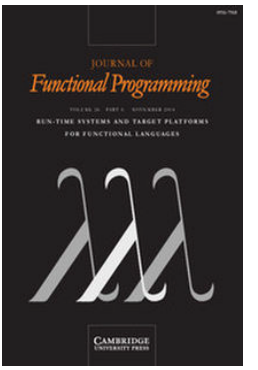

Simple and efficient purely functional queues and deques

Chris Okasaki

Journal of Functional Programming / Volume 5 / Issue 04 / 1995, pp 583 - 592
DOI: 10.1017/S0956796800001489, Published online: 07 November 2008

J. Functional Programming 5 (4): 583–592, October 1995

Simple and efficient purely functional queues and deques

CHRIS OKASAKI

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

Abstract

We present purely functional implementations of queues and double-ended queues (deques) requiring only $O(1)$ time per operation in the worst case. Our algorithms are considerably simpler than previous designs with the same bounds. The inspiration for our approach is the incremental behaviour of certain functions on lazy lists.

Capsule Review

This paper presents another example of the ability to write programs in functional languages that satisfy our desire for clarity while satisfying our need for efficiency. In this case, the subject (often-studied) is the implementation of queues and dequeues that are functional and exhibit constant-time worst-case insertion and deletion operations. Although the problem has been solved previously, this paper presents the simplest algorithm so far. As the author notes, it has a strange feature of requiring some laziness – but not too much!

1 Introduction

Consider the related problems of implementing queues and double-ended queues (deques) in a purely functional programming language. The standard solutions require only $O(1)$ amortized time per operation, but might require $O(n)$ time for any particular operation. Improved solutions, requiring $O(1)$ time per operation in the worst case, have existed for well over a decade but have not seen widespread use, perhaps due to their complexity. We present new implementations of queues and deques, based on lazy lists, which also require $O(1)$ worst-case time per operation but which are considerably simpler than previous solutions.

We begin by reviewing the standard paired-list implementation of queues. We then introduce lazy lists and use them to reduce the worst-case time per operation to $O(\log n)$. Next, by incrementally pre-evaluating the lazy lists, we reduce the worst-case time to $O(1)$. Finally, we extend our approach to double-ended queues and close with a discussion of related and future work.

SML source code for the algorithms described in this paper is available via anonymous `ftp` at `ftp.cs.cmu.edu` in `/afs/cs/project/fox/ftp/queue.tar.Z`.

$$
\begin{aligned}
[]_q &= \langle [], [] \rangle \\
|\langle L, R \rangle|_q &= |L| + |R| \\
\mathit{insert}\ (e, \langle L, R \rangle) &= \langle L, e : R \rangle \\
\mathit{remove}\ \langle L, R \rangle &= \langle \mathit{hd}\ L, \langle \mathit{tl}\ L, R \rangle \rangle \quad \{|L| > 0\} \\
&= \mathit{remove}\ \langle \mathit{rev}\ R, [] \rangle \quad \{|L| = 0\}
\end{aligned}
$$

Fig. 1. The standard implementation of queues as paired lists. Each operation requires only $O(1)$ amortized time, but any particular *remove* might require $O(n)$ time.

2 Queues as paired lists

We desire an implementation of queues with the following specifications:

- $[]_q$ is the empty queue.
- *insert* (e, Q) returns the new queue obtained by adding e to the end of Q.
- *remove* Q returns the pair $\langle e, Q' \rangle$ consisting of the first element of Q and the new queue obtained by deleting e from Q. *remove* Q is undefined if Q is empty.
- $|Q|_q$ returns the length of Q.

The first temptation is to represent a queue as a list. Although removing the head of the list could easily be accomplished in $O(1)$ time, inserting an element at the end of the list would require $O(n)$ time. If the list were maintained in reverse order, then insertions would require $O(1)$ time and removals $O(n)$ time.

The standard trick, reinvented many times (Hood and Melville, 1981; Gries, 1981; Burton, 1982), is to represent the queue as a pair of lists $\langle L, R \rangle$ where L is the front portion of the queue and R is the rear portion of the queue *in reverse order.* In this representation, the first element of the queue is the head of L and the last element of the queue is the head of R. Each is easily accessible in $O(1)$ time. The only complication occurs if we try to remove an element when L is empty. Then, the first element of the queue is the *last* element of R. In this case, we replace $\langle [], R \rangle$ with $\langle \mathit{rev}\ R, [] \rangle$ and try again. (The result is undefined if R is also empty.) Code for this implementation appears in Figure 1.

Assuming that length information is maintained for each list, we can easily see that each queue operation requires $O(1)$ time except for removals that trigger a list reversal. However, a list of length n is only reversed when there have been n insertions since the last list reversal. By amortizing the cost of the reversal over those insertions, we see that each operation requires only $O(1)$ amortized time.

Although simple and efficient, there are two situations when this implementation might be unsatisfactory. The first is in real-time programming, where an $O(n)$ pause to reverse the list might be unacceptable. The second is when previous versions of the queue may be accessed – not just the most recent. (This property, known as *full persistence* (Sarnak, 1986), is always assumed in purely functional languages.) In this case, a list reversal might be triggered many times by repeatedly removing from the same queue.

$$
\begin{aligned}
X +\!\!+ Y &= Y && \{|X| = 0\} \\
&= hd\ X : (tl\ X +\!\!+ Y) && \{|X| > 0\}
\end{aligned}
$$

$$
\begin{aligned}
rev\ X &= rev'(X, [\,]) \\
&\quad \text{where } rev'(X, A) = A && \{|X| = 0\} \\
&\qquad\qquad = rev'(tl\ X, hd\ X : A) && \{|X| > 0\}
\end{aligned}
$$

$$
\begin{aligned}
take\,(n, X) &= [\,] && \{n = 0\} \\
&= hd\ X : take\,(n - 1, tl\ X) && \{n > 0\}
\end{aligned}
$$

$$
\begin{aligned}
drop\,(n, X) &= X && \{n = 0\} \\
&= drop\,(n - 1, tl\ X) && \{n > 0\}
\end{aligned}
$$

Fig. 2. Some useful functions on lists. Note that append (++) and *take* are incremental but *rev* and *drop* are not.

The culprit in both cases is the linear-time list reversal. The solution is to perform the list reversal *incrementally* rather than all at once. However, one must take care to begin the list reversal early enough that it is completed by the time the first element is needed. We next discuss the use of lazy lists in incremental computation and then show how to use lazy lists in implementing queues.

3 Lazy lists and incremental computation

For our purposes, lazy lists (often called *streams*; Abelson and Sussman, 1985) have two important properties. First, the tails of lists are computed on demand, that is, they are not evaluated until and unless they are needed. Second, this evaluation, when it does occur, is memoized (i.e., the first time a tail is demanded, the result is cached so that the next time it is demanded the result can be returned immediately). Such lists are widely useful and appear in many functional programming languages, either built-in or in libraries. From here on, all lists in this paper are assumed to be lazy.

We illustrate the properties of lazy lists through a discussion of the append function (++), shown in Figure 2. How much time does this function require and exactly when does the computation take place? For strict lists, append requires $|X| + 1$ steps, which occur all at once. Each step requires $O(1)$ time. For lazy lists, the answer is more complicated. Again the function requires $|X| + 1$ steps, but these steps occur one at a time. Because the recursive call to $(tl\ X +\!\!+ Y)$ is delayed, the first step returns immediately, and each successive step does not take place until the appropriate tail of the resulting lazy list is demanded. For this reason, we call append an *incremental* function. Examining the other functions in Figure 2, we see that *take* is also incremental, but *rev* and *drop* are not – each performs all of its work at once. The key to implementing queues (and later deques) using lazy lists will be to write *rev* and *drop* as incremental functions.

There is another difference between strict and lazy lists – for lazy lists, each of the $|X| + 1$ steps of the append might require more than $O(1)$ time. During evaluation

of the delayed tail ($tl\ X +\!\!+ Y$), the tail of X is demanded. Evaluation of that tail could involve arbitrary computation. However, since lazy lists are memoized, we can guarantee that no step of the append requires more than $O(1)$ time if we can ensure that the lazy list X has been entirely evaluated prior to the call to append. We will take advantage of this in Section 5. Finally, note that although cons is lazy in its second argument, all other operations are assumed to be strict.

4 Queues as paired lazy lists

For the next implementation, we represent a queue as a pair of lazy lists $\langle L, R\rangle$ with R reversed in the usual way. To prevent the long pauses of the previous implementation, we wish to incrementally reverse R before it is needed, that is, before L becomes empty. We do this by periodically replacing $\langle L, R\rangle$ with $\langle L +\!\!+ rev\ R, [\,]\rangle$. Call this a *rotation*. The trick is to compute $L +\!\!+ rev\ R$ in an incremental fashion. Since append is already incremental, we can make the whole thing incremental by performing one step of the reverse for every step of the append. Of course, this assumes that L and R are about the same length. We will maintain the invariant $|R| \leq |L|$ and perform a rotation whenever the invariant would otherwise be violated. Thus, a rotation occurs whenever $|R| = |L| + 1$.

We implement rotations as follows, where $rot\,(L, R, [\,]) = L +\!\!+ rev\ R$ and A accumulates the partially reversed list:

$$\begin{aligned} rot\,(L, R, A) &= hd\ R : A && \{|L| = 0\} \\ &= hd\ L : rot\,(tl\ L, tl\ R, hd\ R : A) && \{|L| > 0\} \end{aligned}$$

Note that this is nothing more than a fusion of the usual code for append with the usual code for reverse. The complete code for this implementation of queues appears in Figure 3.

Each operation still requires only $O(1)$ amortized time, but we now show that the worst-case time for any particular *remove* has been reduced to $O(\log n)$. Note that *remove* takes $O(1)$ time plus the time to evaluate $tl\ L$. If $tl\ L$ has been evaluated previously, then because of memoization, this evaluation will take only $O(1)$ time. Otherwise, by inspection of *rot*, the tail of L is either of the form A or $rot\,(tl\ L', \ldots)$. Evaluating the former takes $O(1)$ time. Evaluating the latter takes $O(1)$ time plus the time to evaluate $tl\ L'$.[†] Similar arguments hold for evaluating $tl\ L'$. In particular, evaluating $tl\ L'$ might involve evaluating $tl\ L''$ and so on. We must show that no more than a logarithmic number of tails are evaluated by a single removal. However, since *rot* is always called with $|R| = |L| + 1$, we know that if the tail of L is of the form $rot\,(tl\ L', \ldots)$, then $|L'| < |L|/2$. If the tail of L' is also of the form $rot\,(tl\ L'', \ldots)$, then $|L''| < |L'|/2$ and so on. Since the length of the lists is at least halved each step, this pattern cannot repeat more than $\log n$ times.

[†] In evaluating $rot\,(tl\ L', tl\ R', \ldots)$, the evaluation of $tl\ R'$ always takes $O(1)$ time since, by inspection of *insert*, the tail of R' is always of the form R''. In fact, since R never has a non-trivial delayed tail, we could implement it with a strict list instead of a lazy one.

Invariant
$|R| \leq |L|$

$$
\begin{array}{lll}
[]_q & = \langle [], [] \rangle & \\
|\langle L, R \rangle|_q & = |L| + |R| & \\
\mathit{insert}\ (e, \langle L, R \rangle) & = \mathit{makeq}\ \langle L, e : R \rangle & \\
\mathit{remove}\ \langle L, R \rangle & = \langle \mathit{hd}\ L, \mathit{makeq}\ \langle \mathit{tl}\ L, R \rangle \rangle & \\
\mathit{makeq}\ \langle L, R \rangle & = \langle L, R \rangle & \{|R| \leq |L|\} \\
 & = \langle \mathit{rot}\ (L, R, []), [] \rangle & \{|R| = |L| + 1\} \\
\mathit{rot}\ (L, R, A) & = \mathit{hd}\ R : A & \{|L| = 0\} \\
 & = \mathit{hd}\ L : \mathit{rot}\ (\mathit{tl}\ L, \mathit{tl}\ R, \mathit{hd}\ R : A) & \{|L| > 0\}
\end{array}
$$

Fig. 3. Implementation of queues as paired lazy lists. Each operation requires only $O(1)$ amortized time, but any particular *remove* might require $O(\log n)$ time. Note that *makeq* is an auxiliary function that enforces the invariant and that *rot* is always called with $|R| = |L| + 1$.

In addition to improving the worst-case bounds, this implementation of queues also extends the applicability of the amortized bounds, which now hold even when previous versions of the queue may be accessed. For instance, repeated removals from the same queue cause no problems for this implementation – because of memoization, only the first removal might take more than $O(1)$ time. In the next section, we take further advantage of memoization to achieve $O(1)$ worst-case bounds rather than amortized bounds.

5 Queues with pre-evaluation

To ensure that no queue operation takes more than $O(1)$ time in the worst case, we must guarantee that no tail takes more than $O(1)$ time to evaluate. One way to do this is to arrange that whenever there is a tail of the form *rot* (*tl* L,...), the tail of L has already been evaluated and memoized. Since we cannot force the user to call *remove* with sufficient frequency to guarantee this condition, we will *pre-evaluate* L. That is, after each rotation, we will incrementally walk down L, evaluating each tail, so that every tail of L has been evaluated and memoized by the time of the next rotation.

We change the representation of queues to a triple of lazy lists $\langle L, R, \hat{L} \rangle$ where L and R are as before and $\hat{L}$ is some tail of L marking the boundary between the evaluated and unevaluated portions of L. (Think of $\hat{L}$ as a pointer into L rather than as a distinct list.) When $\hat{L} = []$, the entire list has been pre-evaluated. Every call to *insert* and *remove* that does not cause a rotation advances $\hat{L}$ by one position, pre-evaluating the next tail. After a rotation, $\hat{L}$ is set to L. We must guarantee that by the time of the next rotation $\hat{L} = []$. This is easily shown by noting the invariant $|\hat{L}| = |L| - |R|$ so that when $|L| = |R|$, $|\hat{L}| = 0$. In fact, since this occurs just before

Invariants

$$|R| \leq |L| \wedge |\hat{L}| = |L| - |R|$$

$$[]_q = \langle [], [], [] \rangle$$

$$|\langle L, R, \hat{L} \rangle|_q = |L| + |R|$$

$$\mathit{insert}\ (e, \langle L, R, \hat{L} \rangle) = \mathit{makeq}\ \langle L, e : R, \hat{L} \rangle$$

$$\mathit{remove}\ \langle L, R, \hat{L} \rangle = \langle \mathit{hd}\ L, \mathit{makeq}\ \langle \mathit{tl}\ L, R, \hat{L} \rangle \rangle$$

$$\begin{aligned} \mathit{makeq}\ \langle L, R, \hat{L} \rangle &= \langle L, R, \mathit{tl}\ \hat{L} \rangle && \{|\hat{L}| > 0\} \\ &= \text{let } L' = \mathit{rot}\ (L, R, [])\ \text{in } \langle L', [], L' \rangle && \{|\hat{L}| = 0\} \end{aligned}$$

$$\begin{aligned} \mathit{rot}\ (L, R, A) &= \mathit{hd}\ R : A && \{|L| = 0\} \\ &= \mathit{hd}\ L : \mathit{rot}\ (\mathit{tl}\ L, \mathit{tl}\ R, \mathit{hd}\ R : A) && \{|L| > 0\} \end{aligned}$$

Fig. 4. Implementation of queues with pre-evaluation. Each operation requires only $O(1)$ worst-case time. Note that *makeq* is always called with $|\hat{L}| = |L| - |R| + 1$ so that $|R| = |L| + 1$ exactly when $|\hat{L}| = 0$.

the next rotation, we can use the condition $|\hat{L}| = 0$ as the trigger for the rotation. (In practice, this means that we do not need to maintain the lengths of the lists.) This final implementation of queues appears in Figure 4.

6 Double-ended queues

Next we consider an efficient implementation of double-ended queues (deques) with the following specifications:

- $[]_d$ is the empty deque.
- *insertL* (e, Q) returns the new deque obtained by adding e to the left of Q. *insertR* (e, Q) adds e to right of Q.
- *removeL* Q returns the pair $\langle e, Q' \rangle$ consisting of the leftmost element of Q and the new deque obtained by deleting e from Q. *removeR* Q removes the rightmost element of Q. *removeL* Q and *removeR* Q are undefined if Q is empty.
- $|Q|_d$ returns the length of Q.

The implementation of deques closely follows that of queues, differing mainly in the need to treat L and R symmetrically. We represent deques as quadruples of lazy lists $\langle L, R, \hat{L}, \hat{R} \rangle$ where L and R are as before and $\hat{L}$ and $\hat{R}$ are tails of L and R indicating which portions of L and R have been pre-evaluated.

We wish for L and R to be of approximately equal size. In particular, both should be non-empty whenever the deque contains two or more elements. We guarantee this with the invariant

$$|L| \leq c|R| + 1 \wedge |R| \leq c|L| + 1$$

where $c \geq 2$ ($c = 3$ is a typical choice (Hood, 1982; Chuang and Goldberg, 1993)). When the invariant would otherwise be violated, we truncate the longer list to half the combined length of both lists, and rotate the remaining portion onto the back of the shorter list. In the following discussion, assume that R is the longer list. At the beginning of a rotation

$$c|L| + 2 \leq |R| \leq c|L| + c + 1$$

We replace R with $take\,(n, R)$ and L with $L +\!\!+ rev\,(drop\,(n, R))$, where $n = \lfloor(|L| + |R|)/2\rfloor$. As with queues, the trick is to write the latter as an incremental function (*take* is already incremental). We distribute the *rev* and *drop* computations across the append in two phases, the first corresponding to the *drop* and the second corresponding to the *rev*. We process c elements of R for each element of L. At the end of the first phase, we process an extra $n \bmod c$ elements of R, and at the end of the second phase, we process at most $c + 1$ leftover elements of R. If $n \bmod c > 2$ (which can only happen if $c > 3$), we might have fewer than c elements remaining in R when processing the last element of L. This is all made concrete in the following code, where $rot1\,(n, L, R) = L +\!\!+ rev\,(drop\,(n, R))$.

$$\begin{aligned} rot1\,(n, L, R) &= hd\,L : rot1\,(n - c, tl\,L, drop(c, R)) && \{n \geq c\} \\ &= rot2\,(L, drop(n, R), [\,]) && \{n < c\} \end{aligned}$$

$$\begin{aligned} rot2\,(L, R, A) &= hd\,L : rot2\,(tl\,L, drop\,(c, R), \\ &\qquad rev(take(c, R)) +\!\!+ A) && \{|L| > 0 \wedge |R| \geq c\} \\ &= L +\!\!+ rev\,R +\!\!+ A && \{|L| = 0 \vee |R| < c\} \end{aligned}$$

(Note that if $c \leq 3$, *rot2* can be simplified by eliminating the check for $|R| < c$.) Although we make use of the non-incremental versions of *rev* and *drop*, each takes only $O(1)$ time since the size of the arguments is bounded by $c + 1$.

We pre-evaluate each list as before, setting $\hat{L}$ and $\hat{R}$ to L and R after a rotation and advancing each during insertions and removals. However, there is the following complication: if after a rotation the two lists are each of size n, the next rotation could occur after only about $n - n/c$ operations if elements are repeatedly removed from the same side. This is not enough to ensure that the lists are completely pre-evaluated if $\hat{L}$ and $\hat{R}$ are advanced by only one position with each operation. Hence, we advance $\hat{L}$ and $\hat{R}$ by *two* positions for each removal. They need only be advanced by one position for each insertion. The proof that this suffices to completely pre-evaluate the lists relies on the invariant

$$|\hat{L}| \leq \max(2j + 2 - k, 0) \;\wedge\; |\hat{R}| \leq \max(2j + 2 - k, 0)$$

where $j = \min(|L|, |R|)$ and $k = \max(|L|, |R|)$. The bound $2j + 2 - k$ is easily established after a rotation (when $|L|$ and $|R|$ differ by at most one) and is reduced by at most one per insert and by at most two per remove.

Finally, note that when the deque contains only a single element, that element may be stored in either list. If a *removeL* is attempted when the element is stored in the right list (or vice versa), the element must be fetched from that side. Otherwise,

Invariants

$$|L| \leq c|R| + 1 \ \wedge \ |R| \leq c|L| + 1$$
$$|\hat{L}| \leq \max(2j + 2 - k, 0) \ \wedge \ |\hat{R}| \leq \max(2j + 2 - k, 0)$$
where $j = \min(|L|, |R|) \ \wedge \ k = \max(|L|, |R|)$

$$[]_d = \langle [], [], [], [] \rangle$$

$$|\langle L, R, \hat{L}, \hat{R} \rangle|_d = |L| + |R|$$

$$\begin{aligned}
&insertL\,(e, \langle L, R, \hat{L}, \hat{R} \rangle) = makedq\,\langle e : L, R, tl\,\hat{L}, tl\,\hat{R} \rangle \\
&insertR\,(e, \langle L, R, \hat{L}, \hat{R} \rangle) = makedq\,\langle L, e : R, tl\,\hat{L}, tl\,\hat{R} \rangle
\end{aligned}$$

$$\begin{aligned}
removeL\,\langle L, R, \hat{L}, \hat{R} \rangle &= \langle hd\,R, []_d \rangle && \{|L| = 0\} \\
&= \langle hd\,L, makedq\,\langle tl\,L, R, tl\,(tl\,\hat{L}), tl\,(tl\,\hat{R}) \rangle \rangle && \{|L| > 0\} \\
removeR\,\langle L, R, \hat{L}, \hat{R} \rangle &= \langle hd\,L, []_d \rangle && \{|R| = 0\} \\
&= \langle hd\,R, makedq\,\langle L, tl\,R, tl\,(tl\,\hat{L}), tl\,(tl\,\hat{R}) \rangle \rangle && \{|R| > 0\}
\end{aligned}$$

$$\begin{aligned}
makedq\,\langle L, R, \hat{L}, \hat{R} \rangle &= \text{let } n = \lfloor (|L| + |R|)/2 \rfloor && \{|L| > c|R| + 1\} \\
&\quad\ L' = take\,(n, L) \\
&\quad\ R' = rotl\,(n, R, L) \\
&\quad \text{in } \langle L', R', L', R' \rangle \\
&= \text{let } n = \lfloor (|L| + |R|)/2 \rfloor && \{|R| > c|L| + 1\} \\
&\quad\ L' = rotl\,(n, L, R) \\
&\quad\ R' = take\,(n, R) \\
&\quad \text{in } \langle L', R', L', R' \rangle \\
&= \langle L, R, \hat{L}, \hat{R} \rangle && \{\text{otherwise}\}
\end{aligned}$$

$$\begin{aligned}
rotl\,(n, L, R) &= hd\,L : rotl\,(n - c, tl\,L, drop(c, R)) && \{n \geq c\} \\
&= rot2\,(L, drop(n, R), []) && \{n < c\}
\end{aligned}$$

$$\begin{aligned}
rot2\,(L, R, A) &= hd\,L : rot2\,(tl\,L, drop\,(c, R), && \{|L| > 0 \wedge |R| \geq c\} \\
&\qquad\qquad rev(take(c, R)) +\!\!+ A) \\
&= L +\!\!+ revR +\!\!+ A && \{|L| = 0 \vee |R| < c\}
\end{aligned}$$

Fig. 5. Implementation of double-ended queues. Each operation takes only $O(1)$ worst-case time. Note that *makedq* enforces the invariant and that, to avoid special cases when $\hat{L}$ or $\hat{R}$ are [], we assume that $tl\,[] = []$.

removeL and *removeR* always fetch the head of the left and right lists, respectively. The complete code for this implementation of deques appears in Figure 5.

7 Discussion

We have presented purely functional implementations of queues and deques requiring only $O(1)$ time per operation in the worst case. Although not the first such, our implementations are considerably simpler than previous ones.

The standard paired-list implementation of queues has been reinvented many times (Hood and Melville, 1981; Gries, 1981; Burton, 1982). Hoogerwoord (1992)

described a similar implementation of deques. Hood and Melville (1981) gave the first purely functional implementation of queues with $O(1)$ worst-case behaviour. Hood (1982) and later Sarnak (1986), Gajewska and Tarjan (1986), and Chuang and Goldberg (1993) extended this implementation to deques. Many of the techniques used in all these implementations were anticipated by developments in the simulation of multihead Turing machines (Stoß, 1970; Fischer *et al.*, 1972; Leong and Seiferas, 1981).

To see the relative simplicity of our approach, consider the Hood/Melville implementation of queues. (The implementations of deques are similar.) A queue is rotated by first (incrementally) reversing both L and R to get L' and R', and then reversing L' onto R', a total of three reverses. Our implementation corresponds more closely to the single reverse and append that are logically required. Further, in the Hood/Melville implementation, the front elements of L are inaccessible during a rotation since L is reversed. Removing those elements takes special care. In our implementation, no such special treatment is required since the head of a lazy list is available even when 'work' is being performed on some tail of the list.

Some might argue that our implementation is not, in fact, purely functional because of the imperative nature of memoization.‡ However, we feel that this use of memoization is the same as in lazy functional languages – memoization or the lack thereof has no effect on the *correctness* of the algorithms, merely the *efficiency*.

This paper presents a rather unusual use of laziness. Laziness is typically used when one either does not care when a computation takes place or wishes to delay it to the last possible instant. Here, however, we care very much when the computation takes place and we cannot afford to delay it too long. We use pre-evaluation to schedule the computation at precisely the time that we want. These combined techniques of laziness and pre-evaluation may be applicable to other data structures. In the future, we intend to investigate their use in converting other data structures with good amortized time bounds into ones with good worst-case bounds, or in simplifying other incremental algorithms.

Acknowledgements

Thanks to Peter Lee, Mark Leone, Amy Moormann Zaremski and Maria Ebling for comments and suggestions. Thanks also to an anonymous referee for suggesting the second invariant in Section 6.

References

Abelson, H. and Sussman, G. J. (1985) *Structure and Interpretation of Computer Programs.* MIT Press.

Burton, F. W. (1982) An efficient functional implementation of FIFO queues. *Information Processing Letters*, **14**(5):205–206, July.

‡ This question does not arise for the Hood/Melville implementation or its successors since they use strict lists.

Chuang, T.-R. and Goldberg, B. (1993) Real-time deques, multihead Turing machines, and purely functional programming. In: *Proceedings of the Conference on Functional Programming and Computer Architecture, Copenhagen*, pp. 289–298.

Fischer, P. C., Meyer, A. R. and Rosenberg, A. L. (1972) Real-time simulation of multihead tape units. *Journal of the ACM*, **19**(4):590–607, October.

Gajewska, H. and Tarjan, R. E. (1986) Deques with heap order. *Information Processing Letters*, **22**(4):197–200, April.

Gries, D. (1981) *The Science of Programming*. Springer-Verlag.

Hood, R. and Melville, R. (1981) Real-time queue operations in pure Lisp. *Information Processing Letters*, **13**(2):50–53, November.

Hood, R. (1982) *The efficient implementation of very-high-level programming language constructs*. PhD thesis, Department of Computer Science, Cornell University.

Hoogerwoord, R. R. (1992) A symmetric set of efficient list operations. *Journal of Functional Programming*, **2**(4):505–513, October.

Leong, B. L. and Seiferas, J. I. (1981) New real-time simulations of multihead tape units. *Journal of the ACM*, **28**(1):166–180, January.

Sarnak, N. (1986) *Persistent Data Structures*. PhD thesis, Department of Computer Sciences, New York University.

Stoß, H.-J. (1970) K-band simulation von k-Kopf-Turing-maschinen. *Computing*, **6**(3):309–317.