

Real-Time Double-Ended Queue

Balazs Toth and Tobias Nipkow
Technical University of Munich

February 6, 2026

Abstract

A double-ended queue (*deque*) is a queue where one can enqueue and dequeue at both ends. We define and verify the deque implementation by Chuang and Goldberg [1]. It is purely functional and all operations run in constant time.

Contents

1	Double-Ended Queue Specification	2
2	Type Classes	3
3	Stack	4
4	Current Stack	4
5	Idle	5
6	Common	5
7	Bigger End of Deque	7
8	Smaller End of Deque	8
9	Combining Big and Small	9
10	Real-Time Deque Implementation	10
11	Basic Lemma Library	24
12	Stack Proofs	26
13	Idle Proofs	28
14	Current Proofs	29

15 Common Proofs	31
16 Big Proofs	33
17 Small Proofs	35
18 Big + Small Proofs	37
19 Dequeue Proofs	53
20 Enqueue Proofs	54
21 Top-Level Proof	54

1 Double-Ended Queue Specification

```
theory Deque
imports Main
begin
```

Model-oriented specification in terms of an abstraction function to a list.

```
locale Deque =
fixes empty :: 'q
fixes enqL :: 'a ⇒ 'q ⇒ 'q
fixes enqR :: 'a ⇒ 'q ⇒ 'q
fixes firstL :: 'q ⇒ 'a
fixes firstR :: 'q ⇒ 'a
fixes deqL :: 'q ⇒ 'q
fixes deqR :: 'q ⇒ 'q
fixes is-empty :: 'q ⇒ bool
fixes listL :: 'q ⇒ 'a list
fixes invar :: 'q ⇒ bool

assumes list-empty:
  listL empty = []

assumes list-enqL:
  invar q ⇒ listL(enqL x q) = x # listL q
assumes list-enqR:
  invar q ⇒ rev(listL(enqR x q)) = x # rev(listL q)
assumes list-deqL:
  [[invar q; ¬ listL q = []] ⇒ listL(deqL q) = tl(listL q)]
assumes list-deqR:
  [[invar q; ¬ rev(listL q) = []] ⇒ rev(listL(deqR q)) = tl(rev(listL q))]

assumes list-firstL:
  [[invar q; ¬ listL q = []] ⇒ firstL q = hd(listL q)]
assumes list-firstR:
```

$\llbracket \text{invar } q; \neg \text{rev } (\text{listL } q) = [] \rrbracket \implies \text{firstR } q = \text{hd}(\text{rev}(\text{listL } q))$

assumes *list-is-empty*:

invar q \implies *is-empty q* = (*listL q* = [])

assumes *invar-empty*:

invar empty

assumes *invar-enqL*:

invar q \implies *invar(enqL x q)*

assumes *invar-enqR*:

invar q \implies *invar(enqR x q)*

assumes *invar-deqL*:

$\llbracket \text{invar } q; \neg \text{is-empty } q \rrbracket \implies \text{invar}(\text{deqL } q)$

assumes *invar-deqR*:

$\llbracket \text{invar } q; \neg \text{is-empty } q \rrbracket \implies \text{invar}(\text{deqR } q)$

begin

abbreviation *listR* :: 'q \Rightarrow 'a list **where**

listR deque \equiv *rev (listL deque)*

end

end

2 Type Classes

theory *Type-Classes*

imports *Main*

begin

Overloaded functions:

class *is-empty* =

fixes *is-empty* :: 'a \Rightarrow bool

class *invar* =

fixes *invar* :: 'a \Rightarrow bool

class *size-new* =

fixes *size-new* :: 'a \Rightarrow nat

class *step* =

fixes *step* :: 'a \Rightarrow 'a

class *remaining-steps* =

fixes *remaining-steps* :: 'a \Rightarrow nat

end

3 Stack

```
theory Stack
imports Type-Classes
begin
```

A datatype encapsulating two lists. Is used as a base data-structure in different places. It has the operations *push*, *pop* and *first*.

```
datatype (plugins del: size) 'a stack = Stack 'a list 'a list
```

```
fun push :: 'a ⇒ 'a stack ⇒ 'a stack where
  push x (Stack left right) = Stack (x#left) right
```

```
fun pop :: 'a stack ⇒ 'a stack where
  pop (Stack [] []) = Stack [] []
| pop (Stack (x#left) right) = Stack left right
| pop (Stack [] (x#right)) = Stack [] right
```

```
fun first :: 'a stack ⇒ 'a where
  first (Stack (x#left) right) = x
| first (Stack [] (x#right)) = x
```

```
instantiation stack ::(type) is-empty
begin
```

```
fun is-empty-stack where
  is-empty-stack (Stack [] []) = True
| is-empty-stack - = False
```

```
instance(proof)
end
```

end

4 Current Stack

```
theory Current
imports Stack
begin
```

This data structure is composed of:

- the newly added elements to one end of a deque during the rebalancing phase
- the number of these newly added elements

- the originally contained elements
- the number of elements which will be contained after the rebalancing is finished.

datatype (*plugins del: size*) 'a current = Current 'a list nat 'a stack nat

fun *push* :: 'a ⇒ 'a current ⇒ 'a current **where**
push x (Current *extra* *added* *old* *remained*) = Current (x#*extra*) (*added* + 1) *old*
remained

fun *pop* :: 'a current ⇒ 'a * 'a current **where**
pop (Current [] *added* *old* *remained*) =
 (first *old*, Current [] *added* (Stack.pop *old*) (*remained* - 1))
 | *pop* (Current (x#*xs*) *added* *old* *remained*) =
 (x, Current *xs* (*added* - 1) *old* *remained*)

fun *first* :: 'a current ⇒ 'a **where**
first current = fst (*pop* current)

abbreviation *drop-first* :: 'a current ⇒ 'a current **where**
drop-first current ≡ snd (*pop* current)

end

5 Idle

theory *Idle*
imports *Stack*
begin

Represents the ‘idle’ state of one deque end. It contains a *stack* and its size as a natural number.

datatype (*plugins del: size*) 'a idle = Idle 'a stack nat

fun *push* :: 'a ⇒ 'a idle ⇒ 'a idle **where**
push x (Idle *stack* *stackSize*) = Idle (Stack.push x *stack*) (Suc *stackSize*)

fun *pop* :: 'a idle ⇒ ('a * 'a idle) **where**
pop (Idle *stack* *stackSize*) = (Stack.first *stack*, Idle (Stack.pop *stack*) (*stackSize* - 1))

end

6 Common

theory *Common*
imports *Current Idle*

begin

The last two phases of both deque ends during rebalancing:

Copy: Using the *step* function the new elements of this deque end are brought back into the original order.

Idle: The rebalancing of the deque end is finished.

Each phase contains a *current* state, that holds the original elements of the deque end.

```
datatype (plugins del: size)'a common-state =  
  Copy 'a current 'a list 'a list nat  
  | Idle 'a current 'a idle
```

Functions:

push, *pop*: Add and remove elements using the *current* state.

step: Executes one step of the rebalancing, while keeping the invariant.

```
fun normalize :: 'a common-state ⇒ 'a common-state where  
  normalize (Copy current old new moved) = (  
    case current of Current extra added - remained ⇒  
      if moved ≥ remained  
      then Idle current (idle.Idle (Stack extra new) (added + moved))  
      else Copy current old new moved  
  )
```

instantiation common-state ::(type) step

begin

```
fun step-common-state :: 'a common-state ⇒ 'a common-state where  
  step (Idle current idle) = Idle current idle  
| step (Copy current aux new moved) = (  
  case current of Current - - - remained ⇒  
    normalize (  
      if moved < remained  
      then Copy current (tl aux) ((hd aux)#new) (moved + 1)  
      else Copy current aux new moved  
    )  
  )
```

instance⟨proof⟩

end

```
fun push :: 'a ⇒ 'a common-state ⇒ 'a common-state where  
  push x (Idle current (idle.Idle stack stackSize)) =
```

```

    Idle (Current.push x current) (idle.Idle (Stack.push x stack) (Suc stackSize))
  | push x (Copy current aux new moved) = Copy (Current.push x current) aux new
  moved

```

```

fun pop :: 'a common-state ⇒ 'a * 'a common-state where
  pop (Idle current idle) = (let (x, idle) = Idle.pop idle in (x, Idle (drop-first
  current) idle))
  | pop (Copy current aux new moved) =
    (first current, normalize (Copy (drop-first current) aux new moved))

```

end

7 Bigger End of Deque

```

theory Big
imports Common
begin

```

The bigger end of the deque during rebalancing can be in two phases:

Big1: Using the *step* function the originally contained elements, which will be kept in this end, are reversed.

Big2: Specified in theory *Common*. Is used to reverse the elements from the previous phase again to get them in the original order.

Each phase contains a *current* state, which holds the original elements of the deque end.

```

datatype (plugins del: size) 'a big-state =
  Big1 'a current 'a stack 'a list nat
  | Big2 'a common-state

```

Functions:

push, pop: Add and remove elements using the *current* state.

step: Executes one step of the rebalancing

```

instantiation big-state ::(type) step
begin

```

```

fun step-big-state :: 'a big-state ⇒ 'a big-state where
  step (Big2 state) = Big2 (step state)
  | step (Big1 current - aux 0) = Big2 (normalize (Copy current aux [] 0))
  | step (Big1 current big aux count) =
    Big1 current (Stack.pop big) ((Stack.first big)#aux) (count - 1)

```

```

instance<proof>

```

end

```
fun push :: 'a ⇒ 'a big-state ⇒ 'a big-state where  
  push x (Big2 state) = Big2 (Common.push x state)  
| push x (Big1 current big aux count) = Big1 (Current.push x current) big aux  
  count
```

```
fun pop :: 'a big-state ⇒ 'a * 'a big-state where  
  pop (Big2 state) = (let (x, state) = Common.pop state in (x, Big2 state))  
| pop (Big1 current big aux count) =  
  (first current, Big1 (drop-first current) big aux count)
```

end

8 Smaller End of Deque

```
theory Small  
imports Common  
begin
```

The smaller end of the deque during *Rebalancing* can be in one three phases:

Small1: Using the *step* function the originally contained elements are reversed.

Small2: Using the *step* function the newly obtained elements from the bigger end are reversed on top of the ones reversed in the previous phase.

Small3: See theory *Common*. Is used to reverse the elements from the two previous phases again to get them again in the original order.

Each phase contains a *current* state, which holds the original elements of the deque end.

```
datatype (plugins del: size) 'a small-state =  
  Small1 'a current 'a stack 'a list  
| Small2 'a current 'a list 'a stack 'a list nat  
| Small3 'a common-state
```

Functions:

push, *pop*: Add and remove elements using the *current* state.

step: Executes one step of the rebalancing, while keeping the invariant.

```
instantiation small-state::(type) step  
begin
```

```

fun step-small-state :: 'a small-state ⇒ 'a small-state where
  step (Small3 state) = Small3 (step state)
| step (Small1 current small auxS) = (
  if is-empty small
  then Small1 current small auxS
  else Small1 current (Stack.pop small) ((Stack.first small)#auxS)
)
| step (Small2 current auxS big newS count) = (
  if is-empty big
  then Small3 (normalize (Copy current auxS newS count))
  else Small2 current auxS (Stack.pop big) ((Stack.first big)#newS) (count + 1)
)

```

```

instance⟨proof⟩
end

```

```

fun push :: 'a ⇒ 'a small-state ⇒ 'a small-state where
  push x (Small3 state) = Small3 (Common.push x state)
| push x (Small1 current small auxS) = Small1 (Current.push x current) small
  auxS
| push x (Small2 current auxS big newS count) =
  Small2 (Current.push x current) auxS big newS count

```

```

fun pop :: 'a small-state ⇒ 'a * 'a small-state where
  pop (Small3 state) = (
    let (x, state) = Common.pop state
    in (x, Small3 state)
  )
| pop (Small1 current small auxS) =
  (first current, Small1 (drop-first current) small auxS)
| pop (Small2 current auxS big newS count) =
  (first current, Small2 (drop-first current) auxS big newS count)

```

```

end

```

9 Combining Big and Small

```

theory States
imports Big Small
begin

```

```

datatype direction = Left | Right

```

```

datatype 'a states = States direction 'a big-state 'a small-state

```

```

instantiation states::(type) step
begin

```

```

fun step-states :: 'a states ⇒ 'a states where

```

$$\begin{aligned} & \text{step (States dir (Big1 currentB big auxB 0) (Small1 currentS - auxS))} = \\ & \text{States dir (step (Big1 currentB big auxB 0)) (Small2 currentS auxS big [] 0)} \\ | & \text{step (States dir left right) = States dir (step left) (step right)} \end{aligned}$$

instance \langle *proof* \rangle
end

end

10 Real-Time Deque Implementation

theory *RealTimeDeque*
imports *States*
begin

The real-time deque can be in the following states:

Empty: No values stored. No dequeue operation possible.

One: One element in the deque.

Two: Two elements in the deque.

Three: Three elements in the deque.

Idles: Deque with a left and a right end, fulfilling the following invariant:

- $3 * \text{size of left end} \geq \text{size of right end}$
- $3 * \text{size of right end} \geq \text{size of left end}$
- Neither of the ends is empty

Rebal: Deque which violated the invariant of the *Idles* state by non-balanced dequeue and enqueue operations. The invariants during in this state are:

- The rebalancing is not done yet. The deque needs to be in *Idles* state otherwise.
- The rebalancing is in a valid state (Defined in theory *States*)
- The two ends of the deque are in a size window, such that after finishing the rebalancing the invariant of the *Idles* state will be met.

Functions:

is-empty: Checks if a deque is in the *Empty* state

deqL': Dequeues an element on the left end and return the element and the deque without this element. If the deque is in *idle* state and the size invariant is violated either a *rebalancing* is started or if there are 3 or less elements left the respective states are used. On *rebalancing* start, six steps are executed initially. During *rebalancing* state four steps are executed and if it is finished the deque returns to *idle* state.

deqL: Removes one element on the left end and only returns the new deque.

firstL: Removes one element on the left end and only returns the element.

enqL: Enqueues an element on the left and returns the resulting deque. Like in *deqL'* when violating the size invariant in *idle* state, a *rebalancing* with six initial steps is started. During *rebalancing* state four steps are executed and if it is finished the deque returns to *idle* state.

swap: The two ends of the deque are swapped.

deqR', *deqR*, *firstR*, *enqR*: Same behaviour as the left-counterparts. Implemented using the left-counterparts by swapping the deque before and after the operation.

listL, *listR*: Get all elements of the deque in a list starting at the left or right end. They are needed as list abstractions for the correctness proofs.

```

datatype 'a deque =
  Empty
  | One 'a
  | Two 'a 'a
  | Three 'a 'a 'a
  | Idles 'a idle 'a idle
  | Rebal 'a states

definition empty where
  empty = Empty

instantiation deque::(type) is-empty
begin

fun is-empty-deque :: 'a deque  $\Rightarrow$  bool where
  is-empty-deque Empty = True
  | is-empty-deque - = False

instance<proof>
end

fun swap :: 'a deque  $\Rightarrow$  'a deque where

```

```

    swap Empty = Empty
  | swap (One x) = One x
  | swap (Two x y) = Two y x
  | swap (Three x y z) = Three z y x
  | swap (Idles left right) = Idles right left
  | swap (Rebal (States Left big small)) = (Rebal (States Right big small))
  | swap (Rebal (States Right big small)) = (Rebal (States Left big small))

```

fun *small-deque* :: 'a list ⇒ 'a list ⇒ 'a deque **where**

```

  small-deque [] [] = Empty

```

```

  | small-deque (x#[]) [] = One x
  | small-deque [] (x#[]) = One x

```

```

  | small-deque (x#[])(y#[]) = Two y x
  | small-deque (x#y#[]) [] = Two y x
  | small-deque [] (x#y#[]) = Two y x

```

```

  | small-deque [] (x#y#z#[]) = Three z y x
  | small-deque (x#y#z#[]) [] = Three z y x
  | small-deque (x#y#[]) (z#[]) = Three z y x
  | small-deque (x#[]) (y#z#[]) = Three z y x

```

fun *deqL'* :: 'a deque ⇒ 'a * 'a deque **where**

```

  deqL' (One x) = (x, Empty)
  | deqL' (Two x y) = (x, One y)
  | deqL' (Three x y z) = (x, Two y z)
  | deqL' (Idles left (idle.Idle right length-right)) = (
    case Idle.pop left of (x, (idle.Idle left length-left)) ⇒
      if 3 * length-left ≥ length-right
    then
      (x, Idles (idle.Idle left length-left) (idle.Idle right length-right))
    else if length-left ≥ 1
    then
      let length-left' = 2 * length-left + 1 in
      let length-right' = length-right - length-left - 1 in

      let small = Small1 (Current [] 0 left length-left') left [] in
      let big = Big1 (Current [] 0 right length-right') right [] length-right' in

      let states = States Left big small in
      let states = (step6) states in

      (x, Rebal states)
    else
      case right of Stack r1 r2 ⇒ (x, small-deque r1 r2)
  )
  | deqL' (Rebal (States Left big small)) = (
    let (x, small) = Small.pop small in

```

```

    let states = (step~4) (States Left big small) in
  case states of
    States Left
      (Big2 (Common.Idle - big))
      (Small3 (Common.Idle - small))
      ⇒ (x, Idles small big)
    | - ⇒ (x, Rebal states)
  )
| deqL' (Rebal (States Right big small)) = (
  let (x, big) = Big.pop big in
  let states = (step~4) (States Right big small) in
  case states of
    States Right
      (Big2 (Common.Idle - big))
      (Small3 (Common.Idle - small)) ⇒
      (x, Idles big small)
    | - ⇒ (x, Rebal states)
  )
)

fun deqR' :: 'a deque ⇒ 'a * 'a deque where
  deqR' deque = (
    let (x, deque) = deqL' (swap deque)
    in (x, swap deque)
  )

fun deqL :: 'a deque ⇒ 'a deque where
  deqL deque = (let (-, deque) = deqL' deque in deque)

fun deqR :: 'a deque ⇒ 'a deque where
  deqR deque = (let (-, deque) = deqR' deque in deque)

fun firstL :: 'a deque ⇒ 'a where
  firstL deque = (let (x, -) = deqL' deque in x)

fun firstR :: 'a deque ⇒ 'a where
  firstR deque = (let (x, -) = deqR' deque in x)

fun enqL :: 'a ⇒ 'a deque ⇒ 'a deque where
  enqL x Empty = One x
| enqL x (One y) = Two x y
| enqL x (Two y z) = Three x y z
| enqL x (Three a b c) = Idles (idle.Idle (Stack [x, a] []) 2) (idle.Idle (Stack [c, b]
[]) 2)
| enqL x (Idles left (idle.Idle right length-right)) = (
  case Idle.push x left of idle.Idle left length-left ⇒
  if 3 * length-right ≥ length-left
  then
    Idles (idle.Idle left length-left) (idle.Idle right length-right)
  else

```

```

    let length-left = length-left - length-right - 1 in
    let length-right = 2 * length-right + 1 in

    let big = Big1 (Current [] 0 left length-left) left [] length-left in
    let small = Small1 (Current [] 0 right length-right) right [] in

    let states = States Right big small in
    let states = (step6) states in

    Rebal states
  )
| enqL x (Rebal (States Left big small)) = (
  let small = Small.push x small in
  let states = (step4) (States Left big small) in
  case states of
    States Left
      (Big2 (Common.Idle - big))
      (Small3 (Common.Idle - small))
      ⇒ Idles small big
  | - ⇒ Rebal states
)
| enqR x (Rebal (States Right big small)) = (
  let big = Big.push x big in
  let states = (step4) (States Right big small) in
  case states of
    States Right
      (Big2 (Common.Idle - big))
      (Small3 (Common.Idle - small))
      ⇒ Idles big small
  | - ⇒ Rebal states
)
)

fun enqR :: 'a ⇒ 'a deque ⇒ 'a deque where
  enqR x deque = (
    let deque = enqL x (swap deque)
    in swap deque
  )

```

```

end
theory Stack-Aux
imports Stack
begin

```

The function *list* appends the two lists and is needed for the list abstraction of the deque.

```

fun list :: 'a stack ⇒ 'a list where
  list (Stack left right) = left @ right

```

```

instantiation stack ::(type) size

```

```

begin

fun size-stack :: 'a stack ⇒ nat where
  size (Stack left right) = length left + length right

instance⟨proof⟩
end

end
theory Current-Aux
imports Current Stack-Aux
begin

  Specification functions:

  list: list abstraction for the originally contained elements of a deque end
    during transformation.

  invar: Is the stored number of newly added elements correct?

  size: The number of the originally contained elements.

  size-new: Number of elements which will be contained after the transfor-
    mation is finished.

fun list :: 'a current ⇒ 'a list where
  list (Current extra - old -) = extra @ (Stack-Aux.list old)

instantiation current::(type) invar
begin

fun invar-current :: 'a current ⇒ bool where
  invar (Current extra added - -) ⇔ length extra = added

instance⟨proof⟩
end

instantiation current::(type) size
begin

fun size-current :: 'a current ⇒ nat where
  size (Current - added old -) = added + size old

instance⟨proof⟩
end

instantiation current::(type) size-new
begin

fun size-new-current :: 'a current ⇒ nat where

```

```

    size-new (Current - added - remained) = added + remained

instance⟨proof⟩
end

end
theory Idle-Aux
imports Idle Stack-Aux
begin

fun list :: 'a idle ⇒ 'a list where
    list (Idle stack -) = Stack-Aux.list stack

instantiation idle :: (type) size
begin

fun size-idle :: 'a idle ⇒ nat where
    size (Idle stack -) = size stack

instance⟨proof⟩
end

instantiation idle :: (type) is-empty
begin

fun is-empty-idle :: 'a idle ⇒ bool where
    is-empty (Idle stack -) ⟷ is-empty stack

instance⟨proof⟩
end

instantiation idle ::(type) invar
begin

fun invar-idle :: 'a idle ⇒ bool where
    invar (Idle stack stackSize) ⟷ size stack = stackSize

instance⟨proof⟩
end

end
theory Common-Aux
imports Common Current-Aux Idle-Aux
begin

```

Functions:

list: List abstraction of the elements which this end will contain after the rebalancing is finished

list-current: List abstraction of the elements currently in this deque end.

remaining-steps: Returns how many steps are left until the rebalancing is finished.

size-new: Returns the size, that the deque end will have after the rebalancing is finished.

size: Minimum of *size-new* and the number of elements contained in the *current* state.

definition *take-rev* **where**

[*simp*]: $take\text{-}rev\ n\ xs = rev\ (take\ n\ xs)$

fun *list* :: 'a common-state \Rightarrow 'a list **where**

list (Idle - idle) = Idle-Aux.list idle

| *list* (Copy (Current extra - - remained) old new moved)
= extra @ take-rev (remained - moved) old @ new

fun *list-current* :: 'a common-state \Rightarrow 'a list **where**

list-current (Idle current -) = Current-Aux.list current

| *list-current* (Copy current - -) = Current-Aux.list current

instantiation *common-state::(type) invar*

begin

fun *invar-common-state* :: 'a common-state \Rightarrow bool **where**

invar (Idle current idle) \longleftrightarrow

invar idle

\wedge *invar* current

\wedge *size-new* current = *size* idle

\wedge take (*size* idle) (Current-Aux.list current) =

take (*size* current) (Idle-Aux.list idle)

| *invar* (Copy current aux new moved) \longleftrightarrow (
case current of Current - - old remained \Rightarrow

moved < *remained*

\wedge *moved* = length new

\wedge *remained* \leq length aux + *moved*

\wedge *invar* current

\wedge take *remained* (Stack-Aux.list old) = take (*size* old) (take-rev (*remained* -
moved) aux @ new)

)

instance⟨*proof*⟩

end

instantiation *common-state::(type) size*

begin

```

fun size-common-state :: 'a common-state ⇒ nat where
  size (Idle current idle) = min (size current) (size idle)
| size (Copy current - -) = min (size current) (size-new current)

instance⟨proof⟩
end

instantiation common-state::(type) size-new
begin

fun size-new-common-state :: 'a common-state ⇒ nat where
  size-new (Idle current -) = size-new current
| size-new (Copy current - -) = size-new current

instance⟨proof⟩
end

instantiation common-state::(type) remaining-steps
begin

fun remaining-steps-common-state :: 'a common-state ⇒ nat where
  remaining-steps (Idle -) = 0
| remaining-steps (Copy (Current - - - remained) aux new moved) = remained -
  moved

instance⟨proof⟩
end

end
theory Big-Aux
imports Big Common-Aux
begin

```

Functions:

size-new: Returns the size that the deque end will have after the rebalancing is finished.

size: Minimum of *size-new* and the number of elements contained in the current state.

remaining-steps: Returns how many steps are left until the rebalancing is finished.

list: List abstraction of the elements which this end will contain after the rebalancing is finished

list-current: List abstraction of the elements currently in this deque end.

```

fun list :: 'a big-state ⇒ 'a list where
  list (Big2 common) = Common-Aux.list common
| list (Big1 (Current extra - - remained) big aux count) = (
  let reversed = take-rev count (Stack-Aux.list big) @ aux in
  extra @ (take-rev remained reversed)
)

fun list-current :: 'a big-state ⇒ 'a list where
  list-current (Big2 common) = Common-Aux.list-current common
| list-current (Big1 current - -) = Current-Aux.list current

instantiation big-state ::(type) invar
begin

fun invar-big-state :: 'a big-state ⇒ bool where
  invar (Big2 state) ↔ invar state
| invar (Big1 current big aux count) ↔ (
  case current of Current extra added old remained ⇒
  invar current
  ∧ remained ≤ length aux + count
  ∧ count ≤ size big
  ∧ Stack-Aux.list old = rev (take (size old) ((rev (Stack-Aux.list big)) @ aux))
  ∧ take remained (Stack-Aux.list old) =
  rev (take remained (take-rev count (Stack-Aux.list big) @ aux))
)

instance(proof)
end

instantiation big-state ::(type) size
begin

fun size-big-state :: 'a big-state ⇒ nat where
  size (Big2 state) = size state
| size (Big1 current - -) = min (size current) (size-new current)

instance(proof)
end

instantiation big-state ::(type) size-new
begin

fun size-new-big-state :: 'a big-state ⇒ nat where
  size-new (Big2 state) = size-new state
| size-new (Big1 current - -) = size-new current

instance(proof)
end

```

instantiation *big-state* ::(type) *remaining-steps*
begin

fun *remaining-steps-big-state* :: 'a *big-state* \Rightarrow nat **where**
remaining-steps (*Big2 state*) = *remaining-steps state*
| *remaining-steps* (*Big1 (Current - - - remaining) - - count*) = *count + remaining*
+ 1

instance(*proof*)
end

end
theory *Small-Aux*
imports *Small Common-Aux*
begin

Functions:

size-new: Returns the size, that the deque end will have after the rebalancing is finished.

size: Minimum of *size-new* and the number of elements contained in the 'current' state.

list: List abstraction of the elements which this end will contain after the rebalancing is finished. The first phase is not covered, since the elements, which will be transferred from the bigger deque end are not known yet.

list-current: List abstraction of the elements currently in this deque end.

fun *list* :: 'a *small-state* \Rightarrow 'a *list* **where**
list (*Small3 common*) = *Common-Aux.list common*
| *list* (*Small2 (Current extra - - remained) aux big new count*) =
extra @ (take-rev (remained - (count + size big)) aux) @ (rev (Stack-Aux.list big) @ new)

fun *list-current* :: 'a *small-state* \Rightarrow 'a *list* **where**
list-current (*Small3 common*) = *Common-Aux.list-current common*
| *list-current* (*Small2 current - - - -*) = *Current-Aux.list current*
| *list-current* (*Small1 current - -*) = *Current-Aux.list current*

instantiation *small-state*::(type) *invar*
begin

fun *invar-small-state* :: 'a *small-state* \Rightarrow bool **where**
invar (*Small3 state*) = *invar state*
| *invar* (*Small2 current auxS big newS count*) = (
case current of Current - - old remained \Rightarrow

```

    remained = count + size big + size old
  ^ count = List.length newS
  ^ invar current
  ^ List.length auxS ≥ size old
  ^ Stack-Aux.list old = rev (take (size old) auxS)
)
| invar (Small1 current small auxS) = (
  case current of Current - - old remained ⇒
    invar current
  ^ remained ≥ size old
  ^ size small + List.length auxS ≥ size old
  ^ Stack-Aux.list old = rev (take (size old) (rev (Stack-Aux.list small) @ auxS))
)

```

```

instance⟨proof⟩
end

```

```

instantiation small-state::(type) size
begin

```

```

fun size-small-state :: 'a small-state ⇒ nat where
  size (Small3 state) = size state
| size (Small2 current - - -) = min (size current) (size-new current)
| size (Small1 current - -) = min (size current) (size-new current)

```

```

instance⟨proof⟩
end

```

```

instantiation small-state::(type) size-new
begin

```

```

fun size-new-small-state :: 'a small-state ⇒ nat where
  size-new (Small3 state) = size-new state
| size-new (Small2 current - - -) = size-new current
| size-new (Small1 current - -) = size-new current

```

```

instance⟨proof⟩
end

```

```

end
theory States-Aux
imports States Big-Aux Small-Aux
begin

```

```

instantiation states::(type) remaining-steps
begin

```

```

fun remaining-steps-states :: 'a states ⇒ nat where
  remaining-steps (States - big small) = max

```

```

    (remaining-steps big)
    (case small of
      Small3 common  $\Rightarrow$  remaining-steps common
    | Small2 (Current - - - remaining) - big - count  $\Rightarrow$  remaining - count + 1
    | Small1 (Current - - - remaining) - -  $\Rightarrow$ 
      case big of Big1 currentB big auxB count  $\Rightarrow$  remaining + count + 2
    )

instance(proof)
end

fun lists :: 'a states  $\Rightarrow$  'a list * 'a list where
  lists (States - (Big1 currentB big auxB count) (Small1 currentS small auxS)) = (
    Big-Aux.list (Big1 currentB big auxB count),
    Small-Aux.list (Small2 currentS (take-rev count (Stack-Aux.list small)) @ auxS)
  )
  ((Stack.pop  $\sim\sim$  count) big) [] 0
| lists (States - big small) = (Big-Aux.list big, Small-Aux.list small)

fun list-small-first :: 'a states  $\Rightarrow$  'a list where
  list-small-first states = (let (big, small) = lists states in small @ (rev big))

fun list-big-first :: 'a states  $\Rightarrow$  'a list where
  list-big-first states = (let (big, small) = lists states in big @ (rev small))

fun lists-current :: 'a states  $\Rightarrow$  'a list * 'a list where
  lists-current (States - big small) = (Big-Aux.list-current big, Small-Aux.list-current
small)

fun list-current-small-first :: 'a states  $\Rightarrow$  'a list where
  list-current-small-first states = (let (big, small) = lists-current states in small @
(rev big))

fun list-current-big-first :: 'a states  $\Rightarrow$  'a list where
  list-current-big-first states = (let (big, small) = lists-current states in big @ (rev
small))

fun listL :: 'a states  $\Rightarrow$  'a list where
  listL (States Left big small) = list-small-first (States Left big small)
| listL (States Right big small) = list-big-first (States Right big small)

instantiation states::(type) invar
begin

fun invar-states :: 'a states  $\Rightarrow$  bool where
  invar (States dir big small)  $\longleftrightarrow$  (
    invar big
   $\wedge$  invar small
   $\wedge$  list-small-first (States dir big small) = list-current-small-first (States dir big

```

```

small)
  ∧ (case (big, small) of
    (Big1 - big - count, Small1 (Current - - old remained) small -) ⇒
      size big - count = remained - size old ∧ count ≥ size small
    | (-, Small1 - - -) ⇒ False
    | (Big1 - - - -, -) ⇒ False
    | - ⇒ True
  ))

```

```

instance⟨proof⟩
end

```

```

fun size-ok' :: 'a states ⇒ nat ⇒ bool where
  size-ok' (States - big small) steps ↔
    size-new small + steps + 2 ≤ 3 * size-new big
  ∧ size-new big + steps + 2 ≤ 3 * size-new small
  ∧ steps + 1 ≤ 4 * size small
  ∧ steps + 1 ≤ 4 * size big

```

```

abbreviation size-ok :: 'a states ⇒ bool where
  size-ok states ≡ size-ok' states (remaining-steps states)

```

```

abbreviation size-small where size-small states ≡ case states of States - - small
⇒ size small

```

```

abbreviation size-new-small where
  size-new-small states ≡ case states of States - - small ⇒ size-new small

```

```

abbreviation size-big where size-big states ≡ case states of States - big - ⇒ size
big

```

```

abbreviation size-new-big where
  size-new-big states ≡ case states of States - big - ⇒ size-new big

```

```

end
theory RealTimeDeque-Aux
  imports RealTimeDeque States-Aux
begin

```

listL, *listR*: Get all elements of the deque in a list starting at the left or right end. They are needed as list abstractions for the correctness proofs.

```

fun listL :: 'a deque ⇒ 'a list where
  listL Empty = []
| listL (One x) = [x]
| listL (Two x y) = [x, y]

```

```

| listL (Three x y z) = [x, y, z]
| listL (Idles left right) = Idle-Aux.list left @ (rev (Idle-Aux.list right))
| listL (Rebal states) = States-Aux.listL states

```

abbreviation *listR* :: 'a deque ⇒ 'a list **where**
listR deque ≡ rev (*listL deque*)

instantiation *deque*::(type) *invar*
begin

fun *invar-deque* :: 'a deque ⇒ bool **where**

```

  invar Empty = True
| invar (One -) = True
| invar (Two - -) = True
| invar (Three - - -) = True
| invar (Idles left right) ↔
  invar left ∧
  invar right ∧
  ¬ is-empty left ∧
  ¬ is-empty right ∧
  3 * size right ≥ size left ∧
  3 * size left ≥ size right

```

```

| invar (Rebal states) ↔
  invar states ∧
  size-ok states ∧
  0 < remaining-steps states

```

instance⟨*proof*⟩
end

end

11 Basic Lemma Library

theory *RTD-Util*
imports *Main*
begin

lemma *take-last-length*: $\llbracket \text{take } (\text{Suc } 0) (\text{rev } xs) = [\text{last } xs] \rrbracket \implies \text{Suc } 0 \leq \text{length } xs$
 ⟨*proof*⟩

lemma *take-last*: $xs \neq [] \implies \text{take } 1 (\text{rev } xs) = [\text{last } xs]$
 ⟨*proof*⟩

lemma *take-hd* [*simp*]: $xs \neq [] \implies \text{take } (\text{Suc } 0) xs = [\text{hd } xs]$
 ⟨*proof*⟩

lemma *cons-tl*: $x \# xs = ys \implies xs = tl\ ys$
<proof>

lemma *cons-hd*: $x \# xs = ys \implies x = hd\ ys$
<proof>

lemma *take-hd'*: $ys \neq [] \implies take\ (size\ ys)\ (x \# xs) = take\ (Suc\ (size\ xs))\ ys \implies$
 $hd\ ys = x$
<proof>

lemma *rev-app-single*: $rev\ xs\ @\ [x] = rev\ (x \# xs)$
<proof>

lemma *hd-drop-1* [*simp*]: $xs \neq [] \implies hd\ xs \# drop\ (Suc\ 0)\ xs = xs$
<proof>

lemma *hd-drop* [*simp*]: $n < length\ xs \implies hd\ (drop\ n\ xs) \# drop\ (Suc\ n)\ xs =$
 $drop\ n\ xs$
<proof>

lemma *take-1*: $0 < x \wedge 0 < y \implies take\ x\ xs = take\ y\ ys \implies take\ 1\ xs = take\ 1$
 ys
<proof>

lemma *last-drop-rev*: $xs \neq [] \implies last\ xs \# drop\ 1\ (rev\ xs) = rev\ xs$
<proof>

lemma *Suc-min* [*simp*]: $0 < x \implies 0 < y \implies Suc\ (min\ (x - Suc\ 0)\ (y - Suc\ 0)) = min\ x\ y$
<proof>

lemma *rev-tl-hd*: $xs \neq [] \implies rev\ (tl\ xs) \ @\ [hd\ xs] = rev\ xs$
<proof>

lemma *app-rev*: $as\ @\ rev\ bs = cs\ @\ rev\ ds \implies bs\ @\ rev\ as = ds\ @\ rev\ cs$
<proof>

lemma *tl-drop-2*: $tl\ (drop\ n\ xs) = drop\ (Suc\ n)\ xs$
<proof>

lemma *Suc-sub*: $Suc\ n = m \implies n = m - 1$
<proof>

lemma *length-one-hd*: $length\ xs = 1 \implies xs = [hd\ xs]$
<proof>

end

12 Stack Proofs

```
theory Stack-Proof
imports Stack-Aux RTD-Util
begin
```

```
lemma push-list [simp]: list (push x stack) = x # list stack
  <proof>
```

```
lemma pop-list [simp]: list (pop stack) = tl (list stack)
  <proof>
```

```
lemma first-list [simp]:  $\neg$  is-empty stack  $\implies$  first stack = hd (list stack)
  <proof>
```

```
lemma list-empty: list stack = []  $\longleftrightarrow$  is-empty stack
  <proof>
```

```
lemma list-not-empty: list stack  $\neq$  []  $\longleftrightarrow$   $\neg$  is-empty stack
  <proof>
```

```
lemma list-empty-2 [simp]:  $\llbracket$ list stack  $\neq$  []; is-empty stack $\rrbracket \implies$  False
  <proof>
```

```
lemma list-not-empty-2 [simp]:  $\llbracket$ list stack = [];  $\neg$  is-empty stack $\rrbracket \implies$  False
  <proof>
```

```
lemma list-empty-size: list stack = []  $\longleftrightarrow$  size stack = 0
  <proof>
```

```
lemma list-not-empty-size: list stack  $\neq$  []  $\longleftrightarrow$  0 < size stack
  <proof>
```

```
lemma list-empty-size-2 [simp]:  $\llbracket$ list stack  $\neq$  []; size stack = 0 $\rrbracket \implies$  False
  <proof>
```

```
lemma list-not-empty-size-2 [simp]:  $\llbracket$ list stack = []; 0 < size stack $\rrbracket \implies$  False
  <proof>
```

```
lemma size-push [simp]: size (push x stack) = Suc (size stack)
  <proof>
```

```
lemma size-pop [simp]: size (pop stack) = size stack - Suc 0
  <proof>
```

```
lemma size-empty: size (stack :: 'a stack) = 0  $\longleftrightarrow$  is-empty stack
  <proof>
```

```
lemma size-not-empty: size (stack :: 'a stack) > 0  $\longleftrightarrow$   $\neg$  is-empty stack
```

<proof>

lemma *size-empty-2* [simp]: $\llbracket \text{size } (stack :: 'a \text{ stack}) = 0; \neg \text{is-empty stack} \rrbracket \implies \text{False}$
<proof>

lemma *size-not-empty-2* [simp]: $\llbracket 0 < \text{size } (stack :: 'a \text{ stack}); \text{is-empty stack} \rrbracket \implies \text{False}$
<proof>

lemma *size-list-length* [simp]: $\text{length } (list \text{ stack}) = \text{size stack}$
<proof>

lemma *first-pop* [simp]: $\neg \text{is-empty stack} \implies \text{first stack} \# \text{list } (pop \text{ stack}) = \text{list stack}$
<proof>

lemma *push-not-empty* [simp]: $\llbracket \neg \text{is-empty stack}; \text{is-empty } (push \ x \ \text{stack}) \rrbracket \implies \text{False}$
<proof>

lemma *pop-list-length* [simp]: $\neg \text{is-empty stack} \implies \text{Suc } (\text{length } (list \ (pop \ \text{stack}))) = \text{length } (list \ \text{stack})$
<proof>

lemma *first-take*: $\neg \text{is-empty stack} \implies [\text{first stack}] = \text{take } 1 \ (list \ \text{stack})$
<proof>

lemma *first-take-tl* [simp]: $0 < \text{size big} \implies (\text{first big} \# \text{take count } (tl \ (list \ big))) = \text{take } (\text{Suc count}) \ (list \ big)$
<proof>

lemma *first-take-pop* [simp]: $\llbracket \neg \text{is-empty stack}; 0 < x \rrbracket \implies \text{first stack} \# \text{take } (x - \text{Suc } 0) \ (list \ (pop \ \text{stack})) = \text{take } x \ (list \ \text{stack})$
<proof>

lemma [simp]: $\text{first } (Stack \ [] \ []) = \text{undefined}$
<proof>

lemma *first-hd*: $\text{first stack} = \text{hd } (list \ \text{stack})$
<proof>

lemma *pop-tl* [simp]: $\text{list } (pop \ \text{stack}) = \text{tl } (list \ \text{stack})$
<proof>

lemma *pop-drop*: $\text{list } (pop \ \text{stack}) = \text{drop } 1 \ (list \ \text{stack})$
<proof>

lemma *popN-drop* [simp]: $\text{list } ((pop \ \sim n) \ \text{stack}) = \text{drop } n \ (list \ \text{stack})$

<proof>

lemma *popN-size* [*simp*]: $\text{size } ((\text{pop } \sim n) \text{ stack}) = (\text{size } \text{stack}) - n$
<proof>

lemma *take-first*: $[[0 < \text{size } s1; 0 < \text{size } s2; \text{take } (\text{size } s1) (\text{list } s2) = \text{take } (\text{size } s2) (\text{list } s1)]]$
 $\implies \text{first } s1 = \text{first } s2$
<proof>

end

13 Idle Proofs

theory *Idle-Proof*
imports *Idle-Aux Stack-Proof*
begin

lemma *push-list* [*simp*]: $\text{list } (\text{push } x \text{ idle}) = x \# \text{list } \text{idle}$
<proof>

lemma *pop-list* [*simp*]: $[[\neg \text{is-empty } \text{idle}; \text{pop } \text{idle} = (x, \text{idle}')] \implies x \# \text{list } \text{idle}' = \text{list } \text{idle}$
<proof>

lemma *pop-list-tl* [*simp*]:
 $[[\neg \text{is-empty } \text{idle}; \text{pop } \text{idle} = (x, \text{idle}')] \implies x \# (\text{tl } (\text{list } \text{idle})) = \text{list } \text{idle}$
<proof>

lemma *pop-list-tl'* [*simp*]: $[[\text{pop } \text{idle} = (x, \text{idle}')] \implies \text{list } \text{idle}' = \text{tl } (\text{list } \text{idle})$
<proof>

lemma *size-push* [*simp*]: $\text{size } (\text{push } x \text{ idle}) = \text{Suc } (\text{size } \text{idle})$
<proof>

lemma *size-pop* [*simp*]: $[[\neg \text{is-empty } \text{idle}; \text{pop } \text{idle} = (x, \text{idle}')] \implies \text{Suc } (\text{size } \text{idle}') = \text{size } \text{idle}$
<proof>

lemma *size-pop-sub*: $[[\text{pop } \text{idle} = (x, \text{idle}')] \implies \text{size } \text{idle}' = \text{size } \text{idle} - 1$
<proof>

lemma *invar-push*: $\text{invar } \text{idle} \implies \text{invar } (\text{push } x \text{ idle})$
<proof>

lemma *invar-pop*: $[[\text{invar } \text{idle}; \text{pop } \text{idle} = (x, \text{idle}')] \implies \text{invar } \text{idle}'$
<proof>

lemma *size-empty*: $\text{size } \text{idle} = 0 \iff \text{is-empty } (\text{idle} :: 'a \text{ idle})$

$\langle proof \rangle$

lemma *size-not-empty*: $0 < size\ idle \longleftrightarrow \neg is_empty\ (idle :: 'a\ idle)$
 $\langle proof \rangle$

lemma *size-empty-2* [*simp*]: $\llbracket \neg is_empty\ (idle :: 'a\ idle); 0 = size\ idle \rrbracket \implies False$
 $\langle proof \rangle$

lemma *size-not-empty-2* [*simp*]: $\llbracket is_empty\ (idle :: 'a\ idle); 0 < size\ idle \rrbracket \implies False$
 $\langle proof \rangle$

lemma *list-empty*: $list\ idle = [] \longleftrightarrow is_empty\ idle$
 $\langle proof \rangle$

lemma *list-not-empty*: $list\ idle \neq [] \longleftrightarrow \neg is_empty\ idle$
 $\langle proof \rangle$

lemma *list-empty-2* [*simp*]: $\llbracket list\ idle = []; \neg is_empty\ (idle :: 'a\ idle) \rrbracket \implies False$
 $\langle proof \rangle$

lemma *list-not-empty-2* [*simp*]: $\llbracket list\ idle \neq []; is_empty\ (idle :: 'a\ idle) \rrbracket \implies False$
 $\langle proof \rangle$

lemma *list-empty-size*: $list\ idle = [] \longleftrightarrow 0 = size\ idle$
 $\langle proof \rangle$

lemma *list-not-empty-size*: $list\ idle \neq [] \longleftrightarrow 0 < size\ idle$
 $\langle proof \rangle$

lemma *list-empty-size-2* [*simp*]: $\llbracket list\ idle \neq []; 0 = size\ idle \rrbracket \implies False$
 $\langle proof \rangle$

lemma *list-not-empty-size-2* [*simp*]: $\llbracket list\ idle = []; 0 < size\ idle \rrbracket \implies False$
 $\langle proof \rangle$

end

14 Current Proofs

theory *Current-Proof*
imports *Current-Aux Stack-Proof*
begin

lemma *push-list* [*simp*]: $list\ (push\ x\ current) = x \# list\ current$
 $\langle proof \rangle$

lemma *pop-list* [*simp*]:
 $\llbracket 0 < size\ current; invar\ current \rrbracket \implies fst\ (pop\ current) \# tl\ (list\ current) = list$

current
⟨proof⟩

lemma *drop-first-list* [simp]: $\llbracket \text{invar } \text{current}; 0 < \text{size } \text{current} \rrbracket$
 $\implies \text{list } (\text{drop-first } \text{current}) = \text{tl } (\text{list } \text{current})$
⟨proof⟩

lemma *invar-push*: $\text{invar } \text{current} \implies \text{invar } (\text{push } x \text{ current})$
⟨proof⟩

lemma *invar-pop*: $\llbracket 0 < \text{size } \text{current}; \text{invar } \text{current}; \text{pop } \text{current} = (x, \text{current}') \rrbracket$
 $\implies \text{invar } \text{current}'$
⟨proof⟩

lemma *invar-drop-first*: $\llbracket 0 < \text{size } \text{current}; \text{invar } \text{current} \rrbracket \implies \text{invar } (\text{drop-first } \text{current})$
⟨proof⟩

lemma *list-size* [simp]: $\llbracket \text{invar } \text{current}; \text{list } \text{current} = []; 0 < \text{size } \text{current} \rrbracket \implies$
False
⟨proof⟩

lemma *size-new-push* [simp]: $\text{invar } \text{current} \implies \text{size-new } (\text{push } x \text{ current}) = \text{Suc}$
 $(\text{size-new } \text{current})$
⟨proof⟩

lemma *size-push* [simp]: $\text{size } (\text{push } x \text{ current}) = \text{Suc } (\text{size } \text{current})$
⟨proof⟩

lemma *size-new-pop* [simp]: $\llbracket 0 < \text{size-new } \text{current}; \text{invar } \text{current} \rrbracket$
 $\implies \text{Suc } (\text{size-new } (\text{drop-first } \text{current})) = \text{size-new } \text{current}$
⟨proof⟩

lemma *size-pop* [simp]: $\llbracket 0 < \text{size } \text{current}; \text{invar } \text{current} \rrbracket$
 $\implies \text{Suc } (\text{size } (\text{drop-first } \text{current})) = \text{size } \text{current}$
⟨proof⟩

lemma *size-pop-suc* [simp]: $\llbracket 0 < \text{size } \text{current}; \text{invar } \text{current}; \text{pop } \text{current} = (x, \text{current}') \rrbracket$
 $\implies \text{Suc } (\text{size } \text{current}') = \text{size } \text{current}$
⟨proof⟩

lemma *size-pop-sub*: $\llbracket 0 < \text{size } \text{current}; \text{invar } \text{current}; \text{pop } \text{current} = (x, \text{current}') \rrbracket$
 $\implies \text{size } \text{current}' = \text{size } \text{current} - 1$
⟨proof⟩

lemma *size-drop-first-sub*: $\llbracket 0 < \text{size } \text{current}; \text{invar } \text{current} \rrbracket$
 $\implies \text{size } (\text{drop-first } \text{current}) = \text{size } \text{current} - 1$

<proof>

end

15 Common Proofs

theory *Common-Proof*

imports *Common-Aux Idle-Proof Current-Proof*

begin

lemma *take-rev-drop*: $take\text{-}rev\ n\ xs\ @\ acc = drop\ (length\ xs - n)\ (rev\ xs)\ @\ acc$
<proof>

lemma *take-rev-step*: $xs \neq [] \implies take\text{-}rev\ n\ (tl\ xs)\ @\ (hd\ xs\ \# \ acc) = take\text{-}rev\ (Suc\ n)\ xs\ @\ acc$
<proof>

lemma *take-rev-empty* [*simp*]: $take\text{-}rev\ n\ [] = []$
<proof>

lemma *take-rev-tl-hd*:

$0 < n \implies xs \neq [] \implies take\text{-}rev\ n\ xs\ @\ ys = take\text{-}rev\ (n - (Suc\ 0))\ (tl\ xs)\ @\ (hd\ xs\ \# \ ys)$
<proof>

lemma *take-rev-nth*:

$n < length\ xs \implies x = xs\ !\ n \implies x\ \# \ take\text{-}rev\ n\ xs\ @\ ys = take\text{-}rev\ (Suc\ n)\ xs\ @\ ys$
<proof>

lemma *step-list* [*simp*]: $invar\ common \implies list\ (step\ common) = list\ common$
<proof>

lemma *step-list-current* [*simp*]: $invar\ common \implies list\text{-}current\ (step\ common) = list\text{-}current\ common$
<proof>

lemma *push-list* [*simp*]: $list\ (push\ x\ common) = x\ \# \ list\ common$
<proof>

lemma *invar-step*: $invar\ (common\ :: \ 'a\ common\text{-}state) \implies invar\ (step\ common)$
<proof>

lemma *invar-push*: $invar\ common \implies invar\ (push\ x\ common)$
<proof>

lemma *invar-pop*: [
 $0 < size\ common;$
 $invar\ common;$

$pop\ common = (x, common')$
 $\llbracket \implies invar\ common' \rrbracket$
 $\langle proof \rangle$

lemma *push-list-current* [simp]: $list-current\ (push\ x\ left) = x \# list-current\ left$
 $\langle proof \rangle$

lemma *pop-list* [simp]: $invar\ common \implies 0 < size\ common \implies pop\ common = (x, common') \implies x \# list\ common' = list\ common$
 $\langle proof \rangle$

lemma *pop-list-current*: $invar\ common \implies 0 < size\ common \implies pop\ common = (x, common') \implies x \# list-current\ common' = list-current\ common$
 $\langle proof \rangle$

lemma *list-current-size* [simp]:
 $\llbracket 0 < size\ common; list-current\ common = []; invar\ common \rrbracket \implies False$
 $\langle proof \rangle$

lemma *list-size* [simp]: $\llbracket 0 < size\ common; list\ common = []; invar\ common \rrbracket \implies False$
 $\langle proof \rangle$

lemma *step-size* [simp]: $invar\ (common :: 'a\ common-state) \implies size\ (step\ common) = size\ common$
 $\langle proof \rangle$

lemma *step-size-new* [simp]: $invar\ (common :: 'a\ common-state) \implies size-new\ (step\ common) = size-new\ common$
 $\langle proof \rangle$

lemma *remaining-steps-step* [simp]: $\llbracket invar\ (common :: 'a\ common-state); remaining-steps\ common > 0 \rrbracket \implies Suc\ (remaining-steps\ (step\ common)) = remaining-steps\ common$
 $\langle proof \rangle$

lemma *remaining-steps-step-sub* [simp]: $\llbracket invar\ (common :: 'a\ common-state) \rrbracket \implies remaining-steps\ (step\ common) = remaining-steps\ common - 1$
 $\langle proof \rangle$

lemma *remaining-steps-step-0* [simp]: $\llbracket invar\ (common :: 'a\ common-state); remaining-steps\ common = 0 \rrbracket \implies remaining-steps\ (step\ common) = 0$
 $\langle proof \rangle$

lemma *remaining-steps-push* [simp]: $invar\ common \implies remaining-steps\ (push\ x\ common) = remaining-steps\ common$

<proof>

lemma *remaining-steps-pop*: $\llbracket \text{invar } \text{common}; \text{pop } \text{common} = (x, \text{common}') \rrbracket$
 $\implies \text{remaining-steps } \text{common}' \leq \text{remaining-steps } \text{common}$
<proof>

lemma *size-push* [*simp*]: $\text{invar } \text{common} \implies \text{size } (\text{push } x \text{ common}) = \text{Suc } (\text{size } \text{common})$
<proof>

lemma *size-new-push* [*simp*]: $\text{invar } \text{common} \implies \text{size-new } (\text{push } x \text{ common}) = \text{Suc } (\text{size-new } \text{common})$
<proof>

lemma *size-pop* [*simp*]: $\llbracket \text{invar } \text{common}; 0 < \text{size } \text{common}; \text{pop } \text{common} = (x, \text{common}') \rrbracket$
 $\implies \text{Suc } (\text{size } \text{common}') = \text{size } \text{common}$
<proof>

lemma *size-new-pop* [*simp*]: $\llbracket \text{invar } \text{common}; 0 < \text{size-new } \text{common}; \text{pop } \text{common} = (x, \text{common}') \rrbracket$
 $\implies \text{Suc } (\text{size-new } \text{common}') = \text{size-new } \text{common}$
<proof>

lemma *size-size-new*: $\llbracket \text{invar } (\text{common} :: 'a \text{ common-state}); 0 < \text{size } \text{common} \rrbracket \implies 0 < \text{size-new } \text{common}$
<proof>

end

16 Big Proofs

theory *Big-Proof*
imports *Big-Aux Common-Proof*
begin

lemma *step-list* [*simp*]: $\text{invar } \text{big} \implies \text{list } (\text{step } \text{big}) = \text{list } \text{big}$
<proof>

lemma *step-list-current* [*simp*]: $\text{invar } \text{big} \implies \text{list-current } (\text{step } \text{big}) = \text{list-current } \text{big}$
<proof>

lemma *push-list* [*simp*]: $\text{list } (\text{push } x \text{ big}) = x \# \text{list } \text{big}$
<proof>

lemma *list-Big1*: $\llbracket 0 < \text{size } (\text{Big1 } \text{current } \text{big } \text{aux } \text{count}); \text{invar } (\text{Big1 } \text{current } \text{big } \text{aux } \text{count}) \rrbracket$

]] \implies $\text{first current} \# \text{list (Big1 (drop-first current) big aux count)} =$
 $\text{list (Big1 current big aux count)}$
 <proof>

lemma *size-list* [simp]: $\llbracket 0 < \text{size big}; \text{invar big}; \text{list big} = [] \rrbracket \implies \text{False}$
 <proof>

lemma *pop-list* [simp]: $\llbracket 0 < \text{size big}; \text{invar big}; \text{Big.pop big} = (x, \text{big}') \rrbracket$
 $\implies x \# \text{list big}' = \text{list big}$
 <proof>

lemma *pop-list-tl*: $\llbracket 0 < \text{size big}; \text{invar big}; \text{pop big} = (x, \text{big}') \rrbracket \implies \text{list big}' = \text{tl}$
 (list big)
 <proof>

lemma *invar-step*: $\text{invar (big :: 'a big-state)} \implies \text{invar (step big)}$
 <proof>

lemma *invar-push*: $\text{invar big} \implies \text{invar (push x big)}$
 <proof>

lemma *invar-pop*: \llbracket
 $0 < \text{size big};$
 $\text{invar big};$
 $\text{pop big} = (x, \text{big}')$
 $\rrbracket \implies \text{invar big}'$
 <proof>

lemma *push-list-current* [simp]: $\text{list-current (push x big)} = x \# \text{list-current big}$
 <proof>

lemma *pop-list-current* [simp]: $\llbracket \text{invar big}; 0 < \text{size big}; \text{Big.pop big} = (x, \text{big}') \rrbracket$
 $\implies x \# \text{list-current big}' = \text{list-current big}$
 <proof>

lemma *list-current-size*: $\llbracket 0 < \text{size big}; \text{list-current big} = []; \text{invar big} \rrbracket \implies \text{False}$
 <proof>

lemma *step-size*: $\text{invar (big :: 'a big-state)} \implies \text{size big} = \text{size (step big)}$
 <proof>

lemma *remaining-steps-step* [simp]: $\llbracket \text{invar (big :: 'a big-state)}; \text{remaining-steps big} > 0 \rrbracket$
 $\implies \text{Suc (remaining-steps (step big))} = \text{remaining-steps big}$
 <proof>

lemma *remaining-steps-step-0* [simp]: $\llbracket \text{invar (big :: 'a big-state)}; \text{remaining-steps}$

$big = 0$
 $\implies remaining-steps (step\ big) = 0$
 ⟨proof⟩

lemma *remaining-steps-push*: $invar\ big \implies remaining-steps (push\ x\ big) = remaining-steps\ big$
 ⟨proof⟩

lemma *remaining-steps-pop*: $\llbracket invar\ big; pop\ big = (x, big^{\wedge}) \rrbracket$
 $\implies remaining-steps\ big' \leq remaining-steps\ big$
 ⟨proof⟩

lemma *size-push* [simp]: $invar\ big \implies size (push\ x\ big) = Suc (size\ big)$
 ⟨proof⟩

lemma *size-new-push* [simp]: $invar\ big \implies size-new (push\ x\ big) = Suc (size-new\ big)$
 ⟨proof⟩

lemma *size-pop* [simp]: $\llbracket invar\ big; 0 < size\ big; pop\ big = (x, big^{\wedge}) \rrbracket$
 $\implies Suc (size\ big^{\wedge}) = size\ big$
 ⟨proof⟩

lemma *size-new-pop* [simp]: $\llbracket invar\ big; 0 < size-new\ big; pop\ big = (x, big^{\wedge}) \rrbracket$
 $\implies Suc (size-new\ big^{\wedge}) = size-new\ big$
 ⟨proof⟩

lemma *size-size-new*: $\llbracket invar (big :: 'a\ big-state); 0 < size\ big \rrbracket \implies 0 < size-new\ big$
 ⟨proof⟩

end

17 Small Proofs

theory *Small-Proof*
imports *Common-Proof Small-Aux*
begin

lemma *step-size* [simp]: $invar (small :: 'a\ small-state) \implies size (step\ small) = size\ small$
 ⟨proof⟩

lemma *step-size-new* [simp]:
 $invar (small :: 'a\ small-state) \implies size-new (step\ small) = size-new\ small$
 ⟨proof⟩

lemma *size-push* [simp]: $invar\ small \implies size (push\ x\ small) = Suc (size\ small)$
 ⟨proof⟩

lemma *size-new-push* [*simp*]: $\text{invar } \text{small} \implies \text{size-new } (\text{push } x \text{ small}) = \text{Suc } (\text{size-new } \text{small})$
 ⟨*proof*⟩

lemma *size-pop* [*simp*]: $\llbracket \text{invar } \text{small}; 0 < \text{size } \text{small}; \text{pop } \text{small} = (x, \text{small}') \rrbracket$
 $\implies \text{Suc } (\text{size } \text{small}') = \text{size } \text{small}$
 ⟨*proof*⟩

lemma *size-new-pop* [*simp*]: $\llbracket \text{invar } \text{small}; 0 < \text{size-new } \text{small}; \text{pop } \text{small} = (x, \text{small}') \rrbracket$
 $\implies \text{Suc } (\text{size-new } \text{small}') = \text{size-new } \text{small}$
 ⟨*proof*⟩

lemma *size-size-new*: $\llbracket \text{invar } (\text{small} :: 'a \text{ small-state}); 0 < \text{size } \text{small} \rrbracket \implies 0 < \text{size-new } \text{small}$
 ⟨*proof*⟩

lemma *step-list-current* [*simp*]: $\text{invar } \text{small} \implies \text{list-current } (\text{step } \text{small}) = \text{list-current } \text{small}$
 ⟨*proof*⟩

lemma *step-list-common* [*simp*]:
 $\llbracket \text{small} = \text{Small3 } \text{common}; \text{invar } \text{small} \rrbracket \implies \text{list } (\text{step } \text{small}) = \text{list } \text{small}$
 ⟨*proof*⟩

lemma *step-list-Small2* [*simp*]:
assumes
 $\text{small} = (\text{Small2 } \text{current } \text{aux } \text{big } \text{new } \text{count})$
 $\text{invar } \text{small}$
shows
 $\text{list } (\text{step } \text{small}) = \text{list } \text{small}$
 ⟨*proof*⟩

lemma *invar-step*: $\text{invar } (\text{small} :: 'a \text{ small-state}) \implies \text{invar } (\text{step } \text{small})$
 ⟨*proof*⟩

lemma *invar-push*: $\text{invar } \text{small} \implies \text{invar } (\text{push } x \text{ small})$
 ⟨*proof*⟩

lemma *invar-pop*: \llbracket
 $0 < \text{size } \text{small};$
 $\text{invar } \text{small};$
 $\text{pop } \text{small} = (x, \text{small}')$
 $\rrbracket \implies \text{invar } \text{small}'$
 ⟨*proof*⟩

lemma *push-list-common* [*simp*]: $\text{small} = \text{Small3 } \text{common} \implies \text{list } (\text{push } x \text{ small}) = x \# \text{list } \text{small}$

<proof>

lemma *push-list-current* [*simp*]: *list-current (push x small) = x # list-current small*
<proof>

lemma *pop-list-current* [*simp*]: $\llbracket \text{invar } \text{small}; 0 < \text{size } \text{small}; \text{Small.pop } \text{small} = (x, \text{small}') \rrbracket$
 $\implies x \# \text{list-current } \text{small}' = \text{list-current } \text{small}$
<proof>

lemma *list-current-size* [*simp*]: $\llbracket 0 < \text{size } \text{small}; \text{list-current } \text{small} = []; \text{invar } \text{small} \rrbracket \implies \text{False}$
<proof>

lemma *list-Small2* [*simp*]: $\llbracket 0 < \text{size } (\text{Small2 } \text{current } \text{auxS } \text{big } \text{newS } \text{count}); \text{invar } (\text{Small2 } \text{current } \text{auxS } \text{big } \text{newS } \text{count}) \rrbracket \implies$
 $\text{fst } (\text{Current.pop } \text{current}) \# \text{list } (\text{Small2 } (\text{drop-first } \text{current}) \text{auxS } \text{big } \text{newS } \text{count}) =$
 $\text{list } (\text{Small2 } \text{current } \text{auxS } \text{big } \text{newS } \text{count})$
<proof>

end

18 Big + Small Proofs

theory *States-Proof*
imports *States-Aux Big-Proof Small-Proof*
begin

lemmas *state-splits* = *idle.splits common-state.splits small-state.splits big-state.splits*
lemmas *invar-steps* = *Big-Proof.invar-step Common-Proof.invar-step Small-Proof.invar-step*

lemma *invar-list-big-first*:
 $\text{invar } \text{states} \implies \text{list-big-first } \text{states} = \text{list-current-big-first } \text{states}$
<proof>

lemma *step-lists* [*simp*]: $\text{invar } \text{states} \implies \text{lists } (\text{step } \text{states}) = \text{lists } \text{states}$
<proof>

lemma *step-lists-current* [*simp*]:
 $\text{invar } \text{states} \implies \text{lists-current } (\text{step } \text{states}) = \text{lists-current } \text{states}$
<proof>

lemma *push-big*: $\text{lists } (\text{States } \text{dir } \text{big } \text{small}) = (\text{big}', \text{small}')$
 $\implies \text{lists } (\text{States } \text{dir } (\text{Big.push } x \text{big}) \text{small}) = (x \# \text{big}', \text{small}')$
<proof>

lemma *push-small-lists*:

invar (*States dir big small*)
 $\implies \text{lists } (\text{States dir big } (\text{Small.push } x \text{ small})) = (\text{big}', x \# \text{small}') \longleftrightarrow$
 $\text{lists } (\text{States dir big small}) = (\text{big}', \text{small}')$
(*proof*)

lemma *list-small-big*:

$\text{list-small-first } (\text{States dir big small}) = \text{list-current-small-first } (\text{States dir big small}) \longleftrightarrow$
 $\text{list-big-first } (\text{States dir big small}) = \text{list-current-big-first } (\text{States dir big small})$
(*proof*)

lemma *list-big-first-pop-big* [*simp*]: \llbracket

invar (*States dir big small*);
 $0 < \text{size big}$;
 $\text{Big.pop big} = (x, \text{big}')$
 $\implies x \# \text{list-big-first } (\text{States dir big' small}) = \text{list-big-first } (\text{States dir big small})$
(*proof*)

lemma *list-current-big-first-pop-big* [*simp*]: \llbracket

invar (*States dir big small*);
 $0 < \text{size big}$;
 $\text{Big.pop big} = (x, \text{big}')$
 $\implies x \# \text{list-current-big-first } (\text{States dir big' small}) =$
 $\text{list-current-big-first } (\text{States dir big small})$
(*proof*)

lemma *lists-big-first-pop-big*: \llbracket

invar (*States dir big small*);
 $0 < \text{size big}$;
 $\text{Big.pop big} = (x, \text{big}')$
 $\implies \text{list-big-first } (\text{States dir big' small}) = \text{list-current-big-first } (\text{States dir big' small})$
(*proof*)

lemma *lists-small-first-pop-big*: \llbracket

invar (*States dir big small*);
 $0 < \text{size big}$;
 $\text{Big.pop big} = (x, \text{big}')$
 $\implies \text{list-small-first } (\text{States dir big' small}) = \text{list-current-small-first } (\text{States dir big' small})$
(*proof*)

lemma *list-small-first-pop-small* [*simp*]: \llbracket

invar (*States dir big small*);
 $0 < \text{size small}$;
 $\text{Small.pop small} = (x, \text{small}')$
 $\implies x \# \text{list-small-first } (\text{States dir big small}') = \text{list-small-first } (\text{States dir big small})$

small)
 ⟨*proof*⟩

lemma *list-current-small-first-pop-small* [*simp*]: \llbracket
 invar (*States dir big small*);
 $0 < \text{size } \textit{small}$;
 $\textit{Small.pop } \textit{small} = (x, \textit{small}')$ \rrbracket
 $\implies x \# \textit{list-current-small-first } (\textit{States dir big small}') =$
 $\textit{list-current-small-first } (\textit{States dir big small})$
 ⟨*proof*⟩

lemma *lists-small-first-pop-small*: \llbracket
 invar (*States dir big small*);
 $0 < \text{size } \textit{small}$;
 $\textit{Small.pop } \textit{small} = (x, \textit{small}')$ \rrbracket
 $\implies \textit{list-small-first } (\textit{States dir big small}') = \textit{list-current-small-first } (\textit{States dir big}$
small')
 ⟨*proof*⟩

lemma *invars-pop-big*: \llbracket
 invar (*States dir big small*);
 $0 < \text{size } \textit{big}$;
 $\textit{Big.pop } \textit{big} = (x, \textit{big}')$ \rrbracket
 $\implies \textit{invar } \textit{big}' \wedge \textit{invar } \textit{small}$
 ⟨*proof*⟩

lemma *invar-pop-big-aux*: \llbracket
 invar (*States dir big small*);
 $0 < \text{size } \textit{big}$;
 $\textit{Big.pop } \textit{big} = (x, \textit{big}')$ \rrbracket
 $\implies (\textit{case } (\textit{big}', \textit{small}) \textit{ of}$
 $(\textit{Big1} - \textit{big} - \textit{count}, \textit{Small1 } (\textit{Current} - - \textit{old remained}) \textit{small} -) \implies$
 $\textit{size } \textit{big} - \textit{count} = \textit{remained} - \textit{size } \textit{old} \wedge \textit{count} \geq \textit{size } \textit{small}$
 | $(-, \textit{Small1} - -) \implies \textit{False}$
 | $(\textit{Big1} - - - -, -) \implies \textit{False}$
 | $- \implies \textit{True}$
)
 ⟨*proof*⟩

lemma *invar-pop-big*: \llbracket
 invar (*States dir big small*);
 $0 < \text{size } \textit{big}$;
 $\textit{Big.pop } \textit{big} = (x, \textit{big}')$ \rrbracket
 $\implies \textit{invar } (\textit{States dir } \textit{big}' \textit{ small})$
 ⟨*proof*⟩

lemma *invars-pop-small*: \llbracket
 invar (*States dir big small*);
 $0 < \text{size } \textit{small}$;

$Small.pop\ small = (x, small')]$
 $\implies invar\ big \wedge invar\ small'$
 $\langle proof \rangle$

lemma *invar-pop-small-aux*: $[[$
 $invar\ (States\ dir\ big\ small);$
 $0 < size\ small;$
 $Small.pop\ small = (x, small')]$
 $\implies (case\ (big, small')$ of
 $(Big1 - big - count, Small1\ (Current - - old\ remained)\ small -) \implies$
 $size\ big - count = remained - size\ old \wedge count \geq size\ small$
 $| (-, Small1 - - -) \implies False$
 $| (Big1 - - - -, -) \implies False$
 $| - \implies True$
 $)$
 $\langle proof \rangle$

lemma *invar-pop-small*: $[[$
 $invar\ (States\ dir\ big\ small);$
 $0 < size\ small;$
 $Small.pop\ small = (x, small')$
 $]] \implies invar\ (States\ dir\ big\ small')$
 $\langle proof \rangle$

lemma *invar-push-big*: $invar\ (States\ dir\ big\ small) \implies invar\ (States\ dir\ (Big.push\ x\ big)\ small)$
 $\langle proof \rangle$

lemma *invar-push-small*: $invar\ (States\ dir\ big\ small)$
 $\implies invar\ (States\ dir\ big\ (Small.push\ x\ small))$
 $\langle proof \rangle$

lemma *step-invars*: $[[invar\ states; step\ states = States\ dir\ big\ small]] \implies invar\ big$
 $\wedge invar\ small$
 $\langle proof \rangle$

lemma *step-lists-small-first*: $invar\ states \implies$
 $list-small-first\ (step\ states) = list-current-small-first\ (step\ states)$
 $\langle proof \rangle$

lemma *invar-step-aux*: $invar\ states \implies (case\ step\ states\ of$
 $(States - (Big1 - big - count)\ (Small1\ (Current - - old\ remained)\ small -))$
 \implies
 $size\ big - count = remained - size\ old \wedge count \geq size\ small$
 $| (States - - (Small1 - - -)) \implies False$
 $| (States - (Big1 - - - -) -) \implies False$
 $| - \implies True$
 $)$
 $\langle proof \rangle$

lemma *invar-step*: $\text{invar } (states :: 'a \text{ states}) \implies \text{invar } (step \text{ states})$
 ⟨proof⟩

lemma *step-consistent* [simp]:
 $\llbracket \bigwedge states. \text{invar } (states :: 'a \text{ states}) \implies P (step \text{ states}) = P \text{ states}; \text{invar } states \rrbracket$
 $\implies P \text{ states} = P ((step \sim^n) \text{ states})$
 ⟨proof⟩

lemma *step-consistent-2*:
 $\llbracket \bigwedge states. \llbracket \text{invar } (states :: 'a \text{ states}); P \text{ states} \rrbracket \implies P (step \text{ states}); \text{invar } states; P \text{ states} \rrbracket$
 $\implies P ((step \sim^n) \text{ states})$
 ⟨proof⟩

lemma *size-ok'-Suc*: $\text{size-ok}' \text{ states } (Suc \text{ steps}) \implies \text{size-ok}' \text{ states } \text{steps}$
 ⟨proof⟩

lemma *size-ok'-decline*: $\text{size-ok}' \text{ states } x \implies x \geq y \implies \text{size-ok}' \text{ states } y$
 ⟨proof⟩

lemma *remaining-steps-0* [simp]: $\llbracket \text{invar } (states :: 'a \text{ states}); \text{remaining-steps } \text{states} = 0 \rrbracket$
 $\implies \text{remaining-steps } (step \text{ states}) = 0$
 ⟨proof⟩

lemma *remaining-steps-0'*: $\llbracket \text{invar } (states :: 'a \text{ states}); \text{remaining-steps } \text{states} = 0 \rrbracket$
 $\implies \text{remaining-steps } ((step \sim^n) \text{ states}) = 0$
 ⟨proof⟩

lemma *remaining-steps-decline-Suc*:
 $\llbracket \text{invar } (states :: 'a \text{ states}); 0 < \text{remaining-steps } \text{states} \rrbracket$
 $\implies Suc (\text{remaining-steps } (step \text{ states})) = \text{remaining-steps } \text{states}$
 ⟨proof⟩

lemma *remaining-steps-decline-sub* [simp]: $\text{invar } (states :: 'a \text{ states})$
 $\implies \text{remaining-steps } (step \text{ states}) = \text{remaining-steps } \text{states} - 1$
 ⟨proof⟩

lemma *remaining-steps-decline*: $\text{invar } (states :: 'a \text{ states})$
 $\implies \text{remaining-steps } (step \text{ states}) \leq \text{remaining-steps } \text{states}$
 ⟨proof⟩

lemma *remaining-steps-decline-n-steps* [simp]:
 $\llbracket \text{invar } (states :: 'a \text{ states}); \text{remaining-steps } \text{states} \leq n \rrbracket$
 $\implies \text{remaining-steps } ((step \sim^n) \text{ states}) = 0$
 ⟨proof⟩

lemma *remaining-steps-n-steps-plus* [simp]:

$\llbracket n \leq \text{remaining-steps states}; \text{invar } (states :: 'a \text{ states}) \rrbracket$
 $\implies \text{remaining-steps } ((\text{step } \sim n) \text{ states}) + n = \text{remaining-steps states}$
 <proof>

lemma *remaining-steps-n-steps-sub* [simp]: *invar* (states :: 'a states)
 $\implies \text{remaining-steps } ((\text{step } \sim n) \text{ states}) = \text{remaining-steps states} - n$
 <proof>

lemma *step-size-new-small* [simp]:
 $\llbracket \text{invar } (States \text{ dir big small}); \text{step } (States \text{ dir big small}) = States \text{ dir}' \text{ big}' \text{ small}' \rrbracket$
 $\implies \text{size-new small}' = \text{size-new small}$
 <proof>

lemma *step-size-new-small-2* [simp]:
invar states $\implies \text{size-new-small } (\text{step states}) = \text{size-new-small states}$
 <proof>

lemma *step-size-new-big* [simp]:
 $\llbracket \text{invar } (States \text{ dir big small}); \text{step } (States \text{ dir big small}) = States \text{ dir}' \text{ big}' \text{ small}' \rrbracket$
 $\implies \text{size-new big}' = \text{size-new big}$
 <proof>

lemma *step-size-new-big-2* [simp]:
invar states $\implies \text{size-new-big } (\text{step states}) = \text{size-new-big states}$
 <proof>

lemma *step-size-small* [simp]:
 $\llbracket \text{invar } (States \text{ dir big small}); \text{step } (States \text{ dir big small}) = States \text{ dir}' \text{ big}' \text{ small}' \rrbracket$
 $\implies \text{size small}' = \text{size small}$
 <proof>

lemma *step-size-small-2* [simp]:
invar states $\implies \text{size-small } (\text{step states}) = \text{size-small states}$
 <proof>

lemma *step-size-big* [simp]:
 $\llbracket \text{invar } (States \text{ dir big small}); \text{step } (States \text{ dir big small}) = States \text{ dir}' \text{ big}' \text{ small}' \rrbracket$
 $\implies \text{size big}' = \text{size big}$
 <proof>

lemma *step-size-big-2* [simp]:
invar states $\implies \text{size-big } (\text{step states}) = \text{size-big states}$
 <proof>

lemma *step-size-ok-1*: \llbracket
invar (States dir big small);
step (States dir big small) = States dir' big' small';
size-new big + *remaining-steps* (States dir big small) + 2 \leq 3 * *size-new small*
 $\rrbracket \implies \text{size-new big}' + \text{remaining-steps } (States \text{ dir}' \text{ big}' \text{ small}') + 2 \leq 3 * \text{size-new small}$

small'
(proof)

lemma *step-size-ok-2*: \llbracket
 invar (*States dir big small*);
 step (*States dir big small*) = *States dir' big' small'*;
 size-new small + *remaining-steps* (*States dir big small*) + 2 \leq 3 * *size-new big*
 $\rrbracket \implies$ *size-new small'* + *remaining-steps* (*States dir' big' small'*) + 2 \leq 3 *
size-new big'
(proof)

lemma *step-size-ok-3*: \llbracket
 invar (*States dir big small*);
 step (*States dir big small*) = *States dir' big' small'*;
 remaining-steps (*States dir big small*) + 1 \leq 4 * *size small*
 $\rrbracket \implies$ *remaining-steps* (*States dir' big' small'*) + 1 \leq 4 * *size small'*
(proof)

lemma *step-size-ok-4*: \llbracket
 invar (*States dir big small*);
 step (*States dir big small*) = *States dir' big' small'*;
 remaining-steps (*States dir big small*) + 1 \leq 4 * *size big*
 $\rrbracket \implies$ *remaining-steps* (*States dir' big' small'*) + 1 \leq 4 * *size big'*
(proof)

lemma *step-size-ok*: \llbracket *invar states*; *size-ok states* $\rrbracket \implies$ *size-ok* (*step states*)
(proof)

lemma *step-n-size-ok*: \llbracket *invar states*; *size-ok states* $\rrbracket \implies$ *size-ok* ((*step* $\overset{\sim}{\sim}$ *n*) *states*)
(proof)

lemma *step-push-size-small* [*simp*]: \llbracket
 invar (*States dir big small*);
 step (*States dir big (Small.push x small)*) = *States dir' big' small'*
 $\rrbracket \implies$ *size small'* = *Suc (size small)*
(proof)

lemma *step-push-size-new-small* [*simp*]: \llbracket
 invar (*States dir big small*);
 step (*States dir big (Small.push x small)*) = *States dir' big' small'*
 $\rrbracket \implies$ *size-new small'* = *Suc (size-new small)*
(proof)

lemma *step-push-size-big* [*simp*]: \llbracket
 invar (*States dir big small*);
 step (*States dir (Big.push x big) small*) = *States dir' big' small'*
 $\rrbracket \implies$ *size big'* = *Suc (size big)*
(proof)

lemma *step-push-size-new-big* [simp]: \llbracket
 invar (States dir big small);
 step (States dir (Big.push x big) small) = States dir' big' small'
 $\rrbracket \implies \text{size-new big}' = \text{Suc} (\text{size-new big})$
 <proof>

lemma *step-pop-size-big* [simp]: \llbracket
 invar (States dir big small);
 $0 < \text{size big}$;
 Big.pop big = (x, bigP);
 step (States dir bigP small) = States dir' big' small'
 $\rrbracket \implies \text{Suc} (\text{size big}') = \text{size big}$
 <proof>

lemma *step-pop-size-new-big* [simp]: \llbracket
 invar (States dir big small);
 $0 < \text{size big}$; Big.pop big = (x, bigP);
 step (States dir bigP small) = States dir' big' small'
 $\rrbracket \implies \text{Suc} (\text{size-new big}') = \text{size-new big}$
 <proof>

lemma *step-n-size-small* [simp]: \llbracket
 invar (States dir big small);
 (step $\overset{\sim}{\sim}$ n) (States dir big small) = States dir' big' small'
 $\rrbracket \implies \text{size small}' = \text{size small}$
 <proof>

lemma *step-n-size-big* [simp]:
 $\llbracket \text{invar} (\text{States dir big small}); (\text{step } \overset{\sim}{\sim} n) (\text{States dir big small}) = \text{States dir}' \text{big}' \text{small}' \rrbracket$
 $\implies \text{size big}' = \text{size big}$
 <proof>

lemma *step-n-size-new-small* [simp]:
 $\llbracket \text{invar} (\text{States dir big small}); (\text{step } \overset{\sim}{\sim} n) (\text{States dir big small}) = \text{States dir}' \text{big}' \text{small}' \rrbracket$
 $\implies \text{size-new small}' = \text{size-new small}$
 <proof>

lemma *step-n-size-new-big* [simp]:
 $\llbracket \text{invar} (\text{States dir big small}); (\text{step } \overset{\sim}{\sim} n) (\text{States dir big small}) = \text{States dir}' \text{big}' \text{small}' \rrbracket$
 $\implies \text{size-new big}' = \text{size-new big}$
 <proof>

lemma *step-n-push-size-small* [simp]: \llbracket
 invar (States dir big small);
 (step $\overset{\sim}{\sim}$ n) (States dir big (Small.push x small)) = States dir' big' small'
 $\rrbracket \implies \text{size small}' = \text{Suc} (\text{size small})$

<proof>

lemma *step-n-push-size-new-small* [*simp*]: \llbracket
 invar (*States dir big small*);
 (*step* $\overset{\sim}{\sim} n$) (*States dir big (Small.push x small)*) = *States dir' big' small'*
 $\rrbracket \implies \text{size-new small}' = \text{Suc} (\text{size-new small})$
<proof>

lemma *step-n-push-size-big* [*simp*]: \llbracket
 invar (*States dir big small*);
 (*step* $\overset{\sim}{\sim} n$) (*States dir (Big.push x big) small*) = *States dir' big' small'*
 $\rrbracket \implies \text{size big}' = \text{Suc} (\text{size big})$
<proof>

lemma *step-n-push-size-new-big* [*simp*]: \llbracket
 invar (*States dir big small*);
 (*step* $\overset{\sim}{\sim} n$) (*States dir (Big.push x big) small*) = *States dir' big' small'*
 $\rrbracket \implies \text{size-new big}' = \text{Suc} (\text{size-new big})$
<proof>

lemma *step-n-pop-size-small* [*simp*]: \llbracket
 invar (*States dir big small*);
 $0 < \text{size small}$;
 Small.pop small = (*x, smallP*);
 (*step* $\overset{\sim}{\sim} n$) (*States dir big smallP*) = *States dir' big' small'*
 $\rrbracket \implies \text{Suc} (\text{size small}') = \text{size small}$
<proof>

lemma *step-n-pop-size-new-small* [*simp*]: \llbracket
 invar (*States dir big small*);
 $0 < \text{size small}$;
 Small.pop small = (*x, smallP*);
 (*step* $\overset{\sim}{\sim} n$) (*States dir big smallP*) = *States dir' big' small'*
 $\rrbracket \implies \text{Suc} (\text{size-new small}') = \text{size-new small}$
<proof>

lemma *step-n-pop-size-big* [*simp*]: \llbracket
 invar (*States dir big small*);
 $0 < \text{size big}$; *Big.pop big* = (*x, bigP*);
 (*step* $\overset{\sim}{\sim} n$) (*States dir bigP small*) = *States dir' big' small'*
 $\rrbracket \implies \text{Suc} (\text{size big}') = \text{size big}$
<proof>

lemma *step-n-pop-size-new-big*: \llbracket
 invar (*States dir big small*);
 $0 < \text{size big}$; *Big.pop big* = (*x, bigP*);
 (*step* $\overset{\sim}{\sim} n$) (*States dir bigP small*) = *States dir' big' small'*
 $\rrbracket \implies \text{Suc} (\text{size-new big}') = \text{size-new big}$
<proof>

lemma *remaining-steps-push-small* [simp]: $\text{invar } (\text{States dir big small})$
 $\implies \text{remaining-steps } (\text{States dir big small}) =$
 $\text{remaining-steps } (\text{States dir big } (\text{Small.push } x \text{ small}))$
 ⟨proof⟩

lemma *remaining-steps-pop-small*:
 $\llbracket \text{invar } (\text{States dir big small}); 0 < \text{size small}; \text{Small.pop small} = (x, \text{smallP}) \rrbracket$
 $\implies \text{remaining-steps } (\text{States dir big smallP}) \leq \text{remaining-steps } (\text{States dir big small})$
 ⟨proof⟩

lemma *remaining-steps-pop-big*:
 $\llbracket \text{invar } (\text{States dir big small}); 0 < \text{size big}; \text{Big.pop big} = (x, \text{bigP}) \rrbracket$
 $\implies \text{remaining-steps } (\text{States dir bigP small}) \leq \text{remaining-steps } (\text{States dir big small})$
 ⟨proof⟩

lemma *remaining-steps-push-big* [simp]: $\text{invar } (\text{States dir big small})$
 $\implies \text{remaining-steps } (\text{States dir } (\text{Big.push } x \text{ big}) \text{ small}) =$
 $\text{remaining-steps } (\text{States dir big small})$
 ⟨proof⟩

lemma *step-4-remaining-steps-push-big* [simp]: \llbracket
 $\text{invar } (\text{States dir big small});$
 $4 \leq \text{remaining-steps } (\text{States dir big small});$
 $(\text{step } \sim 4) (\text{States dir } (\text{Big.push } x \text{ big}) \text{ small}) = \text{States dir}' \text{ big}' \text{ small}' \rrbracket$
 $\implies \text{remaining-steps } (\text{States dir}' \text{ big}' \text{ small}') = \text{remaining-steps } (\text{States dir big small}) - 4$
 ⟨proof⟩

lemma *step-4-remaining-steps-push-small* [simp]: \llbracket
 $\text{invar } (\text{States dir big small});$
 $4 \leq \text{remaining-steps } (\text{States dir big small});$
 $(\text{step } \sim 4) (\text{States dir big } (\text{Small.push } x \text{ small})) = \text{States dir}' \text{ big}' \text{ small}' \rrbracket$
 $\implies \text{remaining-steps } (\text{States dir}' \text{ big}' \text{ small}') = \text{remaining-steps } (\text{States dir big small}) - 4$
 ⟨proof⟩

lemma *step-4-remaining-steps-pop-big*: \llbracket
 $\text{invar } (\text{States dir big small});$
 $0 < \text{size big};$
 $\text{Big.pop big} = (x, \text{bigP});$
 $4 \leq \text{remaining-steps } (\text{States dir bigP small});$
 $(\text{step } \sim 4) (\text{States dir bigP small}) = \text{States dir}' \text{ big}' \text{ small}' \rrbracket$
 $\implies \text{remaining-steps } (\text{States dir}' \text{ big}' \text{ small}') \leq \text{remaining-steps } (\text{States dir big small}) - 4$
 ⟨proof⟩

lemma *step-4-remaining-steps-pop-small*: \llbracket
invar (*States dir big small*);
 $0 < \text{size small}$;
Small.pop small = (*x*, *smallP*);
 $4 \leq \text{remaining-steps} (\text{States dir big smallP})$;
 $(\text{step} \sim 4) (\text{States dir big smallP}) = \text{States dir}' \text{big}' \text{small}'$
 $\rrbracket \implies \text{remaining-steps} (\text{States dir}' \text{big}' \text{small}') \leq \text{remaining-steps} (\text{States dir big small}) - 4$
<proof>

lemma *step-4-pop-small-size-ok-1*: \llbracket
invar (*States dir big small*);
 $0 < \text{size small}$;
Small.pop small = (*x*, *smallP*);
 $4 \leq \text{remaining-steps} (\text{States dir big smallP})$;
 $(\text{step} \sim 4) (\text{States dir big smallP}) = \text{States dir}' \text{big}' \text{small}'$;
 $\text{remaining-steps} (\text{States dir big small}) + 1 \leq 4 * \text{size small}$
 $\rrbracket \implies \text{remaining-steps} (\text{States dir}' \text{big}' \text{small}') + 1 \leq 4 * \text{size small}'$
<proof>

lemma *step-4-pop-big-size-ok-1*: \llbracket
invar (*States dir big small*);
 $0 < \text{size big}$; *Big.pop big* = (*x*, *bigP*);
 $4 \leq \text{remaining-steps} (\text{States dir bigP small})$;
 $(\text{step} \sim 4) (\text{States dir bigP small}) = \text{States dir}' \text{big}' \text{small}'$;
 $\text{remaining-steps} (\text{States dir big small}) + 1 \leq 4 * \text{size small}$
 $\rrbracket \implies \text{remaining-steps} (\text{States dir}' \text{big}' \text{small}') + 1 \leq 4 * \text{size small}'$
<proof>

lemma *step-4-pop-small-size-ok-2*: \llbracket
invar (*States dir big small*);
 $0 < \text{size small}$;
Small.pop small = (*x*, *smallP*);
 $4 \leq \text{remaining-steps} (\text{States dir big smallP})$;
 $(\text{step} \sim 4) (\text{States dir big smallP}) = \text{States dir}' \text{big}' \text{small}'$;
 $\text{remaining-steps} (\text{States dir big small}) + 1 \leq 4 * \text{size big}$
 $\rrbracket \implies \text{remaining-steps} (\text{States dir}' \text{big}' \text{small}') + 1 \leq 4 * \text{size big}'$
<proof>

lemma *step-4-pop-big-size-ok-2*:
assumes
invar (*States dir big small*)
 $0 < \text{size big}$
Big.pop big = (*x*, *bigP*)
 $\text{remaining-steps} (\text{States dir bigP small}) \geq 4$
 $(\text{step} \sim 4) (\text{States dir bigP small}) = \text{States dir}' \text{big}' \text{small}'$
 $\text{remaining-steps} (\text{States dir big small}) + 1 \leq 4 * \text{size big}$
shows
 $\text{remaining-steps} (\text{States dir}' \text{big}' \text{small}') + 1 \leq 4 * \text{size big}'$

<proof>

lemma *step-4-pop-small-size-ok-3:*

assumes

invar (States dir big small)

0 < size small

Small.pop small = (x, smallP)

remaining-steps (States dir big smallP) ≥ 4

((step \rightsquigarrow 4) (States dir big smallP)) = States dir' big' small'

*size-new small + remaining-steps (States dir big small) + 2 ≤ 3 * size-new big*

shows

*size-new small' + remaining-steps (States dir' big' small') + 2 ≤ 3 * size-new big'*

<proof>

lemma *step-4-pop-big-size-ok-3-aux:* \llbracket

0 < size big;

4 ≤ remaining-steps (States dir big small);

*size-new small + remaining-steps (States dir big small) + 2 ≤ 3 * size-new big*

$\rrbracket \implies$ *size-new small + (remaining-steps (States dir big small) - 4) + 2 ≤ 3 * (size-new big - 1)*

<proof>

lemma *step-4-pop-big-size-ok-3:*

assumes

invar (States dir big small)

0 < size big

Big.pop big = (x, bigP)

remaining-steps (States dir bigP small) ≥ 4

((step \rightsquigarrow 4) (States dir bigP small)) = (States dir' big' small')

*size-new small + remaining-steps (States dir big small) + 2 ≤ 3 * size-new*

big

shows

*size-new small' + remaining-steps (States dir' big' small') + 2 ≤ 3 * size-new big'*

<proof>

lemma *step-4-pop-small-size-ok-4-aux:* \llbracket

0 < size small;

4 ≤ remaining-steps (States dir big small);

*size-new big + remaining-steps (States dir big small) + 2 ≤ 3 * size-new small*

$\rrbracket \implies$ *size-new big + (remaining-steps (States dir big small) - 4) + 2 ≤ 3 * (size-new small - 1)*

<proof>

lemma *step-4-pop-small-size-ok-4:*

assumes

invar (States dir big small)

$0 < \text{size small}$
 $\text{Small.pop small} = (x, \text{smallP})$
 $\text{remaining-steps} (\text{States dir big smallP}) \geq 4$
 $((\text{step} \sim 4) (\text{States dir big smallP})) = (\text{States dir' big' small'})$
 $\text{size-new big} + \text{remaining-steps} (\text{States dir big small}) + 2 \leq 3 * \text{size-new small}$

shows

$\text{size-new big'} + \text{remaining-steps} (\text{States dir' big' small'}) + 2 \leq 3 * \text{size-new small'}$
 <proof>

lemma step-4-pop-big-size-ok-4-aux: \llbracket

$0 < \text{size big};$
 $4 \leq \text{remaining-steps} (\text{States dir big small});$
 $\text{size-new big} + \text{remaining-steps} (\text{States dir big small}) + 2 \leq 3 * \text{size-new small}$
 $\rrbracket \implies \text{size-new big} - 1 + (\text{remaining-steps} (\text{States dir big small}) - 4) + 2 \leq 3 * \text{size-new small}$
 <proof>

lemma step-4-pop-big-size-ok-4:

assumes

$\text{invar} (\text{States dir big small})$
 $0 < \text{size big}$
 $\text{Big.pop big} = (x, \text{bigP})$
 $\text{remaining-steps} (\text{States dir bigP small}) \geq 4$
 $((\text{step} \sim 4) (\text{States dir bigP small})) = (\text{States dir' big' small'})$
 $\text{size-new big} + \text{remaining-steps} (\text{States dir big small}) + 2 \leq 3 * \text{size-new small}$

shows

$\text{size-new big'} + \text{remaining-steps} (\text{States dir' big' small'}) + 2 \leq 3 * \text{size-new small'}$
 <proof>

lemma step-4-push-small-size-ok-1: \llbracket

$\text{invar} (\text{States dir big small});$
 $4 \leq \text{remaining-steps} (\text{States dir big small});$
 $(\text{step} \sim 4) (\text{States dir big} (\text{Small.push } x \text{ small})) = \text{States dir' big' small'};$
 $\text{remaining-steps} (\text{States dir big small}) + 1 \leq 4 * \text{size small}$
 $\rrbracket \implies \text{remaining-steps} (\text{States dir' big' small'}) + 1 \leq 4 * \text{size small'}$
 <proof>

lemma step-4-push-big-size-ok-1: \llbracket

$\text{invar} (\text{States dir big small});$
 $4 \leq \text{remaining-steps} (\text{States dir big small});$
 $(\text{step} \sim 4) (\text{States dir} (\text{Big.push } x \text{ big}) \text{ small}) = \text{States dir' big' small'};$
 $\text{remaining-steps} (\text{States dir big small}) + 1 \leq 4 * \text{size small}$
 $\rrbracket \implies \text{remaining-steps} (\text{States dir' big' small'}) + 1 \leq 4 * \text{size small'}$
 <proof>

lemma step-4-push-small-size-ok-2: \llbracket

invar (*States dir big small*);
 $4 \leq \text{remaining-steps } (\text{States dir big small})$;
 $(\text{step } \sim_4) (\text{States dir big } (\text{Small.push } x \text{ small})) = \text{States dir' big' small'}$;
 $\text{remaining-steps } (\text{States dir big small}) + 1 \leq 4 * \text{size big}$
 $\llbracket \implies \text{remaining-steps } (\text{States dir' big' small'}) + 1 \leq 4 * \text{size big}'$
<proof>

lemma *step-4-push-big-size-ok-2*: \llbracket
invar (*States dir big small*);
 $4 \leq \text{remaining-steps } (\text{States dir big small})$;
 $(\text{step } \sim_4) (\text{States dir } (\text{Big.push } x \text{ big}) \text{ small}) = \text{States dir' big' small'}$;
 $\text{remaining-steps } (\text{States dir big small}) + 1 \leq 4 * \text{size big}$
 $\llbracket \implies \text{remaining-steps } (\text{States dir' big' small'}) + 1 \leq 4 * \text{size big}'$
<proof>

lemma *step-4-push-small-size-ok-3-aux*: \llbracket
 $4 \leq \text{remaining-steps } (\text{States dir big small})$;
 $\text{size-new small} + \text{remaining-steps } (\text{States dir big small}) + 2 \leq 3 * \text{size-new big}$
 $\llbracket \implies \text{Suc } (\text{size-new small}) + (\text{remaining-steps } (\text{States dir big small}) - 4) + 2 \leq$
 $3 * \text{size-new big}$
<proof>

lemma *step-4-push-small-size-ok-3*: \llbracket
invar (*States dir big small*);
 $4 \leq \text{remaining-steps } (\text{States dir big small})$;
 $(\text{step } \sim_4) (\text{States dir big } (\text{Small.push } x \text{ small})) = \text{States dir' big' small'}$;
 $\text{size-new small} + \text{remaining-steps } (\text{States dir big small}) + 2 \leq 3 * \text{size-new big}$
 $\llbracket \implies \text{size-new small}' + \text{remaining-steps } (\text{States dir' big' small'}) + 2 \leq 3 * \text{size-new big}'$
<proof>

lemma *step-4-push-big-size-ok-3-aux*: \llbracket
 $4 \leq \text{remaining-steps } (\text{States dir big small})$;
 $\text{size-new small} + \text{remaining-steps } (\text{States dir big small}) + 2 \leq 3 * \text{size-new big}$
 $\llbracket \implies \text{size-new small} + (\text{remaining-steps } (\text{States dir big small}) - 4) + 2 \leq 3 * \text{Suc } (\text{size-new big})$
<proof>

lemma *step-4-push-big-size-ok-3*: \llbracket
invar (*States dir big small*);
 $4 \leq \text{remaining-steps } (\text{States dir big small})$;
 $(\text{step } \sim_4) (\text{States dir } (\text{Big.push } x \text{ big}) \text{ small}) = \text{States dir' big' small'}$;
 $\text{size-new small} + \text{remaining-steps } (\text{States dir big small}) + 2 \leq 3 * \text{size-new big}$
 $\llbracket \implies \text{size-new small}' + \text{remaining-steps } (\text{States dir' big' small'}) + 2 \leq 3 * \text{size-new big}'$
<proof>

lemma *step-4-push-small-size-ok-4-aux*: \llbracket
 $4 \leq \text{remaining-steps } (\text{States dir big small})$;

$size_new\ big + remaining_steps\ (States\ dir\ big\ small) + 2 \leq 3 * size_new\ small$
 $\] \implies size_new\ big + (remaining_steps\ (States\ dir\ big\ small) - 4) + 2 \leq 3 * Suc$
 $(size_new\ small)$
 <proof>

lemma *step-4-push-small-size-ok-4*: \llbracket
 $invar\ (States\ dir\ big\ small);$
 $4 \leq remaining_steps\ (States\ dir\ big\ small);$
 $(step\ \sim_4)\ (States\ dir\ big\ (Small.push\ x\ small)) = States\ dir'\ big'\ small';$
 $size_new\ big + remaining_steps\ (States\ dir\ big\ small) + 2 \leq 3 * size_new\ small$
 $\] \implies size_new\ big' + remaining_steps\ (States\ dir'\ big'\ small') + 2 \leq 3 * size_new$
 $small'$
 <proof>

lemma *step-4-push-big-size-ok-4-aux*: \llbracket
 $4 \leq remaining_steps\ (States\ dir\ big\ small);$
 $size_new\ big + remaining_steps\ (States\ dir\ big\ small) + 2 \leq 3 * size_new\ small$
 $\] \implies Suc\ (size_new\ big) + (remaining_steps\ (States\ dir\ big\ small) - 4) + 2 \leq 3$
 $* size_new\ small$
 <proof>

lemma *step-4-push-big-size-ok-4*: \llbracket
 $invar\ (States\ dir\ big\ small);$
 $4 \leq remaining_steps\ (States\ dir\ big\ small);$
 $(step\ \sim_4)\ (States\ dir\ (Big.push\ x\ big)\ small) = States\ dir'\ big'\ small';$
 $size_new\ big + remaining_steps\ (States\ dir\ big\ small) + 2 \leq 3 * size_new\ small$
 $\] \implies size_new\ big' + remaining_steps\ (States\ dir'\ big'\ small') + 2 \leq 3 * size_new$
 $small'$
 <proof>

lemma *step-4-push-small-size-ok*: \llbracket
 $invar\ (States\ dir\ big\ small);$
 $4 \leq remaining_steps\ (States\ dir\ big\ small);$
 $size_ok\ (States\ dir\ big\ small)$
 $\] \implies size_ok\ ((step\ \sim_4)\ (States\ dir\ big\ (Small.push\ x\ small)))$
 <proof>

lemma *step-4-push-big-size-ok*: \llbracket
 $invar\ (States\ dir\ big\ small);$
 $4 \leq remaining_steps\ (States\ dir\ big\ small);$
 $size_ok\ (States\ dir\ big\ small)$
 $\] \implies size_ok\ ((step\ \sim_4)\ (States\ dir\ (Big.push\ x\ big)\ small))$
 <proof>

lemma *step-4-pop-small-size-ok*: \llbracket
 $invar\ (States\ dir\ big\ small);$
 $0 < size\ small;$
 $Small.pop\ small = (x, smallP);$
 $4 \leq remaining_steps\ (States\ dir\ big\ smallP);$

size-ok (States dir big small)
 $\llbracket \implies \text{size-ok } ((\text{step} \sim 4) \text{ (States dir big smallP)})$
\langle proof \rangle

lemma *step-4-pop-big-size-ok*: \llbracket
invar (States dir big small);
0 < size big; Big.pop big = (x, bigP);
4 ≤ remaining-steps (States dir bigP small);
size-ok (States dir big small)
 $\llbracket \implies \text{size-ok } ((\text{step} \sim 4) \text{ (States dir bigP small)})$
\langle proof \rangle

lemma *size-ok-size-small*: *size-ok (States dir big small) \implies 0 < size small*
\langle proof \rangle

lemma *size-ok-size-big*: *size-ok (States dir big small) \implies 0 < size big*
\langle proof \rangle

lemma *size-ok-size-new-small*: *size-ok (States dir big small) \implies 0 < size-new small*
\langle proof \rangle

lemma *size-ok-size-new-big*: *size-ok (States dir big small) \implies 0 < size-new big*
\langle proof \rangle

lemma *step-size-ok'*: $\llbracket \text{invar states; size-ok' states } n \rrbracket \implies \text{size-ok' (step states) } n$
\langle proof \rangle

lemma *step-same*: *step (States dir big small) = States dir' big' small' \implies dir = dir'*
\langle proof \rangle

lemma *step-n-same*: *(step $\sim n$) (States dir big small) = States dir' big' small' \implies dir = dir'*
\langle proof \rangle

lemma *step-listL*: *invar states \implies listL (step states) = listL states*
\langle proof \rangle

lemma *step-n-listL*: *invar states \implies listL ((step $\sim n$) states) = listL states*
\langle proof \rangle

lemma *listL-remaining-steps*:
assumes
listL states = []
0 < remaining-steps states
invar states
size-ok states
shows

False
 <proof>

lemma *invar-step-n*: *invar (states :: 'a states) \implies invar ((step \sim n) states)*
 <proof>

lemma *step-n-size-ok'*: \llbracket *invar states; size-ok' states x* $\rrbracket \implies$ *size-ok' ((step \sim n) states) x*
 <proof>

lemma *size-ok-steps*: \llbracket
invar states;
size-ok' states (remaining-steps states - n)
 $\rrbracket \implies$ *size-ok ((step \sim n) states)*
 <proof>

lemma *remaining-steps-idle*: *invar states*
 \implies *remaining-steps states = 0 \longleftrightarrow (*
case states of
States - (Big2 (Common.Idle - -)) (Small3 (Common.Idle - -)) \implies True
| - \implies False)
 <proof>

lemma *remaining-steps-idle'*:
 \llbracket *invar (States dir big small); remaining-steps (States dir big small) = 0* \rrbracket
 \implies \exists *big-current big-idle small-current small-idle. States dir big small =*
States dir
(Big2 (common-state.Idle big-current big-idle))
(Small3 (common-state.Idle small-current small-idle))
 <proof>

end

19 Dequeue Proofs

theory *RealTimeDeque-Dequeue-Proof*
imports *Deque RealTimeDeque-Aux States-Proof*
begin

lemma *list-deqL' [simp]*: \llbracket *invar deque; listL deque \neq []; deqL' deque = (x, deque $\hat{}$)* \rrbracket
 \implies *x # listL deque' = listL deque*
 <proof>

lemma *list-deqL [simp]*:
 \llbracket *invar deque; listL deque \neq []* $\rrbracket \implies$ *listL (deqL deque) = tl (listL deque)*
 <proof>

lemma *list-firstL [simp]*:
 \llbracket *invar deque; listL deque \neq []* $\rrbracket \implies$ *firstL deque = hd (listL deque)*

<proof>

lemma *invar-deqL*:

$\llbracket \text{invar deque}; \neg \text{is-empty deque} \rrbracket \implies \text{invar (deqL deque)}$
<proof>

end

20 Enqueue Proofs

theory *RealTimeDeque-Enqueue-Proof*

imports *Deque RealTimeDeque-Aux States-Proof*

begin

lemma *list-enqL*: $\text{invar deque} \implies \text{listL (enqL x deque)} = x \# \text{listL deque}$
<proof>

lemma *invar-enqL*: $\text{invar deque} \implies \text{invar (enqL x deque)}$
<proof>

end

21 Top-Level Proof

theory *RealTimeDeque-Proof*

imports *RealTimeDeque-Deque-Proof RealTimeDeque-Enqueue-Proof*

begin

lemma *swap-lists-left*: $\text{invar (States Left big small)} \implies$
 $\text{States-Aux.listL (States Left big small)} = \text{rev (States-Aux.listL (States Right big small))}$
<proof>

lemma *swap-lists-right*: $\text{invar (States Right big small)} \implies$
 $\text{States-Aux.listL (States Right big small)} = \text{rev (States-Aux.listL (States Left big small))}$
<proof>

lemma *swap-list [simp]*: $\text{invar } q \implies \text{listR (swap } q) = \text{listL } q$
<proof>

lemma *swap-list'*: $\text{invar } q \implies \text{listL (swap } q) = \text{listR } q$
<proof>

lemma *lists-same*: $\text{lists (States Left big small)} = \text{lists (States Right big small)}$
<proof>

lemma *invar-swap*: $\text{invar } q \implies \text{invar (swap } q)$

<proof>

lemma *listL-is-empty*: *invar deque* \implies *is-empty deque* = (*listL deque* = [])
<proof>

interpretation *RealTimeDeque*: *Deque* **where**

empty = *empty* **and**
enqL = *enqL* **and**
enqR = *enqR* **and**
firstL = *firstL* **and**
firstR = *firstR* **and**
deqL = *deqL* **and**
deqR = *deqR* **and**
is-empty = *is-empty* **and**
listL = *listL* **and**
invar = *invar*
<proof>

end

References

- [1] T. Chuang and B. Goldberg. Real-time dequeues, multihead Turing machines, and purely functional programming. In J. Williams, editor, *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*, pages 289–298. ACM, 1993.