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

11 - Implicit Recursive Slowdown pp. 171-184

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511530104.012

Cambridge University Press

11

Implicit Recursive Slowdown

In Section 9.2.3, we saw how lazy redundant binary numbers support both increment and decrement functions in $O(1)$ amortized time. In Section 10.1.2, we saw how non-uniform types and polymorphic recursion afford particularly simple implementations of numerical representations such as binary random-access lists. In this chapter, we combine and generalize these techniques into a framework called *implicit recursive slowdown*.

Kaplan and Tarjan [KT95, KT96b, KT96a] have studied a related framework, called *recursive slowdown*, that is based on segmented binary numbers (Section 9.2.4) rather than lazy binary numbers. The similarities and differences between implementations based on recursive slowdown and implementations based on implicit recursive slowdown are essentially the same as between those two number systems.

11.1 Queues and Deques

Recall the binary random-access lists of Section 10.1.2, which have the type

```
datatype α RList =
    NIL | ZERO of (α × α) RList | ONE of α × (α × α) RList
```

To simplify later discussions, let us change this type to

```
datatype α Digit = ZERO | ONE of α
datatype α RList = SHALLOW of α Digit | DEEP of α Digit × (α × α) RList
```

A shallow list contains either zero or one elements. A deep list contains either zero or one elements plus a list of pairs. We can play many of the same games with this type that we played with binary random-access lists in Chapter 9. For example, we can support head in $O(1)$ time by switching to a zeroless representation, such as

datatype α Digit = ZERO | ONE **of** α | TWO **of** $\alpha \times \alpha$
datatype α RList = SHALLOW **of** α Digit | DEEP **of** α Digit $\times$ $(\alpha \times \alpha)$ RList

In this representation, the digit in a DEEP node must be ONE or TWO. The ZERO constructor is used only in the empty list, SHALLOW ZERO.

Similarly, by suspending the list of pairs in each DEEP node, we can make either cons or tail run in $O(1)$ amortized time, and the other in $O(\log n)$ amortized time.

datatype α RList =
SHALLOW **of** α Digit
| DEEP **of** α Digit $\times$ $(\alpha \times \alpha)$ RList susp

By allowing a choice of three non-zero digits in each DEEP node, we can make all three of cons, head, and tail run in $O(1)$ time.

datatype α Digit =
ZERO | ONE **of** α | TWO **of** $\alpha \times \alpha$ | THREE **of** $\alpha \times \alpha \times \alpha$

Again, the ZERO constructor is used only in the empty list.

Now, extending this design to support queues and deques is simply a matter of adding a second digit to each DEEP node.

datatype α Queue =
SHALLOW **of** α Digit
| DEEP **of** α Digit $\times$ $(\alpha \times \alpha)$ Queue susp $\times$ α Digit

The first digit represents the first few elements of the queue, and the second digit represents the last few elements. The remaining elements are stored in the suspended queue of pairs, which we call the *middle* queue.

The exact choice of the digit type depends on what functions are to be supported on each end of the queue. The following table lists the allowable values for the front digit of a queue that supports the given combination of functions.

supported functions	allowable digits
cons	ZERO, ONE
cons/head	ONE, TWO
head/tail	ONE, TWO
cons/head/tail	ONE, TWO, THREE

The same choices apply to the rear digit.

As a concrete example, let us develop an implementation of queues supporting snoc on the rear end of the queue, and head and tail on the front end of the queue (i.e., ordinary FIFO queues). Reading from the above table, we choose to allow the front digit of a DEEP node to be ONE or TWO and the rear digit to be ZERO or ONE. We also allow the digit in a SHALLOW node to be ZERO or ONE.

To add a new element y to a deep queue using snoc, we look at the rear digit. If it is ZERO, then we replace the rear digit with ONE y. If it is ONE x, then we replace the rear digit with ZERO and add the pair (x, y) to the middle queue. We also need a few special cases for adding an element to a shallow queue.

```
fun snoc (SHALLOW ZERO, y) = SHALLOW (ONE y)
  | snoc (SHALLOW (ONE x), y) = DEEP (TWO (x, y), $empty, ZERO)
  | snoc (DEEP (f, m, ZERO), y) = DEEP (f, m, ONE y)
  | snoc (DEEP (f, m, ONE x), y) =
      DEEP (f, $snoc (force m, (x, y)), ZERO)
```

To remove an element from a deep queue using tail, we look at the front digit. If it is TWO (x, y), then we discard x and set the front digit to ONE y. If it is ONE x, then we "borrow" a pair (y, z) from the middle queue and set the front digit to TWO (y, z). Again, there are several special cases dealing with shallow queues.

```
fun tail (SHALLOW (ONE x)) = empty
  | tail (DEEP (TWO (x, y), m, r)) = DEEP (ONE y, m, r)
  | tail (DEEP (ONE x, $q, r)) =
      if isEmpty q then SHALLOW r
      else let val (y, z) = head q
           in DEEP (TWO (y, z), $tail q, r) end
```

Note that we force the middle queue in the last clause of tail. The complete code appears in Figure 11.1.

Next, we show that snoc and tail both run in $O(1)$ amortized time. Note that snoc ignores the front digit and tail ignores the rear digit. If we consider each function in isolation, then snoc is analogous to inc on lazy binary numbers and tail is analogous to dec on zeroless lazy binary numbers. By adapting the proofs for inc and dec, we can easily show that snoc and tail run in $O(1)$ amortized time as long as each is used without the other.

The key idea of implicit recursive slowdown is that, when functions like snoc and tail are *almost* independent, then we can combine their proofs by simply adding the debits required by each proof. The proof for snoc allows one debit if the rear digit is ZERO and zero debits if the rear digit is One. The proof for tail allows one debit if the front digit is Two and zero debits if the front digit is One. The following proof combines these debit allowances.

Theorem 11.1 snoc *and* tail *run in* $O(1)$ *amortized time.*

Proof We analyze this implementation using the banker's method. We assign debits to every suspension, each of which is the middle field of some deep queue. We adopt a debit invariant that allows each suspension a number of debits governed by the digits in the front and rear fields. The middle field of a

```
structure ImplicitQueue : QUEUE =
  (* assumes polymorphic recursion! *)
struct
  datatype α Digit = ZERO | ONE of α | TWO of α × α
  datatype α Queue =
        SHALLOW of α Digit
      | DEEP of α Digit × (α × α) Queue susp × α Digit

  val empty = SHALLOW ZERO
  fun isEmpty (SHALLOW ZERO) = true | isEmpty _ = false

  fun snoc (SHALLOW ZERO, y) = SHALLOW (ONE y)
    | snoc (SHALLOW (ONE x), y) = DEEP (TWO (x, y), $empty, ZERO)
    | snoc (DEEP (f, m, ZERO), y) = DEEP (f, m, ONE y)
    | snoc (DEEP (f, m, ONE x), y) =
        DEEP (f, $snoc (force m, (x, y)), ZERO)

  fun head (SHALLOW ZERO) = raise EMPTY
    | head (SHALLOW (ONE x)) = x
    | head (DEEP (ONE x, m, r)) = x
    | head (DEEP (TWO (x, y), m, r)) = x
  fun tail (SHALLOW ZERO) = raise EMPTY
    | tail (SHALLOW (ONE x)) = empty
    | tail (DEEP (TWO (x, y), m, r)) = DEEP (ONE y, m, r)
    | tail (DEEP (ONE x, $q, r)) =
        if isEmpty q then SHALLOW r
        else let val (y, z) = head q
             in DEEP (TWO (y, z), $tail q, r) end
end
```

Figure 11.1. Queues based on implicit recursive slowdown.

deep queue may have up to $|f| - |r|$ debits, where $|f|$ is one or two, and $|r|$ is zero or one.

The unshared cost of each function is $O(1)$, so we must merely show that neither function discharges more than $O(1)$ debits. We describe only the proof for tail. The proof for snoc is slightly simpler.

We argue by debit passing, which is closely related to debit inheritance. Whenever a nested suspension has more debits than it is allowed, we pass those debits to the enclosing suspension, which is the middle field of the previous DEEP node. Debit passing is safe because the outer suspension must be forced before the inner suspension can be forced. Passing responsibility for discharging debits from a nested suspension to the enclosing suspension ensures that those debits will be discharged before the outer suspension is forced, and hence before the inner suspension can be forced.

We show that every call to tail passes one debit to its enclosing suspension, except the outermost call, which has no enclosing suspension. That call simply discharges its excess debit.

Each cascade of tails ends in a call to tail that changes f from TWO to ONE. (For simplicity of presentation, we ignore the possibility of shallow queues). This decreases the debit allowance of m by one, so we pass the excess debit to the enclosing suspension.

Every intermediate call to tail changes f from ONE to TWO and recurses. There are two subcases:

- r is ZERO. m has one debit, which must be discharged before m can be forced. We pass this debit to the enclosing suspension. We create one debit to cover the unshared cost of the suspended recursive call. In addition, this suspension is passed one debit by the recursive call. Since this suspension has a debit allowance of two, we are done.
- r is ONE. m has zero debits, so we can force it for free. We create one debit to cover the unshared cost of the suspended recursive call. In addition, this suspension is passed one debit by the recursive call. Since this suspension has a debit allowance of one, we keep one debit and pass the other to the enclosing suspension.

□

Exercise 11.1 Implement lookup and update functions for these queues. Your functions should run in $O(\log i)$ amortized time. You may find it helpful to augment each queue with a size field.

Exercise 11.2 Implement double-ended queues using the techniques of this section.

11.2 Catenable Double-Ended Queues

Finally, we use implicit recursive slowdown to implement catenable double-ended queues, with the signature shown in Figure 11.2. We first describe a relatively simple implementation that supports ++ in $O(\log n)$ amortized time and all other operations in $O(1)$ amortized time. We then describe a much more complicated implementation that improves the running time of ++ to $O(1)$.

Consider the following representation for catenable double-ended queues, or *c-deques*. A c-deque is either *shallow* or *deep*. A shallow c-deque is simply an ordinary deque, such as the banker's deques of Section 8.4.2. A deep c-deque is decomposed into three segments: a *front*, a *middle*, and a *rear*. The front and

```
signature CATENABLEDEQUE =
sig
  type α Cat

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

  val cons    : α × α Cat → α Cat
  val head    : α Cat → α          (* raises EMPTY if deque is empty *)
  val tail    : α Cat → α Cat      (* raises EMPTY if deque is empty *)

  val snoc    : α Cat × α → α Cat
  val last    : α Cat → α          (* raises EMPTY if deque is empty *)
  val init    : α Cat → α Cat      (* raises EMPTY if deque is empty *)

  val ++      : α Cat × α Cat → α Cat
end
```

Figure 11.2. Signature for catenable double-ended queues.

rear are both ordinary deques containing two or more elements each. The middle is a c-deque of ordinary deques, each containing two or more elements. We assume that D is an implementation of deques satisfying the signature DEQUE, and that all of the functions in D run in $O(1)$ time (amortized or worst-case).

```
datatype α Cat =
    SHALLOW of α D.Queue
  | DEEP of α D.Queue × α D.Queue Cat susp × α D.Queue
```

Note that this definition assumes polymorphic recursion.

To insert an element at either end, we simply insert the element into the front deque or the rear deque. For instance, cons is implemented as

```
fun cons (x, SHALLOW d) = SHALLOW (D.cons (x, d))
  | cons (x, DEEP (f, m, r)) = DEEP (D.cons (x, f), m, r)
```

To remove an element from either end, we remove an element from the front deque or the rear deque. If this drops the length of that deque below two, then we remove the next deque from the middle, add the one remaining element from the old deque, and install the result as the new front or rear. With the addition of the remaining element from the old deque, the new deque contains at least three elements. For example, the code for tail is

```
fun tail (SHALLOW d) = SHALLOW (D.tail d)
  | tail (DEEP (f, m, r)) =
      let val f' = D.tail f
      in
        if not (tooSmall f') then DEEP (f', m, r)
        else if isEmpty (force m) then SHALLOW (dappendL (f', r))
        else DEEP (dappendL (f', head (force m)), $tail (force m), r)
      end
```

where tooSmall tests if the length of a deque is less than two and dappendL appends a deque of length zero or one to a deque of arbitrary length.

Note that calls to tail propagate to the next level of the c-deque only when the length of the front deque is two. In the terminology of Section 9.2.3, we say that a deque of length three or more is *safe* and a deque of length two is *dangerous*. Whenever tail does call itself recursively on the next level, it also changes the front deque from dangerous to safe, so that two successive calls to tail on a given level of the c-deque never both propagate to the next level. We can easily prove that tail runs in $O(1)$ amortized time by allowing one debit per safe deque and zero debits per dangerous deque.

Exercise 11.3 Prove that both tail and init run in $O(1)$ amortized time by combining their respective debit allowances as suggested by implicit recursive slowdown. ◇

Now, what about catenation? To catenate two deep c-deques c_1 and c_2, we retain the front of c_1 as the new front, the rear of c_2 as the new rear, and combine the remaining segments into the new middle by inserting the rear of c_1 into the middle of c_1, and the front of c_2 into the middle of c_2, and then catenating the results.

```
fun (DEEP (f1, m1, r1)) ++ (DEEP (f2, m2, r2)) =
      DEEP (f1, $(snoc (force m1, r1) ++ cons (f2, force m2)), r2)
```

(Of course, there are also cases where c_1 or c_2 are shallow.) Note that ++ recurses to the depth of the shallower c-deque. Furthermore, ++ creates $O(1)$ debits per level, which must be immediately discharged to restore the debit invariant required by the tail and init. Therefore, ++ runs in $O(\min(\log n_1, \log n_2))$ amortized time, where n_i is the size of c_i.

The complete code for this implementation of c-deques appears in Figure 11.3.

To improve the running time of ++ to $O(1)$, we modify the representation of c-deques so that ++ does not call itself recursively. The key is to enable ++ at one level to call only cons and snoc at the next level. Instead of three segments, we expand deep c-deques to contain five segments: (f, a, m, b, r). f, m, and

```
functor SimpleCatenableDeque (D : DEQUE) : CATENABLEDEQUE =
  (* assumes polymorphic recursion! *)
struct
  datatype α Cat =
      SHALLOW of α D.Queue
    | DEEP of α D.Queue × α D.Queue Cat susp × α D.Queue

  fun tooSmall d = D.isEmpty d orelse D.isEmpty (D.tail d)

  fun dappendL (d1, d2) =
      if D.isEmpty d1 then d2 else D.cons (D.head d1, d2)
  fun dappendR (d1, d2) =
      if D.isEmpty d2 then d1 else D.snoc (d1, D.head d2)

  val empty = SHALLOW D.empty
  fun isEmpty (SHALLOW d) = D.isEmpty d
    | isEmpty _ = false

  fun cons (x, SHALLOW d) = SHALLOW (D.cons (x, d))
    | cons (x, DEEP (f, m, r)) = DEEP (D.cons (x, f), m, r)
  fun head (SHALLOW d) = D.head d
    | head (DEEP (f, m, r)) = D.head f
  fun tail (SHALLOW d) = SHALLOW (D.tail d)
    | tail (DEEP (f, m, r)) =
      let val f' = D.tail f
      in
        if not (tooSmall f') then DEEP (f', m, r)
        else if isEmpty (force m) then SHALLOW (dappendL (f', r))
        else DEEP (dappendL (f', head (force m)), $tail (force m), r)
      end

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

  fun (SHALLOW d1) ++ (SHALLOW d2) =
      if tooSmall d1 then SHALLOW (dappendL (d1, d2))
      else if tooSmall d2 then SHALLOW (dappendR (d1, d2))
      else DEEP (d1, $empty, d2)
    | (SHALLOW d) ++ (DEEP (f, m, r)) =
      if tooSmall d then DEEP (dappendL (d, f), m, r)
      else DEEP (d, $cons (f, force m), r)
    | (DEEP (f, m, r)) ++ (SHALLOW d) =
      if tooSmall d then DEEP (f, m, dappendR (r, d))
      else DEEP (f, $snoc (force m, r), d)
    | (DEEP (f1, m1, r1)) ++ (DEEP (f2, m2, r2)) =
      DEEP (f1, $(snoc (force m1, r1) ++ cons (f2, force m2)), r2)
end
```

Figure 11.3. Simple catenable deques.

r are all ordinary deques; f and r contain three or more elements each, and m contains two or more elements. a and b are c-deques of *compound elements*. A degenerate compound element is simply an ordinary deque containing two or more elements. A full compound element has three segments: (f, c, r), where f and r are ordinary deques containing at least two elements each, and c is a c-deque of compound elements. This datatype can be written in Standard ML (with polymorphic recursion) as

```
datatype α Cat =
      SHALLOW of α D.Queue
    | DEEP of α D.Queue                    (* ≥ 3 *)
           × α CmpdElem Cat susp
           × α D.Queue                     (* ≥ 2 *)
           × α CmpdElem Cat susp
           × α D.Queue                     (* ≥ 3 *)
and α CmpdElem =
      SIMPLE of α D.Queue                  (* ≥ 2 *)
    | CMPD of α D.Queue                    (* ≥ 2 *)
           × α CmpdElem Cat susp
           × α D.Queue                     (* ≥ 2 *)
```

Given c-deques c_1 = DEEP (f_1,a_1,m_1,b_1,r_1) and c_2 = DEEP (f_2,a_2,m_2,b_2,r_2), we compute their catenation as follows: First, we retain f_1 as the front of the result, and r_2 as the rear of the result. Next, we build the new middle deque from the last element of r_1 and the first element of f_2. We then combine m_1, b_1, and the rest of r_1 into a compound element, which we snoc onto a_1. This becomes the new a segment of the result. Finally, we combine the rest of f_2, a_2, and m_2 into a compound element, which we cons onto b_2. This becomes the new b segment of the result. Altogether, this is implemented as

```
fun (DEEP (f1, a1, m1, b1, r1)) ++ (DEEP (f2, a2, m2, b2, r2)) =
      let val (r1', m, f2') = share (r1, f2)
          val a1' = $snoc (force a1, CMPD (m1, b1, r1'))
          val b2' = $cons (CMPD (f2', a2, m2), force b2)
      in DEEP (f1, a1', m, b2', r2) end
```

where

```
fun share (f, r) =
      let val m = D.cons (D.last f, D.cons (D.head r, D.empty))
      in (D.init f, m, D.tail r)
fun cons (x, DEEP (f, a, m, b, r)) = DEEP (D.cons (x, f), a, m, b, r)
fun snoc (DEEP (f, a, m, b, r), x) = DEEP (f, a, m, b, D.snoc (r, x))
```

(For simplicity of presentation, we have ignored all cases involving shallow c-deques.)

Unfortunately, in this implementation, tail and init are downright messy. Since the two functions are symmetric, we describe only tail. Given some c-deque c = DEEP (f,a,m,b,r), there are six cases:

- $|f| > 3$.
- $|f| = 3$.
  - a is non-empty.
    - The first compound element of a is degenerate.
    - The first compound element of a is full.
  - a is empty and b is non-empty.
    - The first compound element of b is degenerate.
    - The first compound element of b is full.
  - a and b are both empty.

Here we describe the behavior of tail c in the first three cases. The remaining cases are covered by the complete implementation in Figures 11.4 and 11.5. If $|f| > 3$ then we simply replace f with D.tail f. If $|f| = 3$, then removing an element from f would drop its length below the allowable minimum. Therefore, we remove a new front deque from a and combine it with the remaining two elements of the old f. The new f contains at least four elements, so the next call to tail will fall into the $|f| > 3$ case.

When we remove the first compound element of a to find the new front deque, we get either a degenerate compound element or a full compound element. If we get a degenerate compound element (i.e., a simple deque), then the new value of a is \$tail (force A). If we get a full compound element Cmpd (f',c',r'), then f' becomes the new f (along with the remaining elements of the old f), and the new value of a is

$$\$(\text{force } c' +\!\!+ \text{ cons } (\text{SIMPLE } r', \text{ tail } (\text{force } a)))$$

But note that the effect of the cons and tail is to replace the first element of a. We can do this directly, and avoid an unnecessary call to tail, using the function replaceHead.

```
fun replaceHead (x, SHALLOW d) = SHALLOW (D.cons (x, D.tail d))
  | replaceHead (x, DEEP (f, a, m, b, r)) =
      DEEP (D.cons (x, D.tail f), a, m, b, r)
```

The remaining cases of tail are similar, each doing $O(1)$ work followed by at most one call to tail.

Remark This code can be written much more succinctly and much more perspicuously using a language feature called *views* [Wad87, BC93, PPN96], which allows pattern matching on abstract types. See [Oka97] for further details. Standard ML does not support views. ◇

The cons, snoc, head, and last functions make no use of lazy evaluation,

```
functor ImplicitCatenableDeque (D : DEQUE) : CATENABLEDEQUE =
  (* assumes that D also supports a size function *)
struct
  datatype α Cat =
      SHALLOW of α D.Queue
    | DEEP of α D.Queue × α CmpdElem Cat susp × α D.Queue
                       × α CmpdElem Cat susp × α D.Queue
  and α CmpdElem =
      SIMPLE of α D.Queue
    | CMPD of α D.Queue × α CmpdElem Cat susp × α D.Queue

  val empty = SHALLOW D.empty
  fun isEmpty (SHALLOW d) = D.isEmpty d
    | isEmpty _ = false

  fun cons (x, SHALLOW d) = SHALLOW (D.cons (x, d))
    | cons (x, DEEP (f, a, m, b, r)) = DEEP (D.cons (x, f), a, m, b, r)
  fun head (SHALLOW d) = D.head d
    | head (DEEP (f, a, m, b, r)) = D.head f

  ...snoc and last defined symmetrically...

  fun share (f, r) =
      let val m = D.cons (D.last f, D.cons (D.head r, D.empty))
      in (D.init f, m, D.tail r)
  fun dappendL (d1, d2) =
      if D.isEmpty d1 then d2
      else dappendL (D.init d1, D.cons (D.last d1, d2))
  fun dappendR (d1, d2) =
      if D.isEmpty d2 then d1
      else dappendR (D.snoc (d1, D.head d2), D.tail d2)

  fun (SHALLOW d1) ++ (SHALLOW d2) =
      if D.size d1 < 4 then SHALLOW (dappendL (d1, d2))
      else if D.size d2 < 4 then SHALLOW (dappendR (d1, d2))
      else let val (f, m, r) = share (d1, d2)
           in DEEP (f, $empty, m, $empty, r) end
    | (SHALLOW d) ++ (DEEP (f, a, m, b, r)) =
      if D.size d < 4 then DEEP (dappendL (d, f), a, m, b, r)
      else DEEP (d, $cons (SIMPLE f, force a), m, b, r)
    | (DEEP (f, a, m, b, r)) ++ (SHALLOW d) =
      if D.size d < 4 then DEEP (f, a, m, b, dappendR (r, d))
      else DEEP (f, a, m, $snoc (force b, SIMPLE r), d)
    | (DEEP (f1, a1, m1, b1, r1)) ++ (DEEP (f2, a2, m2, b2, r2)) =
      let val (r1', m, f2') = share (r1, f2)
          val a1' = $snoc (force a1, CMPD (m1, b1, r1'))
          val b2' = $cons (CMPD (f2', a2, m2), force b2)
      in DEEP (f1, a1', m, b2', r2) end

  ...
```

Figure 11.4. Catenable deques using implicit recursive slowdown (part I).

```
...
fun replaceHead (x, SHALLOW d) = SHALLOW (D.cons (x, D.tail d))
  | replaceHead (x, DEEP (f, a, m, b, r)) =
      DEEP (D.cons (x, D.tail f), a, m, b, r)

fun tail (SHALLOW d) = SHALLOW (D.tail d)
  | tail (DEEP (f, a, m, b, r)) =
      if D.size f > 3 then DEEP (D.tail f, a, m, b, r)
      else if not (isEmpty (force a)) then
        case head (force a) of
          SIMPLE d ⇒
            let val f' = dappendL (D.tail f, d)
            in DEEP (f', $tail (force a), m, b, r) end
        | CMPD (f', c', r') ⇒
            let val f'' = dappendL (D.tail f, f')
                val a'' = $(force c' ++ replaceHead (SIMPLE r', force a))
            in DEEP (f'', a'', m, b, r) end
      else if not (isEmpty (force b)) then
        case head (force b) of
          SIMPLE d ⇒
            let val f' = dappendL (D.tail f, m)
            in DEEP (f', $empty, d, $tail (force b), r) end
        | CMPD (f', c', r') ⇒
            let val f'' = dappendL (D.tail f, m)
                val a'' = $cons (SIMPLE f', force c')
            in DEEP (f'', a'', r', $tail (force b), r) end
      else SHALLOW (dappendL (D.tail f, m)) ++ SHALLOW r

  ...replaceLast and init defined symmetrically...
end
```

Figure 11.5. Catenable deques using implicit recursive slowdown (part II).

and are easily seen to take $O(1)$ worst-case time. We analyze the remaining functions using the banker's method and debit passing.

As always, we assign debits to every suspension, each of which is the *a* or *b* segment of a deep c-deque, or the middle (*c*) segment of a compound element. Each *c* field is allowed four debits, but *a* and *b* fields may have from zero to five debits, based on the lengths of the *f* and *r* fields. *a* and *b* have a base allowance of zero debits. If *f* contains more than three elements, then the allowance for *a* increases by four debits and the allowance for *b* increases by one debit. Similarly, if *r* contains more than three elements, then the allowance for *b* increases by four debits and the allowance for *a* increases by one debit.

Theorem 11.2 ++, tail, *and* init *run in* $O(1)$ *amortized time.*

Proof (++) The interesting case is catenating two c-deques DEEP (f_1,a_1,m_1,b_1,r_1) and DEEP (f_2,a_2,m_2,b_2,r_2). In that case, ++ does $O(1)$ unshared work and discharges at most four debits. First, we create two debits for the suspended snoc and cons onto a_1 and b_2, respectively. We always discharge these two debits. In addition, if b_1 or a_2 has five debits, then we must discharge one debit when that segment becomes the middle of a compound element. Also, if f_1 has only three elements but f_2 has more than three elements, then we must discharge a debit from b_2 as it becomes the new b. Similarly for r_1 and r_2. However, note that if b_1 has five debits, then f_1 has more than three elements, and that if a_2 has five debits, then r_2 has more than three elements. Therefore, we must discharge at most four debits altogether, or at least pass those debits to an enclosing suspension.

(tail and init) Since tail and init are symmetric, we include the argument only for tail. By inspection, tail does $O(1)$ unshared work, so we must show that it discharges only $O(1)$ debits. In fact, we show that it discharges at most five debits.

Since tail can call itself recursively, we must account for a cascade of tails. We argue by debit passing. Given some deep c-deque DEEP (f,a,m,b,r), there is one case for each case of tail.

If $|f| > 3$, then this is the end of a cascade. We create no new debits, but removing an element from f might decrease the allowance of a by four debits, and the allowance of b by one debit, so we pass these debits to the enclosing suspension.

If $|f| = 3$, then assume a is non-empty. (The cases where a is empty are similar.) If $|r| > 3$, then a might have one debit, which we pass to the enclosing suspension. Otherwise, a has no debits. If the head of a is a degenerate compound element (i.e., a simple deque of elements), then this becomes the new f along with the remaining elements of the old f. The new a is a suspension of the tail of the old a. This suspension receives at most five debits from the recursive call to tail. Since the new allowance of a is at least four debits, we pass at most one of these debits to the enclosing suspension, for a total of at most two debits. (Actually, the total is at most one debit since we pass one debit here exactly in the case that we did not have to pass one debit for the original a).

Otherwise, if the head of a is a full compound element CMPD (f',c',r'), then f' becomes the new f along with the remaining elements of the old f. The new a involves calls to ++ and replaceHead. The total number of debits on the new a is nine: four debits from c', four debits from the ++, and one newly created debit for the replaceHead. The allowance for the new a is either four or five, so we pass either five or four of these nine debits to the enclosing suspension.

Since we pass four of these debits exactly in the case that we had to pass one debit from the original *a*, we always pass at most five debits. □

Exercise 11.4 Given an implementation D of non-catenable deques, implement catenable lists using the type

```
datatype α Cat =
    SHALLOW of α D.Queue
  | DEEP of α D.Queue × α CmpdElem Cat susp × α D.Queue
and α CmpdElem = CMPD of α D.Queue × α CmpdElem Cat susp
```

where both the front deque of a DEEP node and the deque in a CMPD node contain at least two elements. Prove that every function in your implementation runs in $O(1)$ amortized time, assuming that all the functions in D run in $O(1)$ time (worst-case or amortized).

11.3 Chapter Notes

Recursive Slowdown Kaplan and Tarjan introduced recursive slowdown in [KT95], and used it again in [KT96b], but it is closely related to the regularity constraints of Guibas et al. [GMPR77]. Brodal [Bro95] used a similar technique to implement heaps.

Catenable Deques Buchsbaum and Tarjan [BT95] present a purely functional implementation of catenable deques that supports tail and init in $O(\log^* n)$ worst-case time and all other operations in $O(1)$ worst-case time. Our implementation improves that bound to $O(1)$ for all operations, although in the amortized rather than worst-case sense. Kaplan and Tarjan have independently developed a similar implementation with worst-case bounds [KT96a]. However, the details of their implementation are quite complicated.