Real Time Queue Operations

in Pure LISP*

by

Robert Hood
Robert Melville

TR80-433

Department of Computer Science
Cornell University
Ithaca, New York 14853

*This work has been supported in part by National Science Foundation grants MCS 78-05850 and MCS 76-22360

Real-Time Queue Operations in Pure LISP*

Robert Hood
Robert Melville

Department of Computer Science
Cornell University
Ithaca, New York 14853

ABSTRACT

Several methods of implementing a queue in Pure LISP are presented. A technique to distribute the reversal of a list over a number of operations leads to a real-time queue implementation.

1. Introduction

In Pure LISP, the inability to access the end of a list in constant time increases the asymptotic complexity of some algorithms. It is impossible, for example, to **append** one list to another one without recopying the first. If the end of the first list could be accessed (and modified) then the **append** operation would take constant time rather than time proportional to its length. This restriction on access is of particular significance when implementing a queue. By its very nature a queue requires the ability to access both ends of a list.

2. Queue Operations

2.1. The Straightforward Version

We implement three queue operations: Query, Delete, and Insert. Query[q] returns the element at the front of queue q. Delete[q] returns q with its front element removed. Insert[q, v] returns the queue formed by adding the element v to the end of queue q. A straightforward implementation of these operations is given in Figure 1.

If the head of the queue is kept at the front of a list, as is done in our implementation, then the Delete and Query operations can be performed in constant time. The Insert operation, however, requires the recopying of the queue, and therefore takes time proportional to its length.

```
Query[q]     ≡ car[q]
Delete[q]    ≡ cdr[q]
Insert[q, v] ≡ append[q, list[v]]
```

Figure 1: Queue operations in Pure LISP[1]

2.2. An Asymptotic Improvement

A fairly simple modification allows constant time access to both ends of the queue in most circumstances. Rather than being stored as a single list (say $(q_0 \ldots q_n)$) the queue is kept as a head list and a tail list, with the tail kept in reverse order (e.g. $(q_0\ q_1 \ldots q_{i-1}\ q_i)$ and $(q_n\ q_{n-1} \ldots q_{i+2}\ q_{i+1})$.)

Using this representation, the Insert operation can be performed in constant time since it is sufficient to **cons** an element onto the tail list. We allow the head list to be empty only if the entire queue is empty; thus, the Query operation can be performed in constant time. The Delete operation requires constant time if deleting the front element does not invalidate our invariant of a non-empty head list. When Delete is called and there is only one element left in the head list, it is necessary to replace it by the Reverse of the tail list. Figure 2 shows operations that manipulate queues represented this way.

This representation reduces the cost of n operations from $O(n^2)$ to $O(n)$. This can be easily seen from the fact that the reversal of a list of length k takes place only when there have been k Inserts since the last reversal operation. Since the reversal of the list of length k can be done in $O(k)$ steps, then we are doing $O(k)$ work to reverse only after having done $O(k)$ work to Insert the elements being reversed. Since the other operations require only constant time, n queue operations can be done in $O(n)$ time.

*This work has been supported in part by National Science Foundation grants MCS 78-05850 and MCS 76-22360.

[1]Note that a blackboard dialect of LISP is used.

Suppose a queue is (H . T) where H is the head list and T the tail (in reverse order).

```
Query[q] ≡ car[car[q]] --return car of H

Insert[q, v] ≡
   cons[
        car[q],
        cons[v, cdr[q]]
       ] --tack v onto T

Delete[q] ≡
   if null[cdr[car[q]]] --one left in H
      then
        cons[
             Reverse[cdr[q]],
             NIL
            ] --replace H by T̃
      else
        cons[
             cdr[car[q]],
             cdr[q]
            ] --delete first element
   fi
```

Figure 2: Better queue operations

2.3. Real-Time Operations

2.3.1. The Basic Idea

By distributing the Reverse of the tail list over a number of operations, we can perform the queue operations in real time.[2] The basic idea of performing one step of the Reverse during every operation is easy to implement, if the list being reversed is not changing. For example, if L is of length n, then

```
incr_rev[incr_rev[ ... incr_rev[cons[L, NIL]] ... ]]
|←              n times              →|
```

will return with (NIL . $\overleftarrow{L}$), where $\overleftarrow{L}$ is the reverse of L, and incr_rev is defined as follows:

```
incr_rev[X] ≡ cons[
                   cdr[car[X]],
                   cons[
                        car[car[X]],
                        cdr[X]
                       ]
                  ] .
```

Note that any one call to incr_rev is performed in constant time.

Rather than waiting for the head list H to empty, we periodically Reverse the tail list T and **append** it to the head list, forming a new head list H'. This is a three-step process:

(1) Reverse T forming the tail end of H'
(2) Reverse H forming H_R
(3) Reverse H_R onto the front of H'

Each of these operations is just the reversing of a list and can be done using a function like incr_rev. The first two steps are independent and can be done in parallel. Thus, if at the start of the reversal process we have:

$$\begin{array}{ll} \text{front} & \text{rear} \\ q_0 \cdots q_m & q_{m+1} \cdots q_n \\ \text{--\{ H \}--} & \text{---\{ }\overleftarrow{T}\text{ \}---} \end{array}$$

(where $\overleftarrow{T}$ indicates the Reverse of the list T[3]) then halfway through the above process we have:

$$\begin{array}{ll} \text{front} & \text{rear} \\ q_0 \cdots q_m & q_{m+1} \cdots q_n \\ \text{--\{}\overleftarrow{H_R}\text{\}--} & \text{---\{H'\}--} \end{array}$$

Since q_m is at the front of H_R and q_{m+1} is at the front of H' then **cons**(**car**(H_R), H') will produce ($q_m\ q_{m+1} \cdots q_n$). After m+1 **cons** operations we have:

$$\begin{array}{ll} \text{front} & \text{rear} \\ q_0 \cdots q_m & q_{m+1} \cdots q_n \\ \multicolumn{2}{c}{\text{--------\{H'\}--------}} \end{array}$$

This is the basic idea behind implementing the real-time operations. One minor hitch remains, however: while performing the incremental reverses the queue will not remain constant. Our representation needs to reflect the current state of the queue as well as the current state of the recopying operation.

Taking care of the Inserts that occur while recopying is simple. A new tail list T' is used, and the new elements Cons'ed onto it instead of the old tail T. In order to be able to satisfy Query and Delete requests, there must be two copies of the head list. One head list (h) represents the front end of the queue. Deletes will be performed by replacing h by **cdr**(h), and Query's performed by returning **car**(h). The other head list (H) is used in the reversal process to form H_R. Since there may have been Deletes, not all of H_R should be reversed onto H'. A count, #copy, of the length of h is kept to indicate how much of H_R needs to be copied.

[2] A real-time queue implementation performs any sequence of Inserts, Deletes, and Querys with only a constant amount of processing between operations; note that this is stronger than linear time. In the case of LISP, we assume the existence of real-time **car**, **cdr**, and **cons** primitives [1].

[3] Note that the list H can have its leftmost elements accessed while a reversed list such as T has its rightmost elements accessible.

```
q0 ... q_{d1-1} q_{d1} ... q_j q_{j+1} ... q_{n-1} q_n ... q_k q_{k+1} ... q_2n q_{2n+1} ... q_{2n+a1}
--------{ H_R }--------  ----{ H }----  --{ T }-- ---{ H' }---  -----{ T' }-----
|<- 2 x #operations ->|                 |<- 2 x #ops ->|
-{deletes}- ------------------------{ the queue }-----------------------
            ---------{ h }---------
```

(a) During the first pass of recopying

```
q0 ... q_{d1-1} q_{d1} ... q_{n-1} q_n ... q_2n q_{2n+1} ... q_{2n+a1}
---------{ H_R }----------  --{ H' }--  -----{ T' }-----
-{deletes}-  -------------{ the queue }--------------
             ---{ h }---                     H = T = ()
```

(b) After the first pass of recopying

```
q0 ... q_{d1+d2-1} q_{d1+d2} ... q_i q_{i+1} ... q_{n-1} q_n ... q_2n q_{2n+1} ... q_{2n+a1+a2}
-----------{ H_R }-----------  ---------{ H' }--------  ------{ T' }-------
                 |<- #copy ->|
--{ deletes }-- ---------------------{ the queue }---------------------
                ----------{ h }----------                      H = T = ()
```

(c) During the second pass of recopying

```
q0 ... q_{d1+d2-1} q_{d1+d2} ... q_{n-1} q_n ... q_2n q_{2n+1} ... q_{2n+a1+a2}
-----{ H_R }----  ----------{ H' }----------  ------{ T' }-------
--{ deletes }--  ----------------{ the queue }----------------
                 -----{ h }-----                     H = T = ()
```

(d) After the recopying is finished

Figure 3: The queue at various stages of the recopy process

2.3.2. The Recopy Strategy

Along with the recopy process we need a strategy to indicate when recopying should be done. We choose to start recopying when the length of the tail list T is greater than the length of the head list H. It will be necessary to show that the recopying operation leaves us with $|H'| \geq |T'|$. When the queue is not in "recopy mode", it will behave like the queue of §2.2.

Using the above strategy the recopy operation will start when the tail list is say, n+1 elements long and the head list is of length n. The simultaneous reversing of H and T in the first pass of the recopying will take n+1 incremental steps to perform. The reversing of H_R in the second pass will take n steps to perform. Thus there are a total of 2n+1 steps in the recopying process. The implementation will get into trouble if the head list h is emptied before the recopying is completed. Since h is n elements long, the recopy process must perform 2n+1 incremental steps in at most n queue operations. This will be accomplished by performing the first two steps of the recopy process in the operation that causes T to be longer than H, and two more steps of the process during every subsequent queue operation. This gives us $2 \times (n+1)$ incremental steps before the head list will be emptied. This is sufficient to ensure that the recopy process will be completed in time.

2.3.3. The Recopy Process

When the queue's tail list becomes longer than its head list, as below, the recopying process begins.

$$\begin{array}{cc} \text{front} & \text{rear} \\ q_0 \cdots q_{n-1} & q_n \cdots q_{2n} \\ \text{---\{ H \}---} & \text{---\{ } \overset{+}{T} \text{ \}--} \end{array}$$

After a_1 additions to the queue and d_1 deletions (where $a_1+d_1 < 1/2n$) the queue will be in a state as shown in Figure 3a. H is being reversed forming H_R and T is being reversed forming H'. New elements are Cons'ed to T' and deleted elements are removed from h. Also taking place during this pass is the counting of the length of H' (in lendiff) as it is being formed, and the length of H_R (in ≠copy) as well.

When $a_1+d_1=1/2n$ the queue will have finished the first pass of the recopy operation and will be as pictured in Figure 3b. After $a_2 + d_2$ more additions and deletions, the queue is in the midst of the second pass of recopying and is in a state as shown in Figure 3c. Stored in ≠copy is a count of the number of elements of H_R that have not been deleted from the queue. H_R is being reversed onto H' until #copy = 0. At that time H' and T' can replace H and T as the representation of the queue, as shown shown in Figure 3d. Since the new head list is of length $2n+1-d_1-d_2$ and the new tail list is of length a_1+a_2 and $d_1+d_2+a_1+a_2 = n$ then:

$$\begin{array}{rr} & 2n+1 \geq n \\ & 2n+1 \geq a_1+a_2+d_1+d_2 \\ & 2n+1-d_1-d_2 \geq a_1+a_2 \\ \text{Thus,} & |H'| \geq |T'| \end{array}$$

which we needed to show to prove that our recopying strategy was a valid one.

3. Conclusion

The linear-time algorithm of §2.2 is similar to a Turing Machine construction due to Stoss [5]. Since a one-tape TM can be simulated in real time in Pure LISP, the work of Leong and Seiferas [4] implies the existence of a real-time queue implementation. Their work, which is much more general, leads to a queue implementation which is significantly more complicated than the one presented here, however.

It is believed that all linear time LISP functions can be done incrementally; this could lead to many real-time LISP algorithms. It would be interesting to exhibit a problem for which the lower bound in Pure LISP is worse than some implementation using **rplaca** and **rplacd**.

4. References

[1] Baker, H. List Processing in Real Time on a Serial Computer. Comm. ACM 21, 4 (April 1978), 280-294.

[2] Dijkstra, E. A Discipline of Programming. Prentice-Hall, 1976.

[3] Knuth, D. The Art of Computer Programming, Vol. I. Addison-Wesley, 1973.

[4] Leong, B. and Seiferas, J. New Real-time Simulations of Multi-head Tape Units. Proceedings of the Ninth Annual Symposium on Theory of Computing, Boulder, 1977, 239-248.

[5] Stoss, H-J. K-band-Simulation von k-Kopf-Turing-Maschinen. Computing 6, 309-317 (1970).

5. Appendix--The Real-Time Queue Implementation

A queue is a nine-element list: **list**[recopy, lendiff, #copy, H, T, h, H', T', H_R], where we have given symbolic names to the components. For example, **car**[**car**[q]] is lendiff of queue q. An empty queue has value: **list**[false, 0, 0, NIL, NIL, NIL, NIL, NIL, NIL].

```
Insert[q, v]≡
  if ¬recopy ∧ lendiff > 0 → list[False, lendiff-1, 0, H, cons[v, T], NIL, NIL, NIL, NIL]
   □ ¬recopy ∧ lendiff = 0 → Onestep[Onestep[True, 0, 0, H, cons[v, T], H, NIL, NIL, NIL]]
   □  recopy               → Onestep[Onestep[True, lendiff-1, #copy, H, T, h, H', cons[v, T], H_R]]
  fi

Delete[q]≡
  if ¬recopy ∧ lendiff > 0 → list[False, lendiff-1, 0, cdr[H], T, NIL, NIL, NIL, NIL]
   □ ¬recopy ∧ lendiff = 0 → Onestep[Onestep[True, 0, 0, cdr[H], T, cdr[H], NIL, NIL, NIL]]
   □  recopy               → Onestep[Onestep[True, lendiff-1, #copy, H, T, cdr[h], H', T', H_R]]
  fi

Query[q]≡
  if ¬recopy → car[H]
   □  recopy → car[h]
  fi

Onestep[q]≡
  if ¬recopy → q
   □ recopy ∧ ¬null[H] ∧ ¬null[T]                →
        list[True, lendiff+1, #copy+1, cdr[H], cdr[T], h, cons[car[T], H'], T', cons[car[H], H_R]]

   □ recopy ∧  null[H] ∧ ¬null[T]                →
        list[True, lendiff+1, #copy, NIL, NIL, h, cons[car[T], H'], T', H_R]

   □ recopy ∧  null[H] ∧  null[T] ∧ #copy > 1    →
        list[True, lendiff+1, #copy-1, NIL, NIL, h, cons[car[H_R], H'], T', cdr[H_R]]

   □ recopy ∧  null[H] ∧  null[T] ∧ #copy = 1    →
        list[False, lendiff+1, 0, cons[car[H_R], H'], T', NIL, NIL, NIL, NIL]
  fi
```

Note that the only arithmetic operations performed are: add 1, subtract 1, and test for 0 or 1. They can be performed by appropriate list functions if we encode integers in unary notation. (The integer n is a list of n atoms.)