

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	41
17 Small Proofs	48
18 Big + Small Proofs	53
19 Dequeue Proofs	79
20 Enqueue Proofs	89
21 Top-Level Proof	96

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..
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..  
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..

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..
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..

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..  
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  $\Rightarrow$  nat where
  size (Stack left right) = length left + length right

instance..
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  $\Rightarrow$  'a list where
  list (Current extra - old -) = extra @ (Stack-Aux.list old)

instantiation current::(type) invar
begin

fun invar-current :: 'a current  $\Rightarrow$  bool where
  invar (Current extra added - -)  $\iff$  length extra = added

instance..
end

instantiation current::(type) size
begin

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

instance..
end

instantiation current::(type) size-new
begin

fun size-new-current :: 'a current  $\Rightarrow$  nat where

```

```

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

instance..
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..
end

instantiation idle :: (type) is-empty
begin

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

instance..
end

instantiation idle ::(type) invar
begin

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

instance..
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..

end

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

begin

```

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

instance..
end

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

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

instance..
end

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

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

instance..
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..
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..
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..
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..
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..
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..
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..
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..
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
  ^ 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..
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..
end

end

11 Basic Lemma Library

theory *RTD-Util*

imports *Main*

begin

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

lemma *take-last*: $xs \neq [] \implies \text{take } 1 (\text{rev } xs) = [\text{last } xs]$
by(*induction xs*)(*auto simp: take-last-length*)

lemma *take-hd* [*simp*]: $xs \neq [] \implies \text{take } (\text{Suc } 0) xs = [\text{hd } xs]$
by(*induction xs*) *auto*

lemma *cons-tl*: $x \# xs = ys \implies xs = tl\ ys$
by *auto*

lemma *cons-hd*: $x \# xs = ys \implies x = hd\ ys$
by *auto*

lemma *take-hd'*: $ys \neq [] \implies take\ (size\ ys)\ (x \# xs) = take\ (Suc\ (size\ xs))\ ys \implies hd\ ys = x$
by(*induction ys*) *auto*

lemma *rev-app-single*: $rev\ xs\ @\ [x] = rev\ (x \# xs)$
by *auto*

lemma *hd-drop-1* [*simp*]: $xs \neq [] \implies hd\ xs \# drop\ (Suc\ 0)\ xs = xs$
by(*induction xs*) *auto*

lemma *hd-drop* [*simp*]: $n < length\ xs \implies hd\ (drop\ n\ xs) \# drop\ (Suc\ n)\ xs = drop\ n\ xs$
by(*induction xs*)(*auto simp: list.expand tl-drop*)

lemma *take-1*: $0 < x \wedge 0 < y \implies take\ x\ xs = take\ y\ ys \implies take\ 1\ xs = take\ 1\ ys$
by (*metis One-nat-def bot-nat-0.not-eq-extremum hd-take take-Suc take-eq-Nil*)

lemma *last-drop-rev*: $xs \neq [] \implies last\ xs \# drop\ 1\ (rev\ xs) = rev\ xs$
by (*metis One-nat-def hd-drop-1 hd-rev rev.simps(1) rev-rev-ident*)

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

lemma *rev-tl-hd*: $xs \neq [] \implies rev\ (tl\ xs) \ @\ [hd\ xs] = rev\ xs$
by (*simp add: rev-app-single*)

lemma *app-rev*: $as \ @\ rev\ bs = cs \ @\ rev\ ds \implies bs \ @\ rev\ as = ds \ @\ rev\ cs$
by (*metis rev-append rev-rev-ident*)

lemma *tl-drop-2*: $tl\ (drop\ n\ xs) = drop\ (Suc\ n)\ xs$
by (*simp add: drop-Suc tl-drop*)

lemma *Suc-sub*: $Suc\ n = m \implies n = m - 1$
by *simp*

lemma *length-one-hd*: $length\ xs = 1 \implies xs = [hd\ xs]$
by(*induction xs*) *auto*

end

12 Stack Proofs

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

lemma push-list [simp]: list (push x stack) = x # list stack
  by(cases stack) auto

lemma pop-list [simp]: list (pop stack) = tl (list stack)
  by(induction stack rule: pop.induct) auto

lemma first-list [simp]:  $\neg$  is-empty stack  $\implies$  first stack = hd (list stack)
  by(induction stack rule: first.induct) auto

lemma list-empty: list stack = []  $\longleftrightarrow$  is-empty stack
  by(induction stack rule: is-empty-stack.induct) auto

lemma list-not-empty: list stack  $\neq$  []  $\longleftrightarrow$   $\neg$  is-empty stack
  by(induction stack rule: is-empty-stack.induct) auto

lemma list-empty-2 [simp]:  $\llbracket$ list stack  $\neq$  []; is-empty stack $\rrbracket \implies$  False
  by (simp add: list-empty)

lemma list-not-empty-2 [simp]:  $\llbracket$ list stack = [];  $\neg$  is-empty stack $\rrbracket \implies$  False
  by (simp add: list-empty)

lemma list-empty-size: list stack = []  $\longleftrightarrow$  size stack = 0
  by(induction stack) auto

lemma list-not-empty-size: list stack  $\neq$  []  $\longleftrightarrow$  0 < size stack
  by(induction stack) auto

lemma list-empty-size-2 [simp]:  $\llbracket$ list stack  $\neq$  []; size stack = 0 $\rrbracket \implies$  False
  by (simp add: list-empty-size)

lemma list-not-empty-size-2 [simp]:  $\llbracket$ list stack = []; 0 < size stack $\rrbracket \implies$  False
  by (simp add: list-empty-size)

lemma size-push [simp]: size (push x stack) = Suc (size stack)
  by(cases stack) auto

lemma size-pop [simp]: size (pop stack) = size stack - Suc 0
  by(induction stack rule: pop.induct) auto

lemma size-empty: size (stack :: 'a stack) = 0  $\longleftrightarrow$  is-empty stack
  by(induction stack rule: is-empty-stack.induct) auto

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

by(*induction stack rule: is-empty-stack.induct*) *auto*

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

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

lemma *size-list-length* [*simp*]: $\text{length } (\text{list stack}) = \text{size stack}$
by(*cases stack*) *auto*

lemma *first-pop* [*simp*]: $\neg \text{is-empty stack} \implies \text{first stack} \# \text{list } (\text{pop stack}) = \text{list stack}$
by(*induction stack rule: pop.induct*) *auto*

lemma *push-not-empty* [*simp*]: $\llbracket \neg \text{is-empty stack}; \text{is-empty } (\text{push } x \text{ stack}) \rrbracket \implies \text{False}$
by(*induction x stack rule: push.induct*) *auto*

lemma *pop-list-length* [*simp*]: $\neg \text{is-empty stack} \implies \text{Suc } (\text{length } (\text{list } (\text{pop stack}))) = \text{length } (\text{list stack})$
by(*induction stack rule: pop.induct*) *auto*

lemma *first-take*: $\neg \text{is-empty stack} \implies [\text{first stack}] = \text{take } 1 \text{ (list stack)}$
by (*simp add: list-empty*)

lemma *first-take-tl* [*simp*]: $0 < \text{size big} \implies (\text{first big} \# \text{take count } (\text{tl } (\text{list big}))) = \text{take } (\text{Suc count}) \text{ (list big)}$
by(*induction big rule: Stack.first.induct*) *auto*

lemma *first-take-pop* [*simp*]: $\llbracket \neg \text{is-empty stack}; 0 < x \rrbracket \implies \text{first stack} \# \text{take } (x - \text{Suc } 0) \text{ (list } (\text{pop stack})) = \text{take } x \text{ (list stack)}$
by(*induction stack rule: pop.induct*) (*auto simp: take-Cons'*)

lemma [*simp*]: $\text{first } (\text{Stack } [] []) = \text{undefined}$
by (*meson first.elims list.distinct(1) stack.inject*)

lemma *first-hd*: $\text{first stack} = \text{hd } (\text{list stack})$
by(*induction stack rule: first.induct*)(*auto simp: hd-def*)

lemma *pop-tl* [*simp*]: $\text{list } (\text{pop stack}) = \text{tl } (\text{list stack})$
by(*induction stack rule: pop.induct*) *auto*

lemma *pop-drop*: $\text{list } (\text{pop stack}) = \text{drop } 1 \text{ (list stack)}$
by (*simp add: drop-Suc*)

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

by(*induction n*)(*auto simp: drop-Suc tl-drop*)

lemma *popN-size* [*simp*]: $\text{size } ((\text{pop } \sim n) \text{ stack}) = (\text{size stack}) - n$
by(*induction n*) *auto*

lemma *take-first*: $[[0 < \text{size } s1; 0 < \text{size } s2; \text{take } (\text{size } s1) (\text{list } s2) = \text{take } (\text{size } s2) (\text{list } s1)]]$
 $\implies \text{first } s1 = \text{first } s2$
by(*induction s1 rule: first.induct; induction s2 rule: first.induct*) *auto*

end

13 Idle Proofs

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

lemma *push-list* [*simp*]: $\text{list } (\text{push } x \text{ idle}) = x \# \text{list } \text{idle}$
by(*induction idle arbitrary: x*) *auto*

lemma *pop-list* [*simp*]: $[[\neg \text{is-empty } \text{idle}; \text{pop } \text{idle} = (x, \text{idle}')] \implies x \# \text{list } \text{idle}' = \text{list } \text{idle}$
by(*induction idle arbitrary: x*)(*auto simp: list-not-empty*)

lemma *pop-list-tl* [*simp*]:
 $[[\neg \text{is-empty } \text{idle}; \text{pop } \text{idle} = (x, \text{idle}')] \implies x \# (\text{tl } (\text{list } \text{idle})) = \text{list } \text{idle}$
by(*induction idle arbitrary: x*) (*auto simp: list-not-empty*)

lemma *pop-list-tl'* [*simp*]: $[[\text{pop } \text{idle} = (x, \text{idle}')] \implies \text{list } \text{idle}' = \text{tl } (\text{list } \text{idle})$
by(*induction idle arbitrary: x*)(*auto simp: drop-Suc*)

lemma *size-push* [*simp*]: $\text{size } (\text{push } x \text{ idle}) = \text{Suc } (\text{size } \text{idle})$
by(*induction idle arbitrary: x*) *auto*

lemma *size-pop* [*simp*]: $[[\neg \text{is-empty } \text{idle}; \text{pop } \text{idle} = (x, \text{idle}')] \implies \text{Suc } (\text{size } \text{idle}') = \text{size } \text{idle}$
by(*induction idle arbitrary: x*)(*auto simp: size-not-empty*)

lemma *size-pop-sub*: $[[\text{pop } \text{idle} = (x, \text{idle}')] \implies \text{size } \text{idle}' = \text{size } \text{idle} - 1$
by(*induction idle arbitrary: x*) *auto*

lemma *invar-push*: $\text{invar } \text{idle} \implies \text{invar } (\text{push } x \text{ idle})$
by(*induction x idle rule: push.induct*) *auto*

lemma *invar-pop*: $[[\text{invar } \text{idle}; \text{pop } \text{idle} = (x, \text{idle}')] \implies \text{invar } \text{idle}'$
by(*induction idle arbitrary: x rule: pop.induct*) *auto*

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

```

    by(induction idle)(auto simp: size-empty)

lemma size-not-empty: 0 < size idle  $\longleftrightarrow$   $\neg$ is-empty (idle :: 'a idle)
  by(induction idle)(auto simp: size-not-empty)

lemma size-empty-2 [simp]:  $\llbracket \neg$ is-empty (idle :: 'a idle); 0 = size idle  $\rrbracket \implies$  False
  by (simp add: size-empty)

lemma size-not-empty-2 [simp]:  $\llbracket$ is-empty (idle :: 'a idle); 0 < size idle  $\rrbracket \implies$  False
  by (simp add: size-not-empty)

lemma list-empty: list idle = []  $\longleftrightarrow$  is-empty idle
  by(induction idle)(simp add: list-empty)

lemma list-not-empty: list idle  $\neq$  []  $\longleftrightarrow$   $\neg$  is-empty idle
  by(induction idle)(simp add: list-not-empty)

lemma list-empty-2 [simp]:  $\llbracket$ list idle = [];  $\neg$ is-empty (idle :: 'a idle)  $\rrbracket \implies$  False
  using list-empty by blast

lemma list-not-empty-2 [simp]:  $\llbracket$ list idle  $\neq$  []; is-empty (idle :: 'a idle)  $\rrbracket \implies$  False
  using list-not-empty by blast

lemma list-empty-size: list idle = []  $\longleftrightarrow$  0 = size idle
  by (simp add: list-empty size-empty)

lemma list-not-empty-size: list idle  $\neq$  []  $\longleftrightarrow$  0 < size idle
  by (simp add: list-empty-size)

lemma list-empty-size-2 [simp]:  $\llbracket$ list idle  $\neq$  []; 0 = size idle  $\rrbracket \implies$  False
  by (simp add: list-empty size-empty)

lemma list-not-empty-size-2 [simp]:  $\llbracket$ list idle = []; 0 < size idle  $\rrbracket \implies$  False
  by (simp add: list-empty-size)

end

```

14 Current Proofs

```

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

```

```

lemma push-list [simp]: list (push x current) = x # list current
  by(induction x current rule: push.induct) auto

```

```

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

```

current

by(*induction current rule: pop.induct*)(*auto simp: size-not-empty list-not-empty*)

lemma *drop-first-list* [*simp*]: $\llbracket \text{invar } \text{current}; 0 < \text{size } \text{current} \rrbracket$

$\implies \text{list } (\text{drop-first } \text{current}) = \text{tl } (\text{list } \text{current})$

by(*induction current rule: pop.induct*)(*auto simp: drop-Suc*)

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

by(*induction x current rule: push.induct*) *auto*

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

$\implies \text{invar } \text{current}'$

by(*induction current arbitrary: x rule: pop.induct*) *auto*

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

using *invar-pop*

by (*metis eq-snd-iff*)

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

False

by(*induction current*)(*auto simp: size-not-empty list-empty*)

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

by(*induction x current rule: push.induct*) *auto*

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

by(*induction x current rule: push.induct*) *auto*

lemma *size-new-pop* [*simp*]: $\llbracket 0 < \text{size-new } \text{current}; \text{invar } \text{current} \rrbracket$

$\implies \text{Suc } (\text{size-new } (\text{drop-first } \text{current})) = \text{size-new } \text{current}$

by(*induction current rule: pop.induct*) *auto*

lemma *size-pop* [*simp*]: $\llbracket 0 < \text{size } \text{current}; \text{invar } \text{current} \rrbracket$

$\implies \text{Suc } (\text{size } (\text{drop-first } \text{current})) = \text{size } \text{current}$

by(*induction current rule: pop.induct*) *auto*

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

$\implies \text{Suc } (\text{size } \text{current}') = \text{size } \text{current}$

by(*induction current rule: pop.induct*) *auto*

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

$\implies \text{size } \text{current}' = \text{size } \text{current} - 1$

by(*induction current rule: pop.induct*) *auto*

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

$\implies \text{size } (\text{drop-first current}) = \text{size current} - 1$
by(*induction current rule: pop.induct*) *auto*

end

15 Common Proofs

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

lemma *take-rev-drop*: $\text{take-rev } n \text{ } xs @ acc = \text{drop } (\text{length } xs - n) (\text{rev } xs) @ acc$
unfolding *take-rev-def* **using** *rev-take* **by** *blast*

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

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

lemma *take-rev-tl-hd*:
 $0 < n \implies xs \neq [] \implies \text{take-rev } n xs @ ys = \text{take-rev } (n - (\text{Suc } 0)) (\text{tl } xs) @ (\text{hd } xs \# ys)$
by (*simp add: take-rev-step del: take-rev-def*)

lemma *take-rev-nth*:
 $n < \text{length } xs \implies x = xs ! n \implies x \# \text{take-rev } n xs @ ys = \text{take-rev } (\text{Suc } n) xs @ ys$
by (*simp add: take-Suc-conv-app-nth*)

lemma *step-list* [*simp*]: $\text{invar common} \implies \text{list } (\text{step common}) = \text{list common}$
proof(*induction common rule: step-common-state.induct*)

case (*1 idle*)
then show *?case* **by** *auto*

next

case (*2 current aux new moved*)

then show *?case*

proof(*cases current*)

case (*Current extra added old remained*)

with *2* **have** *aux-not-empty*: $aux \neq []$
by *auto*

from *2 Current* **show** *?thesis*

proof(*cases remained \leq Suc moved*)

case *True*

```

with 2 Current have remained - length new = 1
  by auto

with True Current 2 aux-not-empty show ?thesis
  by auto
next
case False
with Current show ?thesis
  by(auto simp: aux-not-empty take-rev-step Suc-diff-Suc simp del: take-rev-def)
qed
qed
qed

lemma step-list-current [simp]: invar common  $\implies$  list-current (step common) =
list-current common
  by(cases common)(auto split: current.splits)

lemma push-list [simp]: list (push x common) = x # list common
proof(induction x common rule: push.induct)
  case (1 x stack stackSize)
  then show ?case
    by auto
next
  case (2 x current aux new moved)
  then show ?case
    by(induction x current rule: Current.push.induct) auto
qed

lemma invar-step: invar (common :: 'a common-state)  $\implies$  invar (step common)
proof(induction common rule: invar-common-state.induct)
  case (1 idle)
  then show ?case
    by auto
next
  case (2 current aux new moved)
  then show ?case
  proof(cases current)
    case (Current extra added old remained)
    then show ?thesis
    proof(cases aux = [])
      case True
      with 2 Current show ?thesis by auto
    next
      case False
      note AUX-NOT-EMPTY = False

    then show ?thesis
    proof(cases remained  $\leq$  Suc (length new))
      case True

```

```

    with 2 Current False
      have take (Suc (length new)) (Stack-Aux.list old) = take (size old) (hd
aux # new)
      by(auto simp: le-Suc-eq take-Cons')

    with 2 Current True show ?thesis
      by auto
  next
  case False
  with 2 Current AUX-NOT-EMPTY show ?thesis
    by(auto simp: take-rev-step Suc-diff-Suc simp del: take-rev-def)
  qed
  qed
  qed
  qed

```

```

lemma invar-push: invar common  $\implies$  invar (push x common)
proof(induction x common rule: push.induct)
  case (1 x current stack stackSize)
  then show ?case
  proof(induction x current rule: Current.push.induct)
    case (1 x extra added old remained)
    then show ?case
  proof(induction x stack rule: Stack.push.induct)
    case (1 x left right)
    then show ?case by auto
  qed
  qed
  next
  case (2 x current aux new moved)
  then show ?case
  proof(induction x current rule: Current.push.induct)
    case (1 x extra added old remained)
    then show ?case by auto
  qed
  qed

```

```

lemma invar-pop: [
  0 < size common;
  invar common;
  pop common = (x, common')
]  $\implies$  invar common'
proof(induction common arbitrary: x rule: pop.induct)
  case (1 current idle)
  then obtain idle' where idle: Idle.pop idle = (x, idle')
  by(auto split: prod.splits)

```

```

obtain current' where current: drop-first current = current'
by auto

```

```

from 1 current idle show ?case
  using Idle-Proof.size-pop[of idle x idle', symmetric]
    size-new-pop[of current]
    size-pop-sub[of current - current']
  by(auto simp: Idle-Proof.invar-pop invar-pop eq-snd-iff take-tl size-not-empty)
next
case (2 current aux new moved)
then show ?case
proof(induction current rule: Current.pop.induct)
  case (1 added old remained)
  then show ?case
  proof(cases remained - Suc 0 ≤ length new)
    case True

    with 1 have [simp]:
      0 < size old
      Stack-Aux.list old ≠ []
      aux ≠ []
      length new = remained - Suc 0
    by(auto simp: Stack-Proof.size-not-empty Stack-Proof.list-not-empty)

    then have [simp]: Suc 0 ≤ size old
      by linarith

    from 1 have 0 < remained
      by auto

    then have take remained (Stack-Aux.list old)
      = hd (Stack-Aux.list old) # take (remained - Suc 0) (tl (Stack-Aux.list
old)))
      by (metis Suc-pred ‹Stack-Aux.list old ≠ []› list.collapse take-Suc-Cons)

    with 1 True show ?thesis
      using Stack-Proof.pop-list[of old]
      by(auto simp: Stack-Proof.size-not-empty)
  next
  case False
  with 1 have remained - Suc 0 ≤ length aux + length new by auto

  with 1 False show ?thesis
    using Stack-Proof.pop-list[of old]
    apply(auto simp: Suc-diff-Suc take-tl Stack-Proof.size-not-empty tl-append-if)
    by (simp add: Suc-diff-le rev-take tl-drop-2 tl-take)
  qed
next
case (2 x xs added old remained)
then show ?case by auto
qed

```

qed

lemma *push-list-current* [simp]: *list-current (push x left) = x # list-current left*
by(*induction x left rule: push.induct*) *auto*

lemma *pop-list* [simp]: *invar common $\implies 0 < \text{size common} \implies \text{pop common} =$*
(x, common') \implies

x # list common' = list common

proof(*induction common arbitrary: x rule: pop.induct*)

case 1

then show ?*case*

by(*auto simp: size-not-empty split: prod.splits*)

next

case (2 *current aux new moved*)

then show ?*case*

proof(*induction current rule: Current.pop.induct*)

case (1 *added old remained*)

then show ?*case*

proof(*cases remained - Suc 0 \leq length new*)

case *True*

from 1 *True* **have** [simp]:

aux \neq [] 0 < remained

Stack-Aux.list old \neq [] remained - length new = 1

by(*auto simp: Stack-Proof.size-not-empty Stack-Proof.list-not-empty*)

then have *take remained (Stack-Aux.list old) = hd aux # take (size old -*
Suc 0) new

\implies *Stack.first old = hd aux*

by (*metis first-hd hd-take list.sel(1)*)

with 1 *True take-hd[of aux]* **show** ?*thesis*

by(*auto simp: Suc-leI*)

next

case *False*

then show ?*thesis*

proof(*cases remained - length new = length aux*)

case *True*

then have *length-minus-1: remained - Suc (length new) = length aux - 1*

by *simp*

from 1 **have** *not-empty: 0 < remained 0 < size old aux \neq [] \neg is-empty*
old

by(*auto simp: Stack-Proof.size-not-empty*)

from 1 *True not-empty* **have** *take 1 (Stack-Aux.list old) = take 1 (rev aux)*

using *take-1[of*

remained

```

      size old
      Stack-Aux.list old
      (rev aux) @ take (size old + length new - remained) new
    ]
  by(simp)

then have [last aux] = [Stack.first old]
  using take-last first-take not-empty
  by fastforce

then have last aux = Stack.first old
  by auto

with 1 True False show ?thesis
  using not-empty last-drop-rev[of aux]
  by(auto simp: take-rev-drop length-minus-1 simp del: take-rev-def)
next
case False

with 1 have a: take (remained - length new) aux ≠ []
  by auto

from 1 False have b: ¬ is-empty old
  by(auto simp: Stack-Proof.size-not-empty)

from 1 have c: remained - Suc (length new) < length aux
  by auto

from 1 have not-empty:
  0 < remained
  0 < size old
  0 < remained - length new
  0 < length aux
  by auto

with False have
  take remained (Stack-Aux.list old) =
  take (size old) (take-rev (remained - length new) aux @ new)
  ⇒ take (Suc 0) (Stack-Aux.list old) =
  take (Suc 0) (rev (take (remained - length new) aux))
  using take-1[of
    remained
    size old
    Stack-Aux.list old
    (take-rev (remained - length new) aux @ new)
  ]
  by(auto simp: not-empty Suc-le-eq)

with 1 False have

```

```

      take 1 (Stack-Aux.list old) = take 1 (rev (take (remained - length new)
aux))
    by auto

    then have d: [Stack.first old] = [last (take (remained - length new) aux)]
      using take-last first-take a b
      by metis

    have last (take (remained - length new) aux) # rev (take (remained - Suc
(length new)) aux)
      = rev (take (remained - length new) aux)
      using Suc-diff-Suc c not-empty
      by (metis a drop-drop last-drop-rev plus-1-eq-Suc rev-take zero-less-diff)

    with 1(1) 1(3) False not-empty d show ?thesis
      by(cases remained - length new = 1) (auto)
    qed
  qed
next
  case 2
  then show ?case by auto
  qed
qed

```

```

lemma pop-list-current: invar common  $\implies$  0 < size common  $\implies$  pop common =
(x, common')
 $\implies$  x # list-current common' = list-current common
proof(induction common arbitrary: x rule: pop.induct)
  case (1 current idle)
  then show ?case
  proof(induction idle rule: Idle.pop.induct)
    case (1 stack stackSize)
    then show ?case
  proof(induction current rule: Current.pop.induct)
    case (1 added old remained)
    then have Stack.first old = Stack.first stack
      using take-first[of old stack]
      by auto

    with 1 show ?case
      by(auto simp: Stack-Proof.size-not-empty Stack-Proof.list-not-empty)
  next
    case (2 x xs added old remained)
    then have 0 < size stack
      by auto

    with Stack-Proof.size-not-empty Stack-Proof.list-not-empty
    have not-empty:  $\neg$  is-empty stack Stack-Aux.list stack  $\neq$  []
      by auto
  
```

```

with 2 have hd (Stack-Aux.list stack) = x
  using take-hd'[of Stack-Aux.list stack x xs @ Stack-Aux.list old]
  by auto

with 2 show ?case
  using first-list[of stack] not-empty
  by auto
qed
qed
next
  case (2 current)
  then show ?case
  proof(induction current rule: Current.pop.induct)
    case (1 added old remained)
    then have  $\neg$  is-empty old
    by(auto simp: Stack-Proof.size-not-empty)

    with 1 show ?case
    using first-pop
    by(auto simp: Stack-Proof.list-not-empty)
  next
  case 2
  then show ?case by auto
  qed
qed

lemma list-current-size [simp]:
   $\llbracket 0 < \text{size } \text{common}; \text{list-current } \text{common} = []; \text{invar } \text{common} \rrbracket \implies \text{False}$ 
proof(induction common rule: invar-common-state.induct)
  case 1
  then show ?case
  using list-size by auto
next
  case (2 current)
  then have invar current
    Current-Aux.list current = []
     $0 < \text{size } \text{current}$ 
  by(auto split: current.splits)

  then show ?case using list-size by auto
qed

lemma list-size [simp]:  $\llbracket 0 < \text{size } \text{common}; \text{list } \text{common} = []; \text{invar } \text{common} \rrbracket \implies$ 
  False
proof(induction common rule: invar-common-state.induct)
  case 1
  then show ?case
  using list-size Idle-Proof.size-empty

```

```

    by auto
next
case (2 current aux new moved)
then have invar current
      Current-Aux.list current = []
      0 < size current
    by(auto split: current.splits)

then show ?case using list-size by auto
qed

lemma step-size [simp]: invar (common :: 'a common-state)  $\implies$  size (step common) = size common
proof(induction common rule: step-common-state.induct)
  case 1
  then show ?case by auto
next
  case 2
  then show ?case
    by(auto simp: min-def split: current.splits)
qed

lemma step-size-new [simp]: invar (common :: 'a common-state)
 $\implies$  size-new (step common) = size-new common
proof(induction common rule: step-common-state.induct)
  case (1 current idle)
  then show ?case by auto
next
  case (2 current aux new moved)
  then show ?case by(auto split: current.splits)
qed

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
by(induction common)(auto split: current.splits)

lemma remaining-steps-step-sub [simp]:  $\llbracket$ invar (common :: 'a common-state) $\rrbracket$ 
 $\implies$  remaining-steps (step common) = remaining-steps common - 1
by(induction common)(auto split: current.splits)

lemma remaining-steps-step-0 [simp]:  $\llbracket$ invar (common :: 'a common-state); remaining-steps common = 0 $\rrbracket$ 
 $\implies$  remaining-steps (step common) = 0
by(induction common)(auto split: current.splits)

lemma remaining-steps-push [simp]: invar common
 $\implies$  remaining-steps (push x common) = remaining-steps common
by(induction x common rule: Common.push.induct)(auto split: current.splits)

```

```

lemma remaining-steps-pop:  $\llbracket \text{invar common}; \text{pop common} = (x, \text{common}') \rrbracket$ 
 $\implies \text{remaining-steps common}' \leq \text{remaining-steps common}$ 
proof(induction common rule: pop.induct)
  case (1 current idle)
  then show ?case
  proof(induction idle rule: Idle.pop.induct)
    case 1
    then show ?case
    by(induction current rule: Current.pop.induct) auto
  qed
next
  case (2 current aux new moved)
  then show ?case
  by(induction current rule: Current.pop.induct) auto
qed

lemma size-push [simp]: invar common  $\implies \text{size} (\text{push } x \text{ common}) = \text{Suc} (\text{size common})$ 
by(induction x common rule: push.induct) (auto split: current.splits)

lemma size-new-push [simp]: invar common  $\implies \text{size-new} (\text{push } x \text{ common}) = \text{Suc} (\text{size-new common})$ 
by(induction x common rule: Common.push.induct) (auto split: current.splits)

lemma size-pop [simp]:  $\llbracket \text{invar common}; 0 < \text{size common}; \text{pop common} = (x, \text{common}') \rrbracket$ 
 $\implies \text{Suc} (\text{size common}') = \text{size common}$ 
proof(induction common rule: Common.pop.induct)
  case (1 current idle)
  then show ?case
  using size-drop-first-sub[of current] Idle-Proof.size-pop-sub[of idle]
  by(auto simp: size-not-empty split: prod.splits)
next
  case (2 current aux new moved)
  then show ?case
  by(induction current rule: Current.pop.induct) auto
qed

lemma size-new-pop [simp]:  $\llbracket \text{invar common}; 0 < \text{size-new common}; \text{pop common} = (x, \text{common}') \rrbracket$ 
 $\implies \text{Suc} (\text{size-new common}') = \text{size-new common}$ 
proof(induction common rule: Common.pop.induct)
  case (1 current idle)
  then show ?case
  using size-new-pop[of current]
  by(auto split: prod.splits)
next
  case (2 current aux new moved)

```

```

then show ?case
proof(induction current rule: Current.pop.induct)
  case (1 added old remained)
    then show ?case by auto
  next
    case (2 x xs added old remained)
      then show ?case by auto
  qed
qed

lemma size-size-new:  $\llbracket \text{invar } (common :: 'a \text{ common-state}); 0 < \text{size } common \rrbracket \implies$ 
   $0 < \text{size-new } common$ 
  by(cases common) auto

end

```

16 Big Proofs

```

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

lemma step-list [simp]:  $\text{invar } big \implies \text{list } (step \text{ } big) = \text{list } big$ 
proof(induction big rule: step-big-state.induct)
  case 1
    then show ?case
      by auto
  next
    case 2
      then show ?case
        by(auto split: current.splits)
  next
    case 3
      then show ?case
        by(auto simp: rev-take take-drop drop-Suc tl-take rev-drop split: current.splits)
  qed

lemma step-list-current [simp]:  $\text{invar } big \implies \text{list-current } (step \text{ } big) = \text{list-current } big$ 
  by(induction big rule: step-big-state.induct)(auto split: current.splits)

lemma push-list [simp]:  $\text{list } (push \text{ } x \text{ } big) = x \# \text{list } big$ 
proof(induction x big rule: push.induct)
  case (1 x state)
    then show ?case
      by auto
  next
    case (2 x current big aux count)
      then show ?case

```

by(*induction x current rule: Current.push.induct*) *auto*
qed

lemma *list-Big1*: \llbracket

$0 < \text{size } (Big1 \text{ current } big \text{ aux } count);$

$\text{invar } (Big1 \text{ current } big \text{ aux } count)$

$\rrbracket \implies \text{first current } \# \text{ list } (Big1 \text{ (drop-first current) } big \text{ aux } count) =$
 $\text{list } (Big1 \text{ current } big \text{ aux } count)$

proof(*induction current rule: Current.pop.induct*)

case (*1 added old remained*)

then have [*simp*]: $\text{remained} - \text{Suc } 0 < \text{length } (\text{take-rev } count \text{ (Stack-Aux.list } big) @ \text{aux})$

by(*auto simp: le-diff-conv*)

then have

$\llbracket 0 < \text{size } old; 0 < \text{remained}; \text{added} = 0; \text{remained} - \text{count} \leq \text{length } aux; \text{count} \leq \text{size } big;$

$\text{Stack-Aux.list } old =$

$\text{rev } (\text{take } (\text{size } old - \text{size } big) \text{ aux}) @ \text{rev } (\text{take } (\text{size } old) (\text{rev } (\text{Stack-Aux.list } big)));$

$\text{take } \text{remained} (\text{rev } (\text{take } (\text{size } old - \text{size } big) \text{ aux})) @$

$\text{take } (\text{remained} - \text{min } (\text{length } aux) (\text{size } old - \text{size } big))$

$(\text{rev } (\text{take } (\text{size } old) (\text{rev } (\text{Stack-Aux.list } big)))) =$

$\text{rev } (\text{take } (\text{remained} - \text{count}) \text{ aux}) @ \text{rev } (\text{take } \text{remained} (\text{rev } (\text{take } count \text{ (Stack-Aux.list } big))))$

$\implies \text{hd } (\text{rev } (\text{take } (\text{size } old - \text{size } big) \text{ aux}) @ \text{rev } (\text{take } (\text{size } old) (\text{rev } (\text{Stack-Aux.list } big)))) =$

$(\text{rev } (\text{take } count \text{ (Stack-Aux.list } big)) @ \text{aux}) ! (\text{remained} - \text{Suc } 0)$

by (*smt (verit) Suc-pred hd-drop-conv-nth hd-rev hd-take last-snoc length-rev length-take min.absorb2 rev-append take-rev-def size-list-length take-append take-hd-drop*)

with 1 have [*simp*]: $\text{Stack.first } old = (\text{take-rev } count \text{ (Stack-Aux.list } big) @ \text{aux}) ! (\text{remained} - \text{Suc } 0)$

by(*auto simp: take-hd-drop first-hd*)

from 1 show *?case*

using *take-rev-nth*[*of*

$\text{remained} - \text{Suc } 0 \text{ take-rev } count \text{ (Stack-Aux.list } big) @ \text{aux } \text{Stack.first } old$

\llbracket

\rrbracket

by *auto*

next

case *2*

then show *?case* **by** *auto*

qed

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

proof(*induction big rule: list.induct*)

```

case 1
then show ?case
  using list-size by auto
next
case 2
then show ?case
  by (metis list.distinct(1) list-Big1)
qed

```

```

lemma pop-list [simp]:  $\llbracket 0 < \text{size } big; \text{invar } big; \text{Big.pop } big = (x, big^\wedge) \rrbracket$ 
 $\implies x \# \text{list } big' = \text{list } big$ 
proof(induction big arbitrary: x rule: list.induct)
case 1
then show ?case
  by(auto split: prod.splits)
next
case 2
then show ?case
  by (metis Big.pop.simps(2) list-Big1 prod.inject)
qed

```

```

lemma pop-list-tl:  $\llbracket 0 < \text{size } big; \text{invar } big; \text{pop } big = (x, big^\wedge) \rrbracket \implies \text{list } big' = \text{tl}$ 
  (list big)
  using pop-list cons-tl[of x list big' list big]
  by force

```

```

lemma invar-step: invar (big :: 'a big-state)  $\implies$  invar (step big)
proof(induction big rule: step-big-state.induct)
case 1
then show ?case
  by(auto simp: invar-step)
next
case (2 current big aux)

```

```

then obtain extra old remained where current:
  current = Current extra (length extra) old remained
  by(auto split: current.splits)

```

```

with 2 have  $\llbracket$ 
  current = Current extra (length extra) old remained;
  remained  $\leq$  length aux;
  Stack-Aux.list old =
  rev (take (size old - size big) aux) @ rev (take (size old) (rev (Stack-Aux.list
  big)));
  take remained (rev (take (size old - size big) aux)) @
  take (remained - min (length aux) (size old - size big))
  (rev (take (size old) (rev (Stack-Aux.list big)))) =

```

```

    rev (take remained aux)]
  => remained ≤ size old
  by(metis length-rev length-take min.absorb-iff2 size-list-length take-append)

with 2 current have remained - size old = 0
  by auto

with current 2 show ?case
  by(auto simp: take-rev-drop drop-rev)
next
case (3 current big aux count)
then have 0 < size big
  by(auto split: current.splits)

then have big-not-empty: Stack-Aux.list big ≠ []
  by(auto simp: Stack-Proof.size-not-empty Stack-Proof.list-not-empty)

with 3 have a:
  rev (Stack-Aux.list big) @ aux =
  rev (Stack-Aux.list (Stack.pop big)) @ Stack.first big # aux
  by(auto simp: rev-tl-hd first-hd split: current.splits)

from 3 have 0 < size big
  by(auto split: current.splits)

from 3 big-not-empty have
  take-rev (Suc count) (Stack-Aux.list big) @ aux =
  take-rev count (Stack-Aux.list (Stack.pop big)) @ (Stack.first big # aux)
  using take-rev-tl-hd[of Suc count Stack-Aux.list big aux]
  by(auto simp: Stack-Proof.list-not-empty split: current.splits)

with 3 a show ?case
  by(auto split: current.splits)
qed

lemma invar-push: invar big => invar (push x big)
  by(induction x big rule: push.induct)(auto simp: invar-push split: current.splits)

lemma invar-pop: [
  0 < size big;
  invar big;
  pop big = (x, big')
] => invar big'
proof(induction big arbitrary: x rule: pop.induct)
  case (1 state)
  then show ?case
    by(auto simp: invar-pop split: prod.splits)
next

```

```

case (2 current big aux count)
then show ?case
proof(induction current rule: Current.pop.induct)
  case (1 added old remained)
  have linarith:  $\bigwedge x y z. x - y \leq z \implies x - (\text{Suc } y) \leq z$ 
  by linarith

  have a:  $\llbracket \text{remained} \leq \text{count} + \text{length } \text{aux}; 0 < \text{remained}; \text{added} = 0; x = \text{Stack.first } \text{old};$ 
     $\text{big}' = \text{Big1 } (\text{Current } \sqcup 0 (\text{Stack.pop } \text{old}) (\text{remained} - \text{Suc } 0)) \text{ big } \text{aux}$ 
  count;
     $\text{count} \leq \text{size } \text{big}; \text{Stack-Aux.list } \text{old} = \text{rev } \text{aux} @ \text{Stack-Aux.list } \text{big};$ 
     $\text{take } \text{remained} (\text{rev } \text{aux}) @ \text{take } (\text{remained} - \text{length } \text{aux}) (\text{Stack-Aux.list } \text{big}) =$ 
     $\text{drop } (\text{count} + \text{length } \text{aux} - \text{remained}) (\text{rev } \text{aux}) @$ 
     $\text{drop } (\text{count} - \text{remained}) (\text{take } \text{count} (\text{Stack-Aux.list } \text{big}));$ 
     $\neg \text{size } \text{old} \leq \text{length } \text{aux} + \text{size } \text{big} \rrbracket$ 
     $\implies \text{tl } (\text{rev } \text{aux} @ \text{Stack-Aux.list } \text{big}) = \text{rev } \text{aux} @ \text{Stack-Aux.list } \text{big}$ 
  by (metis le-refl length-append length-rev size-list-length)

  have b:  $\llbracket \text{remained} \leq \text{length } (\text{take-rev } \text{count} (\text{Stack-Aux.list } \text{big}) @ \text{aux}); 0 <$ 
     $\text{size } \text{old};$ 
     $0 < \text{remained}; \text{added} = 0;$ 
     $x = \text{Stack.first } \text{old};$ 
     $\text{big}' = \text{Big1 } (\text{Current } \sqcup 0 (\text{Stack.pop } \text{old}) (\text{remained} - \text{Suc } 0)) \text{ big } \text{aux}$ 
  count;
     $\text{remained} - \text{count} \leq \text{length } \text{aux}; \text{count} \leq \text{size } \text{big};$ 
     $\text{Stack-Aux.list } \text{old} =$ 
     $\text{drop } (\text{length } \text{aux} - (\text{size } \text{old} - \text{size } \text{big})) (\text{rev } \text{aux}) @$ 
     $\text{drop } (\text{size } \text{big} - \text{size } \text{old}) (\text{Stack-Aux.list } \text{big});$ 
     $\text{take } \text{remained} (\text{drop } (\text{length } \text{aux} - (\text{size } \text{old} - \text{size } \text{big})) (\text{rev } \text{aux})) @$ 
     $\text{take } (\text{remained} + (\text{length } \text{aux} - (\text{size } \text{old} - \text{size } \text{big})) - \text{length } \text{aux})$ 
     $(\text{drop } (\text{size } \text{big} - \text{size } \text{old}) (\text{Stack-Aux.list } \text{big})) =$ 
     $\text{drop } (\text{length } (\text{take-rev } \text{count} (\text{Stack-Aux.list } \text{big}) @ \text{aux}) - \text{remained})$ 
     $(\text{rev } (\text{take-rev } \text{count} (\text{Stack-Aux.list } \text{big}) @ \text{aux})) \rrbracket$ 
     $\implies \text{tl } (\text{drop } (\text{length } \text{aux} - (\text{size } \text{old} - \text{size } \text{big})) (\text{rev } \text{aux}) @$ 
     $\text{drop } (\text{size } \text{big} - \text{size } \text{old}) (\text{Stack-Aux.list } \text{big})) =$ 
     $\text{drop } (\text{length } \text{aux} - (\text{size } \text{old} - \text{Suc } (\text{size } \text{big}))) (\text{rev } \text{aux}) @$ 
     $\text{drop } (\text{Suc } (\text{size } \text{big}) - \text{size } \text{old}) (\text{Stack-Aux.list } \text{big})$ 
  apply(cases size old - size big  $\leq$  length aux; cases size old  $\leq$  size big)
  by(auto simp: tl-drop-2 Suc-diff-le le-diff-conv le-refl a)

from 1 have remained  $\leq$  length (take-rev count (Stack-Aux.list big) @ aux)
  by(auto)

with 1 show ?case
  apply(auto simp: rev-take take-tl drop-Suc Suc-diff-le tl-drop linarith simp del:
take-rev-def)
  using b

```

apply (*metis* \langle remained \leq length (take-rev count (Stack-Aux.list big) @ aux) \rangle
le-diff-conv rev-append rev-take take-append)
by (*smt* (*verit*, *del-insts*) *Nat.diff-cancel tl-append-if Suc-diff-le append-self-conv2*
diff-add-inverse diff-diff-cancel diff-is-0-eq diff-le-mono drop-eq-Nil2 length-rev nle-le
not-less-eq-eq plus-1-eq-Suc tl-drop-2)
next
case (*2 x xs added old remained*)
then show *?case* **by** *auto*
qed
qed

lemma *push-list-current* [*simp*]: *list-current (push x big) = x # list-current big*
by(*induction x big rule: push.induct*) *auto*

lemma *pop-list-current* [*simp*]: \llbracket *invar big; 0 < size big; Big.pop big = (x, big')* \rrbracket
 \implies *x # list-current big' = list-current big*
proof(*induction big arbitrary: x rule: pop.induct*)
case (*1 state*)
then show *?case*
by(*auto simp: pop-list-current split: prod.splits*)
next
case (*2 current big aux count*)
then show *?case*
proof(*induction current rule: Current.pop.induct*)
case (*1 added old remained*)

then have
 $rev (take (size old - size big) aux) @ rev (take (size old) (rev (Stack-Aux.list$
big))) \neq []
using
order-less-le-trans[of 0 size old size big]
order-less-le-trans[of 0 count size big]
by(*auto simp: Stack-Proof.size-not-empty Stack-Proof.list-not-empty*)

with 1 show *?case*
by(*auto simp: first-hd*)
next
case (*2 x xs added old remained*)
then show *?case*
by *auto*
qed
qed

lemma *list-current-size*: \llbracket *0 < size big; list-current big = []; invar big* $\rrbracket \implies$ *False*
proof(*induction big rule: list-current.induct*)
case 1
then show *?case*
using *list-current-size*
by *simp*

```

next
  case (2 current uu uv uw)
  then show ?case
    apply(cases current)
    by(auto simp: Stack-Proof.size-not-empty Stack-Proof.list-empty)
qed

lemma step-size: invar (big :: 'a big-state)  $\implies$  size big = size (step big)
  by(induction big rule: step-big-state.induct)(auto split: current.splits)

lemma remaining-steps-step [simp]:  $\llbracket$ invar (big :: 'a big-state); remaining-steps big > 0 $\rrbracket$ 
   $\implies$  Suc (remaining-steps (step big)) = remaining-steps big
  by(induction big rule: step-big-state.induct)(auto split: current.splits)

lemma remaining-steps-step-0 [simp]:  $\llbracket$ invar (big :: 'a big-state); remaining-steps big = 0 $\rrbracket$ 
   $\implies$  remaining-steps (step big) = 0
  by(induction big)(auto split: current.splits)

lemma remaining-steps-push: invar big  $\implies$  remaining-steps (push x big) = remaining-steps big
  by(induction x big rule: push.induct)(auto split: current.splits)

lemma remaining-steps-pop:  $\llbracket$ invar big; pop big = (x, big $^\wedge$ ) $\rrbracket$ 
   $\implies$  remaining-steps big'  $\leq$  remaining-steps big
proof(induction big rule: pop.induct)
  case (1 state)
  then show ?case
    by(auto simp: remaining-steps-pop split: prod.splits)
next
  case (2 current big aux count)
  then show ?case
    by(induction current rule: Current.pop.induct) auto
qed

lemma size-push [simp]: invar big  $\implies$  size (push x big) = Suc (size big)
  by(induction x big rule: push.induct)(auto split: current.splits)

lemma size-new-push [simp]: invar big  $\implies$  size-new (push x big) = Suc (size-new big)
  by(induction x big rule: Big.push.induct)(auto split: current.splits)

lemma size-pop [simp]:  $\llbracket$ invar big; 0 < size big; pop big = (x, big $^\wedge$ ) $\rrbracket$ 
   $\implies$  Suc (size big $^\wedge$ ) = size big
proof(induction big rule: pop.induct)
  case 1
  then show ?case
    by(auto split: prod.splits)

```

```

next
  case (2 current big aux count)
  then show ?case
    by(induction current rule: Current.pop.induct) auto
qed

lemma size-new-pop [simp]:  $\llbracket \text{invar } big; 0 < \text{size-new } big; \text{pop } big = (x, big') \rrbracket$ 
   $\implies \text{Suc } (\text{size-new } big') = \text{size-new } big$ 
proof(induction big rule: pop.induct)
  case 1
  then show ?case
    by(auto split: prod.splits)
next
  case (2 current big aux count)
  then show ?case
    by(induction current rule: Current.pop.induct) auto
qed

lemma size-size-new:  $\llbracket \text{invar } (big :: 'a \text{ big-state}); 0 < \text{size } big \rrbracket \implies 0 < \text{size-new } big$ 
  by(induction big)(auto simp: size-size-new)

end

```

17 Small Proofs

```

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

lemma step-size [simp]:  $\text{invar } (small :: 'a \text{ small-state}) \implies \text{size } (\text{step } small) = \text{size } small$ 
  by(induction small rule: step-small-state.induct)(auto split: current.splits)

lemma step-size-new [simp]:
   $\text{invar } (small :: 'a \text{ small-state}) \implies \text{size-new } (\text{step } small) = \text{size-new } small$ 
  by(induction small rule: step-small-state.induct)(auto split: current.splits)

lemma size-push [simp]:  $\text{invar } small \implies \text{size } (\text{push } x \text{ } small) = \text{Suc } (\text{size } small)$ 
  by(induction x small rule: push.induct) (auto split: current.splits)

lemma size-new-push [simp]:  $\text{invar } small \implies \text{size-new } (\text{push } x \text{ } small) = \text{Suc } (\text{size-new } small)$ 
  by(induction x small rule: push.induct) (auto split: current.splits)

lemma size-pop [simp]:  $\llbracket \text{invar } small; 0 < \text{size } small; \text{pop } small = (x, small') \rrbracket$ 
   $\implies \text{Suc } (\text{size } small') = \text{size } small$ 
proof(induction small rule: pop.induct)
  case (1 state)

```

```

then show ?case
  by(auto split: prod.splits)
next
  case (2 current small auxS)
  then show ?case
    using Current-Proof.size-pop[of current]
    by(induction current rule: Current.pop.induct) auto
next
  case (3 current auxS big newS count)
  then show ?case
    using Current-Proof.size-pop[of current]
    by(induction current rule: Current.pop.induct) auto
qed

```

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

$\implies \textit{Suc } (\textit{size-new } \textit{small}') = \textit{size-new } \textit{small}$

proof(*induction small rule: pop.induct*)

case (1 *state*)

then show ?case

by(*auto split: prod.splits*)

next

case (2 *current small auxS*)

then show ?case

by(*induction current rule: Current.pop.induct*) *auto*

next

case (3 *current auxS big newS count*)

then show ?case

by(*induction current rule: Current.pop.induct*) *auto*

qed

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

by(*induction small*)(*auto simp: size-size-new*)

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

by(*induction small rule: step-small-state.induct*)(*auto split: current.splits*)

lemma *step-list-common [simp]:*

$\llbracket \textit{small} = \textit{Small3 } \textit{common}; \textit{invar } \textit{small} \rrbracket \implies \textit{list } (\textit{step } \textit{small}) = \textit{list } \textit{small}$

by *auto*

lemma *step-list-Small2 [simp]:*

assumes

small = (Small2 current aux big new count)

invar small

shows

$\textit{list } (\textit{step } \textit{small}) = \textit{list } \textit{small}$

proof –

have *size-not-empty*: $(0 < \text{size } \text{big}) = (\neg \text{is-empty } \text{big})$
by (*simp add: Stack-Proof.size-not-empty*)

have $\neg \text{is-empty } \text{big}$
 $\implies \text{rev } (\text{Stack-Aux.list } (\text{Stack.pop } \text{big})) @ [\text{Stack.first } \text{big}] = \text{rev } (\text{Stack-Aux.list } \text{big})$
by(*induction big rule: Stack.pop.induct*) *auto*

with *assms show ?thesis*
using *Stack-Proof.size-pop[of big] size-not-empty*
by(*auto simp: Stack-Proof.list-empty split: current.splits*)

qed

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

proof(*induction small rule: step-small-state.induct*)

case (1 *state*)
then show *?case*
by(*auto simp: invar-step*)

next

case (2 *current small aux*)
then show *?case*
proof(*cases is-empty small*)
case *True*
with 2 **show** *?thesis*
by *auto*

next

case *False*

with 2 **have** $\text{rev } (\text{Stack-Aux.list } \text{small}) @ \text{aux} =$
 $\text{rev } (\text{Stack-Aux.list } (\text{Stack.pop } \text{small})) @ \text{Stack.first } \text{small} \# \text{aux}$
by(*auto simp: rev-app-single Stack-Proof.list-not-empty*)

with 2 **show** *?thesis*
by(*auto split: current.splits*)

qed

next

case (3 *current auxS big newS count*)
then show *?case*
proof(*cases is-empty big*)
case *True*

then have *big-size [simp]: size big = 0*
by (*simp add: Stack-Proof.size-empty*)

with *True 3 show ?thesis*

proof(*cases current*)
case (*Current extra added old remained*)

```

with 3 True show ?thesis
proof(cases remained ≤ count)
  case True
  with 3 Current show ?thesis
  using Stack-Proof.size-empty[of big]
  by auto
next
  case False
  with True 3 Current show ?thesis
  by(auto)
qed
next
qed
next
case False
with 3 show ?thesis
using Stack-Proof.size-pop[of big]
by(auto simp: Stack-Proof.size-not-empty split: current.splits)
qed
qed

lemma invar-push: invar small ⇒ invar (push x small)
by(induction x small rule: push.induct)(auto simp: invar-push split: current.splits)

lemma invar-pop: [
  0 < size small;
  invar small;
  pop small = (x, small')
] ⇒ invar small'
proof(induction small arbitrary: x rule: pop.induct)
case (1 state)
  then show ?case
  by(auto simp: invar-pop split: prod.splits)
next
case (2 current small auxS)
  then show ?case
  proof(induction current rule: Current.pop.induct)
  case (1 added old remained)
  then show ?case
  by(cases size small < size old)
  (auto simp: rev-take Suc-diff-le drop-Suc tl-drop)
  next
  case 2
  then show ?case by auto
  qed
next
case (3 current auxS big newS count)
  then show ?case
  by (induction current rule: Current.pop.induct)
  (auto simp: rev-take Suc-diff-le drop-Suc tl-drop)

```

qed

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

lemma *push-list-current* [*simp*]: $list-current\ (push\ x\ small) = x \# list-current\ small$
by(*induction x small rule: push.induct*) *auto*

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

$\implies x \# list-current\ small' = list-current\ small$

proof(*induction small arbitrary: x rule: pop.induct*)

case (1 *state*)

then show *?case*

by(*auto simp: pop-list-current split: prod.splits*)

next

case (2 *current small auxS*)

then have *invar current*

by(*auto split: current.splits*)

with 2 **show** *?case*

by *auto*

next

case (3 *current auxS big newS count*)

then show *?case*

proof(*induction current rule: Current.pop.induct*)

case (1 *added old remained*)

then have $\neg is-empty\ old$

by(*auto simp: Stack-Proof.size-not-empty*)

with 1 **show** *?case*

by(*auto simp: rev-take drop-Suc drop-tl*)

next

case 2

then show *?case*

by *auto*

qed

qed

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

proof(*induction small*)

case (*Small1 current*)

then have *invar current*

by(*auto split: current.splits*)

with *Small1* **show** *?case*

```

    using Current-Proof.list-size
    by auto
next
case Small2
then show ?case
  by(auto split: current.splits)
next
case Small3
then show ?case
  using list-current-size by auto
qed

```

```

lemma list-Small2 [simp]: [
  0 < size (Small2 current auxS big newS count);
  invar (Small2 current auxS big newS count)
]  $\implies$ 
  fst (Current.pop current) # list (Small2 (drop-first current) auxS big newS
count) =
  list (Small2 current auxS big newS count)
  by(induction current rule: Current.pop.induct)
  (auto simp: first-hd rev-take Suc-diff-le)

```

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  $\implies$  list-big-first states = list-current-big-first states
  using app-rev
  by(cases states)(auto split: prod.splits)

```

```

lemma step-lists [simp]: invar states  $\implies$  lists (step states) = lists states
proof(induction states rule: lists.induct)
case (1 dir currentB big auxB count currentS small auxS)
then show ?case
proof(induction
  (States dir (Big1 currentB big auxB count) (Small1 currentS small auxS))
  rule: step-states.induct)
case 1
then show ?case
  by(cases currentB) auto
next

```

```

case (2-1 count')
then have 0 < size big
  by(cases currentB) auto

then have big-not-empty: Stack-Aux.list big ≠ []
  by (simp add: Stack-Proof.size-not-empty Stack-Proof.list-empty)

with 2-1 show ?case
  using
    take-rev-step[of Stack-Aux.list big count' auxB]
    Stack-Proof.list-empty[symmetric, of small]
  apply (cases currentB)
  by(auto simp: first-hd funpow-swap1 take-rev-step simp del: take-rev-def)
qed
next
  case (2-1 dir common small)
  then show ?case
    using step-list-Small2[of small]
    by(auto split: small-state.splits)
next
  case (2-2 dir big current auxS big newS count)
  then show ?case
    using step-list-Small2[of Small2 current auxS big newS count]
    by auto
next
  case (2-3 dir big common)
  then show ?case
    by auto
qed

lemma step-lists-current [simp]:
  invar states ⇒ lists-current (step states) = lists-current states
  by(induction states rule: step-states.induct)(auto split: current.splits)

lemma push-big: lists (States dir big small) = (big', small')
  ⇒ lists (States dir (Big.push x big) small) = (x # big', small')
proof(induction States dir (Big.push x big) small rule: lists.induct)
  case 1
  then show ?case
  proof(induction x big rule: Big.push.induct)
  case 1
  then show ?case
    by auto
  next
  case (2 x current big aux count)
  then show ?case
    by(cases current) auto
qed
next

```

```

case 2-1
then show ?case
  by(cases big) auto
qed auto

```

```

lemma push-small-lists:
  invar (States dir big small)
   $\implies$  lists (States dir big (Small.push x small)) = (big', x # small')  $\longleftrightarrow$ 
    lists (States dir big small) = (big', small')
  apply(induction States dir big (Small.push x small) rule: lists.induct)
  by (auto split: current.splits small-state.splits)

```

```

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)
  using app-rev
  by(auto split: prod.splits)

```

```

lemma list-big-first-pop-big [simp]:  $\llbracket$ 
  invar (States dir big small);
   $0 < \text{size } big$ ;
  Big.pop big = (x, big') $\rrbracket$ 
 $\implies$  x # list-big-first (States dir big' small) = list-big-first (States dir big small)
  by(induction States dir big small rule: lists.induct)(auto split: prod.splits)

```

```

lemma list-current-big-first-pop-big [simp]:  $\llbracket$ 
  invar (States dir big small);
   $0 < \text{size } big$ ;
  Big.pop big = (x, big') $\rrbracket$ 
 $\implies$  x # list-current-big-first (States dir big' small) =
  list-current-big-first (States dir big small)
  by auto

```

```

lemma lists-big-first-pop-big:  $\llbracket$ 
  invar (States dir big small);
   $0 < \text{size } big$ ;
  Big.pop big = (x, big') $\rrbracket$ 
 $\implies$  list-big-first (States dir big' small) = list-current-big-first (States dir big'
small)
  by (metis invar-list-big-first list-big-first-pop-big list-current-big-first-pop-big list.sel(3))

```

```

lemma lists-small-first-pop-big:  $\llbracket$ 
  invar (States dir big small);
   $0 < \text{size } big$ ;
  Big.pop big = (x, big') $\rrbracket$ 
 $\implies$  list-small-first (States dir big' small) = list-current-small-first (States dir big'
small)
  by (meson lists-big-first-pop-big list-small-big)

```

```

lemma list-small-first-pop-small [simp]: [
  invar (States dir big small);
   $0 < \text{size } \text{small}$ ;
  Small.pop small = (x, small')]
 $\implies x \# \text{list-small-first } (\text{States dir big small}') = \text{list-small-first } (\text{States dir big small})$ 
proof(induction States dir big small rule: lists.induct)
  case (1 currentB big auxB count currentS small auxS)
  then show ?case
    by(cases currentS)(auto simp: Cons-eq-appendI)
next
  case (2-1 common)
  then show ?case
proof(induction small rule: Small.pop.induct)
  case (1 common)
  then show ?case
    by(cases Common.pop common)(auto simp: Cons-eq-appendI)
next
  case 2
  then show ?case by auto
next
  case 3
  then show ?case
    by(cases Common.pop common)(auto simp: Cons-eq-appendI)
qed
next
  case (2-2 current)
  then show ?case
    by(induction current rule: Current.pop.induct)
    (auto simp: first-hd rev-take Suc-diff-le)
next
  case (2-3 common)
  then show ?case
    by(cases Common.pop common)(auto simp: Cons-eq-appendI)
qed

```

```

lemma list-current-small-first-pop-small [simp]: [
  invar (States dir big small);
   $0 < \text{size } \text{small}$ ;
  Small.pop small = (x, small')]
 $\implies x \# \text{list-current-small-first } (\text{States dir big small}') =$ 
   $\text{list-current-small-first } (\text{States dir big small})$ 
by auto

```

```

lemma lists-small-first-pop-small: [
  invar (States dir big small);
   $0 < \text{size } \text{small}$ ;
  Small.pop small = (x, small')]

```

$\implies \text{list-small-first (States dir big small')} = \text{list-current-small-first (States dir big small')}$

by (*metis (no-types, opaque-lifting) invar-states.simps list.sel(3)*
list-current-small-first-pop-small list-small-first-pop-small)

lemma *invars-pop-big*: \llbracket
invar (States dir big small);
0 < size big;
Big.pop big = (x, big')
 $\implies \text{invar big}' \wedge \text{invar small}$
by(*auto simp: Big-Proof.invar-pop*)

lemma *invar-pop-big-aux*: \llbracket
invar (States dir big small);
0 < size big;
Big.pop big = (x, big')
 $\implies (\text{case (big', small) of}$
(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
 $)$
by(*auto split: big-state.splits small-state.splits prod.splits*)

lemma *invar-pop-big*: \llbracket
invar (States dir big small);
0 < size big;
Big.pop big = (x, big')
 $\implies \text{invar (States dir big' small)}$
using *invars-pop-big*[of *dir big small x big'*]
lists-small-first-pop-big[of *dir big small x big'*]
invar-pop-big-aux[of *dir big small x big'*]
by *auto*

lemma *invars-pop-small*: \llbracket
invar (States dir big small);
0 < size small;
Small.pop small = (x, small')
 $\implies \text{invar big} \wedge \text{invar small}'$
by(*auto simp: Small-Proof.invar-pop*)

lemma *invar-pop-small-aux*: \llbracket
invar (States dir big small);
0 < size small;
Small.pop small = (x, small')
 $\implies (\text{case (big, small') of}$
(Big1 - big - count, Small1 (Current - - old remained) small -) \implies
size big - count = remained - size old \wedge count \geq size small
 $)$

```

    | (-, Small1 - - -)  $\Rightarrow$  False
    | (Big1 - - - -, -)  $\Rightarrow$  False
    | -  $\Rightarrow$  True
  )
proof(induction small rule: Small.pop.induct)
  case 1
  then show ?case
    by(auto split: big-state.splits small-state.splits prod.splits)
next
  case (2 current)
  then show ?case
  proof(induction current rule: Current.pop.induct)
    case 1
    then show ?case
      by(auto split: big-state.splits)
    next
    case 2
    then show ?case
      by(auto split: big-state.splits)
    qed
  next
  case 3
  then show ?case
    by(auto split: big-state.splits)
  qed

lemma invar-pop-small:  $\llbracket$ 
  invar (States dir big small);
   $0 < size\ small$ ;
   $Small.pop\ small = (x, small')$ 
 $\rrbracket \Rightarrow invar (States\ dir\ big\ small')$ 
using invars-pop-small[of dir big small x small]
  lists-small-first-pop-small[of dir big small x small]
  invar-pop-small-aux[of dir big small x small]
by fastforce

lemma invar-push-big:  $invar (States\ dir\ big\ small) \Rightarrow invar (States\ dir\ (Big.push\ x\ big)\ small)$ 
proof(induction x big arbitrary: small rule: Big.push.induct)
  case 1
  then show ?case
    by(auto simp: Common-Proof.invar-push)
  next
  case (2 x current big aux count)
  then show ?case
    by(cases current)(auto split: prod.splits small-state.splits)
  qed

lemma invar-push-small:  $invar (States\ dir\ big\ small)$ 

```

```

     $\implies$  invar (States dir big (Small.push x small))
proof(induction x small arbitrary: big rule: Small.push.induct)
  case (1 x state)
  then show ?case
    by(auto simp: Common-Proof.invar-push split: big-state.splits)
next
  case (2 x current small auxS)
  then show ?case
    by(induction x current rule: Current.push.induct)(auto split: big-state.splits)
next
  case (3 x current auxS big newS count)
  then show ?case
    by(induction x current rule: Current.push.induct)(auto split: big-state.splits)
qed

lemma step-invars: $\llbracket$ invar states; step states = States dir big small $\rrbracket \implies$  invar big
 $\wedge$  invar small
proof(induction states rule: step-states.induct)
  case (1 dir currentB big' auxB currentS small' auxS)
  with Big-Proof.invar-step have invar (Big1 currentB big' auxB 0)
    by auto
  with 1 have invar-big: invar big
    using Big-Proof.invar-step[of Big1 currentB big' auxB 0]
    by auto

  from 1 have invar-small: invar small
    using Stack-Proof.list-empty-size[of small']
    by(cases currentS) auto

  from invar-small invar-big show ?case
    by simp
next
  case (2-1 dir current big aux count small)
  then show ?case
    using Big-Proof.invar-step[of Big1 current big aux (Suc count)]
      Small-Proof.invar-step[of small]
    by simp
next
  case 2-2
  then show ?case
    by(auto simp: Common-Proof.invar-step Small-Proof.invar-step)
next
  case (2-3 dir big current auxS big' newS count)
  then show ?case
    using Big-Proof.invar-step[of big]
      Small-Proof.invar-step[of Small2 current auxS big' newS count]
    by auto
next
  case 2-4

```

```

then show ?case
  by(auto simp: Common-Proof.invar-step Big-Proof.invar-step)
qed

```

```

lemma step-lists-small-first: invar states  $\implies$ 
  list-small-first (step states) = list-current-small-first (step states)
  using step-lists-current step-lists invar-states.elims(2)
  by fastforce

```

```

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
  )

```

```

proof(induction states rule: step-states.induct)
  case (2-1 dir current big aux count small)
  then show ?case
  proof(cases small)
    case (Small1 current small auxS)
    with 2-1 show ?thesis
    using Stack-Proof.size-empty[symmetric, of small]
    by(auto split: current.splits)
  qed auto
qed (auto split: big-state.splits small-state.splits)

```

```

lemma invar-step: invar (states :: 'a states)  $\implies$  invar (step states)
  using invar-step-aux[of states] step-lists-small-first[of states]
  by(cases step states)(auto simp: step-invars)

```

```

lemma step-consistent [simp]:
   $\llbracket \bigwedge \text{states. invar (states :: 'a states)} \implies P (\text{step states}) = P \text{ states}; \text{invar states} \rrbracket$ 
   $\implies P \text{ states} = P ((\text{step } \sim^n) \text{ states})$ 
  by(induction n arbitrary: states)
  (auto simp: States-Proof.invar-step funpow-swap1)

```

```

lemma step-consistent-2:
   $\llbracket \bigwedge \text{states. } \llbracket \text{invar (states :: 'a states)}; P \text{ states} \rrbracket \implies P (\text{step states}); \text{invar states}; P \text{ states} \rrbracket$ 
   $\implies P ((\text{step } \sim^n) \text{ states})$ 
  by(induction n arbitrary: states)
  (auto simp: States-Proof.invar-step funpow-swap1)

```

```

lemma size-ok'-Suc: size-ok' states (Suc steps)  $\implies$  size-ok' states steps
  by(induction states steps rule: size-ok'.induct) auto

```

```

lemma size-ok'-decline: size-ok' states x  $\implies$  x  $\geq$  y  $\implies$  size-ok' states y

```

```

by(induction states x rule: size-ok'.induct) auto

lemma remaining-steps-0 [simp]:  $\llbracket$ invar (states :: 'a states); remaining-steps states
= 0 $\rrbracket$ 
 $\implies$  remaining-steps (step states) = 0
by(induction states rule: step-states.induct)
(auto split: current.splits small-state.splits)

lemma remaining-steps-0':  $\llbracket$ invar (states :: 'a states); remaining-steps states = 0 $\rrbracket$ 
 $\implies$  remaining-steps ((step  $\sim$  n) states) = 0
by(induction n arbitrary: states)(auto simp: invar-step funpow-swap1)

lemma remaining-steps-decline-Suc:
 $\llbracket$ invar (states :: 'a states); 0 < remaining-steps states $\rrbracket$ 
 $\implies$  Suc (remaining-steps (step states)) = remaining-steps states
proof(induction states rule: step-states.induct)
case 1
then show ?case
by(auto simp: max-def split: big-state.splits small-state.splits current.splits)
next
case (2-1 - - - - small)
then show ?case
by(cases small)(auto split: current.splits)
next
case (2-2 dir big small)
then show ?case
proof(cases small)
case (Small2 current auxS big newS count)
with 2-2 show ?thesis
using Stack-Proof.size-empty-2[of big]
by(cases current) auto
qed auto
next
case (2-3 dir big current auxS big' newS count)
then show ?case
proof(induction big)
case Big1
then show ?case by auto
next
case Big2
then show ?case
using Stack-Proof.size-empty-2[of big']
by(cases current) auto
qed
next
case (2-4 - big)
then show ?case
by(cases big) auto
qed

```

```

lemma remaining-steps-decline-sub [simp]: invar (states :: 'a states)
  ⇒ remaining-steps (step states) = remaining-steps states - 1
using Suc-sub[of remaining-steps (step states) remaining-steps states]
by(cases 0 < remaining-steps states) (auto simp: remaining-steps-decline-Suc)

lemma remaining-steps-decline: invar (states :: 'a states)
  ⇒ remaining-steps (step states) ≤ remaining-steps states
using remaining-steps-decline-sub[of states] by auto

lemma remaining-steps-decline-n-steps [simp]:
  [[invar (states :: 'a states); remaining-steps states ≤ n]]
  ⇒ remaining-steps ((step  $\hat{\sim}$  n) states) = 0
by(induction n arbitrary: states)(auto simp: funpow-swap1 invar-step)

lemma remaining-steps-n-steps-plus [simp]:
  [[n ≤ remaining-steps states; invar (states :: 'a states)]]
  ⇒ remaining-steps ((step  $\hat{\sim}$  n) states) + n = remaining-steps states
by(induction n arbitrary: states)(auto simp: funpow-swap1 invar-step)

lemma remaining-steps-n-steps-sub [simp]: invar (states :: 'a states)
  ⇒ remaining-steps ((step  $\hat{\sim}$  n) states) = remaining-steps states - n
by(induction n arbitrary: states)(auto simp: funpow-swap1 invar-step)

lemma step-size-new-small [simp]:
  [[invar (States dir big small); step (States dir big small) = States dir' big' small]]
  ⇒ size-new small' = size-new small
proof(induction States dir big small rule: step-states.induct)
  case 1
  then show ?case
    by auto
next
  case 2-1
  then show ?case
    by(auto split: small-state.splits)
next
  case 2-2
  then show ?case
    by(auto split: small-state.splits current.splits)
next
  case 2-3
  then show ?case
    by(auto split: current.splits)
next
  case 2-4
  then show ?case
    by auto
qed

```

lemma *step-size-new-small-2* [simp]:
invar states \implies *size-new-small (step states) = size-new-small states*
by(cases states; cases step states) auto

lemma *step-size-new-big* [simp]:
 \llbracket *invar (States dir big small); step (States dir big small) = States dir' big' small'* \rrbracket
 \implies *size-new big' = size-new big*
proof(induction States dir big small rule: step-states.induct)
case 1
then show ?case
by(auto split: current.splits)
next
case 2-1
then show ?case
by auto
next
case 2-2
then show ?case
by auto
next
case 2-3
then show ?case
by(auto split: big-state.splits)
next
case 2-4
then show ?case
by(auto split: big-state.splits)
qed

lemma *step-size-new-big-2* [simp]:
invar states \implies *size-new-big (step states) = size-new-big states*
by(cases states; cases step states) auto

lemma *step-size-small* [simp]:
 \llbracket *invar (States dir big small); step (States dir big small) = States dir' big' small'* \rrbracket
 \implies *size small' = size small*
proof(induction States dir big small rule: step-states.induct)
case 2-3
then show ?case
by(auto split: current.splits)
qed auto

lemma *step-size-small-2* [simp]:
invar states \implies *size-small (step states) = size-small states*
by(cases states; cases step states) auto

lemma *step-size-big* [simp]:
 \llbracket *invar (States dir big small); step (States dir big small) = States dir' big' small'* \rrbracket
 \implies *size big' = size big*

```

proof(induction States dir big small rule: step-states.induct)
  case 1
  then show ?case
    by(auto split: current.splits)
next
  case 2-1
  then show ?case
    by(auto split: small-state.splits current.splits)
next
  case 2-2
  then show ?case
    by(auto split: small-state.splits current.splits)
next
  case 2-3
  then show ?case
    by(auto split: current.splits big-state.splits)
next
  case 2-4
  then show ?case
    by(auto split: big-state.splits)
qed

```

```

lemma step-size-big-2 [simp]:
  invar states  $\implies$  size-big (step states) = size-big states
  by(cases states; cases step states) auto

```

```

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
 $\rrbracket \implies$  size-new big' + remaining-steps (States dir' big' small') + 2  $\leq$  3 * size-new small'
  using step-size-new-small step-size-new-big remaining-steps-decline
  by (smt (verit, ccfv-SIG) add.commute le-trans nat-add-left-cancel-le)

```

```

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'
  using remaining-steps-decline step-size-new-small step-size-new-big
  by (smt (verit, best) add-le-mono le-refl le-trans)

```

```

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'

```

using *remaining-steps-decline step-size-small*
by (*metis Suc-eq-plus1 Suc-le-mono le-trans*)

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'*
using *remaining-steps-decline step-size-big*
by (*metis (no-types, lifting) add-mono-thms-linordered-semiring(3) order.trans*)

lemma *step-size-ok*: \llbracket *invar states; size-ok states* $\rrbracket \implies$ *size-ok* (*step states*)
using *step-size-ok-1 step-size-ok-2 step-size-ok-3 step-size-ok-4*
by (*smt (verit) invar-states.elims(1) size-ok'.elims(3) size-ok'.simps*)

lemma *step-n-size-ok*: \llbracket *invar states; size-ok states* $\rrbracket \implies$ *size-ok* ((*step* \sim *n*) *states*)
using *step-consistent-2*[*of size-ok states n*] *step-size-ok* **by** *blast*

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)*
using
invar-push-small[*of dir big small x*]
step-size-small[*of dir big Small.push x small dir' big' small'*]
size-push[*of small x*]
by *simp*

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)*
using
invar-push-small[*of dir big small x*]
step-size-new-small[*of dir big Small.push x small dir' big' small'*]
size-new-push[*of small x*]
by *simp*

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)*
using
invar-push-big[*of dir big small x*]
Big-Proof.size-push[*of big*]
step-size-big[*of dir Big.push x big small dir' big' small'*]
by *simp*

lemma *step-push-size-new-big* [*simp*]: \llbracket

$invar (States\ dir\ big\ small);$
 $step (States\ dir\ (Big.push\ x\ big)\ small) = States\ dir'\ big'\ small'$
 $\llbracket \implies size\ new\ big' = Suc\ (size\ new\ big)$
using
 $invar\ push\ big[of\ dir\ big\ small\ x]$
 $step\ size\ new\ big[of\ dir\ Big.push\ x\ big\ small\ dir'\ big'\ small']$
 $Big\ Proof.size\ new\ push[of\ big\ x]$
by simp

lemma *step-pop-size-big* [simp]: \llbracket
 $invar (States\ dir\ big\ small);$
 $0 < size\ big;$
 $Big.pop\ big = (x, bigP);$
 $step (States\ dir\ bigP\ small) = States\ dir'\ big'\ small'$
 $\llbracket \implies Suc\ (size\ big') = size\ big$
using
 $invar\ pop\ big[of\ dir\ big\ small\ x\ bigP]$
 $step\ size\ big[of\ dir\ bigP\ small\ dir'\ big'\ small']$
 $Big\ Proof.size\ pop[of\ big\ x\ bigP]$
by simp

lemma *step-pop-size-new-big* [simp]: \llbracket
 $invar (States\ dir\ big\ small);$
 $0 < size\ big; Big.pop\ big = (x, bigP);$
 $step (States\ dir\ bigP\ small) = States\ dir'\ big'\ small'$
 $\llbracket \implies Suc\ (size\ new\ big') = size\ new\ big$
using
 $invar\ pop\ big[of\ dir\ big\ small\ x\ bigP]$
 $Big\ Proof.size\ size\ new[of\ big]$
 $step\ size\ new\ big[of\ dir\ bigP\ small\ dir'\ big'\ small']$
 $Big\ Proof.size\ new\ pop[of\ big\ x\ bigP]$
by simp

lemma *step-n-size-small* [simp]: \llbracket
 $invar (States\ dir\ big\ small);$
 $(step\ \overset{\sim}{\sim} n)\ (States\ dir\ big\ small) = States\ dir'\ big'\ small'$
 $\llbracket \implies size\ small' = size\ small$
using *step-consistent*[of *size-small* *States dir big small n*]
by simp

lemma *step-n-size-big* [simp]:
 $\llbracket invar (States\ dir\ big\ small); (step\ \overset{\sim}{\sim} n)\ (States\ dir\ big\ small) = States\ dir'\ big'\ small'$
 $\implies size\ big' = size\ big$
using *step-consistent*[of *size-big* *States dir big small n*]
by simp

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

small']]
 $\implies \text{size-new } \text{small}' = \text{size-new } \text{small}$
using *step-consistent*[of *size-new-small States dir big small n*]
by *simp*

lemma *step-n-size-new-big [simp]*:
[[*invar (States dir big small); (step \sim n) (States dir big small) = States dir' big'*
small']]
 $\implies \text{size-new } \text{big}' = \text{size-new } \text{big}$
using *step-consistent*[of *size-new-big States dir big small n*]
by *simp*

lemma *step-n-push-size-small [simp]*: [[
invar (States dir big small);
(step \sim n) (States dir big (Small.push x small)) = States dir' big' small'
]] $\implies \text{size } \text{small}' = \text{Suc } (\text{size } \text{small})$
using *step-n-size-small invar-push-small Small-Proof.size-push*
by (*metis invar-states.simps*)

lemma *step-n-push-size-new-small [simp]*: [[
invar (States dir big small);
(step \sim n) (States dir big (Small.push x small)) = States dir' big' small'
]] $\implies \text{size-new } \text{small}' = \text{Suc } (\text{size-new } \text{small})$
by (*metis Small-Proof.size-new-push invar-states.simps invar-push-small step-n-size-new-small*)

lemma *step-n-push-size-big [simp]*: [[
invar (States dir big small);
(step \sim n) (States dir (Big.push x big) small) = States dir' big' small'
]] $\implies \text{size } \text{big}' = \text{Suc } (\text{size } \text{big})$
by (*metis Big-Proof.size-push invar-states.simps invar-push-big step-n-size-big*)

lemma *step-n-push-size-new-big [simp]*: [[
invar (States dir big small);
(step \sim n) (States dir (Big.push x big) small) = States dir' big' small'
]] $\implies \text{size-new } \text{big}' = \text{Suc } (\text{size-new } \text{big})$
by (*metis Big-Proof.size-new-push invar-states.simps invar-push-big step-n-size-new-big*)

lemma *step-n-pop-size-small [simp]*: [[
invar (States dir big small);
0 < size small;
Small.pop small = (x, smallP);
(step \sim n) (States dir big smallP) = States dir' big' small'
]] $\implies \text{Suc } (\text{size } \text{small}') = \text{size } \text{small}$
using *invar-pop-small size-pop step-n-size-small*
by (*metis (no-types, opaque-lifting) invar-states.simps*)

lemma *step-n-pop-size-new-small [simp]*: [[
invar (States dir big small);
0 < size small;

$Small.pop\ small = (x, smallP);$
 $(step \ \widetilde{\sim} \ n) (States\ dir\ big\ smallP) = States\ dir'\ big'\ small'$
 $\llbracket \implies Suc\ (size\ new\ small') = size\ new\ small$
using $invar\ pop\ small\ size\ new\ pop\ step\ n\ size\ new\ small\ size\ size\ new$
by $(metis\ (no\ types,\ lifting)\ invar\ states.\ simps)$

lemma $step\ n\ pop\ size\ big$ [simp]: \llbracket
 $invar\ (States\ dir\ big\ small);$
 $0 < size\ big; Big.pop\ big = (x, bigP);$
 $(step \ \widetilde{\sim} \ n) (States\ dir\ bigP\ small) = States\ dir'\ big'\ small'$
 $\llbracket \implies Suc\ (size\ big') = size\ big$
using $invar\ pop\ big\ Big\ Proof.\ size\ pop\ step\ n\ size\ big$
by $fastforce$

lemma $step\ n\ pop\ size\ new\ big$: \llbracket
 $invar\ (States\ dir\ big\ small);$
 $0 < size\ big; Big.pop\ big = (x, bigP);$
 $(step \ \widetilde{\sim} \ n) (States\ dir\ bigP\ small) = States\ dir'\ big'\ small'$
 $\llbracket \implies Suc\ (size\ new\ big') = size\ new\ big$
using $invar\ pop\ big\ Big\ Proof.\ size\ new\ pop\ step\ n\ size\ new\ big\ Big\ Proof.\ size\ size\ new$
by $(metis\ (no\ types,\ lifting)\ invar\ states.\ simps)$

lemma $remaining\ steps\ push\ small$ [simp]: $invar\ (States\ dir\ big\ small)$
 $\implies remaining\ steps\ (States\ dir\ big\ small) =$
 $remaining\ steps\ (States\ dir\ big\ (Small.push\ x\ small))$
by $(induction\ x\ small\ rule:\ Small.push.induct)(auto\ split:\ current.splits)$

lemma $remaining\ steps\ pop\ small$:
 $\llbracket invar\ (States\ dir\ big\ small); 0 < size\ small; Small.pop\ small = (x, smallP) \rrbracket$
 $\implies remaining\ steps\ (States\ dir\ big\ smallP) \leq remaining\ steps\ (States\ dir\ big\ small)$
proof $(induction\ small\ rule:\ Small.pop.induct)$
case 1
then show ?case
by $(auto\ simp:\ Common\ Proof.\ remaining\ steps\ pop\ max.coboundedI2\ split:\ prod.splits)$
next
case (2 current small auxS)
then show ?case
by $(induction\ current\ rule:\ Current.pop.induct)(auto\ split:\ big\ state.splits)$
next
case (3 current auxS big newS count)
then show ?case
by $(induction\ current\ rule:\ Current.pop.induct)\ auto$
qed

lemma $remaining\ steps\ pop\ big$:
 $\llbracket invar\ (States\ dir\ big\ small); 0 < size\ big; Big.pop\ big = (x, bigP) \rrbracket$
 $\implies remaining\ steps\ (States\ dir\ bigP\ small) \leq remaining\ steps\ (States\ dir\ big\ small)$

```

proof(induction big rule: Big.pop.induct)
  case (1 state)
  then show ?case
  proof(induction state rule: Common.pop.induct)
    case (1 current idle)
    then show ?case
    by(cases idle)(auto split: small-state.splits)
  next
  case (2 current aux new moved)
  then show ?case
  by(induction current rule: Current.pop.induct)(auto split: small-state.splits)
qed
next
  case (2 current big aux count)
  then show ?case
  proof(induction current rule: Current.pop.induct)
    case 1
    then show ?case
    by(auto split: small-state.splits current.splits)
  next
  case 2
  then show ?case
  by(auto split: small-state.splits current.splits)
qed
qed

```

lemma *remaining-steps-push-big* [*simp*]: *invar (States dir big small)*
 \implies *remaining-steps (States dir (Big.push x big) small) =*
remaining-steps (States dir big small)
by(*induction x big rule: Big.push.induct*)(*auto split: small-state.splits current.splits*)

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

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

lemma *step-4-remaining-steps-pop-big*: \llbracket
invar (States dir big small);

$0 < \text{size } \text{big};$
 $\text{Big.pop } \text{big} = (x, \text{bigP});$
 $4 \leq \text{remaining-steps } (\text{States } \text{dir } \text{bigP } \text{small});$
 $(\text{step } \sim^4) (\text{States } \text{dir } \text{bigP } \text{small}) = \text{States } \text{dir}' \text{big}' \text{small}'$
 $\mathbb{I} \implies \text{remaining-steps } (\text{States } \text{dir}' \text{big}' \text{small}') \leq \text{remaining-steps } (\text{States } \text{dir } \text{big}$
 $\text{small}) - 4$
by (*metis add-le-imp-le-diff invar-pop-big remaining-steps-pop-big remaining-steps-n-steps-plus*)

lemma *step-4-remaining-steps-pop-small*: \mathbb{I}
 $\text{invar } (\text{States } \text{dir } \text{big } \text{small});$
 $0 < \text{size } \text{small};$
 $\text{Small.pop } \text{small} = (x, \text{smallP});$
 $4 \leq \text{remaining-steps } (\text{States } \text{dir } \text{big } \text{smallP});$
 $(\text{step } \sim^4) (\text{States } \text{dir } \text{big } \text{smallP}) = \text{States } \text{dir}' \text{big}' \text{small}'$
 $\mathbb{I} \implies \text{remaining-steps } (\text{States } \text{dir}' \text{big}' \text{small}') \leq \text{remaining-steps } (\text{States } \text{dir } \text{big}$
 $\text{small}) - 4$
by (*metis add-le-imp-le-diff invar-pop-small remaining-steps-n-steps-plus remain-*
ing-steps-pop-small)

lemma *step-4-pop-small-size-ok-1*: \mathbb{I}
 $\text{invar } (\text{States } \text{dir } \text{big } \text{small});$
 $0 < \text{size } \text{small};$
 $\text{Small.pop } \text{small} = (x, \text{smallP});$
 $4 \leq \text{remaining-steps } (\text{States } \text{dir } \text{big } \text{smallP});$
 $(\text{step } \sim^4) (\text{States } \text{dir } \text{big } \text{smallP}) = \text{States } \text{dir}' \text{big}' \text{small}';$
 $\text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small}) + 1 \leq 4 * \text{size } \text{small}$
 $\mathbb{I} \implies \text{remaining-steps } (\text{States } \text{dir}' \text{big}' \text{small}') + 1 \leq 4 * \text{size } \text{small}'$
by (*smt (verit, ccfv-SIG) add.left-commute add.right-neutral add-le-cancel-left*
distrib-left-numeral dual-order.trans invar-pop-small le-add-diff-inverse2 mult.right-neutral
plus-1-eq-Suc remaining-steps-n-steps-sub remaining-steps-pop-small step-n-pop-size-small)

lemma *step-4-pop-big-size-ok-1*: \mathbb{I}
 $\text{invar } (\text{States } \text{dir } \text{big } \text{small});$
 $0 < \text{size } \text{big}; \text{Big.pop } \text{big} = (x, \text{bigP});$
 $4 \leq \text{remaining-steps } (\text{States } \text{dir } \text{bigP } \text{small});$
 $(\text{step } \sim^4) (\text{States } \text{dir } \text{bigP } \text{small}) = \text{States } \text{dir}' \text{big}' \text{small}';$
 $\text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small}) + 1 \leq 4 * \text{size } \text{small}$
 $\mathbb{I} \implies \text{remaining-steps } (\text{States } \text{dir}' \text{big}' \text{small}') + 1 \leq 4 * \text{size } \text{small}'$
by (*smt (verit, ccfv-SIG) add-leE add-le-cancel-right invar-pop-big order-trans*
remaining-steps-pop-big step-n-size-small remaining-steps-n-steps-plus)

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

by (*smt* (*z3*) *add.commute add-leE invar-pop-small le-add-diff-inverse2 nat-add-left-cancel-le remaining-steps-n-steps-sub step-n-size-big remaining-steps-pop-small*)

lemma *step-4-pop-big-size-ok-2*:

assumes

invar (*States dir big small*)

$0 < \text{size } big$

Big.pop big = (*x*, *bigP*)

remaining-steps (*States dir bigP small*) ≥ 4

$((\text{step } \sim 4) (\text{States dir bigP small})) = \text{States dir}' big' small'$

remaining-steps (*States dir big small*) + 1 $\leq 4 * \text{size } big$

shows

remaining-steps (*States dir}' big' small'*) + 1 $\leq 4 * \text{size } big'$

proof –

from *assms* **have** *remaining-steps* (*States dir bigP small*) + 1 $\leq 4 * \text{size } big$

by (*meson add-le-cancel-right order.trans remaining-steps-pop-big*)

with *assms* **show** *?thesis*

by (*smt* (*z3*) *Suc-diff-le Suc-eq-plus1 add-mult-distrib2 diff-diff-add diff-is-0-eq invar-pop-big mult-numeral-1-right numerals(1) plus-1-eq-Suc remaining-steps-n-steps-sub step-n-pop-size-big*)

qed

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

assumes

invar (*States dir big small*)

$0 < \text{size } small$

Small.pop small = (*x*, *smallP*)

remaining-steps (*States dir big smallP*) ≥ 4

$((\text{step } \sim 4) (\text{States dir big smallP})) = \text{States dir}' big' small'$

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

shows

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

by (*smt* (*verit*, *best*) *add-leD2 add-mono-thms-linordered-semiring(1) add-mono-thms-linordered-semiring(3) assms(1) assms(2) assms(3) assms(4) assms(5) assms(6) invar-pop-small le-add2 le-add-diff-inverse order-trans plus-1-eq-Suc remaining-steps-n-steps-sub remaining-steps-pop-small step-n-pop-size-new-small step-n-size-new-big*)

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

$0 < \text{size } big$;

$4 \leq \text{remaining-steps} (\text{States dir } big \text{ small})$;

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

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

by *linarith*

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

assumes
invar (States dir big small)
 $0 < \text{size big}$
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 $\leq 3 * \text{size-new big}$*
shows
*size-new small' + remaining-steps (States dir' big' small') + 2 $\leq 3 * \text{size-new big}'$*
proof –
from *assms*
have *size-new small + (remaining-steps (States dir big small) - 4) + 2 $\leq 3 * (\text{size-new big} - 1)$*
by (*meson dual-order.trans remaining-steps-pop-big step-4-pop-big-size-ok-3-aux*)
then
have *size-new small + remaining-steps (States dir' big' small') + 2 $\leq 3 * (\text{size-new big} - 1)$*
by (*smt (verit, ccfv-SIG) add-le-mono assms(1) assms(2) assms(3) assms(4) assms(5) dual-order.trans le-antisym less-or-eq-imp-le nat-less-le step-4-remaining-steps-pop-big*)
with *assms show ?thesis*
by (*metis diff-Suc-1 invar-pop-big step-n-size-new-small step-n-pop-size-new-big*)
qed

lemma *step-4-pop-small-size-ok-4-aux*: \llbracket
 $0 < \text{size small};$
 $4 \leq \text{remaining-steps (States dir big small)};$
 $\text{size-new big} + \text{remaining-steps (States dir big small)} + 2 \leq 3 * \text{size-new small}$
 $\rrbracket \implies \text{size-new big} + (\text{remaining-steps (States dir big small)} - 4) + 2 \leq 3 * (\text{size-new small} - 1)$
by *linarith*

lemma *step-4-pop-small-size-ok-4*:
assumes
invar (States dir big small)
 $0 < \text{size small}$
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 $\leq 3 * \text{size-new small}$*
shows
*size-new big' + remaining-steps (States dir' big' small') + 2 $\leq 3 * \text{size-new small}'$*
proof –
from *assms step-4-pop-small-size-ok-4-aux*

have $size\text{-}new\ big + (remaining\text{-}steps\ (States\ dir\ big\ small) - 4) + 2 \leq 3 * (size\text{-}new\ small - 1)$
by (*smt* (*verit*, *best*) *add-leE le-add-diff-inverse remaining-steps-pop-small*)

with *assms*
have $size\text{-}new\ big + remaining\text{-}steps\ (States\ dir'\ big'\ small') + 2 \leq 3 * (size\text{-}new\ small - 1)$
by (*smt* (*verit*, *best*) *add-le-cancel-left add-mono-thms-linordered-semiring(3) diff-le-mono invar-pop-small order-trans remaining-steps-n-steps-sub remaining-steps-pop-small*)

with *assms* **show** *?thesis*
by (*metis diff-Suc-1 invar-pop-small step-n-size-new-big step-n-pop-size-new-small*)
qed

lemma *step-4-pop-big-size-ok-4-aux*: \llbracket
 $0 < size\ big;$
 $4 \leq remaining\text{-}steps\ (States\ dir\ big\ small);$
 $size\text{-}new\ big + remaining\text{-}steps\ (States\ dir\ big\ small) + 2 \leq 3 * size\text{-}new\ small$
 $\rrbracket \implies size\text{-}new\ big - 1 + (remaining\text{-}steps\ (States\ dir\ big\ small) - 4) + 2 \leq 3 * size\text{-}new\ small$
by *linarith*

lemma *step-4-pop-big-size-ok-4*:
assumes
 $invar\ (States\ dir\ big\ small)$
 $0 < size\ big$
 $Big.pop\ big = (x, bigP)$
 $remaining\text{-}steps\ (States\ dir\ bigP\ small) \geq 4$
 $((step\ \sim\ 4)\ (States\ dir\ bigP\ small)) = (States\ dir'\ big'\ small')$
 $size\text{-}new\ big + remaining\text{-}steps\ (States\ dir\ big\ small) + 2 \leq 3 * size\text{-}new\ small$
shows
 $size\text{-}new\ big' + remaining\text{-}steps\ (States\ dir'\ big'\ small') + 2 \leq 3 * size\text{-}new\ small'$
proof –
from *assms step-4-pop-big-size-ok-4-aux*
have $(size\text{-}new\ big - 1) + (remaining\text{-}steps\ (States\ dir\ big\ small) - 4) + 2 \leq 3 * size\text{-}new\ small$
by *linarith*

with *assms*
have $(size\text{-}new\ big - 1) + remaining\text{-}steps\ (States\ dir'\ big'\ small') + 2 \leq 3 * size\text{-}new\ small$
by (*meson add-le-mono dual-order.eq-iff order-trans step-4-remaining-steps-pop-big*)

with *assms* **show** *?thesis*
by (*metis diff-Suc-1 invar-pop-big step-n-size-new-small step-n-pop-size-new-big*)
qed

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

invar (States dir big small);
 $4 \leq \text{remaining-steps (States dir big small)}$;
 $(\text{step} \sim^4) (\text{States dir big (Small.push x small)}) = \text{States dir' big' small'}$;
 $\text{remaining-steps (States dir big small)} + 1 \leq 4 * \text{size small}$
 $\implies \text{remaining-steps (States dir' big' small')} + 1 \leq 4 * \text{size small'}$
by (*smt (z3) add.commute add-leD1 add-le-mono le-add1 le-add-diff-inverse2*
mult-Suc-right nat-1-add-1 numeral-Bit0 step-n-push-size-small step-4-remaining-steps-push-small)

lemma *step-4-push-big-size-ok-1*: \llbracket
invar (States dir big small);
 $4 \leq \text{remaining-steps (States dir big small)}$;
 $(\text{step} \sim^4) (\text{States dir (Big.push x big) small}) = \text{States dir' big' small'}$;
 $\text{remaining-steps (States dir big small)} + 1 \leq 4 * \text{size small}$
 $\implies \text{remaining-steps (States dir' big' small')} + 1 \leq 4 * \text{size small'}$
by (*smt (verit, cfv-SIG) Nat.le-diff-conv2 add-leD2 invar-push-big le-add1 le-add-diff-inverse2*
remaining-steps-n-steps-sub remaining-steps-push-big step-n-size-small)

lemma *step-4-push-small-size-ok-2*: \llbracket
invar (States dir big small);
 $4 \leq \text{remaining-steps (States dir big small)}$;
 $(\text{step} \sim^4) (\text{States dir big (Small.push x small)}) = \text{States dir' big' small'}$;
 $\text{remaining-steps (States dir big small)} + 1 \leq 4 * \text{size big}$
 $\implies \text{remaining-steps (States dir' big' small')} + 1 \leq 4 * \text{size big'}$
by (*metis (full-types) Suc-diff-le Suc-eq-plus1 invar-push-small less-Suc-eq-le less-imp-diff-less*
step-4-remaining-steps-push-small step-n-size-big)

lemma *step-4-push-big-size-ok-2*: \llbracket
invar (States dir big small);
 $4 \leq \text{remaining-steps (States dir big small)}$;
 $(\text{step} \sim^4) (\text{States dir (Big.push x big) small}) = \text{States dir' big' small'}$;
 $\text{remaining-steps (States dir big small)} + 1 \leq 4 * \text{size big}$
 $\implies \text{remaining-steps (States dir' big' small')} + 1 \leq 4 * \text{size big'}$
by (*smt (verit, cfv-SIG) add.commute add-diff-cancel-left' add-leD1 add-le-mono*
invar-push-big mult-Suc-right nat-le-iff-add one-le-numeral remaining-steps-n-steps-sub
remaining-steps-push-big step-n-push-size-big)

lemma *step-4-push-small-size-ok-3-aux*: \llbracket
 $4 \leq \text{remaining-steps (States dir big small)}$;
 $\text{size-new small} + \text{remaining-steps (States dir big small)} + 2 \leq 3 * \text{size-new big}$
 $\implies \text{Suc (size-new small)} + (\text{remaining-steps (States dir big small)} - 4) + 2 \leq$
 $3 * \text{size-new big}$
using *distrib-left dual-order.trans le-add-diff-inverse2* **by** *force*

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

size-new big'
using *step-n-size-new-big step-n-push-size-new-small step-4-remaining-steps-push-small*
by (*metis invar-push-small step-4-push-small-size-ok-3-aux*)

lemma *step-4-push-big-size-ok-3-aux*: \llbracket
 $4 \leq \text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small});$
 $\text{size-new small} + \text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small}) + 2 \leq 3 * \text{size-new big}$
 $\rrbracket \implies \text{size-new small} + (\text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small}) - 4) + 2 \leq 3 * \text{Suc } (\text{size-new big})$
using *distrib-left dual-order.trans le-add-diff-inverse2* **by** *force*

lemma *step-4-push-big-size-ok-3*: \llbracket
 $\text{invar } (\text{States } \text{dir } \text{big } \text{small});$
 $4 \leq \text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small});$
 $(\text{step } \sim^4) (\text{States } \text{dir } (\text{Big.push } x \text{ big } \text{small})) = \text{States } \text{dir}' \text{big}' \text{small}';$
 $\text{size-new small} + \text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small}) + 2 \leq 3 * \text{size-new big}$
 $\rrbracket \implies \text{size-new small}' + \text{remaining-steps } (\text{States } \text{dir}' \text{big}' \text{small}') + 2 \leq 3 * \text{size-new big}'$
by (*metis invar-push-big remaining-steps-n-steps-sub remaining-steps-push-big step-4-push-big-size-ok-3-aux step-n-push-size-new-big step-n-size-new-small*)

lemma *step-4-push-small-size-ok-4-aux*: \llbracket
 $4 \leq \text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small});$
 $\text{size-new big} + \text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small}) + 2 \leq 3 * \text{size-new small}$
 $\rrbracket \implies \text{size-new big} + (\text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small}) - 4) + 2 \leq 3 * \text{Suc } (\text{size-new small})$
using *distrib-left dual-order.trans le-add-diff-inverse2* **by** *force*

lemma *step-4-push-small-size-ok-4*: \llbracket
 $\text{invar } (\text{States } \text{dir } \text{big } \text{small});$
 $4 \leq \text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small});$
 $(\text{step } \sim^4) (\text{States } \text{dir } \text{big } (\text{Small.push } x \text{ small})) = \text{States } \text{dir}' \text{big}' \text{small}';$
 $\text{size-new big} + \text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small}) + 2 \leq 3 * \text{size-new small}$
 $\rrbracket \implies \text{size-new big}' + \text{remaining-steps } (\text{States } \text{dir}' \text{big}' \text{small}') + 2 \leq 3 * \text{size-new small}'$
by (*metis invar-push-small step-n-size-new-big step-n-push-size-new-small step-4-remaining-steps-push-small step-4-push-small-size-ok-4-aux*)

lemma *step-4-push-big-size-ok-4-aux*: \llbracket
 $4 \leq \text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small});$
 $\text{size-new big} + \text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small}) + 2 \leq 3 * \text{size-new small}$
 $\rrbracket \implies \text{Suc } (\text{size-new big}) + (\text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small}) - 4) + 2 \leq 3 * \text{size-new small}$
using *distrib-left dual-order.trans le-add-diff-inverse2* **by** *force*

lemma *step-4-push-big-size-ok-4*: \llbracket
 $\text{invar } (\text{States } \text{dir } \text{big } \text{small});$
 $4 \leq \text{remaining-steps } (\text{States } \text{dir } \text{big } \text{small});$
 $(\text{step } \sim^4) (\text{States } \text{dir } (\text{Big.push } x \text{ big } \text{small})) = \text{States } \text{dir}' \text{big}' \text{small}';$

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

by (*metis invar-push-big remaining-steps-n-steps-sub remaining-steps-push-big step-4-push-big-size-ok-4-aux step-n-push-size-new-big step-n-size-new-small*)

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

invar (States dir big small);
 $4 \leq \text{remaining-steps (States dir big small)};$
size-ok (States dir big small)

$\llbracket \implies \text{size-ok ((step} \sim 4) (\text{States dir big (Small.push x small)}))$

using *step-4-push-small-size-ok-1 step-4-push-small-size-ok-2 step-4-push-small-size-ok-3 step-4-push-small-size-ok-4*

by (*smt (verit) size-ok'.elims(3) size-ok'.simps*)

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

invar (States dir big small);
 $4 \leq \text{remaining-steps (States dir big small)};$
size-ok (States dir big small)

$\llbracket \implies \text{size-ok ((step} \sim 4) (\text{States dir (Big.push x big) small}))$

using *step-4-push-big-size-ok-1 step-4-push-big-size-ok-2 step-4-push-big-size-ok-3 step-4-push-big-size-ok-4*

by (*smt (verit) size-ok'.elims(3) size-ok'.simps*)

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

invar (States dir big small);
 $0 < \text{size small};$
Small.pop small = (x, smallP);
 $4 \leq \text{remaining-steps (States dir big smallP)};$
size-ok (States dir big small)

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

by (*smt (verit) size-ok'.elims(3) size-ok'.simps step-4-pop-small-size-ok-1 step-4-pop-small-size-ok-2 step-4-pop-small-size-ok-3 step-4-pop-small-size-ok-4*)

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

invar (States dir big small);
 $0 < \text{size big};$ *Big.pop big = (x, bigP);*
 $4 \leq \text{remaining-steps (States dir bigP small)};$
size-ok (States dir big small)

$\llbracket \implies \text{size-ok ((step} \sim 4) (\text{States dir bigP small}))$

using *step-4-pop-big-size-ok-1 step-4-pop-big-size-ok-2 step-4-pop-big-size-ok-3 step-4-pop-big-size-ok-4*

by (*smt (verit) size-ok'.elims(3) size-ok'.simps*)

lemma *size-ok-size-small*: *size-ok (States dir big small) $\implies 0 < \text{size small}$*

by *auto*

lemma *size-ok-size-big*: *size-ok (States dir big small) $\implies 0 < \text{size big}$*

by *auto*

lemma *size-ok-size-new-small*: $\text{size-ok } (\text{States } \text{dir } \text{big } \text{small}) \implies 0 < \text{size-new } \text{small}$
by *auto*

lemma *size-ok-size-new-big*: $\text{size-ok } (\text{States } \text{dir } \text{big } \text{small}) \implies 0 < \text{size-new } \text{big}$
by *auto*

lemma *step-size-ok'*: $\llbracket \text{invar } \text{states}; \text{size-ok}' \text{ states } n \rrbracket \implies \text{size-ok}' (\text{step } \text{states}) n$
by (*smt (verit, ccfv-SIG) size-ok'.elims(2) size-ok'.elims(3) step-size-big step-size-new-big step-size-new-small step-size-small*)

lemma *step-same*: $\text{step } (\text{States } \text{dir } \text{big } \text{small}) = \text{States } \text{dir}' \text{big}' \text{small}' \implies \text{dir} = \text{dir}'$
by(*induction States dir big small rule: step-states.induct*) *auto*

lemma *step-n-same*: $(\text{step } \sim^n) (\text{States } \text{dir } \text{big } \text{small}) = \text{States } \text{dir}' \text{big}' \text{small}' \implies \text{dir} = \text{dir}'$
proof(*induction n arbitrary: big small big' small'*)
case 0
then show *?case*
by *simp*
next
case (*Suc n*)
obtain *big'' small''* **where** $\text{step } (\text{States } \text{dir } \text{big } \text{small}) = \text{States } \text{dir } \text{big}'' \text{small}''$
by (*metis states.exhaust step-same*)

with *Suc* **show** *?case*
by(*auto simp: funpow-swap1*)
qed

lemma *step-listL*: $\text{invar } \text{states} \implies \text{listL } (\text{step } \text{states}) = \text{listL } \text{states}$
proof(*induction states rule: listL.induct*)
case (*1 big small*)
then have $\text{list-small-first } (\text{States } \text{Left } \text{big } \text{small}) =$
 $\text{Small-Aux.list-current } \text{small} @ \text{rev } (\text{Big-Aux.list-current } \text{big})$
by *auto*

then have $\text{list-small-first } (\text{step } (\text{States } \text{Left } \text{big } \text{small})) =$
 $\text{Small-Aux.list-current } \text{small} @ \text{rev } (\text{Big-Aux.list-current } \text{big})$
using *1 step-lists* **by** *fastforce*

then have $\text{listL } (\text{step } (\text{States } \text{Left } \text{big } \text{small})) =$
 $\text{Small-Aux.list-current } \text{small} @ \text{rev } (\text{Big-Aux.list-current } \text{big})$
by (*smt (verit, ccfv-SIG) 1 invar-states.elims(2) States-Proof.invar-step listL.simps(1) step-same*)

with *1* **show** *?case*
by *auto*
next

```

case (2 big small)
then have a: list-big-first (States Right big small) =
    Big-Aux.list-current big @ rev (Small-Aux.list-current small)
using invar-list-big-first[of States Right big small]
by auto

then have list-big-first (step (States Right big small)) =
    Big-Aux.list-current big @ rev (Small-Aux.list-current small)
using 2 step-lists by fastforce

then have listL (step (States Right big small)) =
    Big-Aux.list-current big @ rev (Small-Aux.list-current small)
by (metis(full-types) listL.cases listL.simps(2) step-same)

with 2 show ?case
using a by force
qed

lemma step-n-listL: invar states  $\implies$  listL ((step~n) states) = listL states
using step-consistent[of listL states] step-listL
by metis

lemma listL-remaining-steps:
assumes
    listL states = []
    0 < remaining-steps states
    invar states
    size-ok states
shows
    False
proof(cases states)
case (States dir big small)
with assms show ?thesis
using Small-Proof.list-current-size size-ok-size-small
by(cases dir; cases lists (States dir big small)) auto
qed

lemma invar-step-n: invar (states :: 'a states)  $\implies$  invar ((step~n) states)
by (simp add: invar-step step-consistent-2)

lemma step-n-size-ok': [[invar states; size-ok' states x]  $\implies$  size-ok' ((step~n)
states) x
proof(induction n arbitrary: states x)
case 0
then show ?case by auto
next
case Suc
then show ?case
using invar-step-n step-size-ok'

```

by fastforce
qed

lemma *size-ok-steps*: \llbracket
invar states;
size-ok' *states* (*remaining-steps states - n*)
 $\rrbracket \implies \text{size-ok } ((\text{step } \sim n) \text{ states})$
by (*simp add: step-n-size-ok'*)

lemma *remaining-steps-idle*: *invar states*
 $\implies \text{remaining-steps states} = 0 \longleftrightarrow$ (
case states of
States - (Big2 (Common.Idle - -)) (Small3 (Common.Idle - -)) \implies True
| - \implies False)
by(*cases states*)
(*auto split: big-state.split small-state.split common-state.split current.splits*)

lemma *remaining-steps-idle'*:
 $\llbracket \text{invar } (\text{States dir big small}); \text{remaining-steps } (\text{States dir big small}) = 0 \rrbracket$
 $\implies \exists \text{big-current big-idle small-current small-idle. States dir big small} =$
States dir
(Big2 (common-state.Idle big-current big-idle))
(Small3 (common-state.Idle small-current small-idle))
using *remaining-steps-idle*[*of States dir big small*]
by(*cases big; cases small*) (*auto split!: common-state.splits*)

end

19 Dequeue Proofs

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

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

proof(*induction deque arbitrary: x rule: deqL'.induct*)
case (*4 left right length-right*)

then obtain *left'* **where** *pop-left*[*simp*]: *Idle.pop left* = (*x, left'*)
by(*auto simp: Let-def split: if-splits stack.splits prod.splits idle.splits*)

then obtain *stack-left' length-left'*
where *left'*[*simp*]: *left'* = *idle.Idle stack-left' length-left'*
using *idle.exhaust* **by** *blast*

from *4* **have** *invar-left'*: *invar left'*
using *Idle-Proof.invar-pop*[*of left*]
by *auto*

```

then have size-left' [simp]: size stack-left' = length-left'
  by auto

have size-left'-size-left [simp]: size stack-left' = (size left) - 1
  using Idle-Proof.size-pop-sub[of left x left']
  by auto

show ?case
proof(cases  $3 * \text{length-left}' \geq \text{length-right}$ )
  case True
  with  $4 \text{ pop-left}$  show ?thesis
    using Idle-Proof.pop-list[of left x left']
    by auto
  next
  case False
  note Start-Rebalancing = False

  then show ?thesis
  proof(cases  $\text{length-left}' \geq 1$ )
    case True
    let ?big = Big1 (Current []  $0 \text{ right}$  ( $\text{size right} - \text{Suc length-left}'$ )
      right [] ( $\text{size right} - \text{Suc length-left}'$ ))
    let ?small = Small1 (Current []  $0 \text{ stack-left}'$  ( $\text{Suc } (2 * \text{length-left}')$ )) stack-left'
  []
    let ?states = States Left ?big ?small

    from  $4 \text{ Start-Rebalancing True invar-left'}$  have invar: invar ?states
      by(auto simp: Let-def rev-take rev-drop)

    with  $4 \text{ Start-Rebalancing True invar-left'}$ 
    have States-Aux.listL ?states = tl (Idle-Aux.list left) @ rev (Stack-Aux.list
right)
      using pop-list-tl'[of left x left']
      by (auto simp del: take-rev-def)

    with invar
    have States-Aux.listL ((step  $\sim 6$ ) ?states) =
      tl (Idle-Aux.list left) @ rev (Stack-Aux.list right)
      using step-n-listL[of ?states  $6$ ]
      by presburger

    with  $4 \text{ Start-Rebalancing True}$  show ?thesis
      by(auto simp: Let-def)
  next
  case False
  from False Start-Rebalancing 4 have [simp]: size left = 1
    using size-left' size-left'-size-left by auto

```

```

with False Start-Rebalancing 4 have [simp]: Idle-Aux.list left = [x]
  by(induction left)(auto simp: length-one-hd split: stack.splits)

obtain right1 right2 where right = Stack right1 right2
  using Stack-Aux.list.cases by blast

with False Start-Rebalancing 4 show ?thesis
  by(induction right1 right2 rule: small-deque.induct) auto
qed
qed
next
case (5 big small)

then have start-invar: invar (States Left big small)
  by auto

from 5 have small-invar: invar small
  by auto

from 5 have small-size: 0 < size small
  by auto

with 5(3) obtain small' where pop: Small.pop small = (x, small')
  by(cases small)
  (auto simp: Let-def split: states.splits direction.splits state-splits prod.splits)

let ?states-new = States Left big small'
let ?states-stepped = (step  $\overset{\sim}{4}$ ) ?states-new

have invar: invar ?states-new
  using pop start-invar small-size invar-pop-small[of Left big small x small']
  by auto

have x # Small-Aux.list-current small' = Small-Aux.list-current small
  using small-invar small-size pop Small-Proof.pop-list-current[of small x small']
  by auto

then have listL:
  x # States-Aux.listL ?states-new =
    Small-Aux.list-current small @ rev (Big-Aux.list-current big)
  using invar small-size Small-Proof.pop-list-current[of small x small'] 5(1)
  by auto

from invar have invar ?states-stepped
  using invar-step-n by blast

then have states-listL-list-current [simp]: x # States-Aux.listL ?states-stepped
=
  Small-Aux.list-current small @ rev (Big-Aux.list-current big)

```

```

using States-Proof.step-n-listL invar listL by metis

then have listL (deqL (Rebal (States Left big small))) = States-Aux.listL ?states-stepped
by(auto simp: Let-def pop split: prod.splits direction.splits states.splits state-splits)

then have states-listL-list-current:
  x # listL (deqL (Rebal (States Left big small))) =
  Small-Aux.list-current small @ rev (Big-Aux.list-current big)
by auto

with 5(1) have listL (Rebal (States Left big small)) =
  Small-Aux.list-current small @ rev (Big-Aux.list-current big)
by auto

with states-listL-list-current
have x # listL (deqL (Rebal (States Left big small))) =
  listL (Rebal (States Left big small))
by auto

with 5 show ?case by auto
next
case (6 big small)
then have start-invar: invar (States Right big small)
by auto

from 6 have big-invar: invar big
by auto

from 6 have big-size: 0 < size big
by auto

with 6(3) obtain big' where pop: Big.pop big = (x, big')
by(cases big)
  (auto simp: Let-def split: prod.splits direction.splits states.splits state-splits)

let ?states-new = States Right big' small
let ?states-stepped = (step4) ?states-new

have invar: invar ?states-new
using pop start-invar big-size invar-pop-big[of Right big small]
by auto

have big-list-current: x # Big-Aux.list-current big' = Big-Aux.list-current big
using big-invar big-size pop by auto

then have listL:
  x # States-Aux.listL ?states-new =
  Big-Aux.list-current big @ rev (Small-Aux.list-current small)
proof(cases States-Aux.lists ?states-new)

```

```

    case (Pair bigs smalls)
  with invar big-list-current show ?thesis
    using app-rev[of smalls bigs]
    by(auto split: prod.splits)
qed

from invar have four-steps: invar ?states-stepped
  using invar-step-n by blast

then have [simp]:
  x # States-Aux.listL ?states-stepped =
    Big-Aux.list-current big @ rev (Small-Aux.list-current small)
  using States-Proof.step-n-listL[of ?states-new 4] invar listL
  by auto

then have listL (deqL (Rebal (States Right big small))) =
  States-Aux.listL ?states-stepped
  by(auto simp: Let-def pop split: prod.splits direction.splits states.splits state.splits)

then have listL-list-current:
  x # listL (deqL (Rebal (States Right big small))) =
    Big-Aux.list-current big @ rev (Small-Aux.list-current small)
  by auto

with 6(1) have listL (Rebal (States Right big small)) =
  Big-Aux.list-current big @ rev (Small-Aux.list-current small)
  using invar-list-big-first[of States Right big small] by fastforce

with listL-list-current have
  x # listL (deqL (Rebal (States Right big small))) =
    listL (Rebal (States Right big small))
  by auto

with 6 show ?case by auto
qed auto

lemma list-deqL [simp]:
  [[invar deque; listL deque ≠ []] ⇒ listL (deqL deque) = tl (listL deque)]
  using cons-tl[of fst (deqL' deque) listL (deqL deque) listL deque]
  by(auto split: prod.splits)

lemma list-firstL [simp]:
  [[invar deque; listL deque ≠ []] ⇒ firstL deque = hd (listL deque)]
  using cons-hd[of fst (deqL' deque) listL (deqL deque) listL deque]
  by(auto split: prod.splits)

lemma invar-deqL:
  [[invar deque; ¬ is-empty deque] ⇒ invar (deqL deque)]
  proof(induction deque rule: deqL'.induct)

```

```

case ( $\_4$  left right length-right)
then obtain  $x$  left' where pop-left[simp]: Idle.pop left = ( $x$ , left')
  by fastforce

then obtain stack-left' length-left'
  where left'[simp]: left' = idle.Idle stack-left' length-left'
  using idle.exhaust by blast

from  $\_4$  have invar-left': invar left' invar left
  using Idle-Proof.invar-pop by fastforce+

have [simp]: size stack-left' = size left - 1
  by (metis Idle-Proof.size-pop-sub left' pop-left size-idle.simps)

have [simp]: length-left' = size left - 1
  using invar-left' by auto

from  $\_4$  have list:  $x \#$  Idle-Aux.list left' = Idle-Aux.list left
  using Idle-Proof.pop-list[of left  $x$  left']
  by auto

show ?case
proof(cases length-right  $\leq 3 * \text{size left}'$ )
  case True
    with  $\_4$  invar-left' show ?thesis
      by(auto simp: Stack-Proof.size-empty[symmetric])
  next
    case False
      note Start-Rebalancing = False
      then show ?thesis
        proof(cases  $1 \leq \text{size left}'$ )
          case True
            let ?big =
              Big1
              (Current  $\sqcap$  0 right (size right - Suc length-left'))
              right  $\sqcap$  (size right - Suc length-left')
            let ?small = Small1 (Current  $\sqcap$  0 stack-left' (Suc (2 * length-left'))) stack-left'
             $\square$ 
            let ?states = States Left ?big ?small

          from  $\_4$  Start-Rebalancing True invar-left'
          have invar: invar ?states
            by(auto simp: Let-def rev-take rev-drop)

          then have invar-stepped: invar ((step6) ?states)
            using invar-step-n by blast

          from  $\_4$  Start-Rebalancing True
          have remaining-steps:  $6 < \text{remaining-steps } ?states$ 

```

```

    by auto

  then have remaining-steps-end: 0 < remaining-steps ((step6) ?states)
    by(simp only: remaining-steps-n-steps-sub[of ?states 6] invar)

  from 4 Start-Rebalancing True
  have size-ok': size-ok' ?states (remaining-steps ?states - 6)
    by auto

  then have size-ok: size-ok ((step6) ?states)
    using invar remaining-steps size-ok-steps by blast

  from True Start-Rebalancing 4 show ?thesis
    using remaining-steps-end size-ok invar-stepped
    by(auto simp: Let-def)
next
case False
from False Start-Rebalancing 4 have [simp]: size left = 1
  by auto

with False Start-Rebalancing 4 have [simp]: Idle-Aux.list left = [x]
  using list[symmetric]
  by(auto simp: list Stack-Proof.list-empty-size)

obtain right1 right2 where right = Stack right1 right2
  using Stack-Aux.list.cases by blast

with False Start-Rebalancing 4 show ?thesis
  by(induction right1 right2 rule: small-deque.induct) auto
qed
qed
next
case (5 big small)

obtain x small' where small' [simp]: Small.pop small = (x, small')
  by fastforce

let ?states = States Left big small'
let ?states-stepped = (step4) ?states

obtain big-stepped small-stepped where stepped [simp]:
  ?states-stepped = States Left big-stepped small-stepped
  by (metis remaining-steps-states.cases step-n-same)

from 5 have invar: invar ?states
  using invar-pop-small[of Left big small x small']
  by auto

then have invar-stepped: invar ?states-stepped

```

```

using invar-step-n by blast

show ?case
proof(cases 4 < remaining-steps ?states)
  case True

    then have remaining-steps: 0 < remaining-steps ?states-stepped
      using invar remaining-steps-n-steps-sub[of ?states 4]
      by simp

    from True have size-ok: size-ok ?states-stepped
      using step-4-pop-small-size-ok[of Left big small x small^] 5(1)
      by auto

    from remaining-steps size-ok invar-stepped show ?thesis
      by(cases big-stepped; cases small-stepped) (auto simp: Let-def split!: com-
mon-state.split)
    next
      case False
      then have remaining-steps-stepped: remaining-steps ?states-stepped = 0
        using invar by(auto simp del: stepped)

    then obtain small-current small-idle big-current big-idle where idle [simp]:
      States Left big-stepped small-stepped =
      States Left
      (Big2 (common-state.Idle big-current big-idle))
      (Small3 (common-state.Idle small-current small-idle))

      using remaining-steps-idle' invar-stepped remaining-steps-stepped
      by fastforce

    have size-new-small : 1 < size-new small
      using 5 by auto

    have [simp]: size-new small = Suc (size-new small')
      using 5 by auto

    have [simp]: size-new small' = size-new small-stepped
      using invar step-n-size-new-small stepped
      by metis

    have [simp]: size-new small-stepped = size small-idle
      using idle invar-stepped
      by(cases small-stepped) auto

    have [simp]:  $\neg$ is-empty small-idle
      using size-new-small
      by (simp add: Idle-Proof.size-not-empty)

```

```

have [simp]: size-new big = size-new big-stepped
  by (metis invar step-n-size-new-big stepped)

have [simp]: size-new big-stepped = size big-idle
  using idle invar-stepped
  by(cases big-stepped) auto

have 0 < size big-idle
  using 5 by auto

then have [simp]: ¬is-empty big-idle
  by (auto simp: Idle-Proof.size-not-empty)

have [simp]: size small-idle ≤ 3 * size big-idle
  using 5 by auto

have [simp]: size big-idle ≤ 3 * size small-idle
  using 5 by auto

show ?thesis
  using invar-stepped by auto
qed
next
case (6 big small)

obtain x big' where big' [simp]: Big.pop big = (x, big')
  by fastforce

let ?states = States Right big' small
let ?states-stepped = (step4) ?states

obtain big-stepped small-stepped where stepped [simp]:
  ?states-stepped = States Right big-stepped small-stepped
  by (metis remaining-steps-states.cases step-n-same)

from 6 have invar: invar ?states
  using invar-pop-big[of Right big small x big']
  by auto

then have invar-stepped: invar ?states-stepped
  using invar-step-n by blast

show ?case
proof(cases 4 < remaining-steps ?states)
case True

then have remaining-steps: 0 < remaining-steps ?states-stepped
  using invar remaining-steps-n-steps-sub[of ?states 4]
  by simp

```

```

from True have size-ok: size-ok ?states-stepped
  using step-4-pop-big-size-ok[of Right big small x big^] 6(1)
  by auto

from remaining-steps size-ok invar-stepped show ?thesis
  by(cases big-stepped; cases small-stepped) (auto simp: Let-def split!: com-
mon-state.split)
next
case False
then have remaining-steps-stepped: remaining-steps ?states-stepped = 0
  using invar by(auto simp del: stepped)

then obtain small-current small-idle big-current big-idle where idle [simp]:
  States Right big-stepped small-stepped =
  States Right
  (Big2 (common-state.Idle big-current big-idle))
  (Small3 (common-state.Idle small-current small-idle))

  using remaining-steps-idle' invar-stepped remaining-steps-stepped
  by fastforce

have size-new-big : 1 < size-new big
  using 6 by auto

have [simp]: size-new big = Suc (size-new big')
  using 6 by auto

have [simp]: size-new big' = size-new big-stepped
  using invar step-n-size-new-big stepped
  by metis

have [simp]: size-new big-stepped = size big-idle
  using idle invar-stepped
  by(cases big-stepped) auto

have [simp]: ¬is-empty big-idle
  using size-new-big
  by (simp add: Idle-Proof.size-not-empty)

have [simp]: size-new small = size-new small-stepped
  by (metis invar step-n-size-new-small stepped)

have [simp]: size-new small-stepped = size small-idle
  using idle invar-stepped
  by(cases small-stepped) auto

have 0 < size small-idle
  using 6 by auto

```

```

then have [simp]:  $\neg$ is-empty small-idle
  by (auto simp: Idle-Proof.size-not-empty)

have [simp]: size big-idle  $\leq$  3 * size small-idle
  using 6 by auto

have [simp]: size small-idle  $\leq$  3 * size big-idle
  using 6 by auto

show ?thesis
  using invar-stepped by auto
qed
qed auto

end

```

20 Enqueue Proofs

```

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

```

```

lemma list-engL: invar deque  $\implies$  listL (engL x deque) = x # listL deque
proof(induction x deque rule: engL.induct)
  case (5 x left right length-right)

```

```

  obtain left' length-left' where pushed [simp]:
    Idle.push x left = idle.Idle left' length-left'
  using is-empty-idle.cases by blast

```

```

then have invar-left': invar (idle.Idle left' length-left')
  using Idle-Proof.invar-push[of left x] 5 by auto

```

```

show ?case
proof(cases length-left'  $\leq$  3 * length-right)
  case True
  then show ?thesis
    using Idle-Proof.push-list[of x left]
    by(auto simp: Let-def)

```

```

next
  case False
  let ?length-left = length-left' - length-right - 1
  let ?length-right = 2 * length-right + 1
  let ?big = Big1 (Current [] 0 left' ?length-left) left' [] ?length-left
  let ?small = Small1 (Current [] 0 right ?length-right) right []
  let ?states = States Right ?big ?small
  let ?states-stepped = (step6) ?states

```

```

from False 5 invar-left' have invar: invar ?states
  by(auto simp: rev-drop rev-take)

then have States-Aux.listL ?states = x # Idle-Aux.list left @ rev (Stack-Aux.list
right)
  using Idle-Proof.push-list[of x left]
  by(auto)

then have States-Aux.listL ?states-stepped = x # Idle-Aux.list left @ rev
(Stack-Aux.list right)
  by (metis invar step-n-listL)

with False show ?thesis
  by(auto simp: Let-def)
qed
next
case (6 x big small)
let ?small = Small.push x small
let ?states = States Left big ?small
let ?states-stepped = (step4) ?states

obtain big-stepped small-stepped where stepped:
  ?states-stepped = States Left big-stepped small-stepped
  by (metis remaining-steps-states.cases step-n-same)

from 6 have invar ?states
  using invar-push-small[of Left big small x]
  by auto

then have
  States-Aux.listL ?states-stepped =
  x # Small-Aux.list-current small @ rev (Big-Aux.list-current big)
  using step-n-listL by fastforce

with 6 show ?case
  by(cases big-stepped; cases small-stepped)
  (auto simp: Let-def stepped split!: common-state.split)
next
case (7 x big small)

let ?big = Big.push x big
let ?states = States Right ?big small
let ?states-stepped = (step4) ?states

obtain big-stepped small-stepped where stepped:
  ?states-stepped = States Right big-stepped small-stepped
  by (metis remaining-steps-states.cases step-n-same)

from 7 have list-invar:

```

```

    list-current-small-first (States Right big small) = list-small-first (States Right
big small)
    by auto

from  $\gamma$  have invar: invar ?states
    using invar-push-big[of Right big small x]
    by auto

then have
    States-Aux.listL ?states = x # Big-Aux.list-current big @ rev (Small-Aux.list-current
small)
    using app-rev[of - - - x # Big-Aux.list-current big]
    by(auto split: prod.split)

then have
    States-Aux.listL ?states-stepped =
    x # Big-Aux.list-current big @ rev (Small-Aux.list-current small)
    by (metis invar step-n-listL)

with list-invar show ?case
    using app-rev[of Small-Aux.list-current small Big-Aux.list-current big]
    by(cases big-stepped; cases small-stepped)
    (auto simp: Let-def stepped split!: prod.split common-state.split)
qed auto

lemma invar-enqL: invar deque  $\implies$  invar (enqL x deque)
proof(induction x deque rule: enqL.induct)
    case (5 x left right length-right)
    obtain left' length-left' where pushed [simp]:
        Idle.push x left = idle.Idle left' length-left'
    using is-empty-idle.cases by blast

then have invar-left': invar (idle.Idle left' length-left')
    using Idle-Proof.invar-push[of left x] 5 by auto

have [simp]: size left' = Suc (size left)
    using Idle-Proof.size-push[of x left]
    by auto

show ?case
proof(cases length-left'  $\leq$  3 * length-right)
    case True
    with 5 show ?thesis
    using invar-left' Idle-Proof.size-push[of x left] Stack-Proof.size-not-empty[of
left']
    by auto
next
    case False
    let ?length-left = length-left' - length-right - 1

```

```

let ?length-right = Suc (2 * length-right)
let ?states = States Right
  (Big1 (Current [] 0 left' ?length-left) left' [] ?length-left)
  (Small1 (Current [] 0 right ?length-right) right [])
let ?states-stepped = (step6) ?states

from invar-left' 5 False have invar: invar ?states
  by(auto simp: rev-drop rev-take)

then have invar-stepped: invar ?states-stepped
  using invar-step-n by blast

from False invar-left' 5 have remaining-steps: 6 < remaining-steps ?states
  using Stack-Proof.size-not-empty[of right]
  by auto

then have remaining-steps-stepped: 0 < remaining-steps ?states-stepped
  using invar remaining-steps-n-steps-sub
  by (metis zero-less-diff)

from False invar-left' 5 have size-ok' ?states (remaining-steps ?states - 6)
  using Stack-Proof.size-not-empty[of right]
  size-not-empty
  by auto

then have size-ok-stepped: size-ok ?states-stepped
  using size-ok-steps[of ?states 6] remaining-steps invar
  by blast

from False show ?thesis
  using invar-stepped remaining-steps-stepped size-ok-stepped
  by(auto simp: Let-def)
qed
next
case (6 x big small)
let ?small = Small.push x small
let ?states = States Left big ?small
let ?states-stepped = (step4) ?states

from 6 have invar: invar ?states
  using invar-push-small[of Left big small x]
  by auto

then have invar-stepped: invar ?states-stepped
  using invar-step-n by blast

show ?case
proof(cases 4 < remaining-steps ?states)
  case True

```

```

obtain big-stepped small-stepped where stepped [simp]:
  ?states-stepped = States Left big-stepped small-stepped
  by (metis remaining-steps-states.cases step-n-same)

from True have remaining-steps: 0 < remaining-steps ?states-stepped
  using invar remaining-steps-n-steps-sub[of ?states 4]
  by simp

from True 6(1) have size-ok: size-ok ?states-stepped
  using
    step-4-push-small-size-ok[of Left big small x]
    remaining-steps-push-small[of Left big small x]
  by auto

from remaining-steps size-ok invar-stepped show ?thesis
  by(cases big-stepped; cases small-stepped)
  (auto simp: Let-def split!: common-state.split)
next
case False
then have remaining-steps-stepped: remaining-steps ?states-stepped = 0
  using invar by auto

then obtain small-current small-idle big-current big-idle where idle [simp]:
  ?states-stepped =
  States Left
  (Big2 (common-state.Idle big-current big-idle))
  (Small3 (common-state.Idle small-current small-idle))

using remaining-steps-idle' invar-stepped remaining-steps-stepped step-n-same
  by (smt (verit) invar-states.elims(2))

from 6 have [simp]: size-new (Small.push x small) = Suc (size-new small)
  using Small-Proof.size-new-push by auto

have [simp]: size small-idle = size-new (Small.push x small)
  using invar invar-stepped step-n-size-new-small[of Left big Small.push x small
4]
  by auto

then have [simp]: ¬is-empty small-idle
  using Idle-Proof.size-not-empty[of small-idle]
  by auto

have size-new-big [simp]: 0 < size-new big
  using 6
  by auto

then have [simp]: size big-idle = size-new big

```

```

    using invar invar-stepped step-n-size-new-big[of Left big Small.push x small
4]
    by auto

    then have [simp]: ¬is-empty big-idle
      using Idle-Proof.size-not-empty size-new-big
      by metis

    have size-ok-1: size small-idle ≤ 3 * size big-idle
      using 6 by auto

    have size-ok-2: size big-idle ≤ 3 * size small-idle
      using 6 by auto

    from False show ?thesis
      using invar-stepped size-ok-1 size-ok-2
      by auto
    qed
next
case (7 x big small)
let ?big = Big.push x big
let ?states = States Right ?big small
let ?states-stepped = (step4) ?states

from 7 have invar: invar ?states
  using invar-push-big[of Right big small x]
  by auto

then have invar-stepped: invar ?states-stepped
  using invar-step-n by blast

show ?case
proof(cases 4 < remaining-steps ?states)
  case True

  obtain big-stepped small-stepped where stepped [simp]:
    ?states-stepped = States Right big-stepped small-stepped
  by (metis remaining-steps-states.cases step-n-same)

  from True have remaining-steps: 0 < remaining-steps ?states-stepped
    using invar remaining-steps-n-steps-sub[of ?states 4]
    by simp

  from True 7(1) have size-ok: size-ok ?states-stepped
    using
      step-4-push-big-size-ok[of Right big small x]
      remaining-steps-push-big[of Right big small x]
    by auto

```

```

from remaining-steps size-ok invar-stepped show ?thesis
  by(cases big-stepped; cases small-stepped)
    (auto simp: Let-def split!: common-state.split)
next
case False
then have remaining-steps-stepped: remaining-steps ?states-stepped = 0
  using invar by auto

then obtain small-current small-idle big-current big-idle where idle [simp]:
  ?states-stepped =
  States Right
    (Big2 (common-state.Idle big-current big-idle))
    (Small3 (common-state.Idle small-current small-idle))

using remaining-steps-idle' invar-stepped remaining-steps-stepped step-n-same
  by (smt (verit) invar-states.elims(2))

from 7 have [simp]: size-new (Big.push x big) = Suc (size-new big)
  using Big-Proof.size-new-push by auto

have [simp]: size big-idle = size-new (Big.push x big)
  using invar invar-stepped step-n-size-new-big[of Right Big.push x big small
4]
  by auto

then have [simp]: ¬is-empty big-idle
  using Idle-Proof.size-not-empty[of big-idle]
  by auto

have size-new-small [simp]: 0 < size-new small
  using 7
  by auto

then have [simp]: size small-idle = size-new small
  using invar invar-stepped step-n-size-new-small[of Right Big.push x big small
4]
  by auto

then have [simp]: ¬is-empty small-idle
  using Idle-Proof.size-not-empty size-new-small
  by metis

have size-ok-1: size small-idle ≤ 3 * size big-idle
  using 7 by auto

have size-ok-2: size big-idle ≤ 3 * size small-idle
  using 7 by auto

from False show ?thesis

```

```

    using invar-stepped size-ok-1 size-ok-2
    by auto
  qed
qed auto

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)  $\implies$ 
  States-Aux.listL (States Left big small) = rev (States-Aux.listL (States Right
big small))
  by(auto split: prod.splits big-state.splits small-state.splits)

```

```

lemma swap-lists-right: invar (States Right big small)  $\implies$ 
  States-Aux.listL (States Right big small) = rev (States-Aux.listL (States Left
big small))
  by(auto split: prod.splits big-state.splits small-state.splits)

```

```

lemma swap-list [simp]: invar q  $\implies$  listR (swap q) = listL q
proof(induction q)
  case (Rebal states)
  then show ?case
    apply(cases states)
    using swap-lists-left swap-lists-right
    by (metis (full-types) RealTimeDeque-Aux.listL.simps(6) direction.exhaust in-
var-deque.simps(6) swap.simps(6) swap.simps(7))
qed auto

```

```

lemma swap-list': invar q  $\implies$  listL (swap q) = listR q
  using swap-list rev-swap
  by blast

```

```

lemma lists-same: lists (States Left big small) = lists (States Right big small)
  by(induction States Left big small rule: lists.induct) auto

```

```

lemma invar-swap: invar q  $\implies$  invar (swap q)
  by(induction q rule: swap.induct) (auto simp: lists-same split: prod.splits)

```

```

lemma listL-is-empty: invar deque  $\implies$  is-empty deque = (listL deque = [])
  using Idle-Proof.list-empty listL-remaining-steps
  by(cases deque) auto

```

```

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 (standard, goal-cases)
  case 1
  then show ?case
    by (simp add: empty-def)
next
  case 2
  then show ?case
    by(simp add: list-enqL)
next
  case (3 q x)

  then have listL (enqL x (swap q)) = x # listR q
    by (simp add: list-enqL invar-swap swap-list')

  with 3 show ?case
    by (simp add: invar-enqL invar-swap)
next
  case 4
  then show ?case
    using list-deqL by simp
next
  case (5 q)
  then have listL (deqL (swap q)) = tl (listR q)
    using 5 list-deqL swap-list' invar-swap by fastforce

  then have listR (swap (deqL (swap q))) = tl (listR q)
    using 5 swap-list' invar-deqL invar-swap listL-is-empty swap-list
    by metis

  then show ?case
    by(auto split: prod.splits)
next
  case 6
  then show ?case
    using list-firstL by simp
next
  case (7 q)

  from 7 have [simp]: listR q = listL (swap q)
    by (simp add: invar-swap swap-list')

```

```

from  $\gamma$  have [simp]: firstR q = firstL (swap q)
  by(auto split: prod.splits)

from  $\gamma$  have listL (swap q)  $\neq$  []
  by auto

with  $\gamma$  have firstL (swap q) = hd (listL (swap q))
  using invar-swap list-firstL by blast

then show ?case
  using  $\langle$ firstR q = firstL (swap q) $\rangle$  by auto
next
  case 8
  then show ?case
    using listL-is-empty by auto
next
  case 9
  then show ?case
    by (simp add: empty-def)
next
  case 10
  then show ?case
    by(simp add: invar-enqL)
next
  case 11
  then show ?case
    by (simp add: invar-enqL invar-swap)
next
  case 12
  then show ?case
    using invar-deqL by simp
next
  case (13 q)
  then have invar (swap (deqL (swap q)))
    by (metis invar-deqL invar-swap listL-is-empty rev.simps(1) swap-list)

  then show ?case
    by (auto split: prod.splits)
qed

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.