Real-Time Double-Ended Queue

Balazs Toth and Tobias Nipkow
Technical University of Munich

March 17, 2025

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

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*:

$[\![invar\ q;\ \neg\ rev\ (listL\ q) = []]\!] \Longrightarrow firstR\ q = hd(rev(listL\ q))$

assumes *list-is-empty*:
$invar\ q \Longrightarrow is\text{-}empty\ q = (listL\ q = [])$

assumes *invar-empty*:
invar empty

assumes *invar-enqL*:
$invar\ q \Longrightarrow invar(enqL\ x\ q)$
assumes *invar-enqR*:
$invar\ q \Longrightarrow invar(enqR\ x\ q)$
assumes *invar-deqL*:
$[\![invar\ q;\ \neg\ is\text{-}empty\ q]\!] \Longrightarrow invar(deqL\ q)$
assumes *invar-deqR*:
$[\![invar\ q;\ \neg\ is\text{-}empty\ q]\!] \Longrightarrow invar(deqR\ q)$
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 $is\text{-}empty :: 'a \Rightarrow bool$

class *invar* =
fixes $invar :: 'a \Rightarrow bool$

class *size-new* =
fixes $size\text{-}new :: 'a \Rightarrow nat$

class *step* =
fixes $step :: 'a \Rightarrow 'a$

class *remaining-steps* =
fixes $remaining\text{-}steps :: 'a \Rightarrow nat$

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**

step (*States dir* (*Big1 currentB big auxB 0*) (*Small1 currentS - auxS*)) =
  States dir (*step* (*Big1 currentB big auxB 0*)) (*Small2 currentS auxS big* [] *0*)
| *step* (*States dir left right*) = *States dir* (*step left*) (*step right*)

instance⟨*proof*⟩
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 * size of left end ≥ size of right end
- 3 * size of right end ≥ 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* ⇒ *bool* **where**
is-empty-deque Empty = *True*
| *is-empty-deque* - = *False*

instance⟨*proof*⟩
end

fun *swap* :: *′a deque* ⇒ *′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 = (step^^6) 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 = (step^^6) states in

      Rebal states
  )
| enqL x (Rebal (States Left big small)) = (
    let small = Small.push x small in
    let states = (step^^4) (States Left big small) in
    case states of
      States Left
        (Big2 (Common.Idle - big))
        (Small3 (Common.Idle - small))
       ⇒ Idles small big
     | - ⇒ Rebal states
  )
| enqL x (Rebal (States Right big small)) = (
    let big = Big.push x big in
    let states = (step^^4) (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 transformation 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-rev n xs* = *rev* (*take n xs*)

fun *list* :: *'a common-state* ⇒ *'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* ⇒ *'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* ⇒ *bool* **where**
invar (*Idle current idle*) ⟷
invar idle
∧ *invar current*
∧ *size-new current* = *size idle*
∧ *take* (*size idle*) (*Current-Aux.list current*) =
take (*size current*) (*Idle-Aux.list idle*)
| *invar* (*Copy current aux new moved*) ⟷ (
case current of Current - - *old remained* ⇒
moved < *remained*
∧ *moved* = *length new*
∧ *remained* ≤ *length aux* + *moved*
∧ *invar current*
∧ *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* ⇒ *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* ⇒ *'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* ⇒ *'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* ⇒ *bool* **where**
invar (*Small3 state*) = *invar state*
| *invar* (*Small2 current auxS big newS count*) = (
case current of Current - - old remained ⇒

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 ⇒ *remaining-steps common*
| *Small2* (*Current* - - - *remaining*) - *big* - *count* ⇒ *remaining* − *count* + *1*
| *Small1* (*Current* - - - *remaining*) - - ⇒
case big of Big1 currentB big auxB count ⇒ *remaining* + *count* + *2*
)

instance⟨*proof*⟩
end

fun *lists* :: ′*a states* ⇒ ′*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* ^^ *count*) *big*) [] *0*)
)
| *lists* (*States* - *big small*) = (*Big-Aux.list big*, *Small-Aux.list small*)

fun *list-small-first* :: ′*a states* ⇒ ′*a list* **where**
list-small-first states = (*let* (*big*, *small*) = *lists states in small* @ (*rev big*))

fun *list-big-first* :: ′*a states* ⇒ ′*a list* **where**
list-big-first states = (*let* (*big*, *small*) = *lists states in big* @ (*rev small*))

fun *lists-current* :: ′*a states* ⇒ ′*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* ⇒ ′*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* ⇒ ′*a list* **where**
list-current-big-first states = (*let* (*big*, *small*) = *lists-current states in big* @ (*rev small*))

fun *listL* :: ′*a states* ⇒ ′*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* ⇒ *bool* **where**
invar (*States dir big small*) ⟷ (
invar big
∧ *invar small*
∧ *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*: ⟦*take* (*Suc 0*) (*rev xs*) = [*last xs*]⟧ ⟹ *Suc 0* ≤ *length xs*
  ⟨*proof*⟩

lemma *take-last*: *xs* ≠ [] ⟹ *take 1* (*rev xs*) = [*last xs*]
  ⟨*proof*⟩

lemma *take-hd* [*simp*]: *xs* ≠ [] ⟹ *take* (*Suc 0*) *xs* = [*hd xs*]
  ⟨*proof*⟩

lemma *cons-tl*: $x \# xs = ys \Longrightarrow xs = tl\ ys$
⟨*proof*⟩

lemma *cons-hd*: $x \# xs = ys \Longrightarrow x = hd\ ys$
⟨*proof*⟩

lemma *take-hd'*: $ys \neq [] \Longrightarrow take\ (size\ ys)\ (x \# xs) = take\ (Suc\ (size\ xs))\ ys \Longrightarrow hd\ ys = x$
⟨*proof*⟩

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

lemma *hd-drop-1* [*simp*]: $xs \neq [] \Longrightarrow hd\ xs \# drop\ (Suc\ 0)\ xs = xs$
⟨*proof*⟩

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

lemma *take-1*: $0 < x \wedge 0 < y \Longrightarrow take\ x\ xs = take\ y\ ys \Longrightarrow take\ 1\ xs = take\ 1\ ys$
⟨*proof*⟩

lemma *last-drop-rev*: $xs \neq [] \Longrightarrow last\ xs \# drop\ 1\ (rev\ xs) = rev\ xs$
⟨*proof*⟩

lemma *Suc-min* [*simp*]: $0 < x \Longrightarrow 0 < y \Longrightarrow Suc\ (min\ (x - Suc\ 0)\ (y - Suc\ 0)) = min\ x\ y$
⟨*proof*⟩

lemma *rev-tl-hd*: $xs \neq [] \Longrightarrow rev\ (tl\ xs)\ @\ [hd\ xs] = rev\ xs$
⟨*proof*⟩

lemma *app-rev*: $as\ @\ rev\ bs = cs\ @\ rev\ ds \Longrightarrow bs\ @\ rev\ as = ds\ @\ rev\ cs$
⟨*proof*⟩

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

lemma *Suc-sub*: $Suc\ n = m \Longrightarrow n = m - 1$
⟨*proof*⟩

lemma *length-one-hd*: $length\ xs = 1 \Longrightarrow xs = [hd\ xs]$
⟨*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*]: ¬ *is-empty stack* ⟹ *first stack* = *hd* (*list stack*)
⟨*proof*⟩

lemma *list-empty*: *list stack* = [] ⟷ *is-empty stack*
⟨*proof*⟩

lemma *list-not-empty*: *list stack* ≠ [] ⟷ ¬ *is-empty stack*
⟨*proof*⟩

lemma *list-empty-2* [*simp*]: ⟦*list stack* ≠ []; *is-empty stack*⟧ ⟹ *False*
⟨*proof*⟩

lemma *list-not-empty-2* [*simp*]:⟦*list stack* = []; ¬ *is-empty stack*⟧ ⟹ *False*
⟨*proof*⟩

lemma *list-empty-size*: *list stack* = [] ⟷ *size stack* = *0*
⟨*proof*⟩

lemma *list-not-empty-size*:*list stack* ≠ [] ⟷ *0* < *size stack*
⟨*proof*⟩

lemma *list-empty-size-2* [*simp*]: ⟦*list stack* ≠ []; *size stack* = *0*⟧ ⟹ *False*
⟨*proof*⟩

lemma *list-not-empty-size-2* [*simp*]:⟦*list stack* = []; *0* < *size stack*⟧ ⟹ *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* ⟷ *is-empty stack*
⟨*proof*⟩

lemma *size-not-empty*: *size* (*stack* :: *'a stack*) > *0* ⟷ ¬ *is-empty stack*

⟨*proof*⟩

lemma *size-empty-2*[*simp*]: ⟦*size* (*stack* :: ′*a stack*) = *0*; ¬*is-empty stack*⟧ ⟹ *False*
⟨*proof*⟩

lemma *size-not-empty-2*[*simp*]: ⟦*0* < *size* (*stack* :: ′*a stack*); *is-empty stack*⟧ ⟹ *False*
⟨*proof*⟩

lemma *size-list-length* [*simp*]: *length* (*list stack*) = *size stack*
⟨*proof*⟩

lemma *first-pop* [*simp*]: ¬ *is-empty stack* ⟹ *first stack* # *list* (*pop stack*) = *list stack*
⟨*proof*⟩

lemma *push-not-empty* [*simp*]: ⟦¬ *is-empty stack*; *is-empty* (*push x stack*)⟧ ⟹ *False*
⟨*proof*⟩

lemma *pop-list-length* [*simp*]: ¬ *is-empty stack*
⟹ *Suc* (*length* (*list* (*pop stack*))) = *length* (*list stack*)
⟨*proof*⟩

lemma *first-take*: ¬*is-empty stack* ⟹ [*first stack*] = *take 1* (*list stack*)
⟨*proof*⟩

lemma *first-take-tl* [*simp*]: *0* < *size big*
⟹ (*first big* # *take count* (*tl* (*list big*))) = *take* (*Suc count*) (*list big*)
⟨*proof*⟩

lemma *first-take-pop* [*simp*]: ⟦¬*is-empty stack*; *0* < *x*⟧
⟹ *first stack* # *take* (*x* − *Suc 0*) (*list* (*pop stack*)) = *take x* (*list stack*)
⟨*proof*⟩

lemma [*simp*]: *first* (*Stack* [] []) = *undefined*
⟨*proof*⟩

lemma *first-hd*: *first stack* = *hd* (*list stack*)
⟨*proof*⟩

lemma *pop-tl* [*simp*]: *list* (*pop stack*) = *tl* (*list stack*)
⟨*proof*⟩

lemma *pop-drop*: *list* (*pop stack*) = *drop 1* (*list stack*)
⟨*proof*⟩

lemma *popN-drop* [*simp*]: *list* ((*pop* ^^ *n*) *stack*) = *drop n* (*list stack*)

⟨proof⟩

lemma *popN-size* [*simp*]: *size* ((*pop* ^^ *n*) *stack*) = (*size stack*) − *n*
⟨proof⟩

lemma *take-first*: ⟦*0* < *size s1*; *0* < *size s2*; *take* (*size s1*) (*list s2*) = *take* (*size s2*) (*list s1*)⟧
⟹ *first s1* = *first s2*
⟨proof⟩

end

13 Idle Proofs

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

lemma *push-list* [*simp*]: *list* (*push x idle*) = *x* # *list idle*
⟨proof⟩

lemma *pop-list* [*simp*]: ⟦¬ *is-empty idle*; *pop idle* = (*x*, *idle*′)⟧ ⟹ *x* # *list idle*′ = *list idle*
⟨proof⟩

lemma *pop-list-tl* [*simp*]:
⟦¬ *is-empty idle*; *pop idle* = (*x*, *idle*′)⟧ ⟹ *x* # (*tl* (*list idle*)) = *list idle*
⟨proof⟩

lemma *pop-list-tl*′ [*simp*]: ⟦*pop idle* = (*x*, *idle*′)⟧ ⟹ *list idle*′ = *tl* (*list idle*)
⟨proof⟩

lemma *size-push* [*simp*]: *size* (*push x idle*) = *Suc* (*size idle*)
⟨proof⟩

lemma *size-pop* [*simp*]: ⟦¬*is-empty idle*; *pop idle* = (*x*, *idle*′)⟧ ⟹ *Suc* (*size idle*′) = *size idle*
⟨proof⟩

lemma *size-pop-sub*: ⟦*pop idle* = (*x*, *idle*′)⟧ ⟹ *size idle*′ = *size idle* − *1*
⟨proof⟩

lemma *invar-push*: *invar idle* ⟹ *invar* (*push x idle*)
⟨proof⟩

lemma *invar-pop*: ⟦*invar idle*; *pop idle* = (*x*, *idle*′)⟧ ⟹ *invar idle*′
⟨proof⟩

lemma *size-empty*: *size idle* = *0* ⟷ *is-empty* (*idle* :: ′*a idle*)

⟨*proof*⟩

lemma *size-not-empty*: $0 < size\ idle \longleftrightarrow \neg is\text{-}empty\ (idle :: 'a\ idle)$
⟨*proof*⟩

lemma *size-empty-2* [*simp*]: $[\![\neg is\text{-}empty\ (idle :: 'a\ idle);\ 0 = size\ idle]\!] \Longrightarrow False$
⟨*proof*⟩

lemma *size-not-empty-2* [*simp*]: $[\![is\text{-}empty\ (idle :: 'a\ idle);\ 0 < size\ idle]\!] \Longrightarrow False$

⟨*proof*⟩

lemma *list-empty*: $list\ idle = [] \longleftrightarrow is\text{-}empty\ idle$
⟨*proof*⟩

lemma *list-not-empty*: $list\ idle \neq [] \longleftrightarrow \neg\ is\text{-}empty\ idle$
⟨*proof*⟩

lemma *list-empty-2* [*simp*]: $[\![list\ idle = [];\ \neg is\text{-}empty\ (idle :: 'a\ idle)]\!] \Longrightarrow False$
⟨*proof*⟩

lemma *list-not-empty-2* [*simp*]: $[\![list\ idle \neq [];\ is\text{-}empty\ (idle :: 'a\ idle)]\!] \Longrightarrow False$
⟨*proof*⟩

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

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

lemma *list-empty-size-2* [*simp*]: $[\![list\ idle \neq [];\ 0 = size\ idle]\!] \Longrightarrow False$
⟨*proof*⟩

lemma *list-not-empty-size-2* [*simp*]: $[\![list\ idle = [];\ 0 < size\ idle]\!] \Longrightarrow False$
⟨*proof*⟩

end

14 Current Proofs

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

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

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

current
⟨*proof*⟩

lemma *drop-first-list* [*simp*]: ⟦*invar current*; *0* < *size current*⟧
⟹ *list* (*drop-first current*) = *tl* (*list current*)
⟨*proof*⟩

lemma *invar-push*: *invar current* ⟹ *invar* (*push x current*)
⟨*proof*⟩

lemma *invar-pop*: ⟦*0* < *size current*; *invar current*; *pop current* = (*x*, *current'*)⟧
⟹ *invar current'*
⟨*proof*⟩

lemma *invar-drop-first*: ⟦*0* < *size current*; *invar current*⟧ ⟹ *invar* (*drop-first current*)
⟨*proof*⟩

lemma *list-size* [*simp*]: ⟦*invar current*; *list current* = []; *0* < *size current*⟧ ⟹ *False*
⟨*proof*⟩

lemma *size-new-push* [*simp*]: *invar current* ⟹ *size-new* (*push x current*) = *Suc* (*size-new current*)
⟨*proof*⟩

lemma *size-push* [*simp*]: *size* (*push x current*) = *Suc* (*size current*)
⟨*proof*⟩

lemma *size-new-pop* [*simp*]: ⟦*0* < *size-new current*; *invar current* ⟧
⟹ *Suc* (*size-new* (*drop-first current*)) = *size-new current*
⟨*proof*⟩

lemma *size-pop* [*simp*]: ⟦*0* < *size current*; *invar current* ⟧
⟹ *Suc* (*size* (*drop-first current*)) = *size current*
⟨*proof*⟩

lemma *size-pop-suc* [*simp*]: ⟦*0* < *size current*; *invar current*; *pop current* = (*x*, *current'*) ⟧
⟹ *Suc* (*size current'*) = *size current*
⟨*proof*⟩

lemma *size-pop-sub*: ⟦*0* < *size current*; *invar current*; *pop current* = (*x*, *current'*) ⟧
⟹ *size current'* = *size current* − *1*
⟨*proof*⟩

lemma *size-drop-first-sub*: ⟦*0* < *size current*; *invar current* ⟧
⟹ *size* (*drop-first current*) = *size current* − *1*

⟨*proof*⟩

end

15 Common Proofs

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

lemma *take-rev-drop*: *take-rev n xs* @ *acc* = *drop* (*length xs* − *n*) (*rev xs*) @ *acc*
⟨*proof*⟩

lemma *take-rev-step*: *xs* ≠ [] ⟹ *take-rev n* (*tl xs*) @ (*hd xs* # *acc*) = *take-rev* (*Suc n*) *xs* @ *acc*
⟨*proof*⟩

lemma *take-rev-empty* [*simp*]: *take-rev n* [] = []
⟨*proof*⟩

lemma *take-rev-tl-hd*:
0 < *n* ⟹ *xs* ≠ [] ⟹ *take-rev n xs* @ *ys* = *take-rev* (*n* − (*Suc 0*)) (*tl xs*) @ (*hd xs* #*ys*)
⟨*proof*⟩

lemma *take-rev-nth*:
n < *length xs* ⟹ *x* = *xs* ! *n* ⟹ *x* # *take-rev n xs* @ *ys* = *take-rev* (*Suc n*) *xs* @ *ys*
⟨*proof*⟩

lemma *step-list* [*simp*]: *invar common* ⟹ *list* (*step common*) = *list common*
⟨*proof*⟩

lemma *step-list-current* [*simp*]: *invar common* ⟹ *list-current* (*step common*) = *list-current common*
⟨*proof*⟩

lemma *push-list* [*simp*]: *list* (*push x common*) = *x* # *list common*
⟨*proof*⟩

lemma *invar-step*: *invar* (*common* :: ′*a common-state*) ⟹ *invar* (*step common*)
⟨*proof*⟩

lemma *invar-push*: *invar common* ⟹ *invar* (*push x common*)
⟨*proof*⟩

lemma *invar-pop*: ⟦
0 < *size common*;
invar common;

pop common = (*x*, *common*′)
⟧ ⟹ *invar common*′
⟨*proof*⟩

lemma *push-list-current* [*simp*]: *list-current* (*push x left*) = *x* # *list-current left*
⟨*proof*⟩

lemma *pop-list* [*simp*]: *invar common* ⟹ *0* < *size common* ⟹ *pop common* = (*x*, *common*′) ⟹
x # *list common*′ = *list common*
⟨*proof*⟩

lemma *pop-list-current*: *invar common* ⟹ *0* < *size common* ⟹ *pop common* = (*x*, *common*′)
⟹ *x* # *list-current common*′ = *list-current common*
⟨*proof*⟩

lemma *list-current-size* [*simp*]:
⟦*0* < *size common*; *list-current common* = []; *invar common*⟧ ⟹ *False*
⟨*proof*⟩

lemma *list-size* [*simp*]: ⟦*0* < *size common*; *list common* = []; *invar common*⟧ ⟹ *False*
⟨*proof*⟩

lemma *step-size* [*simp*]: *invar* (*common* :: ′*a common-state*) ⟹ *size* (*step common*) = *size common*
⟨*proof*⟩

lemma *step-size-new* [*simp*]: *invar* (*common* :: ′*a common-state*)
⟹ *size-new* (*step common*) = *size-new common*
⟨*proof*⟩

lemma *remaining-steps-step* [*simp*]: ⟦*invar* (*common* :: ′*a common-state*); *remaining-steps common* > *0*⟧
⟹ *Suc* (*remaining-steps* (*step common*)) = *remaining-steps common*
⟨*proof*⟩

lemma *remaining-steps-step-sub* [*simp*]: ⟦*invar* (*common* :: ′*a common-state*)⟧
⟹ *remaining-steps* (*step common*) = *remaining-steps common* − *1*
⟨*proof*⟩

lemma *remaining-steps-step-0* [*simp*]: ⟦*invar* (*common* :: ′*a common-state*); *remaining-steps common* = *0*⟧
⟹ *remaining-steps* (*step common*) = *0*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *remaining-steps-pop*: ⟦*invar common*; *pop common* = (*x*, *common*′)⟧
⟹ *remaining-steps common*′ ≤ *remaining-steps common*
⟨*proof*⟩

lemma *size-push* [*simp*]: *invar common* ⟹ *size* (*push x common*) = *Suc* (*size common*)
⟨*proof*⟩

lemma *size-new-push* [*simp*]: *invar common* ⟹ *size-new* (*push x common*) = *Suc* (*size-new common*)
⟨*proof*⟩

lemma *size-pop* [*simp*]: ⟦*invar common*; *0* < *size common*; *pop common* = (*x*, *common*′)⟧
⟹ *Suc* (*size common*′) = *size common*
⟨*proof*⟩

lemma *size-new-pop* [*simp*]: ⟦*invar common*; *0* < *size-new common*; *pop common* = (*x*, *common*′)⟧
⟹ *Suc* (*size-new common*′) = *size-new common*
⟨*proof*⟩

lemma *size-size-new*: ⟦*invar* (*common* :: ′*a common-state*); *0* < *size common*⟧ ⟹ *0* < *size-new common*
⟨*proof*⟩

end

16 Big Proofs

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

lemma *step-list* [*simp*]: *invar big* ⟹ *list* (*step big*) = *list big*
⟨*proof*⟩

lemma *step-list-current* [*simp*]: *invar big* ⟹ *list-current* (*step big*) = *list-current big*
⟨*proof*⟩

lemma *push-list* [*simp*]: *list* (*push x big*) = *x* # *list big*
⟨*proof*⟩

lemma *list-Big1*: ⟦
0 < *size* (*Big1 current big aux count*);
invar (*Big1 current big aux count*)

⟧ ⟹ *first current # list (Big1 (drop-first current) big aux count) =*
list (Big1 current big aux count)
⟨*proof*⟩

lemma *size-list* [*simp*]: ⟦*0 < size big*; *invar big*; *list big = []*⟧ ⟹ *False*
⟨*proof*⟩

lemma *pop-list* [*simp*]: ⟦*0 < size big*; *invar big*; *Big.pop big = (x, big′)*⟧
⟹ *x # list big′ = list big*
⟨*proof*⟩

lemma *pop-list-tl*: ⟦*0 < size big*; *invar big*; *pop big = (x, big′)*⟧ ⟹ *list big′ = tl (list big)*
⟨*proof*⟩

lemma *invar-step*: *invar (big :: ′a big-state)* ⟹ *invar (step big)*
⟨*proof*⟩

lemma *invar-push*: *invar big* ⟹ *invar (push x big)*
⟨*proof*⟩

lemma *invar-pop*: ⟦
0 < size big;
invar big;
pop big = (x, big′)
⟧ ⟹ *invar big′*
⟨*proof*⟩

lemma *push-list-current* [*simp*]: *list-current (push x big) = x # list-current big*
⟨*proof*⟩

lemma *pop-list-current* [*simp*]: ⟦*invar big*; *0 < size big*; *Big.pop big = (x, big′)*⟧
⟹ *x # list-current big′ = list-current big*
⟨*proof*⟩

lemma *list-current-size*: ⟦*0 < size big*; *list-current big = []*; *invar big*⟧ ⟹ *False*
⟨*proof*⟩

lemma *step-size*: *invar (big :: ′a big-state)* ⟹ *size big = size (step big)*
⟨*proof*⟩

lemma *remaining-steps-step* [*simp*]: ⟦*invar (big :: ′a big-state)*; *remaining-steps big > 0*⟧
⟹ *Suc (remaining-steps (step big)) = remaining-steps big*
⟨*proof*⟩

lemma *remaining-steps-step-0* [*simp*]: ⟦*invar (big :: ′a big-state)*; *remaining-steps*

big = *0*⟧
  ⟹ *remaining-steps* (*step big*) = *0*
⟨*proof*⟩

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

lemma *remaining-steps-pop*: ⟦*invar big*; *pop big* = (*x*, *big'*)⟧
  ⟹ *remaining-steps big'* ≤ *remaining-steps big*
⟨*proof*⟩

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

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

lemma *size-pop* [*simp*]: ⟦*invar big*; *0* < *size big*; *pop big* = (*x*, *big'*)⟧
  ⟹ *Suc* (*size big'*) = *size big*
⟨*proof*⟩

lemma *size-new-pop* [*simp*]: ⟦*invar big*; *0* < *size-new big*; *pop big* = (*x*, *big'*)⟧
  ⟹ *Suc* (*size-new big'*) = *size-new big*
⟨*proof*⟩

lemma *size-size-new*: ⟦*invar* (*big* :: *'a big-state*); *0* < *size big*⟧ ⟹ *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*) ⟹ *size* (*step small*) = *size small*
⟨*proof*⟩

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

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

lemma *size-new-push* [*simp*]: *invar small* ⟹ *size-new* (*push x small*) = *Suc* (*size-new small*)
⟨*proof*⟩

lemma *size-pop* [*simp*]: ⟦*invar small*; *0* < *size small*; *pop small* = (*x*, *small*′)⟧
⟹ *Suc* (*size small*′) = *size small*
⟨*proof*⟩

lemma *size-new-pop* [*simp*]: ⟦*invar small*; *0* < *size-new small*; *pop small* = (*x*, *small*′)⟧
⟹ *Suc* (*size-new small*′) = *size-new small*
⟨*proof*⟩

lemma *size-size-new*: ⟦*invar* (*small* :: ′*a small-state*); *0* < *size small*⟧ ⟹ *0* < *size-new small*
⟨*proof*⟩

lemma *step-list-current* [*simp*]: *invar small* ⟹ *list-current* (*step small*) = *list-current small*
⟨*proof*⟩

lemma *step-list-common* [*simp*]:
⟦*small* = *Small3 common*; *invar small*⟧ ⟹ *list* (*step small*) = *list small*
⟨*proof*⟩

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

lemma *invar-step*: *invar* (*small* :: ′*a small-state*) ⟹ *invar* (*step small*)
⟨*proof*⟩

lemma *invar-push*: *invar small* ⟹ *invar* (*push x small*)
⟨*proof*⟩

lemma *invar-pop*: ⟦
0 < *size small*;
invar small;
pop small = (*x*, *small*′)
⟧ ⟹ *invar small*′
⟨*proof*⟩

lemma *push-list-common* [*simp*]: *small* = *Small3 common* ⟹ *list* (*push x small*) = *x* # *list small*

⟨*proof*⟩

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

lemma *pop-list-current* [*simp*]: ⟦*invar small*; *0* < *size small*; *Small.pop small* = (*x*, *small′*)⟧
⟹ *x* # *list-current small′* = *list-current small*
⟨*proof*⟩

lemma *list-current-size* [*simp*]: ⟦*0* < *size small*; *list-current small* = []; *invar small*⟧ ⟹ *False*
⟨*proof*⟩

lemma *list-Small2* [*simp*]: ⟦
0 < *size* (*Small2 current auxS big newS count*);
invar (*Small2 current auxS big newS count*)
⟧ ⟹
fst (*Current.pop current*) # *list* (*Small2* (*drop-first current*) *auxS big newS count*) =
list (*Small2 current auxS big newS 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*:
invar states ⟹ *list-big-first states* = *list-current-big-first states*
⟨*proof*⟩

lemma *step-lists* [*simp*]: *invar states* ⟹ *lists* (*step states*) = *lists states*
⟨*proof*⟩

lemma *step-lists-current* [*simp*]:
invar states ⟹ *lists-current* (*step states*) = *lists-current states*
⟨*proof*⟩

lemma *push-big*: *lists* (*States dir big small*) = (*big′*, *small′*)
⟹ *lists* (*States dir* (*Big.push x big*) *small*) = (*x* # *big′*, *small′*)
⟨*proof*⟩

lemma *push-small-lists*:
invar (*States dir big small*)
$\Longrightarrow$ *lists* (*States dir big* (*Small.push x small*)) = (*big'*, *x* # *small'*) $\longleftrightarrow$
lists (*States dir big small*) = (*big'*, *small'*)
⟨*proof*⟩

lemma *list-small-big*:
list-small-first (*States dir big small*) = *list-current-small-first* (*States dir big small*) $\longleftrightarrow$
list-big-first (*States dir big small*) = *list-current-big-first* (*States dir big small*)
⟨*proof*⟩

lemma *list-big-first-pop-big* [*simp*]: ⟦
invar (*States dir big small*);
0 < *size big*;
Big.pop big = (*x*, *big'*)⟧
$\Longrightarrow$ *x* # *list-big-first* (*States dir big' small*) = *list-big-first* (*States dir big small*)
⟨*proof*⟩

lemma *list-current-big-first-pop-big* [*simp*]: ⟦
invar (*States dir big small*);
0 < *size big*;
Big.pop big = (*x*, *big'*)⟧
$\Longrightarrow$ *x* # *list-current-big-first* (*States dir big' small*) =
list-current-big-first (*States dir big small*)
⟨*proof*⟩

lemma *lists-big-first-pop-big*: ⟦
invar (*States dir big small*);
0 < *size big*;
Big.pop big = (*x*, *big'*)⟧
$\Longrightarrow$ *list-big-first* (*States dir big' small*) = *list-current-big-first* (*States dir big' small*)
⟨*proof*⟩

lemma *lists-small-first-pop-big*: ⟦
invar (*States dir big small*);
0 < *size big*;
Big.pop big = (*x*, *big'*)⟧
$\Longrightarrow$ *list-small-first* (*States dir big' small*) = *list-current-small-first* (*States dir big' small*)
⟨*proof*⟩

lemma *list-small-first-pop-small* [*simp*]: ⟦
invar (*States dir big small*);
0 < *size small*;
Small.pop small = (*x*, *small'*)⟧
$\Longrightarrow$ *x* # *list-small-first* (*States dir big small'*) = *list-small-first* (*States dir big*

small)
⟨*proof*⟩

lemma *list-current-small-first-pop-small* [*simp*]: ⟦
invar (*States dir big small*);
0 < *size small*;
Small.pop small = (*x*, *small'*)⟧
⟹ *x* # *list-current-small-first* (*States dir big small'*) =
list-current-small-first (*States dir big small*)
⟨*proof*⟩

lemma *lists-small-first-pop-small*: ⟦
invar (*States dir big small*);
0 < *size small*;
Small.pop small = (*x*, *small'*)⟧
⟹ *list-small-first* (*States dir big small'*) = *list-current-small-first* (*States dir big small'*)
⟨*proof*⟩

lemma *invars-pop-big*: ⟦
invar (*States dir big small*);
0 < *size big*;
Big.pop big = (*x*, *big'*)⟧
⟹ *invar big'* ∧ *invar small*
⟨*proof*⟩

lemma *invar-pop-big-aux*: ⟦
invar (*States dir big small*);
0 < *size big*;
Big.pop big = (*x*, *big'*)⟧
⟹ (*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*
)
⟨*proof*⟩

lemma *invar-pop-big*: ⟦
invar (*States dir big small*);
0 < *size big*;
Big.pop big = (*x*, *big'*)⟧
⟹ *invar* (*States dir big' small*)
⟨*proof*⟩

lemma *invars-pop-small*: ⟦
invar (*States dir big small*);
0 < *size small*;

Small.pop small = (*x*, *small'*)⟧
⟹ *invar big* ∧ *invar small'*
⟨*proof*⟩

lemma *invar-pop-small-aux*: ⟦
invar (*States dir big small*);
0 < *size small*;
Small.pop small = (*x*, *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*
)
⟨*proof*⟩

lemma *invar-pop-small*: ⟦
invar (*States dir big small*);
0 < *size small*;
Small.pop small = (*x*, *small'*)
⟧ ⟹ *invar* (*States dir big small'*)
⟨*proof*⟩

lemma *invar-push-big*: *invar* (*States dir big small*) ⟹ *invar* (*States dir* (*Big.push x big*) *small*)
⟨*proof*⟩

lemma *invar-push-small*: *invar* (*States dir big small*)
⟹ *invar* (*States dir big* (*Small.push x small*))
⟨*proof*⟩

lemma *step-invars*:⟦*invar states*; *step states* = *States dir big small*⟧ ⟹ *invar big* ∧ *invar small*
⟨*proof*⟩

lemma *step-lists-small-first*: *invar states* ⟹
list-small-first (*step states*) = *list-current-small-first* (*step states*)
⟨*proof*⟩

lemma *invar-step-aux*: *invar states* ⟹(*case step states of*
(*States* - (*Big1* - *big* - *count*) (*Small1* (*Current* - - *old remained*) *small* -)) ⇒
size big − *count* = *remained* − *size old* ∧ *count* ≥ *size small*
| (*States* - - (*Small1* - - -)) ⇒ *False*
| (*States* - (*Big1* - - - -) -) ⇒ *False*
| - ⇒ *True*
)
⟨*proof*⟩

lemma *invar-step*: *invar* (*states* :: *'a states*) $\Longrightarrow$ *invar* (*step states*)
⟨*proof*⟩

lemma *step-consistent* [*simp*]:
⟦⋀*states*. *invar* (*states* :: *'a states*) $\Longrightarrow$ *P* (*step states*) = *P states*; *invar states*⟧
$\Longrightarrow$ *P states* = *P* ((*step* ^^ *n*) *states*)
⟨*proof*⟩

lemma *step-consistent-2*:
⟦⋀*states*. ⟦*invar* (*states* :: *'a states*); *P states*⟧ $\Longrightarrow$ *P* (*step states*); *invar states*; *P states*⟧
$\Longrightarrow$ *P* ((*step* ^^ *n*) *states*)
⟨*proof*⟩

lemma *size-ok'-Suc*: *size-ok' states* (*Suc steps*) $\Longrightarrow$ *size-ok' states steps*
⟨*proof*⟩

lemma *size-ok'-decline*: *size-ok' states x* $\Longrightarrow$ $x \geq y$ $\Longrightarrow$ *size-ok' states y*
⟨*proof*⟩

lemma *remaining-steps-0* [*simp*]: ⟦*invar* (*states* :: *'a states*); *remaining-steps states* = *0*⟧
$\Longrightarrow$ *remaining-steps* (*step states*) = *0*
⟨*proof*⟩

lemma *remaining-steps-0'*: ⟦*invar* (*states* :: *'a states*); *remaining-steps states* = *0*⟧
$\Longrightarrow$ *remaining-steps* ((*step* ^^ *n*) *states*) = *0*
⟨*proof*⟩

lemma *remaining-steps-decline-Suc*:
⟦*invar* (*states* :: *'a states*); *0* < *remaining-steps states*⟧
$\Longrightarrow$ *Suc* (*remaining-steps* (*step states*)) = *remaining-steps states*
⟨*proof*⟩

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

lemma *remaining-steps-decline*: *invar* (*states* :: *'a states*)
$\Longrightarrow$ *remaining-steps* (*step states*) ≤ *remaining-steps states*
⟨*proof*⟩

lemma *remaining-steps-decline-n-steps* [*simp*]:
⟦*invar* (*states* :: *'a states*); *remaining-steps states* ≤ *n*⟧
$\Longrightarrow$ *remaining-steps* ((*step* ^^ *n*) *states*) = *0*
⟨*proof*⟩

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

$\llbracket n \leq$ *remaining-steps states*; *invar* (*states* :: *'a states*)$\rrbracket$
$\Longrightarrow$ *remaining-steps* ((*step* ^^ *n*) *states*) + *n* = *remaining-steps states*
⟨*proof*⟩

lemma *remaining-steps-n-steps-sub* [*simp*]: *invar* (*states* :: *'a states*)
$\Longrightarrow$ *remaining-steps* ((*step* ^^ *n*) *states*) = *remaining-steps states* − *n*
⟨*proof*⟩

lemma *step-size-new-small* [*simp*]:
$\llbracket$*invar* (*States dir big small*); *step* (*States dir big small*) = *States dir' big' small'*$\rrbracket$
$\Longrightarrow$ *size-new small'* = *size-new small*
⟨*proof*⟩

lemma *step-size-new-small-2* [*simp*]:
invar states $\Longrightarrow$ *size-new-small* (*step states*) = *size-new-small states*
⟨*proof*⟩

lemma *step-size-new-big* [*simp*]:
$\llbracket$*invar* (*States dir big small*); *step* (*States dir big small*) = *States dir' big' small'*$\rrbracket$
$\Longrightarrow$ *size-new big'* = *size-new big*
⟨*proof*⟩

lemma *step-size-new-big-2* [*simp*]:
invar states $\Longrightarrow$ *size-new-big* (*step states*) = *size-new-big states*
⟨*proof*⟩

lemma *step-size-small* [*simp*]:
$\llbracket$*invar* (*States dir big small*); *step* (*States dir big small*) = *States dir' big' small'*$\rrbracket$
$\Longrightarrow$ *size small'* = *size small*
⟨*proof*⟩

lemma *step-size-small-2* [*simp*]:
invar states $\Longrightarrow$ *size-small* (*step states*) = *size-small states*
⟨*proof*⟩

lemma *step-size-big* [*simp*]:
$\llbracket$*invar* (*States dir big small*); *step* (*States dir big small*) = *States dir' big' small'*$\rrbracket$
$\Longrightarrow$ *size big'* = *size big*
⟨*proof*⟩

lemma *step-size-big-2* [*simp*]:
invar states $\Longrightarrow$ *size-big* (*step states*) = *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* ≤ *3* ∗ *size-new small*
$\rrbracket \Longrightarrow$ *size-new big'* + *remaining-steps* (*States dir' big' small'*) + *2* ≤ *3* ∗ *size-new*

small'
⟨*proof*⟩

lemma *step-size-ok-2*: ⟦
invar (*States dir big small*);
step (*States dir big small*) = *States dir' big' small'*;
size-new small + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new big*
⟧ ⟹ *size-new small'* + *remaining-steps* (*States dir' big' small'*) + *2* ≤ *3* ∗ *size-new big'*
⟨*proof*⟩

lemma *step-size-ok-3*: ⟦
invar (*States dir big small*);
step (*States dir big small*) = *States dir' big' small'*;
remaining-steps (*States dir big small*) + *1* ≤ *4* ∗ *size small*
⟧ ⟹ *remaining-steps* (*States dir' big' small'*) + *1* ≤ *4* ∗ *size small'*
⟨*proof*⟩

lemma *step-size-ok-4*: ⟦
invar (*States dir big small*);
step (*States dir big small*) = *States dir' big' small'*;
remaining-steps (*States dir big small*) + *1* ≤ *4* ∗ *size big*
⟧ ⟹ *remaining-steps* (*States dir' big' small'*) + *1* ≤ *4* ∗ *size big'*
⟨*proof*⟩

lemma *step-size-ok*: ⟦*invar states*; *size-ok states*⟧ ⟹ *size-ok* (*step states*)
⟨*proof*⟩

lemma *step-n-size-ok*: ⟦*invar states*; *size-ok states*⟧ ⟹ *size-ok* ((*step* ^^ *n*) *states*)
⟨*proof*⟩

lemma *step-push-size-small* [*simp*]: ⟦
invar (*States dir big small*);
step (*States dir big* (*Small.push x small*)) = *States dir' big' small'*
⟧ ⟹ *size small'* = *Suc* (*size small*)
⟨*proof*⟩

lemma *step-push-size-new-small* [*simp*]: ⟦
invar (*States dir big small*);
step (*States dir big* (*Small.push x small*)) = *States dir' big' small'*
⟧ ⟹ *size-new small'* = *Suc* (*size-new small*)
⟨*proof*⟩

lemma *step-push-size-big* [*simp*]: ⟦
invar (*States dir big small*);
step (*States dir* (*Big.push x big*) *small*) = *States dir' big' small'*
⟧ ⟹ *size big'* = *Suc* (*size big*)
⟨*proof*⟩

lemma *step-push-size-new-big* [*simp*]: ⟦
invar (*States dir big small*);
step (*States dir* (*Big.push x big*) *small*) = *States dir′ big′ small′*
⟧ ⟹ *size-new big′* = *Suc* (*size-new big*)
⟨*proof*⟩

lemma *step-pop-size-big* [*simp*]: ⟦
invar (*States dir big small*);
0 < *size big*;
Big.pop big = (*x*, *bigP*);
step (*States dir bigP small*) = *States dir′ big′ small′*
⟧ ⟹ *Suc* (*size big′*) = *size big*
⟨*proof*⟩

lemma *step-pop-size-new-big* [*simp*]: ⟦
invar (*States dir big small*);
0 < *size big*; *Big.pop big* = (*x*, *bigP*);
step (*States dir bigP small*) = *States dir′ big′ small′*
⟧ ⟹ *Suc* (*size-new big′*) = *size-new big*
⟨*proof*⟩

lemma *step-n-size-small* [*simp*]: ⟦
invar (*States dir big small*);
(*step* ^^ *n*) (*States dir big small*) = *States dir′ big′ small′*
⟧ ⟹ *size small′* = *size small*
⟨*proof*⟩

lemma *step-n-size-big* [*simp*]:
⟦*invar* (*States dir big small*); (*step* ^^ *n*) (*States dir big small*) = *States dir′ big′ small′*⟧
⟹ *size big′* = *size big*
⟨*proof*⟩

lemma *step-n-size-new-small* [*simp*]:
⟦*invar* (*States dir big small*); (*step* ^^ *n*) (*States dir big small*) = *States dir′ big′ small′*⟧
⟹ *size-new small′* = *size-new small*
⟨*proof*⟩

lemma *step-n-size-new-big* [*simp*]:
⟦*invar* (*States dir big small*); (*step* ^^ *n*) (*States dir big small*) = *States dir′ big′ small′*⟧
⟹ *size-new big′* = *size-new big*
⟨*proof*⟩

lemma *step-n-push-size-small* [*simp*]: ⟦
invar (*States dir big small*);
(*step* ^^ *n*) (*States dir big* (*Small.push x small*)) = *States dir′ big′ small′*
⟧ ⟹ *size small′* = *Suc* (*size small*)

⟨proof⟩

lemma *step-n-push-size-new-small* [*simp*]: ⟦
invar (*States dir big small*);
(*step* ^^ *n*) (*States dir big* (*Small.push x small*)) = *States dir′ big′ small′*
⟧ ⟹ *size-new small′* = *Suc* (*size-new small*)
⟨proof⟩

lemma *step-n-push-size-big* [*simp*]: ⟦
invar (*States dir big small*);
(*step* ^^ *n*) (*States dir* (*Big.push x big*) *small*) = *States dir′ big′ small′*
⟧ ⟹ *size big′* = *Suc* (*size big*)
⟨proof⟩

lemma *step-n-push-size-new-big* [*simp*]: ⟦
invar (*States dir big small*);
(*step* ^^ *n*) (*States dir* (*Big.push x big*) *small*) = *States dir′ big′ small′*
⟧ ⟹ *size-new big′* = *Suc* (*size-new big*)
⟨proof⟩

lemma *step-n-pop-size-small* [*simp*]: ⟦
invar (*States dir big small*);
0 < *size small*;
Small.pop small = (*x*, *smallP*);
(*step* ^^ *n*) (*States dir big smallP*) = *States dir′ big′ small′*
⟧ ⟹ *Suc* (*size small′*) = *size small*
⟨proof⟩

lemma *step-n-pop-size-new-small* [*simp*]: ⟦
invar (*States dir big small*);
0 < *size small*;
Small.pop small = (*x*, *smallP*);
(*step* ^^ *n*) (*States dir big smallP*) = *States dir′ big′ small′*
⟧ ⟹ *Suc* (*size-new small′*) = *size-new small*
⟨proof⟩

lemma *step-n-pop-size-big* [*simp*]: ⟦
invar (*States dir big small*);
0 < *size big*; *Big.pop big* = (*x*, *bigP*);
(*step* ^^ *n*) (*States dir bigP small*) = *States dir′ big′ small′*
⟧ ⟹ *Suc* (*size big′*) = *size big*
⟨proof⟩

lemma *step-n-pop-size-new-big*: ⟦
invar (*States dir big small*);
0 < *size big*; *Big.pop big* = (*x*, *bigP*);
(*step* ^^ *n*) (*States dir bigP small*) = *States dir′ big′ small′*
⟧ ⟹ *Suc* (*size-new big′*) = *size-new big*
⟨proof⟩

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

lemma *remaining-steps-pop-small*:
⟦*invar* (*States dir big small*); *0* < *size small*; *Small.pop small* = (*x*, *smallP*)⟧
$\Longrightarrow$ *remaining-steps* (*States dir big smallP*) ≤ *remaining-steps* (*States dir big small*)
⟨*proof*⟩

lemma *remaining-steps-pop-big*:
⟦*invar* (*States dir big small*); *0* < *size big*; *Big.pop big* = (*x*, *bigP*)⟧
$\Longrightarrow$ *remaining-steps* (*States dir bigP small*) ≤ *remaining-steps* (*States dir big small*)
⟨*proof*⟩

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

lemma *step-4-remaining-steps-push-big* [*simp*]: ⟦
invar (*States dir big small*);
4 ≤ *remaining-steps* (*States dir big small*);
(*step*^^*4*) (*States dir* (*Big.push x big*) *small*) = *States dir′ big′ small′*⟧
$\Longrightarrow$ *remaining-steps* (*States dir′ big′ small′*) = *remaining-steps* (*States dir big small*) − *4*
⟨*proof*⟩

lemma *step-4-remaining-steps-push-small* [*simp*]: ⟦
invar (*States dir big small*);
4 ≤ *remaining-steps* (*States dir big small*);
(*step*^^*4*) (*States dir big* (*Small.push x small*)) = *States dir′ big′ small′*
⟧ $\Longrightarrow$ *remaining-steps* (*States dir′ big′ small′*) = *remaining-steps* (*States dir big small*) − *4*
⟨*proof*⟩

lemma *step-4-remaining-steps-pop-big*: ⟦
invar (*States dir big small*);
0 < *size big*;
Big.pop big = (*x*, *bigP*);
4 ≤ *remaining-steps* (*States dir bigP small*);
(*step*^^*4*) (*States dir bigP small*) = *States dir′ big′ small′*
⟧ $\Longrightarrow$ *remaining-steps* (*States dir′ big′ small′*) ≤ *remaining-steps* (*States dir big small*) − *4*
⟨*proof*⟩

lemma *step-4-remaining-steps-pop-small*: ⟦
invar (*States dir big small*);
0 < *size small*;
Small.pop small = (*x*, *smallP*);
4 ≤ *remaining-steps* (*States dir big smallP*);
(*step*^^*4*) (*States dir big smallP*) = *States dir′ big′ small′*
⟧ ⟹ *remaining-steps* (*States dir′ big′ small′*) ≤ *remaining-steps* (*States dir big small*) − *4*
⟨*proof*⟩

lemma *step-4-pop-small-size-ok-1*: ⟦
invar (*States dir big small*);
0 < *size small*;
Small.pop small = (*x*, *smallP*);
4 ≤ *remaining-steps* (*States dir big smallP*);
(*step*^^*4*) (*States dir big smallP*) = *States dir′ big′ small′*;
remaining-steps (*States dir big small*) + *1* ≤ *4* ∗ *size small*
⟧ ⟹ *remaining-steps* (*States dir′ big′ small′*) + *1* ≤ *4* ∗ *size small′*
⟨*proof*⟩

lemma *step-4-pop-big-size-ok-1*: ⟦
invar (*States dir big small*);
0 < *size big*; *Big.pop big* = (*x*, *bigP*);
4 ≤ *remaining-steps* (*States dir bigP small*);
(*step*^^*4*) (*States dir bigP small*) = *States dir′ big′ small′*;
remaining-steps (*States dir big small*) + *1* ≤ *4* ∗ *size small*
⟧ ⟹ *remaining-steps* (*States dir′ big′ small′*) + *1* ≤ *4* ∗ *size small′*
⟨*proof*⟩

lemma *step-4-pop-small-size-ok-2*: ⟦
invar (*States dir big small*);
0 < *size small*;
Small.pop small = (*x*, *smallP*);
4 ≤ *remaining-steps* (*States dir big smallP*);
(*step*^^*4*) (*States dir big smallP*) = *States dir′ big′ small′*;
remaining-steps (*States dir big small*) + *1* ≤ *4* ∗ *size big*
⟧ ⟹ *remaining-steps* (*States dir′ big′ small′*) + *1* ≤ *4* ∗ *size big′*
⟨*proof*⟩

lemma *step-4-pop-big-size-ok-2*:
assumes
invar (*States dir big small*)
0 < *size big*
Big.pop big = (*x*, *bigP*)
remaining-steps (*States dir bigP small*) ≥ *4*
((*step* ^^ *4*) (*States dir bigP small*)) = *States dir′ big′ small′*
remaining-steps (*States dir big small*) + *1* ≤ *4* ∗ *size big*
shows
remaining-steps (*States dir′ big′ small′*) + *1* ≤ *4* ∗ *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* ^^ *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*: ⟦
0 < *size big*;
4 ≤ *remaining-steps* (*States dir big small*);
size-new small + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new big*
⟧ ⟹ *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* ^^ *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*: ⟦
0 < *size small*;
4 ≤ *remaining-steps* (*States dir big small*);
size-new big + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new small*
⟧ ⟹ *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 < size small
Small.pop small = (x, smallP)
remaining-steps (States dir big smallP) ≥ 4
((step ^^ 4) (States dir big smallP)) = (States dir′ big′ small′)
size-new big + remaining-steps (States dir big small) + 2 ≤ 3 ∗ size-new small
shows
size-new big′ + remaining-steps (States dir′ big′ small′) + 2 ≤ 3 ∗ size-new small′
⟨*proof*⟩

lemma *step-4-pop-big-size-ok-4-aux*: ⟦
0 < size big;
4 ≤ remaining-steps (States dir big small);
size-new big + remaining-steps (States dir big small) + 2 ≤ 3 ∗ size-new small
⟧ ⟹ *size-new big − 1 + (remaining-steps (States dir big small) − 4) + 2 ≤ 3 ∗ size-new small*
⟨*proof*⟩

lemma *step-4-pop-big-size-ok-4*:
assumes
invar (States dir big small)
0 < size big
Big.pop big = (x, bigP)
remaining-steps (States dir bigP small) ≥ 4
((step ^^ 4) (States dir bigP small)) = (States dir′ big′ small′)
size-new big + remaining-steps (States dir big small) + 2 ≤ 3 ∗ size-new small
shows
size-new big′ + remaining-steps (States dir′ big′ small′) + 2 ≤ 3 ∗ size-new small′
⟨*proof*⟩

lemma *step-4-push-small-size-ok-1*: ⟦
invar (States dir big small);
4 ≤ remaining-steps (States dir big small);
(step^^4) (States dir big (Small.push x small)) = States dir′ big′ small′;
remaining-steps (States dir big small) + 1 ≤ 4 ∗ size small
⟧ ⟹ *remaining-steps (States dir′ big′ small′) + 1 ≤ 4 ∗ size small′*
⟨*proof*⟩

lemma *step-4-push-big-size-ok-1*: ⟦
invar (States dir big small);
4 ≤ remaining-steps (States dir big small);
(step^^4) (States dir (Big.push x big) small) = States dir′ big′ small′;
remaining-steps (States dir big small) + 1 ≤ 4 ∗ size small
⟧ ⟹ *remaining-steps (States dir′ big′ small′) + 1 ≤ 4 ∗ size small′*
⟨*proof*⟩

lemma *step-4-push-small-size-ok-2*: ⟦

invar (*States dir big small*);
4 ≤ *remaining-steps* (*States dir big small*);
(*step*^^*4*) (*States dir big* (*Small.push x small*)) = *States dir′ big′ small′*;
remaining-steps (*States dir big small*) + *1* ≤ *4* ∗ *size big*
⟧ ⟹ *remaining-steps* (*States dir′ big′ small′*) + *1* ≤ *4* ∗ *size big′*
⟨*proof*⟩

lemma *step-4-push-big-size-ok-2*: ⟦
invar (*States dir big small*);
4 ≤ *remaining-steps* (*States dir big small*);
(*step*^^*4*) (*States dir* (*Big.push x big*) *small*) = *States dir′ big′ small′*;
remaining-steps (*States dir big small*) + *1* ≤ *4* ∗ *size big*
⟧ ⟹ *remaining-steps* (*States dir′ big′ small′*) + *1* ≤ *4* ∗ *size big′*
⟨*proof*⟩

lemma *step-4-push-small-size-ok-3-aux*: ⟦
4 ≤ *remaining-steps* (*States dir big small*);
size-new small + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new big*
⟧ ⟹ *Suc* (*size-new small*) + (*remaining-steps* (*States dir big small*) − *4*) + *2* ≤ *3* ∗ *size-new big*
⟨*proof*⟩

lemma *step-4-push-small-size-ok-3*: ⟦
invar (*States dir big small*);
4 ≤ *remaining-steps* (*States dir big small*);
(*step*^^*4*) (*States dir big* (*Small.push x small*)) = *States dir′ big′ small′*;
size-new small + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new big*
⟧ ⟹ *size-new small′* + *remaining-steps* (*States dir′ big′ small′*) + *2* ≤ *3* ∗ *size-new big′*
⟨*proof*⟩

lemma *step-4-push-big-size-ok-3-aux*: ⟦
4 ≤ *remaining-steps* (*States dir big small*);
size-new small + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new big*
⟧ ⟹ *size-new small* + (*remaining-steps* (*States dir big small*) − *4*) + *2* ≤ *3* ∗ *Suc* (*size-new big*)
⟨*proof*⟩

lemma *step-4-push-big-size-ok-3*: ⟦
invar (*States dir big small*);
4 ≤ *remaining-steps* (*States dir big small*);
(*step*^^*4*) (*States dir* (*Big.push x big*) *small*) = *States dir′ big′ small′*;
size-new small + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new big*
⟧ ⟹ *size-new small′* + *remaining-steps* (*States dir′ big′ small′*) + *2* ≤ *3* ∗ *size-new big′*
⟨*proof*⟩

lemma *step-4-push-small-size-ok-4-aux*: ⟦
4 ≤ *remaining-steps* (*States dir big small*);

size-new big + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new small*
⟧ ⟹ *size-new big* + (*remaining-steps* (*States dir big small*) − *4*) + *2* ≤ *3* ∗ *Suc* (*size-new small*)
⟨*proof*⟩

lemma *step-4-push-small-size-ok-4*: ⟦
invar (*States dir big small*);
4 ≤ *remaining-steps* (*States dir big small*);
(*step*^^*4*) (*States dir big* (*Small.push x small*)) = *States dir′ big′ small′*;
size-new big + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new small*
⟧ ⟹ *size-new big′* + *remaining-steps* (*States dir′ big′ small′*) + *2* ≤ *3* ∗ *size-new small′*
⟨*proof*⟩

lemma *step-4-push-big-size-ok-4-aux*: ⟦
4 ≤ *remaining-steps* (*States dir big small*);
size-new big + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new small*
⟧ ⟹ *Suc* (*size-new big*) + (*remaining-steps* (*States dir big small*) − *4*) + *2* ≤ *3* ∗ *size-new small*
⟨*proof*⟩

lemma *step-4-push-big-size-ok-4*: ⟦
invar (*States dir big small*);
4 ≤ *remaining-steps* (*States dir big small*);
(*step*^^*4*) (*States dir* (*Big.push x big*) *small*) = *States dir′ big′ small′*;
size-new big + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new small*
⟧ ⟹ *size-new big′* + *remaining-steps* (*States dir′ big′ small′*) + *2* ≤ *3* ∗ *size-new small′*
⟨*proof*⟩

lemma *step-4-push-small-size-ok*: ⟦
invar (*States dir big small*);
4 ≤ *remaining-steps* (*States dir big small*);
size-ok (*States dir big small*)
⟧ ⟹ *size-ok* ((*step*^^*4*) (*States dir big* (*Small.push x small*)))
⟨*proof*⟩

lemma *step-4-push-big-size-ok*: ⟦
invar (*States dir big small*);
4 ≤ *remaining-steps* (*States dir big small*);
size-ok (*States dir big small*)
⟧ ⟹ *size-ok* ((*step*^^*4*) (*States dir* (*Big.push x big*) *small*))
⟨*proof*⟩

lemma *step-4-pop-small-size-ok*: ⟦
invar (*States dir big small*);
0 < *size small*;
Small.pop small = (*x*, *smallP*);
4 ≤ *remaining-steps* (*States dir big smallP*);

size-ok (*States dir big small*)
⟧ ⟹ *size-ok* ((*step*⁀⁀*4*) (*States dir big smallP*))
⟨*proof*⟩

lemma *step-4-pop-big-size-ok*: ⟦
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*)
⟧ ⟹ *size-ok* ((*step*⁀⁀*4*) (*States dir bigP small*))
⟨*proof*⟩

lemma *size-ok-size-small*: *size-ok* (*States dir big small*) ⟹ *0* < *size small*
⟨*proof*⟩

lemma *size-ok-size-big*: *size-ok* (*States dir big small*) ⟹ *0* < *size big*
⟨*proof*⟩

lemma *size-ok-size-new-small*: *size-ok* (*States dir big small*) ⟹ *0* < *size-new small*
⟨*proof*⟩

lemma *size-ok-size-new-big*: *size-ok* (*States dir big small*) ⟹ *0* < *size-new big*
⟨*proof*⟩

lemma *step-size-ok'*: ⟦*invar states*; *size-ok' states n*⟧ ⟹ *size-ok'* (*step states*) *n*
⟨*proof*⟩

lemma *step-same*: *step* (*States dir big small*) = *States dir' big' small'* ⟹ *dir* = *dir'*
⟨*proof*⟩

lemma *step-n-same*: (*step*⁀⁀*n*) (*States dir big small*) = *States dir' big' small'* ⟹ *dir* = *dir'*
⟨*proof*⟩

lemma *step-listL*: *invar states* ⟹ *listL* (*step states*) = *listL states*
⟨*proof*⟩

lemma *step-n-listL*: *invar states* ⟹ *listL* ((*step*⁀⁀*n*) *states*) = *listL states*
⟨*proof*⟩

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*) ⟹ *invar* ((*step*^^*n*) *states*)
⟨*proof*⟩

lemma *step-n-size-ok'*: ⟦*invar states*; *size-ok' states x*⟧ ⟹ *size-ok'* ((*step* ^^ *n*) *states*) *x*
⟨*proof*⟩

lemma *size-ok-steps*: ⟦
invar states;
size-ok' states (*remaining-steps states* − *n*)
⟧ ⟹ *size-ok* ((*step* ^^ *n*) *states*)
⟨*proof*⟩

lemma *remaining-steps-idle*: *invar states*
⟹ *remaining-steps states* = *0* ⟷ (
case states of
States - (*Big2* (*Common.Idle* - -)) (*Small3* (*Common.Idle* - -)) ⇒ *True*
| - ⇒ *False*)
⟨*proof*⟩

lemma *remaining-steps-idle'*:
⟦*invar* (*States dir big small*); *remaining-steps* (*States dir big small*) = *0*⟧
⟹ ∃ *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*]: ⟦*invar deque*; *listL deque* ≠ []; *deqL' deque* = (*x*, *deque'*)⟧
⟹ *x* # *listL deque'* = *listL deque*
⟨*proof*⟩

lemma *list-deqL* [*simp*]:
⟦*invar deque*; *listL deque* ≠ []⟧ ⟹ *listL* (*deqL deque*) = *tl* (*listL deque*)
⟨*proof*⟩

lemma *list-firstL* [*simp*]:
⟦*invar deque*; *listL deque* ≠ []⟧ ⟹ *firstL deque* = *hd* (*listL deque*)

⟨*proof*⟩

lemma *invar-deqL*:
 ⟦*invar deque*; ¬ *is-empty deque*⟧ ⟹ *invar* (*deqL deque*)
⟨*proof*⟩

end

20 Enqueue Proofs

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

lemma *list-enqL*: *invar deque* ⟹ *listL* (*enqL x deque*) = *x* # *listL deque*
⟨*proof*⟩

lemma *invar-enqL*: *invar deque* ⟹ *invar* (*enqL x deque*)
⟨*proof*⟩

end

21 Top-Level Proof

theory *RealTimeDeque-Proof*
imports *RealTimeDeque-Dequeue-Proof RealTimeDeque-Enqueue-Proof*
begin

lemma *swap-lists-left*: *invar* (*States Left big small*) ⟹
 States-Aux.listL (*States Left big small*) = *rev* (*States-Aux.listL* (*States Right big small*))
⟨*proof*⟩

lemma *swap-lists-right*: *invar* (*States Right big small*) ⟹
 States-Aux.listL (*States Right big small*) = *rev* (*States-Aux.listL* (*States Left big small*))
⟨*proof*⟩

lemma *swap-list* [*simp*]: *invar q* ⟹ *listR* (*swap q*) = *listL q*
⟨*proof*⟩

lemma *swap-list′*: *invar q* ⟹ *listL* (*swap q*) = *listR q*
⟨*proof*⟩

lemma *lists-same*: *lists* (*States Left big small*) = *lists* (*States Right big small*)
⟨*proof*⟩

lemma *invar-swap*: *invar q* ⟹ *invar* (*swap q*)

⟨*proof*⟩

lemma *listL-is-empty*: *invar deque* $\Longrightarrow$ *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 deques, 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.