as

```
fun head Empty = raise EMPTY
  | head (Unit $x$) = $x$
  | head (App ($s$, $t$)) = head $s$
```

However, a moment's reflection reveals that the third clause of *head* is incorrect. For example, *head* (*App* (*Empty*, *Unit* x)) raises an exception instead of returning x. One way to fix this problem is to handle the exception.

```
  | head (App ($s$, $t$)) = head $s$ handle EMPTY $\Rightarrow$ head $t$
```

A better solution is to insist that all trees be well-formed by disallowing occurrences of *Empty* beneath *App* nodes. We enforce this by having $+\!\!+$ check for *Empty* ($+\!\!+$ can be viewed as a pseudo-constructor for *App*).

```
fun $Empty +\!\!+ s = s$
  | $s +\!\!+ Empty = s$
  | $s +\!\!+ t$ = App ($s$, $t$)
```

Even though the naive version of *head* is now correct, it is inefficient. In particular it takes time proportional to the length of the *left spine* (i.e., the path from the root to the leftmost leaf), which could be as much as $O(n)$. For now, we accept this inefficiency.

For *tail*, there are four cases. We have little choice in three of these cases.

```
fun tail Empty = raise EMPTY
  | tail (Unit $x$) = Empty
  | tail (App (Unit $x$, $s$)) = $s$
```

However, for the fourth case, *tail* (*App* (*App* (s, t), u)), we have at least two choices:

```
  | tail (App (App ($s$, $t$), $u$)) = App (tail (App ($s$, $t$)), $u$)
```

or

```
  | tail (App (App ($s$, $t$), $u$)) = tail (App ($s$, App ($t$, $u$)))
```

The first choice deletes the leftmost leaf while leaving the rest of the tree intact. The second choice dynamically applies the associative rule along the left spine, converting a tree with a long left spine into a tree with a long right spine. Both versions take the same amount of time, but the second choice drastically reduces the cost of subsequent operations. In fact, successive *tail* operations using the first choice will frequently take quadratic time, but successive *tail* operations using the second choice will take only linear time (see Exercise 12). Hence, we will use the second version. The complete code for this implementation appears in Figure 7.

```
structure Cat0 : CATENABLE =
struct
  datatype α Cat = Empty | Unit of α | App of α Cat × α Cat

  exception EMPTY

  val empty = Empty
  fun isEmpty Empty = true | isEmpty _ = false

  fun unit x = Unit x
  fun Empty ++ s = s
    | s ++ Empty = s
    | s ++ t = App (s, t)

  fun head Empty = raise EMPTY
    | head (Unit x) = x
    | head (App (s, t)) = head s
  fun tail Empty = raise EMPTY
    | tail (Unit x) = Empty
    | tail (App (Unit x, s)) = s
    | tail (App (App (s, t), u)) = tail (App (s, App (t, u)))
end
```

Fig. 7. Catenable lists as binary leaf trees.

Exercise 12. Prove that executing n successive *tail* operations on any catenable list of size n takes only $O(n)$ time. (Hint: Consider the number of bad nodes, where a bad node is any left child or any descendent of a left child.) ◇

Exercise 13. Rather than relying on the programmer to refrain from building *App* nodes containing *Empty*, we could build this requirement into the type.

```
datatype α Tree = Unit of α | App of α Tree × α Tree
datatype α Cat = Empty | NonEmpty of α Tree
```

(a) Modify the implementation in Figure 7 to support this new type.
(b) Discuss the advantages and disadvantages of each approach. ◇

3.2 Persistent Catenable Lists

Even if the above data structure is efficient in some circumstances, it is clearly not efficient when used persistently. For example, consider building a tree with a long left spine and then repeatedly taking the *head* or *tail* of that tree. We next combine lazy evaluation with a clever representation of left spines to obtain an implementation that requires only $O(1)$ amortized time per operation.

Consider how the left spine is treated by the various list operations. The append operation adds a new node at the top of the left spine, and the *head* and *tail* operations access the leaf at the bottom of the left spine. But these are just the operations supported by queues! The *left spine view* of a binary leaf tree is thus a tree in which every left spine from the binary leaf tree is represented recursively as an element (the leaf at the bottom of the spine) together with a queue of left spine views corresponding to the right children of nodes in the spine. Figure 8 shows a binary leaf tree and its corresponding left spine view.

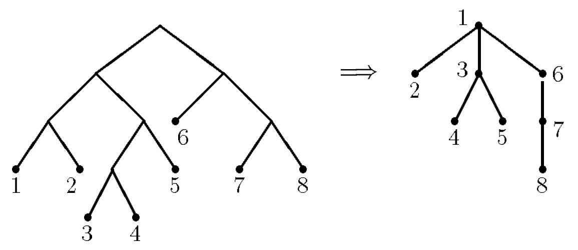

Fig. 8. A binary leaf tree and its left spine view.

If q is a structure implementing queues, then the datatype of left spine views can be written

```
datatype α Cat = Empty | Cat of α × α Cat Q.Queue
```

This type can independently be viewed as a type of multiway tree with elements stored at every node and ordered in left-to-right preorder. Again, we insist that all trees be well-formed (i.e., that *Empty* never appears in the child queue of a *Cat* node).

Now, simply translating the operations *unit*, ++, and *head* from binary leaf trees to left spine views, we get

```
fun unit x = Cat (x, Q.empty)
fun Empty ++ s = s
  | s ++ Empty = s
  | (Cat (x, q)) ++ s = Cat (x, Q.snoc (q, s))
fun head Empty = raise EMPTY
  | head (Cat (x, q)) = x
```

The third clause of ++ *links* the two trees by making the second tree the last child of the first tree.

The translation of the *tail* function is a little trickier. Given a tree *Cat* (x, q), where q is non-empty, it discards x and links the elements of q from right to left.

```
fun tail Empty = raise EMPTY
  | tail (Cat (x, q)) = if Q.isEmpty q then Empty else linkAll q
```

where *linkAll* is defined as

```
fun linkAll q = if Q.size q = 1 then Q.head q
                else Q.head q ++ linkAll (Q.tail q)
```

This code assumes that queues support a constant-time *size* operator, but it could easily be rewritten to use *Q.isEmpty.* This code can also be viewed as an instance of the standard *foldr1* schema. If q provides a *foldr1* function, then *linkAll* can be rewritten

```
fun linkAll q = Q.foldr1 (op ++) q
```

Figure 9 illustrates the overall effect of *tail* operation.

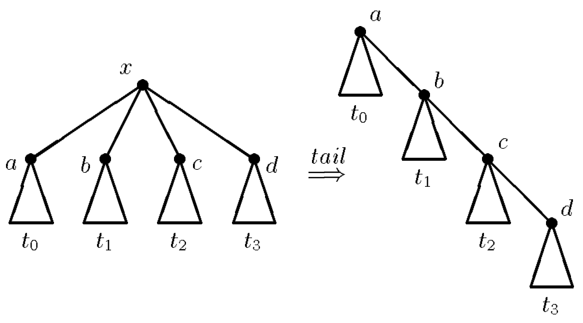

Fig. 9. Illustration of the *tail* operation on left spine views.

The *unit* and *head* operations clearly take $O(1)$ worst-case time. If we use one of the constant-amortized-time queues from the previous section, then ++ takes $O(1)$ amortized time. However, *tail* takes $O(|q|)$ time, which could be as large as $O(n)$. To reduce this to $O(1)$ amortized time, we need to make one further change — we execute *linkAll* lazily. We alter the datatype so that each tree in a child queue is suspended.

```
datatype α Cat = Empty | Cat of α × α Cat susp Q.Queue
```

Then, *linkAll* is written

```
fun linkAll q = if Q.size q = 1 then force (Q.head q)
                else link (force (Q.head q), $linkAll (Q.tail q))
```

where *link* is like ++ but expects its second argument to be suspended.

```
fun link (Cat (x, q), d) = Cat (x, Q.snoc (q, d))
```

To make the types work out, ++ must be rewritten to suspend its second argument.

```
fun Empty ++ s = s
  | s ++ Empty = s
  | s ++ t = link (s, $t)
```

The complete code for this implementation appears in Figure 10. It is written as a functor that is parameterized over the particular implementation of queues.

```
functor Cat1 (structure Q : QUEUE) : CATENABLE =
struct
  datatype α Cat = Empty | Cat of α × α Cat susp Q.Queue

  exception EMPTY

  val empty = Empty
  fun isEmpty Empty = true | isEmpty _ = false

  fun link (Cat (x, q), d) = Cat (x, Q.snoc (q, d))
  fun linkAll q = if Q.size q = 1 then force (Q.head q)
                  else link (force (Q.head q), $linkAll (Q.tail q))

  fun unit x = Cat (x, Q.empty)
  fun Empty ++ s = s
    | s ++ Empty = s
    | s ++ t = link (s, $t)

  fun head Empty = raise EMPTY
    | head (Cat (x, q)) = x
  fun tail Empty = raise EMPTY
    | tail (Cat (x, q)) = if Q.isEmpty q then Empty else linkAll q
end
```

Fig. 10. Catenable lists using lazy left spine views.

Exercise 14. Implement *cons* and *snoc* directly for this implementation instead of in terms of *unit* and ++. ◇

We now prove that ++ and *tail* take only $O(1)$ amortized time using a debit argument. Each performs only $O(1)$ work aside from forcing suspensions, so we must show that discharging $O(1)$ debits per ++ and *tail* suffices to discharge all debits before their associated suspensions are forced.

Proof. Let $d_t(i)$ be the number of debits on the ith node of tree t and let $D_t(i) = \sum_{j=0}^{i} d_t(i)$ be the cumulative number of debits on all nodes up to and including the ith node of t. Finally, let D_t be the total number debits on all nodes in t (i.e., $D_t = D_t(|t| - 1)$). We maintain two invariants on debits.

First, we require that the number of debits on any node be bounded by the degree of the node (i.e., $d_t(i) \leq \mathit{degree}_t(i)$). Since the sum of degrees of all nodes in a non-empty tree is one less than the size of the tree, this implies that the total number of debits in a tree is bounded by the size of the tree (i.e., $D_t < |t|$). We will maintain this invariant by incrementing the number of debits on a node only when we also increment its degree.

Second, we insist that the $D_t(i)$ be bounded by some linear function on i. The particular linear function we choose is

$$D_t(i) \leq i + \mathit{depth}_t(i)$$

where $\mathit{depth}_t(i)$ is the length of the path in t from the root to node i. This invariant is called the *left-linear debit invariant.* Note that the left-linear debit invariant guarantees that $d_t(0) = D_t(0) \leq 0 + 0 = 0$, so all debits on a node must be discharged by the time it reaches the root. (Recall that the root is not even suspended!) The only time we actually force a suspension is when the suspended node is to become the new root.

We first show that $+\!\!+$ maintains both invariants by discharging only a single debit. The only debit created by append is for the trivial suspension of its second argument. Since we are not increasing the degree of this node, we immediately discharge the new debit. Now, assume that t_1 and t_2 are non-empty and let $t = t_1 +\!\!+ t_2$. Let $n = |t_1|$. Note that the index, depth, and cumulative debits of each node in t_1 are unaffected by the append, so for $i < n$

$$\begin{aligned} D_t(i) &= D_{t_1}(i) \\ &\leq i + \mathit{depth}_{t_1}(i) \\ &= i + \mathit{depth}_t(i) \end{aligned}$$

The nodes in t_2 increase in index by n, increase in depth by one, and accumulate the total debits of t_1, so for $i \geq n$

$$\begin{aligned} D_t(n+i) &= D_{t_1} + D_{t_2}(i) \\ &< n + D_{t_2}(i) \\ &\leq n + i + \mathit{depth}_{t_2}(i) \\ &= n + i + \mathit{depth}_t(n+i) - 1 \\ &< (n+i) + \mathit{depth}_t(n+i) \end{aligned}$$

Thus, we do not need to discharge any further debits to maintain the left-linear debit invariant.

Finally, we show that *tail* maintains both invariants by discharging three debits. Let $t' = \mathit{tail}\, t$. After discarding the root of t, we link the children $t_0 \ldots t_{m-1}$ from right to left. Let t'_j be the partial result of linking $t_j \ldots t_{m-1}$. Then $t' = t'_0$. Since every link except the outermost is suspended, we assign a single debit to the root of each t_j, $0 < j < m - 1$. Note that the degree of each of these nodes increases by one. We also assign a debit to the root of t'_{m-1} because the last call to *linkAll* is suspended even though it does not call *link*. Since the degree of this node does not change, we immediately discharge this final debit.

Now, suppose the ith node of t appears in t_j. We know that $D_t(i) < i + depth_t(i)$, but consider how each of these quantities changes with the *tail*. i decreases by one because the first element is discarded. The depth of each node in t_j increases by $j-1$ (see Figure 9) while the cumulative debits of each node in t_j increases by j. Thus,

$$\begin{aligned} D'_t(i-1) &= D_t(i) + j \\ &\leq i + depth_t(i) + j \\ &= i + (depth'_t(i-1) - (j-1)) + j \\ &= (i-1) + depth'_t(i-1) + 2 \end{aligned}$$

Discharging the first two debits restores the invariant, for a total of three debits. □

Exercise 15. This exercise explores conversion functions from lists to catenable lists.

(a) Write a function $makeCat : \alpha\ list \rightarrow \alpha\ Cat$ that runs in $O(1)$ amortized time.

(b) Write a function $flatten : \alpha\ list\ list \rightarrow \alpha\ Cat$ that runs in $O(1+E)$ amortized time, where E is the number of empty lists in the original list. ◇

3.3 Bibliographic Notes

Hughes [13, 14] has investigated several implementations of catenable lists, including ones similar to Figure 7. However, he does not support efficient *head* and *tail* operations; rather, he supplies a single operation for converting a catenable list to an ordinary list. This is comparable to requiring that all of the appends precede all the heads and tails.

Kaplan and Tarjan [16] gave the first functional implementation of catenable lists to support all operations in $O(1)$ worst-case time, based on the technique of *recursive slowdown*. Shortly thereafter, Okasaki [22] developed the implementation in Figure 10. This implementation is much simpler than Kaplan and Tarjan's approach, but supports all operations in $O(1)$ amortized time, rather than worst-case time.

4 Heaps

For queues and catenable lists, the usual imperative implementations do not translate well to a functional setting. For heaps (priority queues), however, many standard imperative solutions translate quite nicely. In Section 4.1, we consider one such example — leftist heaps [6, 18]. In Section 4.2, though, we consider a second example — pairing heaps [8] — that is more problematical.

Figure 11 gives a minimal signature for mergeable heaps. Note that these heaps are not polymorphic; rather, the type of elements is fixed, as is the ordering relation on those elements. Not all heap data structures support an efficient *merge* operation, but we will consider only those that do. Heap data structures typically also support an *insert* operation, but, as we did for catenable lists, we have eliminated this operation in favor of a *unit* operation since *insert* (x, h) can be simulated by *merge* (*unit* x, h).

```
signature ORDERED =
sig
   type T                    (* type of ordered elements *)
   val leq : T × T → bool    (* total ordering relation *)
end

signature HEAP =
sig
   structure Elem : ORDERED

   type Heap

   exception EMPTY

   val empty     : Heap
   val isEmpty   : Heap → bool

   val unit      : Elem.T → Heap         (* create a singleton heap *)
   val merge     : Heap × Heap → Heap

   val findMin   : Heap → Elem.T         (* raises EMPTY if heap is empty *)
   val deleteMin : Heap → Heap           (* raises EMPTY if heap is empty *)
end
```

Fig. 11. Signature for heaps (priority queues).

4.1 Leftist Heaps

A *heap-ordered* tree is one in which the root of each subtree contains the minimum element in that subtree. Thus, the root of a heap-ordered tree is always the overall minimum element in the tree.

Define the *r-height* of a binary tree to be the length of its right spine (i.e., the length of the rightmost path to an empty node). Leftist heaps are heap-ordered binary trees that satisfy the *leftist property*: the r-height of any left child is $\geq$ the r-height of its right sibling. Note that the right spine of a leftist heap is always the shortest path from the root to an empty node.

Exercise 16. Prove that the r-height of a leftist heap of size $n > 0$ is $\leq \log_2 n+1$. ◇

Leftist heaps are represented by the following type:

```
datatype Heap = Empty | Node of int × Elem.T × Heap × Heap
```

where the integer records the length of the right spine.

The *unit*, *findMin*, and *deleteMin* operations on leftist heaps are trivial.

```
fun unit x = Node (1, x, Empty, Empty)
fun findMin Empty = raise EMPTY
    | findMin (Node (r, x, a, b)) = x
fun deleteMin Empty = raise EMPTY
    | deleteMin (Node (r, x, a, b)) = merge (a, b)
```

To merge two leftist heaps, you merge their right spines in the same way that you would merge two ordered lists. Then you swap children of nodes along the merge path as necessary to restore the leftist property. This swapping is performed by the pseudo-constructor *node*.

```
fun node (x, a, Empty) = Node (1, x, a, Empty)
    | node (x, Empty, b) = Node (1, x, b, Empty)
    | node (x, a as Node (ra, _, _, _), b as Node (rb, _, _, _)) =
        if ra ≤ rb then Node (ra+1, x, b, a) else Node (rb+1, x, a, b)
```

Finally, *merge* may be implemented as

```
fun merge (a, Empty) = a
    | merge (Empty, b) = b
    | merge (a as Node (_, x, a1, a2), b as Node (_, y, b1, b2)) =
        if Elem.leq (x, y) then node (x, a1, merge (a2, b))
        else node (y, b1, merge (b2, a))
```

The complete code for leftist heaps appears in Figure 12.

Because the right spine of a leftist heap has logarithmic height, and *merge* traverses the right spines of its two arguments, *merge* takes $O(\log n)$ worst-case time. *deleteMin* calls *merge*, so it also takes $O(\log n)$ worst-case time. *unit* and *findMin* run in $O(1)$ time.

Exercise 17. Implement *insert* directly instead of via *unit* and *merge*. ◇

Exercise 18. Implement a function *makeHeap* : *Elem.T list* → *Heap* that produces a leftist heap from an unordered list of elements in $O(n)$ time. (Note that the naive approach of folding *insert* across the list takes $O(n \log n)$ time.) ◇

Exercise 19 (Weight-biased Leftist Heaps [4]). Weight-biased leftist heaps are an alternative to leftist heaps that replace the leftist property with the *weight-biased leftist property*: the size of any left child is ≥ the size of its right sibling.

(a) Prove that the right spine of a weight-biased leftist heap of size $n > 0$ has length $\leq \log_2 n + 1$.

```
functor Leftist (structure E : ORDERED) : HEAP =
struct
  structure Elem = E

  datatype Heap = Empty | Node of int × Elem.T × Heap × Heap

  exception EMPTY

  val empty = Empty
  fun isEmpty Empty = true | isEmpty _ = false

  fun node (x, a, Empty) = Node (1, x, a, Empty)
    | node (x, Empty, b) = Node (1, x, b, Empty)
    | node (x, a as Node (ra, _, _, _), b as Node (rb, _, _, _)) =
        if ra ≤ rb then Node (ra+1, x, b, a) else Node (rb+1, x, a, b)

  fun unit x = Node (1, x, Empty, Empty)
  fun merge (a, Empty) = a
    | merge (Empty, b) = b
    | merge (a as Node (_, x, a1, a2), b as Node (_, y, b1, b2)) =
        if Elem.leq (x, y) then node (x, a1, merge (a2, b))
        else node (y, b1, merge (b2, a))

  fun findMin Empty = raise EMPTY
    | findMin (Node (r, x, a, b)) = x
  fun deleteMin Empty = raise EMPTY
    | deleteMin (Node (r, x, a, b)) = merge (a, b)
end
```

Fig. 12. Leftist heaps.

(b) Modify the implementation in Figure 12 to obtain weight-biased leftist heaps.

(c) Currently, *merge* operates in two passes: a top-down pass consisting of calls to *merge*, and a bottom-up pass consisting of calls to the pseudo-constructor *node*. Modify *merge* for weight-biased leftist heaps to operate in a single, top-down pass.

(d) What advantages would the top-down version of *merge* have in a lazy environment? In a concurrent environment?

4.2 Pairing Heaps

Pairing heaps are heap-ordered multiway trees, as defined by the following datatype:

```
datatype Heap = Empty | Node of Elem.T × Heap list
```

We insist that all pairing heaps be well-formed (i.e., that *Empty* never occur in a child list of *Node*).

The *unit* and *findMin* operations are trivial.

```
fun unit x = Node (x, [])
fun findMin Empty = raise EMPTY
  | findMin (Node (x, cs)) = x
```

The *merge* operation makes the node with the larger root the leftmost child of the node with the smaller root.

```
fun merge (a, Empty) = a
  | merge (Empty, b) = b
  | merge (a as Node (x, cs), b as Node (y, ds)) =
      if Elem.leq (x, y) then Node (x, b :: cs) else Node (y, a :: ds)
```

Pairing heaps get their name from the *deleteMin* operation. *deleteMin* discards the root and then merges the children in two passes. The first pass merges children in pairs from left to right (i.e., the first child with the second, the third with the fourth, and so on). The second pass merges the remaining trees from right to left. This can be coded concisely as

```
fun mergeAll [] = Empty
  | mergeAll [a] = a
  | mergeAll (a :: b :: rest) = merge (merge (a, b), mergeAll rest)
```

Then, *deleteMin* is simply

```
fun deleteMin Empty = raise EMPTY
  | deleteMin (Node (x, cs)) = mergeAll cs
```

The complete code for this implementation is given in Figure 13.

In practice, pairing heaps are among the fastest of all imperative priority queues [15, 19, 21]. Since pairing heaps are also among the simplest of priority queues, they are the data structure of choice for many applications. Surprisingly, however, tight bounds for pairing heaps are not known. Clearly, *unit*, *findMin*, and *merge* run in $O(1)$ worst-case time, but *deleteMin* might take as much as $O(n)$ worst-case time. It has been conjectured [8] that the amortized running time of *merge* and *deleteMin* are $O(1)$ and $O(\log n)$ respectively, but, even if true of an imperative implementation, this amortized bound is clearly false in the presence of persistence.

Suppose we call *deleteMin* on some pairing heap $h = Node\ (x, cs)$. The result is simply *mergeAll cs*. Now, suppose we merge h with two other pairing heaps a and b, where the root of h is less than the roots of a and b. The result is $h' = Node\ (x, b :: a :: cs)$. If we now call *deleteMin* on h', we will duplicate the work of *mergeAll cs*.

To cope with persistence, we must prevent this duplicated work. We once again turn to lazy evaluation. Instead of a *Heap list*, we instead represent the children of a node as a *Heap susp*. This suspension is equal to *\$mergeAll cs*. Since *mergeAll* operates on pairs of children, we extend the suspension with two children at once. Therefore, each node will contain an extra *Heap* field that will contain any partnerless child. If there are no partnerless children (i.e., if the

```
functor Pairing0 (structure E : ORDERED) : HEAP =
struct
  structure Elem = E

  datatype Heap = Empty | Node of Elem.T × Heap list

  exception EMPTY

  val empty = Empty
  fun isEmpty Empty = true | isEmpty _ = false

  fun unit x = Node (x, [])
  fun merge (a, Empty) = a
    | merge (Empty, b) = b
    | merge (a as Node (x, cs), b as Node (y, ds)) =
        if Elem.leq (x, y) then Node (x, b :: cs) else Node (y, a :: ds)

  fun mergeAll [] = Empty
    | mergeAll [a] = a
    | mergeAll (a :: b :: rest) = merge (merge (a, b), mergeAll rest)

  fun findMin Empty = raise EMPTY
    | findMin (Node (x, cs)) = x
  fun deleteMin Empty = raise EMPTY
    | deleteMin (Node (x, cs)) = mergeAll cs
end
```

Fig. 13. Pairing heaps.

number of children is even), then this extra field will be empty. Since this field is in use only when the number of children is odd, call this the *odd field*. The new datatype is thus

```
datatype Heap = Empty | Node of Elem.T × Heap × Heap susp
```

As usual, the *unit* and *findMin* operations trivial.

```
fun unit x = Node (x, Empty, $Empty)
fun findMin Empty = raise EMPTY
  | findMin (Node (x, a, m)) = x
```

Previously, the *merge* operation was simple and the *deleteMin* operation was complex. Now, the situation is reversed — all the complexity of *mergeAll* has been shifted to *merge*, which must set up the appropriate suspensions. *deleteMin* is simply

```
fun deleteMin Empty = raise EMPTY
  | deleteMin (Node (x, a, m)) = merge (a, force m)
```

We define *merge* in two steps. The first step checks for empty arguments and otherwise compares the two arguments to see which has the smaller root.

```
fun merge (a, Empty) = a
   | merge (Empty, b) = b
   | merge (a as Node (x, _, _), b as Node (y, _, _)) =
      if Elem.leq (x, y) then link (a, b) else link (b, a)
```

The second step, embodied in the *link* helper function, adds a new child to a node. If the odd field is empty, then this child is placed in the odd field.

```
fun link (Node (x, Empty, m), a) = Node (x, a, m)
```

Otherwise, the new child is paired with the child in the odd field, and both are added to the suspension. In other words, we extend the suspension $m =$ \$*mergeAll cs* to \$*mergeAll* $(a :: b :: cs)$. Observe that

$$\begin{aligned}\$\text{mergeAll } (a :: b :: cs) &= \$\text{merge (merge } (a, b), \text{ mergeAll } cs) \\ &= \$\text{merge (merge } (a, b), \text{ force } (\$\text{mergeAll } cs)) \\ &= \$\text{merge (merge } (a, b), \text{ force } m)\end{aligned}$$

so the second clause of *link* may be written

```
fun link (Node (x, b, m), a) =
      Node (x, Empty, $merge (merge (a, b), force m)
```

The complete code for this implementation is given in Figure 14.

Although it now deals gracefully with persistence, this implementation of pairing heaps is relatively slow in practice, because of overheads associated with lazy evaluation. Still, in an entirely lazy language, such as Haskell, where all data structures will pay these overheads regardless of whether they actually gain any benefit from lazy evaluation, this implementation should be competitive.

Exercise 20. Design an experiment to test whether the implementation in Figure 14 really does support *merge* in $O(1)$ amortized time and *deleteMin* in $O(\log n)$ amortized time for non-single-threaded sequences of operations. ◇

4.3 Bibliographic Notes

Leftist heaps were invented by Crane [6], and first presented in their current form by Knuth [18, pages 150–152]. Núñez, Palao, and Peña developed a functional implementation of leftist heaps similar to that in Figure 12.

Fredman, Sedgewick, Sleator, and Tarjan introduced pairing heaps in [8], and conjectured that *insert* and *merge* run in $O(1)$ amortized time, while *deleteMin* runs in $O(\log n)$ amortized time. Stasko and Vitter [26] proved that the bounds on *insert* and *deleteMin* did in fact hold for a variant of pairing heaps, but they did not consider the *merge* operation. Several studies have shown that pairing heaps are among the fastest implementations of priority queues in practice [15, 19, 21].

Many other implementations of priority queues can be adapted very easily to a functional setting. For example, King [17] and Okasaki [24] have described functional implementations of binomial queues. Brodal and Okasaki [2]

```
functor Pairing1 (structure E : ORDERED) : HEAP =
struct
  structure Elem = E

  datatype Heap = Empty | Node of Elem.T × Heap × Heap susp

  exception EMPTY

  val empty = Empty
  fun isEmpty Empty = true | isEmpty _ = false

  fun unit x = Node (x, Empty, $Empty)
  fun merge (a, Empty) = a
    | merge (Empty, b) = b
    | merge (a as Node (x, _, _), b as Node (y, _, _)) =
        if Elem.leq (x, y) then link (a, b) else link (b, a)
  and link (Node (x, Empty, m), a) = Node (x, a, m)
    | link (Node (x, b, m), a) =
        Node (x, Empty, $merge (merge (a, b), force m))

  fun findMin Empty = raise EMPTY
    | findMin (Node (x, a, m)) = x
  fun deleteMin Empty = raise EMPTY
    | deleteMin (Node (x, a, m)) = merge (a, force m)
end
```

Fig. 14. Persistent pairing heaps.

have modified binomial queues to obtain an efficient functional implementation that supports *findMin*, *insert*, and *merge* in $O(1)$ worst-case time, and *deleteMin* in $O(\log n)$ worst-case time.

5 Closing Remarks

We have presented efficient functional implementations of three common abstractions: FIFO queues, catenable lists, and mergeable heaps. Besides being useful in practice, particularly for applications requiring persistence, these implementations illustrate many of the techniques of functional data structure design.

Some imperative data structures can easily be adapted to functional languages, but most cannot. We have seen one example (leftist heaps) of a data structure that is essentially the same in ML as in C, and another (pairing heaps) that can easily be translated into a functional language, but that then needs some non-trivial modifications to handle persistence efficiently. The remaining two data structures (FIFO queues and catenable lists) must be redesigned from scratch — the usual imperative solutions are completely unsuitable for functional implementations.

Functional data structures are not of interest only to functional programmers. Functional languages provide a convenient framework for designing persistent data structures. If desired, these data structures can then be implemented in imperative languages. In fact, for some problems, such as catenable lists [16, 22], the best known persistent solutions were designed in exactly this way.

References

1. Stephen Adams. Efficient sets—a balancing act. *Journal of Functional Programming*, 3(4):553–561, October 1993.
2. Gerth Stølting Brodal and Chris Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6), December 1996. To appear.
3. F. Warren Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14(5):205–206, July 1982.
4. Seonghun Cho and Sartaj Sahni. Weight biased leftist trees and modified skip lists. In *International Computing and Combinatorics Conference*, page ??, June 1996.
5. Tyng-Ruey Chuang and Benjamin Goldberg. Real-time deques, multihead Turing machines, and purely functional programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 289–298, June 1993.
6. Clark Allan Crane. *Linear lists and priority queues as balanced binary trees.* PhD thesis, Computer Science Department, Stanford University, February 1972. Available as STAN-CS-72-259.
7. James R. Driscoll, Neil Sarnak, Daniel D. K. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.
8. Michael L. Fredman, Robert Sedgewick, Daniel D. K. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.
9. David Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1981.
10. Robert Hood. *The Efficient Implementation of Very-High-Level Programming Language Constructs.* PhD thesis, Department of Computer Science, Cornell University, August 1982. (Cornell TR 82-503).
11. Robert Hood and Robert Melville. Real-time queue operations in pure Lisp. *Information Processing Letters*, 13(2):50–53, November 1981.
12. Rob R. Hoogerwoord. A symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513, October 1992.
13. John Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, March 1986.
14. John Hughes. The design of a pretty-printing library. In *First International Spring School on Advanced Functional Programming Techniques*, volume 519 of *LNCS*, pages 53–96. Springer-Verlag, May 1995.
15. Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, April 1986.
16. Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *ACM Symposium on Theory of Computing*, pages 93–102, May 1995.
17. David J. King. Functional binomial queues. In *Glasgow Workshop on Functional Programming*, pages 141–150, September 1994.
18. Donald E. Knuth. *Searching and Sorting*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, 1973.

19. Andrew M. Liao. Three priority queue applications revisited. *Algorithmica*, 7(4):415–427, 1992.
20. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.
21. Bernard M. E. Moret and Henry D. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *Workshop on Algorithms and Data Structures*, volume 519 of *LNCS*, pages 400–411. Springer-Verlag, August 1991.
22. Chris Okasaki. Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. In *IEEE Symposium on Foundations of Computer Science*, pages 646–654, October 1995.
23. Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995.
24. Chris Okasaki. The role of lazy evaluation in amortized data structures. In *ACM SIGPLAN International Conference on Functional Programming*, pages 62–72, May 1996.
25. David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.
26. John T. Stasko and Jeffrey S. Vitter. Pairing heaps: experiments and analysis. *Communications of the ACM*, 30(3):234–249, March 1987.
27. Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

This article was processed using the LaTeX macro package with LLNCS style