Pre-publication version! Appears in: Schrijvers and Thiemann (Eds), *Symposium on Functional and Logic Programming FLOPS 2012, LNCS 7294*, pp. 62–72, 2012, Springer. The original publication is available at `www.springerlink.com`

Real-time Persistent Queues and Deques with Logic Variables (Declarative Pearl)

Gerlof Bouma

Spraakbanken, Department of Swedish, University of Gothenburg
Box 200, 405 30 Gothenburg, Sweden
`gerlof.bouma@gu.se`

Abstract. We present a Prolog implementation of real-time persistent queues and double-ended queues. Our implementation is inspired by Okasaki's lazy-functional approach, but relies only on standard Prolog, comprising of the pure subset plus if-then-else constructs to efficiently implement guards and meta-calls for convenience. The resulting data structure is a nice demonstration of the fact that the use of logic variables to hold the outcome of an unfinished computation can sometimes give the same kind of elegant and compact solutions as lazy evaluation.

Keywords: Queues, deques, persistence, Prolog, logic variables, pearl.

1 Introduction

The well-known, classic way to implement efficient queues in a language without mutable data structures is to use a pair of lists $\langle F_i, R_i \rangle$, where F_i holds the front and R_i the rear in reversed order for version i of the queue. Adding to the queue (henceforth: *inject*) is prepending to R_i, which takes constant time. Removal (*pop*) generally takes constant time as well, as we simply take the first element from F_i. A linear time pop occurs when $|F_i| = 1$, which triggers reversal of R_i in order to use it as F_{i+1}. Since the linear time pop costing $k = |R_i|$ steps occurs after k injections, amortized time complexity of pop is constant ([1], etc.).

Also well-known is that this amortized constant time implementation of queues is not well-suited for persistent use. If the same version i of a queue is accessed multiple times in the course of programme execution, the linear time pop may occur somewhere down the road for each of these uses. As O'Keefe [2] points out, this aspect of amortized queues can become particularly nasty in a language like Prolog, where intensive backtracking can make repeated use of the same version of a data structure the rule rather than the exception.

A *real-time persistent* implementation avoids these problems. Following Hood & Melville [1], we call data structures real-time when all basic operations have worst-case constant time complexity. For a queue, these operations are *empty* (testing for or creating the empty queue), *pop* and *inject*. Furthermore, a data structure is (fully) persistent if, after an operation, the old version of the data structure is accessible for further operations with time complexity as good as that of operations on the most current version. Thus, if we have version i of a real-time persistent queue, we can perform empty, pop or inject on i as well as on any previous version $i-1, i-2, \ldots$ in constant time. We will call a data structure that allows access to previous versions with degraded efficiency or functionality *partially* persistent. The canonical Prolog implementation of a queue as an open difference list is an example of a partially persistent real-time data structure, as inject restricts a version's reusability.

An early implementation of a real-time persistent queue is the *Hood-Melville queue* [1]. Here, worst-case constant time is achieved by performing a little bit of the reversal of R at each operation ahead of time, so that the result is ready when it is needed. Okasaki [3] later showed that the use of lazy evaluation and memoization leads to a particularly elegant implementation of this scheduled reversal approach and applies his technique to queues and double-ended queues. Our paper is concerned with the implementation of real-time persistent (double-ended) queues in Prolog. Although we could give a translation of Okasaki's proposal with the help of delayed evaluation (`freeze/2`, `block/N` directives, etc.), these techniques go beyond basic Prolog, and may be associated with portability and performance issues. We will show that the availability of logic variables alone is enough to capture most of the simplicity of Okasaki's proposal. We consider the resulting implementations to be declarative pearls. The implementations use completely standard Prolog with very little reliance on side-effects. To be precise: we use the pure subset with addition of arithmetic, the if-then-else construct to allow for an efficient implementation of guards, and `call/2` as an inconsequential but convenient bridge between data and code.

We begin by briefly reviewing Hood-Melville queues and Okasaki's lazy implementation. We then demonstrate how logic variables are enough to achieve the same kind of simplifications as with lazy evaluation. After that, we give an extension to real-time persistent double-ended queues. Discussion and a brief comparison to the canonical Prolog queue ends the paper.

2 Real-time queues

The real-time and fully persistent *Hood-Melville queues* are based on the pair-of-lists idea, but achieve worst-case constant time for all operations by making sure that reversing the rear list R is done before its reverse R' is needed. In addition, R' is appended to the front list F, instead of R' replacing F when the latter is empty. We shall use the term *rotation* to refer to the combined reverse and append. We need to schedule a bit of rotation for each operation on the queue. The intermediate results are passed around as part of the queue data structure, which makes using tail-recursive versions of reverse and append crucial if we want to maintain constant time for each update of the rotation. A full rotation is thus characterized by the following three stages:

1. Reverse R to give R' ($|R|$ steps)
2. Reverse F to give F' ($|F|$ steps)
3. Reverse F' onto R' to give FR' ($|F| - \#\mathit{pops}$ steps).

$$\begin{array}{lll}
\mathit{inject}(x, \langle F, R, F_{\text{ev}}\rangle) & = \mathit{makeq}(F, [x|R], F_{\text{ev}}) & \\[1ex]
\mathit{pop}(\langle F, R, F_{\text{ev}}\rangle) & = \langle x, \mathit{makeq}(F', R, F_{\text{ev}})\rangle, \text{ where } F = [x|F'] & (\text{when } F \neq [\,]) \\[1ex]
\mathit{makeq}(F, R, F_{\text{ev}}) & = \langle F, R, \mathit{tail}(F_{\text{ev}})\rangle & (\text{when } F_{\text{ev}} \neq [\,]) \\
 & = \langle F'_{\text{ev}}, [\,], F'_{\text{ev}}\rangle, \text{ where } F'_{\text{ev}} = \mathit{rotate}(F, R, [\,]) & (\text{when } F_{\text{ev}} = [\,]) \\[1ex]
\mathit{rotate}(F, R, A) & = [\mathit{head}(R)|A] & (\text{when } F = [\,]) \\
 & = [\mathit{head}(F)|\mathit{rotate}(\mathit{tail}(F), \mathit{tail}(R), [\mathit{head}(R)|A])] & (\text{when } F \neq [\,])
\end{array}$$

Fig. 1. Okasaki queues [3]; real-time and persistent on the assumption that the function *rotate*, which yields the new front list, is lazily evaluated.

Stages (1) and (2) can be executed in parallel, stage (3) needs to follow the others. We will leave out details of deciding when to start a new rotation and how many updates per operation are needed. We do, however, note that the implementation of Hood-Melville queues is complicated by the fact that the front of F as used in the rotation is not available for popping. A version of F at the start of the rotation is kept around to be used for this purpose and the rotation has to be completed before this list is exhausted. However, by the time the rotation is finished, *its* version of F may be outdated, as it may contain items that where popped during the rotation. Book-keeping of *#pops* is needed to ensure that the new front list does not contain such illicit items.

As mentioned in the introduction, Okasaki [3] implements the scheduled rotation idea using lazy evaluation. *Okasaki queues* are triples $\langle F, R, F_{\text{ev}}\rangle$, where F is a lazy list representing the front of the queue and F_{ev} is the unevaluated tail of F. Unlike in Hood-Melville queues, a representation of intermediate results of the rotation computation is not itself part of the data structure. Rather, rotations are independent, delayed computations, whose evaluation is forced stepwise by traversing F_{ev}. We might say that only the synchronization device is part of the data structure. Pseudo-code for Okasaki queues can be found in Fig. 1.

The implementation is kept minimal by the fact that there is no need to maintain a separate F accessible for pops during a rotation and therefore no need either to keep track of the number of pops. Memoization ensures that a front list is only constructed once by a rotation, even when the same version of a queue is reused.

The functions *pop* and *inject* do not directly return a queue, but rely on the auxiliary function *make_q* to assemble a new queue triple given the possibly new F and R, and the old F_{ev}. The function *make_q* either updates an ongoing rotation (the first case) or starts a new one (the second case). In any valid Okasaki queue we have invariants $|F_i| \geq |R_i|$ and $|F_{\text{ev}\,i}| = |F_i| - |R_i|$. Per the latter, *make_q* is called with $|F_{\text{ev}}| = |F| - |R| + 1$. The former invariant is thus threatened when $|F_{\text{ev}}| = 0$ and a new rotation is needed to save it. So, a new rotation is started as soon as the previous one is finished. The decision to do so can be made without explicitly tracking $|F|$ or $|R|$.

Because of lazy evaluation, Okasaki can a) control the scheduled execution of a rotation without having to pass on a representation of the rotation itself and b) incrementally append two lists without an additional reversal of the front list. The first is possible because the rotation updates are forced implicitly in lazy evaluation, the second because one can carry around the outcome of an unfinished computation like any other outcome. In Prolog, *logic variables* give us this second capability. Consider the canonical `append/3` implementation, which is tail-recursive (in Prolog) and incrementally constructs its result:

```
append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]):-
    append(Xs,Ys,Zs).
```

Procedurally, in a call with the first two arguments instantiated and the third one unbound, the second clause creates an open copy of its first argument in its third argument. At the end of the computation, the first clause instantiates the open copy's tail with the second argument.

The open copy thus becomes more and more filled in for each recursion, but is passed around as if it were completely filled in from the start. Moreover, and this is crucial for our purposes, we have direct access to this incremental result through the binding of the third argument at the top level. By returning the recursive call rather than executing it, we can also carry around a pointer to the remainder of the computation after each step at the top level.

```
append([],Xs,Xs,done).
append([X|Xs],Ys,[X|Zs],append(Xs,Ys,Zs)).
```

We can now evaluate, say, `append([1,2],[3,4,5],Zs)` by composing three calls to `append/4`, which completely instantiates `Zs` and returns the dummy state `done`. The meta-predicate `call/2` offers a convenient way of implementing this composition.[1] Evaluating `append([1,2],[3,4,5],Zs)` using `append/4` and `call/2` is then:

```
call(append([1,2],[3,4,5],Zs),A1), call(A1,A2), call(A2,A3)
```

which results in the bindings (written to reflect the incremental binding of `Zs`):

```
Zs=[1|Zs1],  A1=append([2],[3,4,5],Zs1),
Zs1=[2|Zs2], A2=append([],[3,4,5],Zs2),
Zs2=[3,4,5], A3=done.
```

This way, logic variables present a third opportunity to implement scheduled rotation, situated between Okasaki's implicit, lazy evaluation and Hood-Melville's explicit, tail-recursive 'reverse, reverse-onto'. On the one hand, we will use the (future) outcome of a rotation directly as the front list, but on the other, we *do* carry around an explicit representation of a rotation. Concretely, the rotation schedule will look like this:

[1] The relation `call/2` holds between `p(A1,...,An)` and `Am` iff `p(A1,...,An,Am)` holds. It is offered as a built-in in many current Prologs. Note that if all possible functor/arity combinations of the first argument are known in advance, `call/2` is purely a convenience predicate and its occurrences can be spelled out into first-order, user-defined predicates.

1. Reverse `Rs` to give `Rs_rev` (`Rs_len` steps)
2. Open `Fs` to give `Fs_open` resp. `Fs_tail` (`Fs_len` steps)
3. `Fs_tail = Rs_rev` to give the concatenation of `Fs` and `Rs_rev`.

Stage (2) and (3) are to be compared to the append definition above. As before, stage (1) and (2) will be executed simultaneously. `F_open` is the new, partially instantiated front list. Because the front of such a list behaves like any other list, we can use it as the front list immediately. Unlike Hood & Melville, we do not need to book-keep pops during a rotation.

To guarantee real-time behaviour, we have to ensure that each rotation update can be performed in constant time. However, if we are not careful, the one unification step carried out in stage (3) may incur linear time in the context of persistence. There are two cases to consider. First, the first time a rotation reaches this point, `Fs_tail` is unbound and unification is constant time. Secondly, in subsequent completions of the *same* rotation, caused by reuse of one version of a queue, `Fs_tail` will already be bound. Although unification at this point is guaranteed to succeed, we may take linear time to redundantly check this if `Rs_rev` is rebuilt each time. Luckily, we can avoid the linear time case by making sure that `Rs_rev` is not only structurally identical (i.e., unifiable) between completions, but also referentially so. Checking `A=A` is a constant time operation irrespective of the binding of `A`. Stage (1) is therefore implemented as analyzing a list that contains the intermediate results of an accumulator-based reversal. The first item on this list is the empty list, the last item is `Rs_rev` itself. Logic variables again are of great help here, as constructing the list (the first case) and traversing it (the second) need not be distinguished in the implementation.

Rotations are represented by the following data structure:[2]

```
:- type state(T) ---> ready
                      %      Rs_revs
                      ; wait(list(T))
                      %     Fs      Fs_open Rs      Rs_revs
                      ; rot(list(T),list(T),list(T),list(list(T))).
```

The rotation itself is performed when rotation state is of the form `rot/4`, which implements stages (1)–(3). The state `wait/1` is used to count down to the next rotation, which will be started at `ready/0`.[3] Rotation state updates are handled by predicates with the same functor as the rotation state, but with an extra argument that holds the remainder of the rotation (cf the use of `append/4` to implement `append/3`, above).

```
% wait/2     +Rs_rev -State
:- pred(wait(list(T),state(T))).
wait([],ready).
wait([_|Rs_rev],wait(Rs_rev)).
```

[2] We use the syntax of the Hindley-Milner typing library for (Yap and SWI) Prolog described in [4].

[3] Since we start the next rotation at `ready` and not `wait([])`, we start our waiting counter with $|R| - 1$.

```
% rot/5     +Fs     ?Fs_tail +Rs     ?Rs_revs      -State
:- pred(rot(list(T),list(T), list(T),list(list(T)),state(T))).
rot([],[R|Rs_rev],[R],[Rs_rev],wait(Rs_rev)).
rot([F|Fs],[F|Fs_tail],[R|Rs],[Rs_rev|Rs_revs],
                                        rot(Fs,Fs_tail,Rs,Rs_revs)):-
    Rs_revs = [[R|Rs_rev]|_].
```

As before, for a valid queue, $|F_i| \geq |R_i|$ holds. A rotation starts with $|R| = |F|+1$. By performing one update at the start of a rotation, plus one for each operation on the queue, we arrive at `ready/0` exactly when $|F_i| = |R_i|$. A new rotation will be started at the next operation. The only time `Fs` is empty is when the queue is empty. Fig. 2 contains specification of the queue data structure, implementation of the basic queue operations and the auxiliary predicate that maintains the queue invariant.

Without presenting a formal proof, we reason our implementation has the following time complexity properties with respect to the length of the queue. Throughout we assume that input arguments are sufficiently instantiated and output arguments are unbound at call time. A rotation update then takes constant time because only a fixed portion of each list in a rotation state is (de)composed per update and no unification of whole lists is involved, except for in the first clause of the `rot/5` predicate (stage 3), where we avoid a linear time hit because of the referential identity of the lists. Queue assembly in `make_q/4` calls one constant time rotation update and only unifies the rotation state lists with unbound variables, and is thus constant time as a whole. The operations *inject* and *pop* cost a constant time call to `make_q` plus the time taken to (de)compose the top cell of the front and/or rear list, which takes constant time. Finally, *empty* involves unification of a queue with a fixed size term representing the empty queue and is therefore constant time. No operation is nondeterministic. None of the operations destroys a previous version of the queue although they may fill in a little bit of the deterministic future of a rotation, which does not affect future operations on the same version because these will always result in the exact same update. Our queue thus qualifies as a real-time persistent queue.

The implementation in Fig. 2 uses if-then-else and `call/2` for clarity and consistency of presentation between this section and the next. However, the deterministic choice in `make_q` can also be implemented through Prolog's standard first-argument indexing, because all functor/arities of the `State` argument are known. Transforming the predicate in this way and compiling away `call/2` – possible for the same reason – results in the pure, first-order Prolog implementation of `make_q` given in Fig. 3. This alternative implementation of the invariant maintaining code can be used as a drop-in replacement, modulo the order of arguments of the predicate.

3 Real-time persistent deques

Queues provide removal from one end of the queue and addition to the other. Double-ended queues, deques for short, allow addition and removal at either

```
:- type queue(T) ---> q(list(T),list(T),state(T)).


% empty_queue/1     ?Q
:- pred empty_queue(queue(T)).
empty_queue(q([],[],ready)).


% pop_queue/3     +Q       ?El -Q1
:- pred pop_queue(queue(T),T,  queue(T)).
pop_queue(q([F|Fs],Rs,St),F,Q1):-
    make_q(Fs,Rs,St,Q1).


% inject_queue/2     ?El +Q       -Q1
:- pred inject_queue(T,  queue(T),queue(T)).
inject_queue(R,q(Fs,Rs,St),Q1):-
    make_q(Fs,[R|Rs],St,Q1).


% make_q/4     +Fs     +Rs     +State  -Q1
:- pred make_q(list(T),list(T),state(T),queue(T)).
make_q(Fs,Rs,St,Q):-
    ( St == ready
      -> call(rot(Fs,Fs1,Rs,[[]|_]),St1),
         Q = q(Fs1,[],St1)

    ; call(St,St1),
      Q = q(Fs,Rs,St1)
    ).
```

Fig. 2. Basic queue operations and invariant maintaining code.

```
% make_q/4     +State, +Fs     +Rs     -Q1
:- pred make_q(state(T),list(T),list(T),queue(T)).
make_q(ready,Fs,Rs,q(Fs1,[],St1)):-
    rot(Fs,Fs1,Rs,[[]|_],St1).
make_q(wait(A),Fs,Rs,q(Fs,Rs,St1)):-
    wait(A,St1).
make_q(rot(A1,A2,A3,A4),Fs,Rs,q(Fs,Rs,St1)):-
    rot(A1,A2,A3,A4,St1).
```

Fig. 3. Queue invariant maintaining code in pure, first-order Prolog.

end. We call *pop* and *inject*'s mirror operations *push* and *eject*.[4] The scheduled rotation technique used above can also be employed to implement constant time eject and push. Such extensions, or descriptions thereof, can be found in [5, 6, 3].

Because of the symmetry of the operations, it does not do to guarantee $|F_i| \geq |R_i|$. A deque requires a different invariant. In addition, protecting the invariant may involve moving only some elements of one list to the other. We thus need a different rotation scheme. Instead of just constraining $|R_i|$ by $|F_i|$, we keep the two lists balanced with respect to each other. In particular, for deques with more than one item, we maintain the invariant $|L_i| \leq 3 \cdot |S_i|$,[5] where L and S are the longer and shorter list, respectively. When the invariant is threatened, we rebalance the lists by moving approximately a third of L to S. A full rotation then comprises the following stages:

1. Open `Ss` to give `Ss_open` resp. `Ss_tail` (`Ss_len` steps),
2. split `Ls` into `Lks`, to keep, and `Lgs`, to give away (`Ss_len` double steps),
3. reverse `Lgs` to `Lgs_rev` (`Lgs_len` = `Ss_len` steps),
4. `Ss_tail = Lgs_rev`.

In a deque, we cannot easily predict when the invariant will be threatened again, so instead of counting down to the start of the next rotation, we keep track of the length of $|F|$ and $|R|$ explicitly. The rotation state data structure contains two active states, corresponding to stages (1)–(2) and (3)–(4), respectively, and a ready state that does nothing indefinitely. Note that the `rot/5` state has a slot for `Lgs_revs`, but this is only to pass it on to the `rot/3` state, where this list is analyzed.

```
:- type state(T) ---> ready
                    %     Lgs      Lgs_revs      Ss_tail
                    ; rot(list(T),list(list(T)),list(T))

                    %     Ss       Ss_open Ls       Lks      Lgs_revs
                    ; rot(list(T),list(T),list(T),list(T),
                                                        list(list(T))).

% ready/1 -State
:- pred ready(state(T)).
ready(ready).

% rot/4     +Lgs     ?Lgs_revs      ?Ss_tail -State
:- pred rot(list(T),list(list(T)),list(T), state(T)).
rot([],[Ss_tail],Ss_tail,ready).
rot([Lg|Lgs],[Lgs_rev|Lgs_revs],Ss_tail,rot(Lgs,Lgs_revs,Ss_tail)):-
    Lgs_revs = [[Lg|Lgs_rev]|_].
```

[4] Restricted deques lack one of the four operations. The queue implementation of the previous section needs only minor alterations to yield an output-restricted deque.

[5] Other ratios are possible. Hood [5] and Chuang & Goldberg [6] use 3 as well, whereas Okasaki [3] allows either 2 or 3. In our case, a ratio of 3 relieves us from the task of actually counting the elements we move from L to S, because there will be an integer multiple of $|S|$ number of them, which means we can use S as a counter.

```
% rot/6     +Ss     ?Ss_open +Ls     ?Lks     ?Lgs_revs     -State
:- pred rot(list(T),list(T), list(T),list(T),list(list(T)),state(T)).
rot([],Ss_tail,[L|Lgs],[L],Lgs_revs,rot(Lgs,Lgs_revs,Ss_tail)).
rot([S|Ss],[S|Ss_open],[L1,L2|Ls],[L1,L2|Lks],Lgs_revs,
                                        rot(Ss,Ss_open,Ls,Lks,Lgs_revs)).
```

A rotation starts in the `rot/5` state. We execute stages (1)–(2) in parallel, using `Ss` as a counter to pick $2 \cdot |S| + 1$ elements for `Lks`. It requires $|S| + 1$ calls to the `rot/6` predicate to arrive in the `rot/3` state. Stages (3)–(4) take $|L_{give}| + 1 = |L| - (2 \cdot |S| + 1) + 1$ calls to the `rot/4` predicate, where $|L|$ is between $3 \cdot |S| + 1$ and $3 \cdot (|S| + 1)$ at the start of the rotation – the latter occurs when we remove an item from the shorter list of a borderline balanced deque. We may thus need $|S| + 3$ calls to the `rot/4` predicate to complete these stages and arrive at `ready/0`. To be ready before the next rotation needs to start, we therefore have to perform $2 \cdot |S| + 4$ updates in $|S|$ operations, that is, 4 updates at the start of a rotation plus 2 for each operation on the deque.

Code to maintain the deque invariant and for the operations *pop*, *push*, and *reverse* is in Fig. 4. Reversing a deque is simply a matter of exchanging F and R. Injecting and ejecting are combinations of reversing and pushing or popping. Deque rotations are blissfully unaware of whether they are moving elements from the front to the rear or the other way around, so reversal of front and rear lists does not affect them. Finally, because the one element in a singleton deque may appear in either list, the implementation of *pop* involves a first, special case for singleton deques with an empty front list.

Through comparison to the queue implementation, rotation updates, deque assembly, *push*, and the second case of *pop* take constant time, modulo the costs of arithmetic. Furthermore, recognizing and executing the first case of *pop* only involves fixed size terms, and *inject* and *eject* are simple compositions of other operations – each of these only add constant time. In *reverse*, the deque lists and state are only unified with unbound variables, which takes constant time. Like before, operations and updates are deterministic. So, on the (deficient) assumption that the arithmetic operations are constant in the length of the deque, our deque implementation qualifies as a real-time persistent data structure.

4 Discussion

We have presented real-time persistent queues and deques using logic variables in Prolog. We consider the data structures, and in particular the pure Prolog version of the real-time persistent deque, to be declarative pearls, because they demonstrate that logic variables allow for some of the programming solutions that other languages get through lazy evaluation: the outcome of a computation can be used before the computation is completely finished. What logic variables alone do not automatically give you is the implicit, need-based forcing of the computation, but in the case of scheduled rotation this is not an issue as we want to have explicit control of the delayed computation anyway.

```
:- type deque(T) ---> deq(integer,list(T),integer,list(T),state(T)).

% push_deque/3     ?El +DEQ_old -DEQ_new
:- pred push_deque(T,  deque(T),deque(T)).
push_deque(F,deq(Fs_len,Fs,Rs_len,Rs,St),DEQ_new):-
    Fs1_len is Fs_len+1,
    make_deq(Fs1_len,[F|Fs],Rs_len,Rs,St,DEQ_new).


% pop_deque/3     +DEQ_old ?El -DEQ_new
:- pred pop_deque(deque(T),T,  deque(T)).
pop_deque(deq(Fs_len,Fs,Rs_len,Rs,St),F,DEQ_new):-
    ( Fs == []
      -> Rs = [F],
         empty_deque(DEQ_new)
    ; Fs = [F|Fs1],
      Fs1_len is Fs_len-1,
      make_deq(Fs1_len,Fs1,Rs_len,Rs,St,DEQ_new)
    ).


% reverse_deque/2     +DEQ     -DEQ_rev
:- pred reverse_deque(deque(T),deque(T)).
reverse_deque(deq(Fs_len,Fs,Rs_len,Rs,St),deq(Rs_len,Rs,Fs_len,Fs,St)).


% make_deq/6     +Fs_len +Fs     +Rs_len +Rs     +State   -DEQ
:- pred make_deq(integer,list(T),integer,list(T),state(T),deque(T)).
make_deq(Fs_len,Fs,Rs_len,Rs,St,DEQ):-
    ( Rs_len > 3*Fs_len
      -> Rs1_len is 2*Fs_len+1,
         Fs1_len is Rs_len-Fs_len-1,
         fourcalls(rot(Fs,Fs1,Rs,Rs1,[[]|_]),St1),
         DEQ = deq(Fs1_len,Fs1,Rs1_len,Rs1,St1)

    ; Fs_len > 3*Rs_len
      -> Fs1_len is 2*Rs_len+1,
         Rs1_len is Fs_len-Rs_len-1,
         fourcalls(rot(Rs,Rs1,Fs,Fs1,[[]|_]),St1),
         DEQ = deq(Fs1_len,Fs1,Rs1_len,Rs1,St1)

    ; twocalls(St,St1),
      DEQ = deq(Fs_len,Fs,Rs_len,Rs,St1)
    ).

twocalls(A,R):- call(A,AR), call(AR,R).
fourcalls(A,R):- twocalls(A,AR), twocalls(AR,R).
```

Fig. 4. Basic deque operations and invariant maintaining code.

Although we consider our queues a clear improvement over an implementation of the same ideas without reliance on logic variables, they are nowhere near as light-weight as the canonical queue in Prolog, that was already considered standard by the time [7] was written. In this implementation, a queue is a single open difference list – pop is constant time, as well as inject because of the constant time append available for open difference lists. That is, the resulting queue qualifies as real-time. The approach can be extended to output-restricted deques (no efficient eject), but as far as we are aware there is no easy way of using the approach for unrestricted deques.

The canonical Prolog queue is real-time, but it is only partially persistent. Injecting element e into version i of a queue, is done by instantiating the tail of the queue at i, t_i, with the list $[e|t_{i+1}]$. Constant time reuses of i are then restricted to popping or to injecting the exact same e. In fact, because all versions of the queue that only differ by a pop share the same tail, even past versions of the queue may become restricted in this way. The partial persistence does not cause trouble during backtracking, though, since the limiting variable binding will be undone for each pass. As should be clear, this problem does not occur in our implementation. During a rotation, we also append to an open list, but these appends are completely determined by and encapsulated in the rotation, and will therefore be identical for as long as the rotation lasts, irrespective of how the queue is used. The partial persistence of open difference lists is sufficient for the full persistence of the complete data structure.

Acknowledgements I would like to thank Markus Forsberg, Richard Johansson, and three anonymous reviewers for discussion and comments. The research presented here is carried out in the context of the Centre for Language Technology of the University of Gothenburg and Chalmers University of Technology.

References

1. Hood, R., Melville, R.: Real-time queue operations in pure LISP. Computer science technical report, Cornell University (1980) http://hdl.handle.net/1813/6273
2. O'Keefe, R.A.: The Craft of Prolog. The MIT Press, Cambridge, MA (1990)
3. Okasaki, Ch.: Simple and Efficient Purely Functional Queues and Deques, J. Functional Programming 5(4), 583–592 (1995)
4. Schrijvers, T., Santos Costa, V., Wielemaker, J., Demoen, B.: Towards Typed Prolog. In: Garcia de la Banda, M., Pontelli, E. (eds) Logic Programming. LNCS vol. 5366, pp. 603–697. Springer, Heidelberg (2008) http://dx.doi.org/10.1007/978-3-540-89982-2_59
5. Hood, R.T.: The Efficient Implementation of Very-High-Level Programming Language Constructs. Computer science technical report, Cornell University (1982) http://hdl.handle.net/1813/6343
6. Chuang, T.-R, Goldberg, B.: Real-time deques, multihead turing machines, and purely functional programming. In: Proceedings of the Conference on Functional Programming and Computer Architecture, pp. 289 298. ACM, New York (1993)
7. Sterling, L., Shapiro, E.: The Art of Prolog. The MIT Press, Cambridge, MA (1986)