

# Hood-Melville Queue

Alejandro Gómez-Londoño

March 17, 2025

## Abstract

This is a verified implementation of a constant time queue. The original design is due to Hood and Melville [1]. This formalization follows the presentation by Okasaki [2].

```
theory Hood-Melville-Queue
imports
  HOL-Data-Structures.Queue-Spec
begin

datatype 'a status =
  Idle
  | Rev nat 'a list 'a list 'a list 'a list
  | App nat 'a list 'a list
  | Done 'a list

record 'a queue = lenf :: nat
  front :: 'a list
  status :: 'a status
  rear :: 'a list
  lenr :: nat

fun exec :: 'a status  $\Rightarrow$  'a status where
  exec (Rev ok (x#f) f' (y#r) r') = Rev (ok+1) f (x#f') r (y#r')
| exec (Rev ok [] f' [y] r') = App ok f' (y#r')
| exec (App 0 f' r') = Done r'
| exec (App ok (x#f') r') = App (ok-1) f' (x#r')
| exec s = s

fun invalidate where
```

```

  invalidate (Rev ok f f' r r') = Rev (ok-1) f f' r r'
| invalidate (App 0 f' (x#r')) = Done r'
| invalidate (App ok f' r') = App (ok-1) f' r'
| invalidate s = s

```

**fun** *exec2* :: 'a queue ⇒ 'a queue **where**

```

exec2 q =
  (case exec (exec (status q)) of
    Done newf ⇒ q(status := Idle, front := newf) |
    newstate ⇒ q(status := newstate))

```

**definition** *check* :: 'a queue ⇒ 'a queue **where**

```

check q = (if lenr q ≤ lenf q
  then exec2 q
  else let newstate = Rev 0 (front q) [] (rear q) []
    in exec2 (q(lenf := lenf q + lenr q, status := newstate, rear := [],
lenr := 0)))

```

**definition** *empty* :: 'a queue **where**

```

empty = queue.make 0 [] Idle [] 0

```

**fun** *enq* **where**

```

enq x q = check (q(rear := x#(rear q), lenr := lenr q + 1))

```

**fun** *deq* **where**

```

deq q = check (q(lenf := lenf q - 1
, front := tl (front q)
, status := invalidate (status q)))

```

**fun** *front-list* :: 'a queue ⇒ 'a list **where**

```

front-list q = (case status q of
  Idle ⇒ front q
| Done f ⇒ f
| Rev ok f f' r r' ⇒ rev (take ok f') @ f @ rev r @ r'
| App ok f' r' ⇒ rev (take ok f') @ r')

```

**definition** *rear-list* :: 'a queue ⇒ 'a list **where**

```

rear-list = rev o rear

```

**fun** *list* :: 'a queue ⇒ 'a list **where**

```

list q = front-list q @ rear-list q

```

**fun** *first* :: 'a queue  $\Rightarrow$  'a **where**  
*first* q = hd (front q)

**fun** *rem-steps* :: 'a status  $\Rightarrow$  nat **where**  
*rem-steps* (Rev ok f f' r r') = 2\*length f + ok + 2  
| *rem-steps* (App ok f' r') = ok + 1  
| *rem-steps* - = 0

**fun** *inv-st* :: 'a status  $\Rightarrow$  bool **where**  
*inv-st* (Rev ok f f' r r') = (length f + 1 = length r  $\wedge$   
length f' = length r'  $\wedge$   
ok  $\leq$  length f')  
| *inv-st* (App ok f' r') = (ok  $\leq$  length f'  $\wedge$  length f' < length r')  
| *inv-st* - = True

**fun** *steps* :: nat  $\Rightarrow$  'a status  $\Rightarrow$  'a status **where**  
*steps* n st = (exec  $\overset{\sim}{\sim}$  n) st

**lemma** *rev-steps-app*:

**assumes** *inv*: *inv-st* (Rev ok f f' r r')  
**shows** *steps* (length f + 1) (Rev ok f f' r r') = App (length f + ok) (rev f @ f')  
(rev r @ r')

**proof** -

**show** ?thesis **using** *inv*

**proof** (*induction* f arbitrary: ok f' r r')

**case** Nil

**then obtain** x **where** r = [x]

**by** (*metis* One-nat-def Suc-length-conv add.right-neutral add-Suc-right length-0-conv  
*inv-st.simps*(1))

**then show** ?case **using** Nil **by** *simp*

**next**

**case** (Cons a f)

**then obtain** x and xs **where** r = x # xs

**by** (*metis* One-nat-def Suc-length-conv add-Suc-right *inv-st.simps*(1))

**hence** r-x: r = x # xs **by** *simp*

**then show** ?case **using** Cons Nat.funpow-add **by** (*simp* add: Nat.funpow-swap1)

**qed**

**qed**

**lemma** *inv-st-steps*:

**assumes** *inv* : *inv-st* s

**assumes** *not-idle* : s  $\neq$  Idle

**shows**  $\exists x. \text{steps } (rem\text{-steps } s) s = Done\ x$  (**is** ?reach-done s)

```

proof –
  let ?steps = λx. steps (rem-steps x)
  have app-inv: inv-st (App ok f r) ⇒ ?reach-done (App ok f r)
    for ok f r
  proof (induct f arbitrary: ok r)
    case (Cons a f') then show ?case
      by (induct ok; simp add: Nat.funpow-swap1)
  qed simp
  show ?thesis
  proof (cases s)
    case (App ok f' r')
      then show ?thesis using inv app-inv unfolding App by simp
  next
    case (Rev ok f f' r r')
      have rep-split: rem-steps (Rev ok f f' r r') = (length f + ok + 1) + (length f
+ 1) by simp
      then have split: ∧stp. ?steps (Rev ok f f' r r') stp = (steps (length f + ok +
1)) ((steps (length f + 1)) stp)
        unfolding rep-split Nat.funpow-add steps.simps by simp
      also have f: inv-st (App (length f + ok) (rev f @ f') (rev r @ r'))
        using Rev inv by simp
      thus ?thesis using inv f [THEN app-inv]
        unfolding Rev split inv[simplified Rev, THEN rev-steps-app] by simp
      qed (auto simp add: not-idle)
  qed

```

```

lemma inv-st-exec:
  assumes inv-st: inv-st s
  shows inv-st (exec s)
proof (cases s)
next
  case (Rev ok f f' r r')
    show ?thesis
    proof (cases f)
      case Nil
        then show ?thesis using inv-st unfolding Rev
          by (simp; cases r; cases ok; cases f'; simp)
      next
        case C-a: (Cons a as)
          then obtain x xs where r = x # xs using inv-st unfolding Rev Cons
            by (metis One-nat-def length-Suc-conv list.size(4) inv-st.simps(1))
          hence r-x: r = x # xs by simp
          then show ?thesis
            proof (cases as)
              case Nil then show ?thesis using inv-st unfolding Rev C-a Nil r-x by
(simp; cases xs; simp)
            next
              case (Cons b bs)

```

```

    then show ?thesis using inv-st unfolding Rev C-a r-x by (simp; cases xs;
simp)
  qed
  qed
next
case (App ok f r)
then show ?thesis
proof (cases ok)
  case (Suc ok')
  then obtain x xs where f = x # xs using inv-st unfolding App Suc
  by (metis Suc-le-D Zero-not-Suc list.exhaust list.size(3) inv-st.simps(2))
  then show ?thesis using inv-st unfolding App Suc
  by (cases ok'; cases xs; simp)
qed simp
qed simp+

```

```

lemma inv-st-exec2:
  assumes inv-st: inv-st s
  shows inv-st (exec (exec s))
proof -
  show ?thesis using inv-st inv-st-exec
  by auto
qed

```

```

lemma inv-st-invalidate:
  assumes inv-st: inv-st s
  shows inv-st (invalidate s)
proof (cases s)
next
case (Rev ok f f' r r')
show ?thesis using inv-st unfolding Rev by auto
next
case (App ok f r)
then show ?thesis
  using inv-st unfolding App
  by (cases ok; cases r; simp)
qed simp+

```

**definition invar where**

$$\begin{aligned}
\text{invar } q = & (\text{lenf } q = \text{length } (\text{front-list } q) \wedge \\
& \text{lenr } q = \text{length } (\text{rear-list } q) \wedge \\
& \text{lenr } q \leq \text{lenf } q \wedge \\
& (\text{case status } q \text{ of} \\
& \quad \text{Rev } ok \ f \ f' \ r \ r' \Rightarrow 2 * \text{lenr } q \leq \text{length } f' \wedge ok \neq 0 \wedge 2 * \text{length } f + ok \\
+ 2 \leq & 2 * \text{length } (\text{front } q) \\
& \quad | \text{App } ok \ f \ r \Rightarrow 2 * \text{lenr } q \leq \text{length } r \wedge ok + 1 \leq 2 * \text{length } (\text{front } q) \\
& \quad | - \Rightarrow \text{True}) \wedge
\end{aligned}$$

$$(\exists \text{rest. front-list } q = \text{front } q @ \text{rest}) \wedge$$

$$(\neg(\exists \text{fr. status } q = \text{Done fr})) \wedge$$

$$\text{inv-st } (\text{status } q)$$

**lemma** *invar-empty: invar empty*  
**by**(*simp add: invar-def empty-def make-def rear-list-def*)

**lemma** *tl-rev-take*:  $\llbracket 0 < \text{ok}; \text{ok} \leq \text{length } f \rrbracket \implies \text{rev } (\text{take } \text{ok } (x \# f)) = \text{tl } (\text{rev } (\text{take } \text{ok } f)) @ [x]$   
**by**(*simp add: rev-take Suc-diff-le drop-Suc tl-drop*)

**lemma** *tl-rev-take-Suc*:  
 $n + 1 \leq \text{length } l \implies \text{rev } (\text{take } n l) = \text{tl } (\text{rev } (\text{take } (\text{Suc } n) l))$   
**by**(*simp add: rev-take tl-drop Suc-diff-Suc flip: drop-Suc*)

**lemma** *invar-deq*:  
**assumes** *inv: invar q*  
**shows** *invar (deq q)*  
**proof** (*cases q*)  
**case** (*fields lenf front status rear lenr*)  
**have** *pre-inv:  $\exists \text{rest. front } @ \text{rest} = \text{front-list } q$  using *inv unfolding fields**  
**by**(*simp add: invar-def check-def; cases status; auto simp add: invar-def Let-def rear-list-def*)  
**have** *tl-app:  $\text{status} \neq \text{Idle} \implies \forall l. \text{tl front } @ l = \text{tl } (\text{front } @ l)$  using *inv unfolding fields**  
**by** (*simp add: invar-def check-def; cases status; cases front; auto simp add: invar-def Let-def rear-list-def*)  
**then show** *?thesis*  
**proof** (*cases status rule: exec.cases*)  
**case** *st: (1 ok x f f' y r r')*  
**then show** *?thesis*  
**proof** (*cases f*)  
**case** *Nil*  
**have** *pre:  $\exists \text{rest. front-list } (\text{deq } q) = \text{tl front } @ \text{rest}$  using *inv pre-inv unfolding fields st Nil**  
**apply** (*simp add: invar-def check-def; cases r; simp add: invar-def Let-def rear-list-def*)  
**apply** (*erule exE*)  
**apply** (*rule-tac x=rest in exI*)  
**apply** (*simp add: tl-app st tl-rev-take*)  
**apply** (*cases f'; auto*)  
**done**  
**then show** *?thesis using inv unfolding fields st Nil*  
**by** (*simp add: invar-def check-def rear-list-def; cases r; auto simp add: min-absorb2 invar-def rear-list-def Let-def*)

```

next
  case (Cons a list)
  then show ?thesis using pre-inv inv
    unfolding fields st Nil
    apply (simp add: invar-def check-def inv ; cases r; simp add: invar-def inv
min-absorb2 rear-list-def)
    apply (erule exE)
    apply (rule conjI, force)
    apply (rule-tac x=rest in exI)
    apply (simp add: tl-app st tl-rev-take)
    apply (cases f'; auto)
  done
qed
next
case st: (2 ok f y r)
then show ?thesis
proof(cases ok)
  case ok: 0
  then show ?thesis using inv unfolding fields st
    by (simp add: invar-def check-def rear-list-def)
next
case (Suc ok')
obtain fx fs where f = fx # fs
  using inv lessI less-le-trans not-less-zero
  unfolding fields st Suc invar-def
  by (metis list.exhaust list.size(3) select-convs(3) inv-st.simps(1))
hence f-x: f = fx # fs by simp
obtain rx rs where r = rx # rs
  using inv lessI less-le-trans not-less-zero
  unfolding fields st Suc invar-def
  by (metis list.exhaust list.size(3) select-convs(3) inv-st.simps(1))
hence r-x: r = rx # rs by simp
  then show ?thesis using pre-inv inv unfolding fields st Suc invar-def
rear-list-def r-x f-x
  apply (simp add: check-def; cases ok'; simp add: check-def min-absorb2)

  apply (erule exE)
  apply (rule conjI, arith)
  apply (rule-tac x=rest in exI)
  apply (simp add: tl-app st tl-rev-take-Suc)
  by (metis Suc-le-length-iff length-take list.sel(3) min-absorb2 n-not-Suc-n
rev-is-Nil-conv take-tl tl-Nil tl-append2)
qed
next
case st: (3 f r)
show ?thesis using inv unfolding fields st
  by(cases r; simp add: invar-def check-def rear-list-def)
next
case st: (4 ok x f r)

```

```

then show ?thesis
proof(cases ok)
  case 0
  then show ?thesis using inv unfolding fields st invar-def rear-list-def
  by (simp add: check-def)
next
  case (Suc ok')
  then show ?thesis using pre-inv inv unfolding fields st invar-def rear-list-def
Suc
  apply (cases f; cases ok'; simp add: invar-def rear-list-def check-def
min-absorb2)
  apply (erule exE)
  apply (rule conjI, arith)
  apply (rule-tac x=rest in exI)
  apply (simp add: tl-app st tl-rev-take-Suc)
  by (metis length-take list.size(3) min.absorb2 nat.distinct(1) rev.simps(1)
rev-rev-ident tl-append2)
qed
next
  case st: 5-1
  then show ?thesis
proof (cases lenr ≤ lenf - 1)
  case True
  then show ?thesis using inv unfolding st fields
  by (simp add: check-def rear-list-def invar-def)
next
  case overflows: False
  then have f-eq-r: length front = length rear using inv unfolding st fields
  by (simp add: le-antisym rear-list-def invar-def)
  then show ?thesis
proof (cases front)
  case Nil
  show ?thesis using inv overflows unfolding st fields Nil
  by (cases rear; auto simp add: rear-list-def check-def Let-def invar-def)
next
  case C-a : (Cons a as)
  then obtain x xs where rear = x # xs
  using inv overflows unfolding st fields Cons invar-def
  by (metis f-eq-r length-Suc-conv C-a)
  hence rear-x: rear = x # xs by simp
  then show ?thesis
proof (cases as)
  case Nil
  then show ?thesis using inv overflows unfolding st fields Nil rear-x C-a
  by (cases xs; simp add: invar-def check-def Let-def rear-list-def)
next
  case (Cons b bs)
  then show ?thesis using inv overflows unfolding st fields Cons rear-x
C-a invar-def

```



```

      by (cases xs; cases bs; simp add: check-def Let-def rear-list-def)
    qed
  qed
  qed
next
  case st: 5-2 then show ?thesis using inv unfolding fields st
    by (simp add: invar-def inv fields st)
next
  case st: 5-3 then show ?thesis using inv unfolding fields st
    by (simp add: invar-def)
next
  case st: 5-4 then show ?thesis using inv unfolding fields st
    by (simp add: invar-def)
next
  case st: 5-5 then show ?thesis using inv unfolding fields st
    by (simp add: invar-def)
next
  case st: (5-6 v) then show ?thesis using inv unfolding fields st
    by (simp add: invar-def)
  qed
qed

```

lemma *invar-enq*:

```

  assumes inv: invar q
  shows invar (enq x q)
proof (cases q)
  case (fields lenf front status rear lenr)
  then show ?thesis
  proof (cases status rule: exec.cases)
    case st: (1 ok x f f' y r r')
    then show ?thesis
    proof (cases f)
      case Nil
      then show ?thesis using inv unfolding fields st Nil
        by (simp add: invar-def check-def rear-list-def; cases r; auto simp add:
min-absorb2 invar-def rear-list-def Let-def)
    next
      case (Cons a list)
      then show ?thesis using inv check-def rear-list-def
        unfolding fields st Nil invar-def check-def rear-list-def
        by (simp; cases r; auto simp add: min-absorb2 check-def)
    qed
  next
  case st: (2 ok f y r)
  then show ?thesis
  proof (cases ok)
    case ok: 0
    then show ?thesis using inv unfolding fields st

```

```

    by (simp add: invar-def check-def rear-list-def)
  next
  case (Suc ok')
  obtain fx fs where f = fx # fs
    using inv lessI less-le-trans not-less-zero
    unfolding fields st Suc invar-def
    by (metis list.exhaust list.size(3) select-convs(3) inv-st.simps(1))
  hence f-x: f = fx # fs by simp
  obtain rx rs where r = rx # rs
    using inv lessI less-le-trans not-less-zero
    unfolding fields st Suc invar-def
    by (metis list.exhaust list.size(3) select-convs(3) inv-st.simps(1))
  hence r-x: r = rx # rs by simp
  then show ?thesis using inv unfolding fields st Suc invar-def rear-list-def
r-x f-x
    by (simp add: check-def; cases ok'; simp add: check-def min-absorb2)
  qed
next
case st: (3 f r)
then show ?thesis
proof(cases r)
  case Nil
  then show ?thesis using inv unfolding fields st
    by (simp add: check-def rear-list-def Let-def invar-def)
next
case (Cons a list)
then show ?thesis using inv unfolding fields st
  by (simp add: check-def rear-list-def Let-def invar-def)
qed
next
case st: (4 ok x f r)
then show ?thesis
proof(cases ok)
  case 0
  then show ?thesis using inv unfolding fields st invar-def rear-list-def
    by (simp add: check-def)
next
case (Suc ok')
then show ?thesis using inv unfolding fields st invar-def rear-list-def Suc
  by (cases f; cases ok'; auto simp add: invar-def rear-list-def check-def
min-absorb2)
qed
next
case st: 5-1
then show ?thesis
proof (cases lenr + 1 ≤ lenf)
  case True
  then show ?thesis using inv unfolding st fields
    by (simp add: check-def rear-list-def invar-def)

```

```

next
  case overflows: False
  then have f-eq-r: length front = length rear using inv unfolding st fields
    by (simp add: le-antisym rear-list-def invar-def)
  then show ?thesis
  proof (cases front)
    case Nil
    show ?thesis using inv overflows unfolding st fields Nil
      by (cases rear; auto simp add: rear-list-def check-def Let-def invar-def)
  next
  case C-a : (Cons a as)
  then obtain x xs where rear = x # xs
    using inv overflows unfolding st fields Cons invar-def
    by (metis f-eq-r length-Suc-conv C-a)
  hence rear-x: rear = x # xs by simp
  then show ?thesis
  proof (cases as)
    case Nil
    then show ?thesis using inv overflows unfolding st fields Nil rear-x C-a
      by (cases xs; simp add: invar-def check-def Let-def rear-list-def)
  next
  case (Cons b bs)
  then show ?thesis using inv overflows unfolding st fields Cons rear-x
    C-a invar-def
    by (cases xs; cases bs; simp add: check-def Let-def rear-list-def)
  qed
  qed
  qed
next
  case st: 5-2 then show ?thesis using inv unfolding fields st
    by (simp add: invar-def)
next
  case st: 5-3 then show ?thesis using inv unfolding fields st
    by (simp add: invar-def)
next
  case st: 5-4 then show ?thesis using inv unfolding fields st
    by (simp add: invar-def)
next
  case st: 5-5 then show ?thesis using inv unfolding fields st
    by (simp add: invar-def)
next
  case st: 5-6 then show ?thesis using inv unfolding fields st
    by (simp add: invar-def)
  qed
qed

```

```

lemma queue-correct-deq :
  assumes inv: invar q

```

```

shows list (deq q) = tl (list q)
proof (cases q)
  case (fields lenf front status rear lenr)
  have inv-deq: invar (deq q) using inv invar-deq by simp
  then show ?thesis
  proof (cases status rule: exec.cases)
    case st: (1 ok x f f' y r r')
    then show ?thesis using inv inv-deq unfolding st fields
    apply (cases f; cases r; simp add: invar-def check-def rear-list-def min-absorb2
tl-rev-take)
    by (metis le-zero-eq length-take list.size(3) min.absorb2 not-le rev-is-Nil-conv
tl-append2)+
  next
  case st: (2 ok f y r)
  then show ?thesis
  proof(cases ok)
    case ok: 0
    then show ?thesis using inv unfolding fields st
    by (simp add: invar-def check-def rear-list-def)
  next
  case (Suc ok')
  then show ?thesis using inv inv-deq unfolding fields st Suc invar-def
  apply (cases f; cases r; cases ok'; simp add: check-def min-absorb2 rear-list-def
tl-rev-take-Suc)
  by (metis (no-types) length-greater-0-conv less-le-trans old.nat.distinct(2)
rev-is-Nil-conv take-eq-Nil tl-append2 zero-less-Suc)
  qed
next
  case st: (3 f r)
  then show ?thesis using inv inv-deq unfolding st fields
  by (cases f; cases r; simp add: invar-def check-def rear-list-def)
next
  case st: (4 ok x f r)
  then show ?thesis
  proof(cases ok)
    case 0
    then show ?thesis using inv inv-deq unfolding fields st invar-def rear-list-def
    by (simp add: check-def rear-list-def)
  next
  case (Suc ok')
  then show ?thesis using inv inv-deq unfolding fields st invar-def rear-list-def
Suc
  apply (cases f; cases ok'; auto simp add: rear-list-def check-def min-absorb2
tl-rev-take-Suc)
  by (metis Nitpick.size-list-simp(2) length-rev length-take min.absorb2 nat.simps(3)
tl-append2)
  qed
next
  case st: 5-1

```

```

then show ?thesis
proof (cases lenr ≤ lenf - 1)
  case True
  then show ?thesis
  using inv inv-deq Nil-is-rev-conv append-Nil diff-is-0-eq diff-zero length-0-conv
  unfolding st fields
  by (simp add: check-def rear-list-def invar-def; metis list.sel(2) tl-append2)
next
  case overflows: False
  then have f-eq-r: length front = length rear using inv unfolding st fields
  by (simp add: le-antisym rear-list-def invar-def)
  then show ?thesis
  proof (cases front)
    case Nil
    show ?thesis using inv overflows inv-deq unfolding st fields Nil
    by (cases rear; auto simp add: rear-list-def check-def Let-def invar-def)
  next
  case C-a : (Cons a as)
  then obtain x xs where rear = x # xs
  using inv overflows unfolding st fields Cons invar-def
  by (metis f-eq-r length-Suc-conv C-a)
  hence rear-x: rear = x # xs by simp
  then show ?thesis
  proof (cases as)
    case Nil
    then show ?thesis using inv overflows unfolding st fields Nil rear-x C-a
    by (cases xs; simp add: invar-def check-def Let-def rear-list-def)
  next
  case (Cons b bs)
  then show ?thesis using inv overflows unfolding st fields Cons rear-x
  C-a invar-def
  by (cases xs; cases bs; simp add: check-def Let-def rear-list-def)
  qed
  qed
  qed
next
  case st: 5-2 then show ?thesis using inv inv-deq unfolding st fields
  by (simp add: invar-def check-def rear-list-def)
next
  case st: 5-3 then show ?thesis using inv inv-deq unfolding st fields
  by (simp add: invar-def check-def rear-list-def)
next
  case st: 5-4 then show ?thesis using inv inv-deq unfolding st fields
  by (simp add: invar-def check-def rear-list-def)
next
  case st: 5-5 then show ?thesis using inv inv-deq unfolding st fields
  by (simp add: invar-def check-def rear-list-def)
next
  case st: 5-6 then show ?thesis using inv inv-deq unfolding st fields

```

```

    by (simp add: invar-def check-def rear-list-def)
  qed
qed

lemma queue-correct-enq :
  assumes inv: invar q
  shows list (enq x q) = (list q) @ [x]
proof (cases q)
  case (fields lenf front status rear lenr)
  have inv-deq: invar (enq x q) using inv invar-enq by simp
  then show ?thesis
  proof (cases status rule: exec.cases)
    case st: (1 ok x f f' y r r')
    then show ?thesis using inv inv-deq unfolding st fields
    by (cases f; cases r; simp add: invar-def check-def rear-list-def min-absorb2
tl-rev-take)
  next
    case st: (2 ok f y r)
    then show ?thesis
    proof (cases ok)
      case ok: 0
      then show ?thesis using inv unfolding fields st
      by (simp add: invar-def check-def rear-list-def)
    next
      case (Suc ok')
      then show ?thesis using inv inv-deq unfolding fields st Suc invar-def
      by (cases f; cases r; cases ok'; simp add: check-def min-absorb2 rear-list-def
tl-rev-take-Suc)
    qed
  next
    case st: (3 f r)
    then show ?thesis using inv inv-deq unfolding st fields
    by (cases f; cases r; simp add: invar-def check-def rear-list-def)
  next
    case st: (4 ok x f r)
    then show ?thesis
    proof (cases ok)
      case 0
      then show ?thesis using inv inv-deq unfolding fields st invar-def rear-list-def
      by (simp add: check-def rear-list-def)
    next
      case (Suc ok')
      then show ?thesis using inv inv-deq unfolding fields st invar-def rear-list-def
      by (cases f; cases ok'; auto simp add: rear-list-def check-def min-absorb2
tl-rev-take-Suc)
    qed
  next

```

```

case st: 5-1
then show ?thesis
proof (cases lenr + 1 ≤ lenf)
  case True
  then show ?thesis
  using inv inv-deq Nil-is-rev-conv append-Nil diff-is-0-eq diff-zero length-0-conv
  unfolding st fields
  by (simp add: check-def rear-list-def invar-def; metis list.sel(2) tl-append2)
next
case overflows: False
then have f-eq-r: length front = length rear using inv unfolding st fields
  by (simp add: le-antisym rear-list-def invar-def)
then show ?thesis
proof (cases front)
  case Nil
  show ?thesis using inv overflows inv-deq unfolding st fields Nil
  by (cases rear; auto simp add: rear-list-def check-def Let-def invar-def)
next
case C-a : (Cons a as)
then obtain x xs where rear = x # xs
  using inv overflows unfolding st fields Cons invar-def
  by (metis f-eq-r length-Suc-conv C-a)
hence rear-x: rear = x # xs by simp
then show ?thesis
proof (cases as)
  case Nil
  then show ?thesis using inv overflows unfolding st fields Nil rear-x C-a
  by (cases xs; simp add: invar-def check-def Let-def rear-list-def)
next
case (Cons b bs)
  then show ?thesis using inv overflows unfolding st fields Cons rear-x
C-a invar-def
  by (cases xs; cases bs; simp add: check-def Let-def rear-list-def)
  qed
qed
qed
next
case st: 5-2 then show ?thesis using inv inv-deq unfolding st fields
  by (simp add: invar-def check-def rear-list-def)
next
case st: 5-3 then show ?thesis using inv inv-deq unfolding st fields
  by (simp add: invar-def check-def rear-list-def)
next
case st: 5-4 then show ?thesis using inv inv-deq unfolding st fields
  by (simp add: invar-def check-def rear-list-def)
next
case st: 5-5 then show ?thesis using inv inv-deq unfolding st fields
  by (simp add: invar-def check-def rear-list-def)
next

```

```

    case st: 5-6 then show ?thesis using inv inv-deq unfolding st fields
    by (simp add: invar-def check-def rear-list-def)
  qed
qed

```

```

datatype 'a action = Deq | Enq 'a

```

```

type-synonym 'a actions = 'a action list

```

```

fun do-act :: 'a action  $\Rightarrow$  'a queue  $\Rightarrow$  'a queue where
  do-act Deq q = deq q
| do-act (Enq x) q = enq x q

```

```

definition qfa :: 'a actions  $\Rightarrow$  'a queue where
  qfa = ( $\lambda$ acts. foldr do-act acts empty)

```

```

lemma invar-qfa : invar (qfa l)
proof(induction l)
  case Nil
  then show ?case by (simp add: qfa-def invar-empty)
next
  case (Cons x xs)
  have qfa-cons: qfa (x#xs) = do-act x (qfa xs) by (simp add: qfa-def)
  then show ?case
  proof(cases x)
    case Deq
    then show ?thesis using invar-deq[of qfa xs] unfolding qfa-cons
    by (simp add: Cons)
  next
    case (Enq a)
    then show ?thesis using invar-enq[of qfa xs] unfolding qfa-cons
    by (simp add: Cons)
  qed
qed

```

```

lemma qfa-deq-correct: list (deq (qfa l)) = tl (list (qfa l))
  using invar-qfa queue-correct-deq by blast

```

```

lemma qfa-enq-correct: list (enq x (qfa l)) = (list (qfa l)) @ [x]
  by (meson invar-qfa queue-correct-enq)

```

```

lemma first-correct :
  assumes inv: invar q
  assumes not-nil : list q  $\neq$  []

```



```

shows          first q = hd (list q)
proof (cases front q)
  obtain rest where front-l: front-list q = front q @ rest
    using inv
    by (auto simp add: invar-def simp del: front-list.simps)
  case front-nil: Nil
  have rear-nil: rear-list q = [] using inv unfolding invar-def rear-list-def front-nil
    by (simp; cases status q; simp add: front-nil)
  have front-nil: front-list q = [] using inv unfolding invar-def rear-list-def
front-nil
    by (simp; cases status q; simp add: front-nil)
  show ?thesis using not-nil unfolding list.simps rear-nil front-nil
    by simp
next
  case front-cons: (Cons x xs)
  show ?thesis using inv unfolding list.simps first.simps front-cons front-list.simps

    apply (simp add: invar-def rear-list-def)
    by (metis append-Cons front-cons list.sel(1))
qed

fun is-empty :: 'a queue ⇒ bool where
  is-empty q = (list q = [])

interpretation HMQueue: Queue where
  empty = empty and
  enq = enq and
  first = first and
  deq = deq and
  is-empty = is-empty and
  list = list and
  invar = invar
proof (standard, goal-cases)
  case 1 thus ?case
    by (simp add: empty-def make-def rear-list-def)
next
  case 2 thus ?case using queue-correct-enq by simp
next
  case 3 thus ?case using queue-correct-deq by simp
next
  case 4 thus ?case using first-correct by simp
next
  case 5 thus ?case by simp
next
  case 6 thus ?case
    by (simp add: empty-def invar-def make-def rear-list-def)
next
  case 7 thus ?case using invar-enq by simp
next

```

```
case 8 thus ?case using invar-deq by simp
qed

end
```

## References

- [1] R. Hood and R. Melville. Real-time queue operation in pure LISP. *Inf. Process. Lett.*, 13(2):50–54, 1981.
- [2] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.