Purely Functional Data Structures

Chris Okasaki

Book DOI: http://dx.doi.org/10.1017/CBO9780511530104

Online ISBN: 9780511530104

Hardback ISBN: 9780521631242

Paperback ISBN: 9780521663502

Chapter

8 - Lazy Rebuilding pp. 99-114

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511530104.009

Cambridge University Press

8

Lazy Rebuilding

The remaining four chapters describe general techniques for designing functional data structures. We begin in this chapter with *lazy rebuilding*, a variant of *global rebuilding* [Ove83].

8.1 Batched Rebuilding

Many data structures obey balance invariants that guarantee efficient access. The canonical example is balanced binary search trees, which improve the worst-case running times of many tree operations from the $O(n)$ required by unbalanced trees to $O(\log n)$. One approach to maintaining a balance invariant is to rebalance the structure after every update. For most balanced structures, there is a notion of *perfect balance*, which is a configuration that minimizes the cost of subsequent operations. However, since it is usually too expensive to restore perfect balance after every update, most implementations settle for approximations of perfect balance that are at most a constant factor slower. Examples of this approach include AVL trees [AVL62] and red-black trees [GS78].

However, provided no update disturbs the balance too drastically, an attractive alternative is to postpone rebalancing until after a sequence of updates, and then to rebalance the entire structure, restoring it to perfect balance. We call this approach *batched rebuilding*. Batched rebuilding yields good amortized time bounds provided that (1) the data structure is not rebuilt too often, and (2) individual updates do not excessively degrade the performance of later operations. More precisely, condition (1) states that, if one hopes to achieve a bound of $O(f(n))$ amortized time per operation, and the rebuilding transformation requires $O(g(n))$ time, then the rebuilding transformation cannot be executed any more frequently than every $c \cdot g(n)/f(n)$ operations, for some constant c. For example, consider binary search trees. Rebuilding a tree to per-

fect balance takes $O(n)$ time, so if one wants each operation to take $O(\log n)$ amortized time, then the data structure must not be rebuilt more often than every $c \cdot n/\log n$ operations, for some constant c.

Assume that a data structure is to be rebuilt every $c \cdot g(n)/f(n)$ operations, and that an individual operation on a newly rebuilt data structure takes $O(f(n))$ time (worst-case or amortized). Then, condition (2) states that, after up to $c \cdot g(n)/f(n)$ updates to a newly rebuilt data structure, individual operations must still take only $O(f(n))$ time. In other words, the cost of an individual operation can only degrade by a constant factor. Update functions satisfying condition (2) are called *weak updates*.

For example, consider the following approach to implementing a delete function on binary search trees. Instead of physically removing the specified node from the tree, leave it in the tree but mark it as deleted. Then, whenever half the nodes in the tree have been deleted, make a global pass removing the deleted nodes and restoring the tree to perfect balance. Does this approach satisfy both conditions, assuming we want deletions to take $O(\log n)$ amortized time?

Suppose a tree contains n nodes, up to half of which are marked as deleted. Then removing the deleted nodes and restoring the tree to perfect balance takes $O(n)$ time. We execute the transformation only every $\frac{1}{2}n$ delete operations, so condition (1) is satisfied. In fact, condition (1) would allow us to rebuild the data structure even more often, as often as every $c \cdot n/\log n$ operations. The naive delete algorithm finds the desired node and marks it as deleted. This takes $O(\log n)$ time, even if up to half the nodes have been marked as deleted, so condition (2) is satisfied. Note that, even if half the nodes in the tree are marked as deleted, the average depth per active node is only about one greater than it would be if the deleted nodes had been physically removed. The extra depth degrades each operation by only a constant additive factor, whereas condition (2) allows for each operation to be degraded by a constant multiplicative factor. Hence, condition (2) would allow us to rebuild the data structure even less often.

In the above discussion, we described only deletions, but of course binary search trees typically support insertions as well. Unfortunately, insertions are not weak because they can create a deep path very quickly. However, a hybrid approach is possible, in which insertions are handled by local rebalancing after every update, as in AVL trees or red-black trees, but deletions are handled via batched rebuilding.

Exercise 8.1 Extend the red-black trees of Section 3.3 with a delete function using these ideas. Add a boolean field to the T constructor and maintain es-

timates of the numbers of valid and invalid elements in the tree. Assume for the purposes of these estimates that every insertion adds a new valid element and that every deletion invalidates a previously valid element. Correct the estimates during rebuilding. You will find Exercise 3.9 helpful in rebuilding the tree. ◇

As a second example of batched rebuilding, consider the batched queues of Section 5.2. The rebuilding transformation reverses the rear list into the front list, restoring the queue to a state of perfect balance in which every element is contained in the front list. As we have already seen, batched queues have good amortized efficiency, but only when used ephemerally. Under persistent usage, the amortized bounds degrade to the cost of the rebuilding transformation because it is possible to trigger the transformation arbitrarily often. In fact, this is true for all data structures based on batched rebuilding.

8.2 Global Rebuilding

Overmars [Ove83] describes a technique for eliminating the amortization from batched rebuilding. He calls this technique *global rebuilding*. The basic idea is to execute the rebuilding transformation incrementally, performing a few steps per normal operation. This can be usefully viewed as running the rebuilding transformation as a coroutine. The tricky part of global rebuilding is that the coroutine must be started early enough that it can finish by the time the rebuilt structure is needed.

Concretely, global rebuilding is accomplished by maintaining two copies of each object. The primary, or *working*, copy is the ordinary structure. The secondary copy is the one that is being gradually rebuilt. All queries and updates operate on the working copy. When the secondary copy is completed, it becomes the new working copy and the old working copy is discarded. A new secondary copy might be started immediately, or the object may carry on for a while without a secondary structure, before eventually starting the next rebuilding phase.

There is a further complication to handle updates that occur while the secondary copy is being rebuilt. The working copy will be updated in the normal fashion, but the secondary copy must be updated as well or the effect of the update will be lost when the secondary copy takes over. However, the secondary copy will not in general be represented in a form that can be efficiently updated. Thus, these updates to the secondary copy are buffered and executed, a few at a time, after the secondary copy has been rebuilt, but before it takes over as the working copy.

Global rebuilding can be implemented purely functionally, and has been several times. For example, the real-time queues of Hood and Melville [HM81] are based on this technique. Unlike batched rebuilding, global rebuilding has no problems with persistence. Since no one operation is particularly expensive, arbitrarily repeating operations has no effect on the time bounds. Unfortunately, global rebuilding is often quite complicated. In particular, representing the secondary copy, which amounts to capturing the intermediate state of a coroutine, can be quite messy.

8.2.1 Example: Hood–Melville Real-Time Queues

Hood and Melville's implementation of real-time queues [HM81] is similar in many ways to the real-time queues of Section 7.2. Both implementations maintain two lists representing the front and rear of the queue, respectively, and incrementally rotate elements from the rear list to the front list beginning when the rear list becomes one longer than the front list. The differences lie in the details of this incremental rotation.

First, consider how we might reverse a list in an incremental fashion by keeping two lists and gradually transferring elements from one to the other.

```
datatype α ReverseState = WORKING of α list × α list | DONE of α list

fun startReverse xs = WORKING (xs, [])

fun exec (WORKING (x :: xs, xs')) = WORKING (xs, x :: xs')
  | exec (WORKING ([], xs')) = DONE xs'
```

To reverse a list *xs*, we first create a new state WORKING (*xs*, [ ]) and then repeatedly call exec until it returns DONE with the reversed list. Altogether, this takes $n + 1$ calls to exec, where n is the initial length of *xs*.

We can incrementally append two lists *xs* and *ys* by applying this trick twice. First we reverse *xs* to get *xs'*, then we reverse *xs'* onto *ys*.

```
datatype α AppendState =
    REVERSING of α list × α list × α list
  | APPENDING of α list × α list
  | DONE of α list

fun startAppend (xs, ys) = REVERSING (xs, [], ys)

fun exec (REVERSING (x :: xs, xs', ys)) = REVERSING (xs, x :: xs', ys)
  | exec (REVERSING ([], xs', ys)) = APPENDING (xs', ys)
  | exec (APPENDING (x :: xs', ys)) = APPENDING (xs', x :: ys)
  | exec (APPENDING ([], ys)) = DONE ys
```

Altogether, this takes $2m + 2$ calls to exec, where m is the initial length of *xs*.

Now, to append *f* onto reverse *r* in this fashion, we perform a total of three

reversals. First, we reverse f and r in parallel to get f' and r' and then we reverse f' onto r'. The following code assumes that r is initially one longer than f.

```
datatype α RotationState =
      REVERSING of α list × α list × α list × α list
    | APPENDING of α list × α list
    | DONE of α list

fun startRotation (f, r) = REVERSING (f, [], r, [])

fun exec (REVERSING (x :: f, f', y :: r, r')) = REVERSING (f, x :: f', r, y :: r')
  | exec (REVERSING ([], f', [y], r')) = APPENDING (f', y :: r')
  | exec (APPENDING (x :: f', r')) = APPENDING (f', x :: r')
  | exec (APPENDING ([], r')) = DONE r'
```

Again, this finishes after a total of $2m + 2$ calls to exec, where m is the initial length of f.

Unfortunately, there is a major problem with this method of performing rotations. If we only call exec a few times per call to snoc or tail, then by the time the rotation finishes, the answer may no longer be the one we want! In particular, if tail has been called k times during the rotation, then the first k elements of the resulting list are invalid. There are two basic ways we can fix this problem. One is to keep a count of the number of invalid elements and extend RotationState with a third phase, Deleting, that deletes elements from the list a few at a time until there are no more invalid elements. This is the approach that corresponds most closely with the definition of global rebuilding. However, a better approach in this case is to avoid placing the invalid elements on the answer list to begin with. We keep track of the number of valid elements in f', and quit copying elements from f' to r' when this number reaches zero. Every call to tail during the rotation decrements the number of valid elements.

```
datatype α RotationState =
      REVERSING of int × α list × α list × α list × α list
    | APPENDING of int × α list × α list
    | DONE of α list

fun startRotation (f, r) = REVERSING (0, f, [], r, [])

fun exec (REVERSING (ok, x :: f, f', y :: r, r')) =
      REVERSING (ok+1, f, x :: f', r, y :: r')
  | exec (REVERSING (ok, [], f', [y], r')) = APPENDING (ok, f', y :: r')
  | exec (APPENDING (0, f', r')) = DONE r'
  | exec (APPENDING (ok, x :: f', r')) = APPENDING (ok−1, f', x :: r')

fun invalidate (REVERSING (ok, f, f', r, r')) = REVERSING (ok−1, f, f', r, r')
  | invalidate (APPENDING (0, f', x :: r')) = DONE r'
  | invalidate (APPENDING (ok, f', r')) = APPENDING (ok−1, f', r')
```

This process finishes after a total of $2m + 2$ calls to exec and invalidate, where m is the initial length of *f*.

There are three more tricky details to consider. The first is that, during a rotation, the first few elements of the queue lie at the back of the f' field within the rotation state. How then are we to answer a head query? The solution to this dilemma is to keep a working copy of the old front list. We just have to make sure that the new copy of the front list is ready by the time the working copy is exhausted. During a rotation, the *lenf* field measures the length of the list that is under construction, rather than of the working copy *f*. In between rotations, the *lenf* field contains the length of *f*.

The second detail is exactly how many calls to exec we must issue per snoc and tail to guarantee that the rotation completes before either the next rotation is ready to begin or the working copy of the front list is exhausted. Assume that *f* has length m and *r* has length $m + 1$ at the beginning of a rotation. Then, the next rotation will begin after any combination of $2m+2$ insertions or deletions, but the working copy of the front list will be exhausted after just m deletions. Altogether, the rotation requires at most $2m + 2$ steps to complete. If we call exec twice per operation, including the operation that begins the rotation, then the rotation will complete at most m operations after it begins.

The third detail is that, since each rotation finishes long before the next rotation begins, we add an IDLE state to RotationState, such that exec IDLE = IDLE. Then we can blindly call exec without worrying about whether we are in the middle of a rotation or not.

The remaining details are by now routine and the complete implementation is shown in Figure 8.1.

Exercise 8.2 Prove that calling exec twice at the beginning of each rotation, and once for every remaining insertion or deletion is enough to finish the rotation on time. Modify the code accordingly.

Exercise 8.3 Replace the *lenf* and *lenr* fields with a single diff field that maintains the difference between the lengths of *f* and *r*. *diff* may be inaccurate during rebuilding, but must be accurate by the time rebuilding is finished.

8.3 Lazy Rebuilding

The implementation of physicist's queues in Section 6.4.2 is closely related to global rebuilding, but there is an important difference. As in global rebuilding, this implementation keeps two copies of the front list, the working copy, *w*, and the secondary copy, *f*, with all queries being answered by the working

```
structure HoodMelvilleQueue : QUEUE =
struct
  datatype α RotationState =
        IDLE
      | REVERSING of int × α list × α list × α list × α list
      | APPENDING of int × α list × α list × α list
      | DONE of α list

  type α Queue = int × α list × α RotationState × int × α list

  fun exec (REVERSING (ok, x :: f, f', y :: r, r')) =
        REVERSING (ok+1, f, x :: f', r, y :: r')
    | exec (REVERSING (ok, [], f', [y], r')) = APPENDING (ok, f', y :: r')
    | exec (APPENDING (0, f', r')) = DONE r'
    | exec (APPENDING (ok, x :: f', r')) = APPENDING (ok−1, f', x :: r')
    | exec state = state

  fun invalidate (REVERSING (ok, f, f', r, r')) = REVERSING (ok−1, f, f', r, r')
    | invalidate (APPENDING (0, f', x :: r')) = DONE r'
    | invalidate (APPENDING (ok, f', r')) = APPENDING (ok−1, f', r')
    | invalidate state = state

  fun exec2 (lenf, f, state, lenr, r) =
        case exec (exec state) of
          DONE newf ⇒ (lenf, newf, IDLE, lenr, r)
        | newstate ⇒ (lenf, f, newstate, lenr, r)

  fun check (q as (lenf, f, state, lenr, r)) =
        if lenr ≤ lenf then exec2 q
        else let val newstate = REVERSING (0, f, [], r, [])
             in exec2 (lenf+lenr, f, newstate, 0, []) end

  val empty = (0, [], IDLE, 0, [])
  fun isEmpty (lenf, f, state, lenr, r) = (lenf = 0)

  fun snoc ((lenf, f, state, lenr, r), x) = check (lenf, f, state, lenr+1, x :: r)
  fun head (lenf, [], state, lenr, r) = raise EMPTY
    | head (lenf, x :: f, state, lenr, r) = x
  fun tail (lenf, [], state, lenr, r) = raise EMPTY
    | tail (lenf, x :: f, state, lenr, r) =
        check (lenf−1, f, invalidate state, lenr, r)
end
```

Figure 8.1. Real-time queues based on global rebuilding.

copy. Updates to f (i.e., tail operations) are buffered, to be executed at the end of the rotation, by writing

```
... $tl (force f) ...
```

In addition, this implementation takes care to start (or at least set up) the rotation long before its result is needed. However, unlike global rebuilding, this

implementation does not *execute* the rebuilding transformation (i.e., the rotation) concurrently with the normal operations; rather, it *pays for* the rebuilding transformation concurrently with the normal operations, but then executes the transformation all at once at some point after it has been paid for. In essence, we have replaced the complications of explicitly or implicitly coroutining the rebuilding transformation with the simpler mechanism of lazy evaluation. We call this variant of global rebuilding *lazy rebuilding*.

The implementation of banker's queues in Section 6.3.2 reveals a further simplification possible under lazy rebuilding. By incorporating nested suspensions into the basic data structure — for instance, by using streams instead of lists — we can often eliminate the distinction between the working copy and the secondary copy and employ a single structure that combines aspects of both. The "working" portion of that structure is the part that has already been paid for, and the "secondary" portion is the part that has not yet been paid for.

Global rebuilding has two advantages over batched rebuilding: it is suitable for implementing persistent data structures and it yields worst-case bounds rather than amortized bounds. Lazy rebuilding shares the first advantage, but, at least in its simplest form, yields amortized bounds. However, if desired, worst-case bounds can often be recovered using the scheduling techniques of Chapter 7. For example, the real-time queues in Section 7.2 combine lazy rebuilding with scheduling to achieve worst-case bounds. In fact, the combination of lazy rebuilding and scheduling can be viewed as an instance of global rebuilding in which the coroutines are reified in a particularly simple way using lazy evaluation.

8.4 Double-Ended Queues

As further examples of lazy rebuilding, we next present several implementations of double-ended queues, also known as *deques*. Deques differ from FIFO queues in that elements can be both inserted and deleted from either end of the queue. A signature for deques appears in Figure 8.2. This signature extends the signature for queues with three new functions: cons (insert an element at the front), last (return the rearmost element), and init (remove the rearmost element).

Remark Notice that the signature for queues is a strict subset of the signature for deques — the same names have been chosen for the type and the overlapping functions. Because deques are thus a strict extension of queues, Standard ML will allow us to use a deque module wherever a queue module is expected.

```
signature DEQUE =
sig
  type α Queue

  val empty   : α Queue
  val isEmpty : α Queue → bool

  (* insert, inspect, and remove the front element *)
  val cons    : α × α Queue → 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 *)

  (* insert, inspect, and remove the rear element *)
  val snoc    : α Queue × α → α Queue
  val last    : α Queue → α          (* raises EMPTY if queue is empty *)
  val init    : α Queue → α Queue    (* raises EMPTY if queue is empty *)
end
```

Figure 8.2. Signature for double-ended queues.

8.4.1 Output-Restricted Deques

First, note that extending the queue implementations from Chapters 6 and 7 to support cons, in addition to snoc, is trivial. A queue that supports insertions at both ends, but deletions from only one end, is called an *output-restricted deque*.

For example, we can implement a cons function for the banker's queues of Section 6.3.2 as follows:

```
fun cons (x, (lenf, f, lenr, r)) = (lenf+1, $CONS (x, f), lenr, r)
```

Note that there is no need to call the check helper function because adding an element to *f* cannot possibly make *f* shorter than *r*.

Similarly, we can easily implement a cons function for the real-time queues of Section 7.2.

```
fun cons (x, (f, r, s)) = ($CONS (x, f), r, $CONS (x, s))
```

We add *x* to *s* only to maintain the invariant that $|s| = |f| - |r|$.

Exercise 8.4 Unfortunately, we cannot extend Hood and Melville's real-time queues with a cons function quite so easily, because there is no easy way to insert the new element into the rotation state. Instead, write a functor that extends *any* implementation of queues with a constant-time cons function, using the type

```
type α Queue = α list × α Q.Queue
```

where Q is the parameter to the functor. cons should insert elements into the new list, and head and tail should remove elements from the new list whenever it is non-empty.

8.4.2 Banker's Deques

Deques can be represented in essentially the same way as queues, as two streams (or lists), f and r, plus some associated information to help maintain balance. For queues, the notion of perfect balance is for all the elements to be in the front stream. For deques, the notion of perfect balance is for the elements to be evenly divided between the front and rear streams. Since we cannot afford to restore perfect balance after every operation, we will settle for guaranteeing that neither stream is more than about c times longer than the other, for some constant $c > 1$. Specifically, we maintain the following balance invariant:

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

The "+1" in each term allows for the only element of a singleton deque to be stored in either stream. Note that both streams are non-empty whenever the deque contains at least two elements. Whenever the invariant would otherwise be violated, we restore the deque to perfect balance by transferring elements from the longer stream to the shorter stream until both streams have the same length.

Using these ideas, we can adapt either the banker's queues of Section 6.3.2 or the physicist's queues of Section 6.4.2 to obtain deques that support every operation in $O(1)$ amortized time. Because the banker's queues are slightly simpler, we choose to work with that implementation.

The type of banker's deques is precisely the same as for banker's queues.

```
type α Queue = int × α Stream × int × α Stream
```

The functions on the front element are defined as follows:

```
fun cons (x, (lenf, f, lenr, r)) = check (lenf+1, $Cons (x, f), lenr, r)
fun head (lenf, $Nil, lenr, $Cons (x, _)) = x
  | head (lenf, $Cons (x, f'), lenr, r) = x
fun tail (lenf, $Nil, lenr, $Cons (x, _)) = empty
  | tail (lenf, $Cons (x, f'), lenr, r) = check (lenf−1, f', lenr, r)
```

The first clauses of head and tail handle singleton deques where the single element is stored in the rear stream. The functions on the rear element — snoc, last, and init — are defined symmetrically.

The interesting portion of this implementation is the check helper function,

which restores the deque to perfect balance when one stream becomes too long by first truncating the longer stream to half the combined length of both streams and then transferring the remaining elements of the longer stream onto the back of the shorter stream. For example, if $|f| > c|r| + 1$, then check replaces f with take (i, f) and r with r ++ reverse (drop (i, f)), where $i = \lfloor(|f| + |r|)/2\rfloor$. The full definition of check is

```
fun check (q as (lenf, f, lenr, r)) =
      if lenf > c*lenr + 1 then
         let val i = (lenf + lenr) div 2      val j = lenf + lenr − i
             val f' = take (i, f)             val r' = r ++ reverse (drop (i, f))
         in (i, f', j, r') end
      else if lenr > c*lenf + 1 then
         let val j = (lenf + lenr) div 2      val i = lenf + lenr − j
             val r' = take (j, r)             val f' = f ++ reverse (drop (j, r))
         in (i, f', j, r') end
      else q
```

This implementation is summarized in Figure 8.3.

Remark Because of the symmetry of this implementation, we can reverse a deque in $O(1)$ time by simply swapping the roles of f and r.

```
fun reverse (lenf, f, lenr, r) = (lenr, r, lenf, f)
```

Many other implementations of deques share this property [Hoo92, CG93]. Rather than essentially duplicating the code for the functions on the front element and the functions on the rear element, we could define the functions on the rear element in terms of reverse and the corresponding functions on the front element. For example, we could implement init as

```
fun init q = reverse (tail (reverse q))
```

Of course, init will be slightly faster if implemented directly. ◇

To analyze these deques, we again turn to the banker's method. For both the front and rear streams, let $d(i)$ be the number of debits on element i of the stream, and let $D(i) = \sum_{j=0}^{i} d(j)$. We maintain the debit invariants that, for both the front and rear streams,

$$D(i) \leq \min(ci + i, cs + 1 - t)$$

where $s = \min(|f|, |r|)$ and $t = \max(|f|, |r|)$. Since $d(0) = 0$, the heads of both streams are free of debits and so can be accessed at any time by head or last.

Theorem 8.1 cons *and* tail *(symmetrically,* snoc *and* init*) maintain the debit*

```
functor BankersDeque (val c : int) : DEQUE =      (* c > 1 *)
struct
  type α Queue = int × α Stream × int × α Stream

  val empty = (0, $NIL, 0, $NIL)
  fun isEmpty (lenf, f, lenr, r) = (lenf+lenr = 0)

  fun check (q as (lenf, f, lenr, r)) =
        if lenf > c*lenr + 1 then
          let val i = (lenf + lenr) div 2     val j = lenf + lenr − i
              val f' = take (i, f)            val r' = r ++ reverse (drop (i, f))
          in (i, f', j, r') end
        else if lenr > c*lenf + 1 then
          let val j = (lenf + lenr) div 2     val i = lenf + lenr − j
              val r' = take (j, r)            val f' = f ++ reverse (drop (j, r))
          in (i, f', j, r') end
        else q

  fun cons (x, (lenf, f, lenr, r)) = check (lenf+1, $CONS (x, f), lenr, r)
  fun head (lenf, $NIL, lenr, $NIL) = raise EMPTY
    | head (lenf, $NIL, lenr, $CONS (x, _)) = x
    | head (lenf, $CONS (x, f'), lenr, r) = x
  fun tail (lenf, $NIL, lenr, $NIL) = raise EMPTY
    | tail (lenf, $NIL, lenr, $CONS (x, _)) = empty
    | tail (lenf, $CONS (x, f'), lenr, r) = check (lenf−1, f', lenr, r)

  ...snoc, last, and init defined symmetrically...
end
```

Figure 8.3. An implementation of deques based on lazy rebuilding and the banker's method.

invariants on both the front and rear streams by discharging at most 1 *and* $c + 1$ *debits per stream, respectively.*

Proof Similar to the proof of Theorem 6.1 on page 66. □

By inspection, every operation has an $O(1)$ unshared cost, and by Theorem 8.1, every operation discharges no more than $O(1)$ debits. Therefore, every operation runs in $O(1)$ amortized time.

Exercise 8.5 Prove Theorem 8.1.

Exercise 8.6 Explore the tradeoffs in the choice of the balance constant c. Construct a sequence of operations for which the choice $c = 4$ would be significantly faster than $c = 2$. Now, construct a sequence of operations for which $c = 2$ would be significantly faster than $c = 4$.

8.4.3 Real-Time Deques

Real-time deques support every operation in $O(1)$ worst-case time. We obtain real-time deques from the deques of the previous section by scheduling both the front and rear streams.

As always, the first step in applying the scheduling technique is to convert all monolithic functions to incremental functions. In the previous implementation, the rebuilding transformation rebuilt *f* and *r* as *f* ++ reverse (drop (*j*, *r*)) and take (*j*, *r*) (or vice versa). take and ++ are already incremental, but reverse and drop are monolithic. We therefore rewrite *f* ++ reverse (drop (*j*, *r*)) as rotateDrop (*f*, *j*, *r*) where rotateDrop performs c steps of the drop for every step of the ++ and eventually calls rotateRev, which in turn performs c steps of the reverse for every remaining step of the ++. rotateDrop can be implemented as

```
fun rotateDrop (f, j, r) =
      if j < c then rotateRev (f, drop (j, r), $NIL)
      else let val ($CONS (x, f')) = f
           in $CONS (x, rotateDrop (f', j − c, drop (c, r))) end
```

Initially, $|r| = c|f| + 1 + k$ where $1 \leq k \leq c$. Every call to rotateDrop drops c elements of *r* and processes one element of *f*, except the last, which drops $j \bmod c$ elements of *r* and leaves *f* unchanged. Therefore, at the time of the first call to rotateRev, $|r| = c|f| + 1 + k - (j \bmod c)$. It will be convenient to insist that $|r| \geq c|f|$, so we require that $1 + k - (j \bmod c) \geq 0$. This is guaranteed only for $c < 4$. Since c must be greater than one, the only values of c that we allow are two and three. Then we can implement rotateRev as

```
fun rotateRev ($NIL, r, a) = reverse r ++ a
  | rotateRev ($CONS (x, f), r, a) =
      $CONS (x, rotateRev (f, drop (c, r), reverse (take (c, r)) ++ a))
```

Note that rotateDrop and rotateRev make frequent calls to drop and reverse, which were exactly the functions we were trying to eliminate. However, now drop and reverse are always called with arguments of bounded size, and therefore execute in $O(1)$ steps.

Once we have converted the monolithic functions to incremental functions, the next step is to schedule the execution of the suspensions in *f* and *r*. We maintain a separate schedule for each stream and execute a few suspensions per operation from each schedule. As with the real-time queues of Section 7.2, the goal is to ensure that both schedules are completely evaluated before the next rotation, so that the suspensions that are forced within rotateDrop and rotateRev are guaranteed to have already been memoized.

Exercise 8.7 Show that executing one suspension per stream per insertion and

```
functor RealTimeDeque (val c : int) : DEQUE =    (* c = 2 or c = 3 *)
struct
  type α Queue =
        int × α Stream × α Stream × int × α Stream × α Stream

  val empty = (0, $NIL, $NIL, 0, $NIL, $NIL)
  fun isEmpty (lenf, f, sf, lenr, r, sr) = (lenf+lenr = 0)

  fun exec1 ($CONS (x, s)) = s
    | exec1 s = s
  fun exec2 s = exec1 (exec1 s)

  fun rotateRev ($NIL, r, a) = reverse r ++ a
    | rotateRev ($CONS (x, f), r, a) =
      $CONS (x, rotateRev (f, drop (c, r), reverse (take (c, r)) ++ a))
  fun rotateDrop (f, j, r) =
      if j < c then rotateRev (f, drop (j, r), $NIL)
      else let val ($CONS (x, f')) = f
           in $CONS (x, rotateDrop (f', j − c, drop (c, r))) end

  fun check (q as (lenf, f, sf, lenr, r, sr)) =
      if lenf > c*lenr + 1 then
        let val i = (lenf + lenr) div 2     val j = lenf + lenr − i
            val f' = take (i, f)            val r' = rotateDrop (r, i, f)
        in (i, f', f', j, r', r') end
      else if lenr > c*lenf + 1 then
        let val j = (lenf + lenr) div 2     val i = lenf + lenr − j
            val r' = take (j, r)            val f' = rotateDrop (f, j, r)
        in (i, f', f', j, r', r') end
      else q

  fun cons (x, (lenf, f, sf, lenr, r, sr)) =
      check (lenf+1, $CONS (x, f), exec1 sf, lenr, r, exec1 sr)
  fun head (lenf, $NIL, sf, lenr, $NIL, sr) = raise EMPTY
    | head (lenf, $NIL, sf, lenr, $CONS (x, _), sr) = x
    | head (lenf, $CONS (x, f'), sf, lenr, r, sr) = x
  fun tail (lenf, $NIL, sf, lenr, $NIL, sr) = raise EMPTY
    | tail (lenf, $NIL, sf, lenr, $CONS (x, _), sr) = empty
    | tail (lenf, $CONS (x, f'), sf, lenr, r, sr) =
      check (lenf−1, f', exec2 sf, lenr, r, exec2 sr)

  ...snoc, last, and init defined symmetrically...
end
```

Figure 8.4. Real-time deques via lazy rebuilding and scheduling.

two suspensions per stream per deletion is enough to guarantee that both schedules are completely evaluated before the next rotation. ◇

This implementation is summarized in Figure 8.4.

8.5 Chapter Notes

Global Rebuilding Overmars introduced global rebuilding in [Ove83]. It has since been used in many situations, including real-time queues [HM81], real-time deques [Hoo82, GT86, Sar86, CG93], catenable deques [BT95], and the order maintenance problem [DS87].

Deques Hood [Hoo82] first modified the real-time queues of [HM81] to obtain real-time deques based on global rebuilding. Several other researchers later duplicated this work [GT86, Sar86, CG93]. These implementations are all similar to techniques used to simulate multihead Turing machines [Sto70, FMR72, LS81]. Hoogerwoord [Hoo92] proposed amortized deques based on batched rebuilding, but, as always with batched rebuilding, his implementation is not efficient when used persistently. The real-time deques in Figure 8.4 first appeared in [Oka95c].

Coroutines and Lazy Evaluation Streams (and other lazy data structures) have frequently been used to implement a form of coroutining between the producer of a stream and the consumer of a stream. Landin [Lan65] first pointed out this connection between streams and coroutines. See Hughes [Hug89] for some compelling applications of this feature.