Real-Time Double-Ended Queue

Balazs Toth and Tobias Nipkow
Technical University of Munich

October 27, 2022

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 5
5 Idle 6
6 Common 7
7 Bigger End of Deque 11
8 Smaller End of Deque 14
9 Combining Big and Small 17
10 Real-Time Deque Implementation 19
11 Basic Lemma Library 25
12 Stack Proofs 26
13 Idle Proofs 28
14 Current Proofs 30
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:  

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

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

**assumes** \text{invar-empty}: 
\text{invar empty}

**assumes** \text{invar-enqL}: 
\text{invar } q \implies \text{invar (enqL } x \ q)

**assumes** \text{invar-enqR}: 
\text{invar } q \implies \text{invar (enqR } x \ q)

**assumes** \text{invar-deqL}: 
[\text{invar } q; \neg \text{is-empty } q] \implies \text{invar (deqL } q)

**assumes** \text{invar-deqR}: 
[\text{invar } q; \neg \text{is-empty } q] \implies \text{invar (deqR } q)

**begin**

**abbreviation** \text{listR } :: 'q \Rightarrow 'a list \text{ where}
\text{listR deque} \equiv \text{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. The function list
appends the two lists and is needed for the list abstraction of the deque.

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

definition empty where
  empty ≡ Stack [] []

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

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

instantiation stack ::(type) is-empty
begin

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

instance..
end

instantiation stack ::(type) size
begin

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

instance..

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 transformation phase
- the number of these newly added elements
- the originally contained elements
- the number of elements which will be contained after the transformation is finished.

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

Specification functions:

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

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

size: The number of the originally contained elements.

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

fun 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)

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

instantiation current::(type) is-empty
begin

fun is-empty-current :: 'a current ⇒ bool where
  is-empty (Current extra - old remained) ←→ is-empty old ∧ extra = [] ∨ remained = 0

instance.. end

instantiation current::(type) invar
begin

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

instance.. end

instantiation current::(type) size
begin

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

instance.. end

instantiation current::(type) size-new
begin

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

instance.. end

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 list :: 'a idle ⇒ 'a list where
  list (Idle stack _) = Stack.list stack

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))

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

6 Common

theory Common
imports Current Idle

begin

The last two phases of both deque ends during transformation:

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

*Idle*: The transformation of the deque end is finished.

Each phase contains a \textit{current} state, that holds the original elements of the deque end.

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

Functions:

\textit{push, pop}: Add and remove elements using the \textit{current} state.

\textit{list}: List abstraction of the elements which this end will contain after the transformation is finished

\textit{list-current}: List abstraction of the elements currently in this deque end.

\textit{step}: Executes one step of the transformation, while keeping the invariant.

\textit{remaining-steps}: Returns how many steps are left until the transformation is finished.

\textit{size-new}: Returns the size, that the deque end will have after the transformation is finished.

\textit{size}: Minimum of \textit{size-new} and the number of elements contained in the \textit{current} state.

definition reverseN where
  \[\text{simp}]: \text{reverseN n xs acc} \equiv \text{rev (take n xs) @ acc}\n
fun list :: 'a state \Rightarrow 'a list where
  list (Idle - idle) = Idle.list idle
  \text{list (Copy (Current extra - - remained) old new moved)}
  \quad= extra @ reverseN (remained - moved) old new

fun list-current :: 'a state \Rightarrow 'a list where
  list-current (Idle current -) = Current.list current
  \text{list-current (Copy current - - -)} = Current.list current
fun normalize :: 'a state ⇒ 'a 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  
  )
| normalize state = state

instantiation state ::(type) step begin

fun step-state :: 'a state ⇒ 'a 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 state ⇒ 'a state where
  push x (Idle current (idle.Idle stack stackSize)) =  
    Idle (Current.push x current) (idle.Idle (Stack.push stack) (Suc stackSize))  
| push x (Copy current aux new moved) = Copy (Current.push x current) aux new moved

fun pop :: 'a state ⇒ 'a * 'a 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))

instantiation state ::(type) is-empty begin

fun is-empty-state :: 'a state ⇒ bool where
  is-empty (Idle current idle) ←→ is-empty current ∨ is-empty idle  
| is-empty (Copy current - - -) ←→ is-empty current

instance..
end
instantiation state::(type) invar begin

fun invar-state :: 'a state ⇒ bool where
  invar (Idle current idle) ⟷
    invar idle
  ∧ invar current
  ∧ size-new current = size idle
  ∧ take (size idle) (Current.list current) =
    take (size current) (Idle.list idle)
| invar (Copy current aux new moved) ⟷ ( 
  case current of Current - - old remained ⇒
    moved < remained
  ∧ moved = length new
  ∧ remained ≤ length aux + moved
  ∧ invar current
  ∧ take remained (Stack.list old) = take (size old) (reverseN (remained − moved) aux new)
) instance.. end

instantiation state::(type) size begin

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

instantiation state::(type) size-new begin

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

instantiation state::(type) remaining-steps begin

fun remaining-steps-state :: 'a state ⇒ nat where
  remaining-steps (Idle - -) = 0
7 Bigger End of Deque

theory Big
imports Common
begin

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

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

```haskell
datatype (plugins del: size) 'a state =
    Reverse 'a current 'a stack 'a list nat
  | Common 'a Common.state
```

Functions:

- **step**: Executes one step of the transformation
- **size-new**: Returns the size that the deque end will have after the transfor-
  mation 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 transformation
  is finished.
- **list**: List abstraction of the elements which this end will contain after the
  transformation is finished
- **list-current**: List abstraction of the elements currently in this deque end.

```haskell
fun list :: 'a state ⇒ 'a list where
  list (Common common) = Common.list common
```
list (Reverse (Current extra - - remained) big aux count) = (  
  let reversed = reverseN count (Stack.list big) aux in  
  extra @ (reverseN remained reversed [])  
)  

fun list-current :: 'a state ⇒ 'a list where  
  list-current (Common common) = Common.list-current common  
  list-current (Reverse current - - -) = Current.list-current

instantiation state ::= (type) step  
begin  
  fun step-state :: 'a state ⇒ 'a state where  
    step (Common state) = Common (step state)  
    | step (Reverse current - aux 0) = Common (normalize (Copy current aux [] 0))  
    | step (Reverse current big aux count) =  
      Reverse current (Stack.pop big) ((Stack.first big)#aux) (count - 1)  
  
instance..
end

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

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

instantiation state ::= (type) is-empty  
begin  
  fun is-empty-state :: 'a state ⇒ bool where  
    is-empty (Common state) = is-empty state  
    | is-empty (Reverse current - - count) = (  
      case current of Current extra added old remained ⇒  
        is-empty current ∨ remained ≤ count  
    )  
  
instance..
end

instantiation state ::= (type) invar  
begin  
  fun invar-state :: 'a state ⇒ bool where  
    invar (Common state) ←→ invar state

12
invar (Reverse current big aux count) \leftrightarrow ( 
  case current of Current extra added old remained \Rightarrow 
  invar current 
  \land \text{List.length aux} \geq \text{remained} - \text{count} 
  \land \text{count} \leq \text{size big} 
  \land \text{Stack.list old} = \text{rev} \left( \text{take} \left( \text{size old} \right) (\text{rev} \left( \text{Stack.list big} \right)) \odot \text{aux} \right) 
  \land \text{take remained} \left( \text{Stack.list old} \right) = \text{rev} \left( \text{take remained} \left( \text{reverseN count} \left( \text{Stack.list big} \right) \odot \text{aux} \right) \right) 
)

instance.. 
end

initialization state \:::(\text{type}) \text{size}
begin

fun size-state :: 'a state \Rightarrow \text{nat} \text{ where}
  size (\text{Common state}) = \text{size state}
  \mid \text{size} (\text{Reverse current - - -}) = \min \left( \text{size current} \right) \left( \text{size-new current} \right)

instance.. 
end

initialization state ::(\text{type}) \text{size-new}
begin

fun size-new-state :: 'a state \Rightarrow \text{nat} \text{ where}
  size-new (\text{Common state}) = \text{size-new state}
  \mid \text{size-new} (\text{Reverse current - - -}) = \text{size-new current}

instance.. 
end

initialization state ::(\text{type}) \text{remaining-steps}
begin

fun remaining-steps-state :: 'a state \Rightarrow \text{nat} \text{ where}
  remaining-steps (\text{Common state}) = \text{remaining-steps state}
  \mid \text{remaining-steps} (\text{Reverse (Current - - - remaining) - - count}) = \text{count} + \text{remaining} + 1

instance.. 
end

end
8 Smaller End of Deque

theory Small
imports Common
begin

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

*Reverse1*: Using the \textit{step} function the originally contained elements are reversed.

*Reverse2*: Using the \textit{step} function the newly obtained elements from the bigger end are reversed on top of the ones reversed in the previous phase.

*Common*: See theory \textit{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 \textit{current} state, which holds the original elements of the deque end.

\textbf{datatype} (plugins del: size) 'a state =
Reverse1 'a current 'a stack 'a list
| Reverse2 'a current 'a list 'a stack 'a list nat
| Common 'a Common.state

Functions:

\textit{push, pop}: Add and remove elements using the \textit{current} state.

\textit{step}: Executes one step of the transformation, while keeping the invariant.

\textit{size-new}: Returns the size, that the deque end will have after the transformation is finished.

\textit{size}: Minimum of \textit{size-new} and the number of elements contained in the \textit{current} state.

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

\textit{list-current}: List abstraction of the elements currently in this deque end.

\textbf{fun} list :: 'a state ⇒ 'a list \textbf{where}
list (Common common) = Common.list common
| list (Reverse2 (Current extra - - remained) aux big new count) =
extra @ reverseN (remained - (count + size big)) aux (rev (Stack.list big) @
new)

fun list-current :: 'a state ⇒ 'a list where
  list-current (Common common) = Common.list-current common
| list-current (Reverse2 current - - - ) = Current.list current
| list-current (Reverse1 current - - ) = Current.list current

instantiation state::(type) step
begin

fun step-state :: 'a state ⇒ 'a state where
  step (Common state) = Common (step state)
| step (Reverse1 current small auxS) = (if is-empty small
     then Reverse1 current small auxS
     else Reverse1 current (Stack.pop small) ((Stack.first small)#auxS)
     )
| step (Reverse2 current auxS big newS count) = (if is-empty big
     then Common (normalize (Copy current auxS newS count))
     else Reverse2 current auxS (Stack.pop big) ((Stack.first big)#newS) (count +
1)
   )

instance..
end

fun push :: 'a ⇒ 'a state ⇒ 'a state where
  push x (Common state) = Common (Common.push x state)
| push x (Reverse1 current small auxS) = Reverse1 (Current.push x current) small
   auxS
| push x (Reverse2 current auxS big newS count) = Reverse2 (Current.push x current) auxS big newS count

fun pop :: 'a state ⇒ 'a * 'a state where
  pop (Common state) = (let (x, state) = Common.pop state
    in (x, Common state)
  )
| pop (Reverse1 current small auxS) =
    (first current, Reverse1 (drop-first current) small auxS)
| pop (Reverse2 current auxS big newS count) =
    (first current, Reverse2 (drop-first current) auxS big newS count)

instantiation state::(type) is-empty
begin

fun is-empty-state :: 'a state ⇒ bool where
is-empty (Common state) = is-empty state
| is-empty (Reverse1 current - -) = is-empty current
| is-empty (Reverse2 current - - - -) = is-empty current

instance..
end

instantiation state::(type) invar
begin

fun invar-state :: 'a state ⇒ bool where
invar (Common state) = invar state
| invar (Reverse2 current auxS big newS count) = (case current of Current - - old remained ⇒ remained = count + size big + size old ∧ remained ≥ size old ∧ count = List.length newS ∧ invar current ∧ List.length auxS ≥ size old ∧ Stack.list old = rev (take (size old) auxS))
| invar (Reverse1 current small auxS) = (case current of Current - - old remained ⇒ invar current ∧ remained ≥ size old ∧ size small + List.length auxS ≥ size old ∧ Stack.list old = rev (take (size old) (rev (Stack.list small) @ auxS)))

instance..
end

instantiation state::(type) size
begin

fun size-state :: 'a state ⇒ nat where
size (Common state) = size state
| size (Reverse2 current - - - -) = min (size current) (size-new current)
| size (Reverse1 current - -) = min (size current) (size-new current)

instance..
end

instantiation state::(type) size-new
begin

fun size-new-state :: 'a state ⇒ nat where
size-new (Common state) = size-new state
| size-new (Reverse2 current - - - -) = size-new current
| size-new (Reverse1 current - -) = size-new current

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

instatiation states::(type) step
begin

fun step-states :: 'a states ⇒ 'a states where
  step (States dir (Reverse currentB big auxB 0) (Reverse1 currentS - auxS)) =
    States dir (step (Reverse currentB big auxB 0)) (Reverse2 currentS auxS big [] 0)
  | step (States dir left right) = States dir (step left) (step right)

instance..
end

instatiation 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
      Small.Common common ⇒ remaining-steps common
        | Reverse2 (Current - - remaining) - big - count ⇒ (remaining - (count + size big)) + size big + 1
        | Reverse1 (Current - - remaining) - - ⇒
          case big of
            Reverse currentB big auxB count ⇒ size big + (remaining + count - size big) + 2
    )

instance..
end

fun lists :: 'a states ⇒ 'a list * 'a list where
lists (States - (Reverse currentB big auxB count) (Reverse1 currentS small auxS)) = 
  Big.list (Reverse currentB big auxB count),
  Small.list (Reverse2 currentS (reverseN count (Stack.list small) auxS) ((Stack.pop ~ count) big) [] 0)
  
| lists (States - big small) = (Big.list big, Small.list small)

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

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

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

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

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

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

instantiation states::(type) invar
begin

fun invar-states :: 'a states ⇒ bool where
  invar (States dir big small) ←→ (invar big
  ∧ invar small
  ∧ list-small-first (States dir big small) = list-current-small-first (States dir big small)
  ∧ (case (big, small) of
    (Reverse - big - count, Reverse1 (Current - - old remained) small -) ⇒ size big - count = remained - size old ∧ count ≥ size small
    | (Reverse1 - - -) ⇒ False
    | (Reverse - - - -) ⇒ 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)

instantiation states::(type) is-empty
begin
fun is-empty-states : 'a states ⇒ bool where
is-empty (States - big small) ←→ is-empty big ∨ is-empty small
instance..
end

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

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.
**Idle:** Deque with a left and a right end, fulfilling the following invariant:

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

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

- The transformation is not done yet. The deque needs to be in idle state otherwise.
- The transformation 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 transformation the invariant of the idle 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 transformation is started or if there are 3 or less elements left the respective states are used. On transformation start, six steps are executed initially. During transformation 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 transformation with six initial steps is started. During transformation 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
  | Idle 'a idle 'a idle
  | Transforming 'a states

definition empty where
    empty ≡ Empty

instantiation deque::(type) is-empty
begin

fun is-empty-deque :: 'a deque ⇒ bool where
    is-empty-deque Empty = True
  | is-empty-deque - = False

instance
end

fun swap :: 'a deque ⇒ 'a deque where
    swap Empty = Empty
  | swap (One x) = One x
  | swap (Two x y) = Two y x
  | swap (Three x y z) = Three z y x
  | swap (Idle left right) = Idle right left
  | swap (Transforming (States Left big small)) = (Transforming (States Right big small))
  | swap (Transforming (States Right big small)) = (Transforming (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#[]) (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′ (Idle 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, Idle (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 = Reverse1 (Current [] 0 left length-left') left [] in let big = Reverse (Current [] 0 right length-right') right [] length-right' in let states = States Left big small in let states = (step^^6) states in (x, Transforming states)
else case right of Stack r1 r2 ⇒ (x, small-deque r1 r2)
)
deqL′ (Transforming (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
  (Big.Common (Common.Idle - big))
  (Small.Common (Common.Idle - small)) ⇒ (x, Idle small big)
  | - ⇒ (x, Transforming states)
)
deqL′ (Transforming (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
  (Big.Common (Common.Idle - big))
  (Small.Common (Common.Idle - small)) ⇒ (x, Idle big small)
  | - ⇒ (x, Transforming 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) = Idle (idle.Idle (Stack [x, a] [])) 2) (idle.Idle (Stack [c, b] [])) 2)
| enqL x (Idle left (idle.Idle right length-right)) = (case Idle.push x left of idle.Idle left length-left ⇒
  if 3 * length-right ≥ length-left
  then
  Idle (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 = Reverse (Current [] 0 left length-left) left [] length-left in
  let small = Reverse1 (Current [] 0 right length-right) right [] in

  let states = States Right big small in
  let states = (step^^6) states in
  Transforming states
)| enqL x (Transforming (States Left big small)) = (let small = Small.push x small in
  let states = (step^^4) (States Left big small) in
  case states of
  States Left
  (Big.Common (Common.Idle - big))
  (Small.Common (Common.Idle - small))
  ⇒ Idle small big
  | ⇒ Transforming states
)
| enqL x (Transforming (States Right big small)) = (let big = Big.push x big in
  let states = (step^^4) (States Right big small) in
  case states of
  |
States Right

\[(\text{Big.Common (Common.Idle - big)})\]
\[(\text{Small.Common (Common.Idle - small)})\]
\[\Rightarrow \text{Idle big small}\]
\[| - \Rightarrow \text{Transforming states}\]

\begin{verbatim}
fun enqR :: 'a ⇒ 'a deque ⇒ 'a deque where
  enqR x deque = (let deque = enqL x (swap deque)
                     in swap deque)

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 (Idle left right) = Idle.list left ⊕ (rev (Idle.list right))
  listL (Transforming states) = States.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 (Idle left right) ←→
    invar left ∧
    invar right ∧
    ¬ is-empty left ∧
    ¬ is-empty right ∧
    3 * size right ≥ size left ∧
    3 * size left ≥ size right
  invar (Transforming states) ←→
    invar states ∧
    size-ok states ∧
    0 < remaining-steps states

instance..
end
\end{verbatim}
11 Basic Lemma Library

theory RTD-Util
imports Main
begin

lemma tl-append-if: tl (xs @ ys) = (if xs = [] then tl ys else tl xs @ ys)
  by (simp)

lemma take-last-length: [take (Suc 0) (rev xs) = [last xs]] \implies Suc 0 \leq length xs
  by (induction xs) auto

lemma take-last: xs \neq [] \implies take 1 (rev xs) = [last xs]
  by (induction xs) simp: take-last-length

lemma take-hd: [simp]: xs \neq [] \implies take (size xs) (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 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 \land 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))
\[
0)) = \min x y
\]
by auto

**Lemma rev-tl-hd**: \(x : \not = [] \Rightarrow \text{rev} (\text{tl} \; x) @ \text{hd} \; x = \text{rev} \; x\)
by (simp add: rev-app-single)

**Lemma app-rev**: \(\text{as} @ \text{rev} \; \text{bs} = \text{cs} @ \text{rev} \; \text{ds} \Rightarrow \text{bs} @ \text{rev} \; \text{as} = \text{ds} @ \text{rev} \; \text{cs}\)
by (metis rev-append rev-rev-ident)

**Lemma tl-drop-2**: \(\text{tl} \; (\text{drop} \; n \; x) = \text{drop} \; (\text{Suc} \; n) \; x\)
by (simp add: drop-Suc tl-drop)

**Lemma Suc-sub**: \(\text{Suc} \; n = m \Rightarrow n = m - 1\)
by simp

**Lemma length-one-hd**: \(\text{length} \; x = 1 \Rightarrow x = [\text{hd} \; x]\)
by (induction x) auto

end

## 12 Stack Proofs

theory Stack-Proof
imports Stack RTD-Util
begin

**Lemma push-list** [simp]: \(\text{list} \; (\text{push} \; x \; \text{stack}) = x \# \text{list} \; \text{stack}\)
by (cases stack) auto

**Lemma pop-list** [simp]: \(\neg \text{is-empty} \; \text{stack} \Rightarrow \text{list} \; (\text{pop} \; \text{stack}) = \text{tl} \; (\text{list} \; \text{stack})\)
by (induction stack rule: pop.induct) auto

**Lemma first-list** [simp]: \(\neg \text{is-empty} \; \text{stack} \Rightarrow \text{first} \; \text{stack} = \text{hd} \; (\text{list} \; \text{stack})\)
by (induction stack rule: first.induct) auto

**Lemma list-empty**: \(\text{list} \; \text{stack} = [] \Leftrightarrow \text{is-empty} \; \text{stack}\)
by (induction stack rule: is-empty-stack.induct) auto

**Lemma list-not-empty**: \(\text{list} \; \text{stack} \not = [] \Leftrightarrow \neg \text{is-empty} \; \text{stack}\)
by (induction stack rule: is-empty-stack.induct) auto

**Lemma list-empty-2** [simp]: \([\text{list} \; \text{stack} \not = [] ; \; \text{is-empty} \; \text{stack}] \Rightarrow \text{False}\)
by (simp add: list-empty)

**Lemma list-not-empty-2** [simp]: \([\text{list} \; \text{stack} = [] ; \; \neg \; \text{is-empty} \; \text{stack}] \Rightarrow \text{False}\)
by (simp add: list-empty)

**Lemma list-empty-size**: \(\text{list} \; \text{stack} = [] \Leftrightarrow \text{size} \; \text{stack} = 0\)
by (induction stack) auto
lemma list-not-empty-size: list stack \neq [] \iff 0 < \text{size} stack
by (induction stack) auto

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

lemma list-not-empty-size-2 [simp]: [list stack = []; 0 < size stack] \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 \iff \text{is-empty} stack
by (induction stack rule: is-empty-stack.induct) auto

lemma size-not-empty: size (stack :: 'a stack) > 0 \iff \neg \text{is-empty} stack
by (induction stack rule: is-empty-stack.induct) auto

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

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

lemma size-list-length [simp]: length (list stack) = size stack
by (cases stack) auto

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

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

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

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

lemma first-take-tl [simp]: 0 < size big
\[ \leading{\text{Lemma}} \] \text{first-take-pop [simp]:} [\neg \text{is-empty stack}; 0 < x]  
\[ \dots \implies \text{first stack} \not\# (x - \text{Suc } 0) \text{ (list (pop stack))} = \text{take } x \text{ (list stack)} \]
\[ \text{by (induction stack rule: pop.induct) (auto simp: take-Cons')} \]

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

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

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

\[ \text{lemma popN-drop [simp]:} \text{list ((pop} \cdots \text{n stack)} = \text{drop n (list stack)} \]
\[ \text{by (induction n) (auto simp: drop-Suc tl-drop)} \]

\[ \text{Lemma popN-size [simp]:} \text{size ((pop} \cdots \text{n stack)} = \text{size stack} - n \]
\[ \text{by (induction n) auto} \]

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

\[ \text{end} \]

13 Idle Proofs

\[ \text{theory Idle-Proof} \]
\[ \text{imports Idle Stack-Proof} \]
\[ \text{begin} \]

\[ \text{Lemma push-list [simp]: list (push x idle)} = x \not\# \text{ list idle} \]
\[ \text{by (induction idle arbitrary: x) auto} \]

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

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

28
lemma pop-list-tl' \[\text{simp}]: [\text{pop idle} = (x, \text{idle}')] \implies \text{list idle}' = \text{tl (list idle)}
  by(induction idle arbitrary: x)(auto simp: drop-Suc)

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

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

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

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

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

lemma size-empty \[\text{simp}]: \text{size idle} = 0 \iff \text{is-empty \text{idle} :: 'a idle)}
  by(induction idle)(auto simp: size-empty)

lemma size-not-empty \[\text{simp}]: 0 < \text{size idle} \iff \neg \text{is-empty \text{idle} :: 'a idle)}
  by(induction idle)(auto simp: size-not-empty)

lemma size-empty-2 \[\text{simp}]: \text{list idle} = []; \neg \text{is-empty \text{idle} :: 'a idle)}
  by (simp add: size-empty)

lemma size-not-empty-2 \[\text{simp}]: \text{list idle} \neq []; \text{is-empty \text{idle} :: 'a idle)}
  by (simp add: size-not-empty)

lemma list-empty \[\text{simp}]: \text{list idle} = [] \iff \text{is-empty idle}
  by(induction idle)(simp add: list-empty)

lemma list-not-empty \[\text{simp}]: \text{list idle} \neq [] \iff \neg \text{is-empty idle}
  by(induction idle)(simp add: list-not-empty)

lemma list-empty-2 \[\text{simp}]: \text{list idle} = []; \neg \text{is-empty \text{idle} :: 'a idle)}
  by list-empty by blast

lemma list-not-empty-2 \[\text{simp}]: \text{list idle} \neq []; \text{is-empty \text{idle} :: 'a idle)}
  by list-not-empty by blast

lemma list-empty-size \[\text{simp}]: \text{list idle} = [] \iff \text{0 = size idle}
  by (simp add: list-empty size-empty)

lemma list-not-empty-size \[\text{simp}]: \text{list idle} \neq [] \iff \text{0 < size idle}
  by list-not-empty-size

29
by (simp add: list-empty-size)

lemma list-empty-size-2 [simp]: \([\text{list idle} \neq []]; \ 0 = \text{size idle} \) \implies \text{False}
by (simp add: list-empty size-empty)

lemma list-not-empty-size-2 [simp]: \([\text{list idle} = []]; \ 0 < \text{size idle} \) \implies \text{False}
by (simp add: list-empty-size)

end

14 Current Proofs

theory Current-Proof
imports Current Stack-Proof
begin

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

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

lemma pop-list-size: \([\text{invar current}; \ 0 < \text{size current}; \ \text{pop current} = (x, \text{current}')] \)
\implies \(x \# \text{list current}' = \text{list current} \)
by (induction current arbitrary: x rule: pop.induct)(auto simp: size-not-empty list-not-empty)

lemma drop-first-list [simp]:
\(\neg \text{is-empty current} \implies \text{list (drop-first current)} = \text{tl (list current)} \)
by (induction current rule: pop.induct)(auto simp: drop-Suc)

lemma pop-list-2 [simp]: \([0 < \text{size current}; \ \text{invar current}] \implies \text{fst (pop current)} \# \text{tl (list current)} = \text{list current} \)
by (induction current rule: pop.induct)(auto simp: size-not-empty list-not-empty)

lemma drop-first-list-size [simp]: \([\text{invar current}; \ 0 < \text{size current}] \)
\implies \(\text{list (drop-first current)} = \text{tl (list current)} \)
by (induction current rule: pop.induct)(auto simp: drop-Suc)

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

lemma invar-pop: \([\neg \text{is-empty current}; \ \text{invar current}; \ \text{pop current} = (x, \text{current}')] \)
\implies \text{invar current'}
by (induction current arbitrary: x rule: pop.induct) auto

lemma invar-size-pop: \([0 < \text{size current}; \ \text{invar current}; \ \text{pop current} = (x, \text{cur-}
\[
\begin{align*}
\text{lemma } \text{invar-size-drop-first}: [0 < \text{size current}; \text{invar current}] & \implies \text{invar (drop-first current)} \\
\text{using } \text{invar-size-pop} \\
\text{by } (\text{induction current arbitrary: } x \text{ rule: pop.induct}) \text{ auto}
\end{align*}
\]

\[
\begin{align*}
\text{lemma } \text{invar-drop-first}: [\neg \text{is-empty current}; \text{invar current}] & \implies \text{invar (drop-first current)} \\
\text{by } (\text{induction current rule: pop.induct}) \text{ auto}
\end{align*}
\]

\[
\begin{align*}
\text{lemma } \text{push-not-empty } [\text{simp}]: [\neg \text{is-empty current}; \text{is-empty (push x current)}] & \implies \text{False} \\
\text{by } (\text{induction x current rule: push.induct}) \text{ auto}
\end{align*}
\]

\[
\begin{align*}
\text{lemma } \text{size-empty}: \text{invar (current :: 'a current)} & \implies \text{size current} = 0 \implies \text{is-empty current} \\
\text{by } (\text{induction current}) (\text{auto simp: size-empty})
\end{align*}
\]

\[
\begin{align*}
\text{lemma } \text{size-new-empty}: \text{invar (current :: 'a current)} & \implies \text{size-new current} = 0 \implies \text{is-empty current} \\
\text{by } (\text{induction current}) (\text{auto simp: size-empty})
\end{align*}
\]

\[
\begin{align*}
\text{lemma } \text{list-not-empty } [\text{simp}]: [\text{list current} = []; \neg \text{is-empty current}] & \implies \text{False} \\
\text{by } (\text{induction current}) (\text{auto simp: list-empty})
\end{align*}
\]

\[
\begin{align*}
\text{lemma } \text{list-size } [\text{simp}]: [\text{invar current}; \text{list current} = []; 0 < \text{size current}] & \implies \text{False} \\
\text{by } (\text{induction current}) (\text{auto simp: size-not-empty list-empty})
\end{align*}
\]

\[
\begin{align*}
\text{lemma } \text{size-new-push } [\text{simp}]: \text{invar current} & \implies \text{size-new (push x current)} = \text{Suc (size-new current)} \\
\text{by } (\text{induction x current rule: push.induct}) \text{ auto}
\end{align*}
\]

\[
\begin{align*}
\text{lemma } \text{size-push } [\text{simp}]: \text{size (push x current)} & = \text{Suc (size current)} \\
\text{by } (\text{induction x current rule: push.induct}) \text{ auto}
\end{align*}
\]

\[
\begin{align*}
\text{lemma } \text{size-new-pop } [\text{simp}]: [0 < \text{size-new current}; \text{invar current}] & \implies \text{Suc (size-new (drop-first current)) = size-new current} \\
\text{by } (\text{induction current rule: pop.induct}) \text{ auto}
\end{align*}
\]

\[
\begin{align*}
\text{lemma } \text{size-pop } [\text{simp}]: [0 < \text{size current}; \text{invar current}] & \implies \text{Suc (size (drop-first current)) = size current} \\
\text{by } (\text{induction current rule: pop.induct}) \text{ auto}
\end{align*}
\]

\[
\begin{align*}
\text{lemma } \text{size-pop-suc } [\text{simp}]: [0 < \text{size current}; \text{invar current}; \text{pop current} = (x, \\
\end{align*}
\]
\[ current' \] 
\[ \Rightarrow Suc (size current') = size current \]
by (induction current rule: pop.induct) auto

lemma size-pop-sub: \[ 0 < size current; invar current; pop current = (x, current') \]
\[ \Rightarrow size current' = size current - 1 \]
by (induction current rule: pop.induct) auto

lemma size-drop-first-sub: \[ 0 < size current; invar current \]
\[ \Rightarrow size (drop-first current) = size current - 1 \]
by (induction current rule: pop.induct) auto

end

15 Common Proofs

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

lemma reverseN-drop: reverseN n xs acc = drop (length xs - n) (rev xs) @ acc
unfolding reverseN-def using rev-take by blast

lemma reverseN-step: xs \[ \neq \] [] \[ \Rightarrow \] reverseN n (tl xs) (hd xs # acc) = reverseN \mbox{(Suc n)} xs acc
by (simp add: take-Suc)

lemma reverseN-finish: reverseN n [] acc = acc
by (simp)

lemma reverseN-tl-hd: 0 < n \[ \Rightarrow \] xs \[ \neq \] [] \[ \Rightarrow \] reverseN n xs ys = reverseN (n - (Suc 0)) (tl xs) (hd xs # ys)
by (simp add: reverseN-step del: reverseN-def)

lemma reverseN-nth: n < length xs \[ \Rightarrow \] x = xs ! n \[ \Rightarrow \] x \# reverseN n xs ys = reverseN (Suc n) xs ys
by (simp add: take-Suc-conv-app-nth)

lemma step-list [simp]: invar common \[ \Rightarrow \] list (step common) = list common
proof (induction common rule: step-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 ≠ []
   by auto

from 2 Current show ?thesis
proof (cases remained ≤ 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 simp:)

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

lemma step-list-current [simp]: invar common ⇒ 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 state) ⇒ invar (step common)
proof (induction common rule: invar-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 = [])

33
case True
  with 2 Current show ?thesis by auto
next
case False
  note AUX-NOT-EMPTY = False
then show ?thesis
proof(cases remained ≤ Suc (length new))
  case True
    with 2 Current False
    have take (Suc (length new)) (Stack.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: reverseN-step Suc-diff-Suc simp del: reverseN-def)
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: []
  \theta < size common;
  invar common;
\[\text{pop common} = (x, \text{common'})\]
\[\implies \text{invar common'}\]

**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-size-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 \(\leq\) length new)

**case** True

with 1 have [simp]:

0 < size old

Stack.list old \(\neq\) []

aux \(\neq\) []

length new = remained - Suc 0

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

then have [simp]: Suc 0 \(\leq\) size old

by linarith

**from** 1 have 0 < remained

by auto

then have take remained (Stack.list old)

= hd (Stack.list old) \# take (remained - Suc 0) (tl (Stack.list old))

by (metis Suc-pred \(\langle\) Stack.list old \(\neq\) []\(\rangle\), 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 \(\leq\) 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

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 ⇒ 0 < size common ⇒ pop common =
  (x, common') ⇒
  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 ≤ length new)
case True

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

then have take remained (Stack.list old) = hd aux ≠ take (size old − Suc 0) new
  ⇒ 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

36
from 1 have not-empty: 0 < remained 0 < size old aux ≠ [] ¬ is-empty old
  by(auto simp: Stack-Proof.size-not-empty)

from 1 True not-empty have take 1 (Stack.list old) = take 1 (rev aux)
  using take-1[of remained size old Stack.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: reverseN-drop length-minus-1 simp del: reverseN-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.list old) = take (size old) (reverseN (remained - length new) aux new)
      ⇒ take (Suc 0) (Stack.list old) = take (Suc 0) (rev (take (remained - length new) aux))
  using take-1[of remained size old Stack.list old]
(reverseN (remained - length new) aux new)
by (auto simp: not-empty Suc-le-eq)

with 1 False have take 1 (Stack.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

next
case 2
then show ?case by auto
qed

lemma pop-list-current: invar common ⇒ 0 < size common ⇒ pop common = (x, common')
  ⇒ 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: ¬ is-empty stack Stack.list stack ≠ []
  by auto

with 2 have hd (Stack.list stack) = x
  using take-hd[of Stack.list stack x xs @ Stack.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 ¬ 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]:
  [0 < size common; list-current common = []; invar common] ⇒ False
proof(induction common rule: invar-state.induct)
case 1
  then show ?case
    using list-size by auto
next
case (2 current)
  then have invar current
    Current.list current = []
    0 < size current
    by(auto split: current.splits)
then show ?case using list-size by auto
qed

lemma list-size [simp]: [0 < size common; list common = []; invar common] ⇒ False
proof (induction common rule: invar-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.list current = []
  0 < size current
  by (auto split: current.splits)
then show ?case using list-size by auto
qed

lemma size-empty: invar (common :: 'a state) \implies size common = 0 \implies is-empty common
proof (induction common rule: is-empty-state.induct)
case 1
then show ?case
  by (auto simp: min-def size-empty size-new-empty split: if-splits)
next
case (2 current)
then have invar current
  by (auto split: current.splits)
with 2 show ?case
  by (auto simp: min-def size-empty size-new-empty split: if-splits)
qed

lemma step-size [simp]: invar (common :: 'a state) \implies size (step common) = size common
proof (induction common rule: step-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 state) \implies size-new (step common) = size-new common
proof (induction common rule: step-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]: \[\text{invar (common :: 'a state); remaining-steps common > 0} \implies \text{Suc (remaining-steps (step common)) = remaining-steps common} \]
by (induction common) (auto split: current.splits)

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

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

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

lemma remaining-steps-pop [simp]: \[\text{invar common; 0 < size common; pop common = (x, common')} \]
\[\implies \text{remaining-steps common'} \leq \text{remaining-steps common} \]
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]: \text{invar common} \implies \text{size (push x common) = Suc (size common)} \]
by (induction x common rule: push.induct) (auto split: current.splits)

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

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

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

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

end

16 Big Proofs

theory Big-Proof
imports Big Common-Proof
begin

lemma step-list \([\text{simp}]\): \text{invar big} \implies \text{list (step big)} = \text{list big}
proof(induction big rule: step-state.induct)
  case 1
  then show \(?\text{case}\)
    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]: invar big \implies list-current (step big) = list-current big
  by (induction big rule: step-state.induct)(auto split: current.splits)

lemma push-list [simp]: list (push x big) = x # 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-Reverse: 
  \[ 0 < \text{size (Reverse current big aux count)}; \]
  \[ \text{invar (Reverse current big aux count)}; \]
  \[ \implies \text{first current} \neq \text{list (Reverse (drop-first current) big aux count)} = \]
  \[ \text{list (Reverse current big aux count)}; \]
proof (induction current rule: Current.pop.induct)
  case (1 added old remained)
  then have [simp]: remained − Suc 0 < length (reverseN count (Stack.list big) aux)
    by (auto simp: le-diff-conv)

then have
  \[ [0 < \text{size old}; 0 < \text{remained}; \text{added} = 0; \text{remained} − \text{count} \leq \text{length aux}; \text{count} \]
  \[ \leq \text{size big}; \]
  \[ \text{Stack.list old} = \]
  \[ \text{rev (take (size old − size big) aux) @ rev (take (size old) (rev (Stack.list big))}); \]
  \[ \text{take remained} \ (\text{rev (take (size old − size big) aux)) @} \]
  \[ \text{take (remained} − \text{min (length aux) (size old − size big))} \]
  \[ \text{(rev (take (size old) (rev (Stack.list big))})) = \]
  \[ \text{rev (take (remained − count) aux) @ rev (take remained} \ (\text{rev (take count} \]
  \[ \text{Stack.list big))}); \]
  \[ \implies \text{hd (rev (take (size old − size big) aux) @ rev (take (size old) (rev (Stack.list big))})} = \]
  \[ \text{(rev (take count (Stack.list big)) @ aux)) ! (remained − Suc 0)} \]
  by (smt (verit) Suc-pred hd-drop-conv-nth hd-rev hd-take last-snoc length-rev
length-take min.absorb rev-append reverseN-def size-list-length take-append take-hd-drop

with 1 have simp: Stack.first old = reverseN count (Stack.list big) aux !
  (remained = Suc 0)
  by(auto simp: take-hd-drop first-hd)

from 1 show ?case
  using reverseN-nth[of
    remained = Suc 0 reverseN count (Stack.list big) aux Stack.first old []
  ] by auto
next
  case 2
  then show ?case by auto
qed

lemma size-list simp: 
  \[ \begin{align*}
    0 < \text{size big; invar big; list big } & = [] \implies \text{False}
    \end{align*} \]
proof(induction big arbitrary: x 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-Reverse)
qed

lemma pop-list simp: 
  \[ \begin{align*}
    0 < \text{size big; invar big; Big.pop big } & = (x, \text{big}') \implies \text{x # list big'} = \text{tl list big}
    \end{align*} \]
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-Reverse prod.inject)
qed

lemma pop-list-tl: 
  \[ \begin{align*}
    0 < \text{size big; invar big; Big.pop big } & = (x, \text{big}') \implies \text{Big.list big'} = \text{tl (Big.list big)}
    \end{align*} \]
using pop-list cons-tl[of x list big' list big]
by force

lemma invar-step: invar (big :: 'a state) \implies invar (step big)
proof(induction big rule: step-state.induct)
  case 1
  then show ?case

44
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 \(\text{current = Current extra (length extra) old remained; remained} \leq \text{length aux}\):

  Stack.list old =
  rev (take (size old - size big) aux) @ rev (take (size old) (rev (Stack.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.list big))));
  rev (take remained aux);

  \(\Rightarrow\) remained \leq 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: reverseN-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.list big \# []
by (auto simp: Stack-Proof.size-not-empty Stack-Proof.list-not-empty)

with 3 have a:
  rev (Stack.list big) @ aux =
  rev (Stack.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
  reverseN (Suc count) (Stack.list big) aux =
  reverseN count (Stack.list (Stack.pop big)) (Stack.first big \# aux)
using reverseN-tl-hd[of Suc count Stack.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 \(\implies\) invar (push x big)
by (induction x big rule: push.induct) (auto simp: invar-push split: current.splits)

**lemma** invar-pop: 
\[\begin{align*}
0 < \text{size big} \\
\text{invar big} \\
\text{pop big} = (x, \text{big'}) \\
\end{align*}\] \(\implies\) invar big'
proof (induction big arbitrary; x rule: pop.induct)
\text{case (1 state)}
then show ?case by (auto simp: invar-pop split: prod.splits)
next
\text{case (2 current big aux count)}
then show ?case by (auto simp: invar-pop split: prod.splits)

have \(x y z. x - y \leq z \implies x - (Suc y) \leq z\)
by linarith

have a: \[\begin{align*}
\text{remained} \leq \text{count} + \text{length aux}; 
0 < \text{remained}; 
\text{added} = 0; 
x = \text{Stack.first old}; 
\text{big'} = \text{Reverse (Current [] 0 (Stack.pop old) (remained - Suc 0)) big aux count}; 
\text{count} \leq \text{size big}; 
\text{Stack.list old} = \text{rev aux @ Stack.list big}; 
\text{take remained (rev aux) @ take (remained - length aux) (Stack.list big)} = 
\text{drop (count + length aux - remained) (rev aux) @}
\text{drop (count - remained) (take count (Stack.list big))}; 
\neg \text{size old} \leq \text{length aux} + \text{size big} \\
\implies \text{tl (rev aux @ Stack.list big)} = \text{rev aux @ Stack.list big} \\
by \text{metis le-refl length-append length-rev size-list-length}
\]

have b: \[\begin{align*}
\text{remained} \leq \text{length} (\text{reverseN count (Stack.list big) aux}); 
0 < \text{size old}; 
\text{0 < remained}; 
\text{added} = 0; 
x = \text{Stack.first old}; 
\text{big'} = \text{Reverse (Current [] 0 (Stack.pop old) (remained - Suc 0)) big aux count}; 
\text{remained - count} \leq \text{length aux}; 
\text{count} \leq \text{size big}; 
\text{Stack.list old} = 
\text{drop (length aux - (size old - size big)) (rev aux) @}
\text{drop (size big - size old) (Stack.list big)}; 
\text{take remained (drop (length aux - (size old - size big)) (rev aux)) @}
\text{take (remained + (length aux - (size old - size big)) - length aux)}
\text{drop (size big - size old) (Stack.list big)} = 
\text{drop (length (reverseN count (Stack.list big) aux) - remained)}
\]

46
\[(\text{rev}\ (\text{reverseN}\ \text{count}\ (\text{Stack}.\text{list}\ \text{big})\ \text{aux}))\]  
\[\implies\ \text{tl}\ (\text{drop}\ (\text{length}\ \text{aux} - (\text{size}\ \text{old} - \text{size}\ \text{big}))\ (\text{rev}\ \text{aux})\ @\]  
\[\text{drop}\ (\text{size}\ \text{big} - \text{size}\ \text{old})\ (\text{Stack}.\text{list}\ \text{big})) =\]  
\[\text{drop}\ (\text{length}\ \text{aux} - (\text{size}\ \text{old} - \text{Suc}\ (\text{size}\ \text{big})))\ (\text{rev}\ \text{aux})\ @\]  
\[\text{drop}\ (\text{Suc}\ (\text{size}\ \text{big}) - \text{size}\ \text{old})\ (\text{Stack}.\text{list}\ \text{big}))\]  
\[\text{apply}\ (\text{cases}\ \text{size}\ \text{old} - \text{size}\ \text{big} \leq \text{length}\ \text{aux};\ \text{cases}\ \text{size}\ \text{old} \leq \text{size}\ \text{big})\]  
\[\text{by}\ (\text{auto}\ \text{simp}:\ \text{tl-drop-2}\ \text{Suc-diff-le}\ \text{le-diff-conv}\ \text{le-refl}\ a)\]  

\text{from}\ 1\ \text{have}\ \text{remained} \leq \text{length}\ (\text{reverseN}\ \text{count}\ (\text{Stack}.\text{list}\ \text{big})\ \text{aux})\]  
\[\text{by}(\text{auto})\]  

\text{with}\ 1\ \text{show}\ \text{?case}\  
\[\text{apply}(\text{auto}\ \text{simp}:\ \text{rev-take}\ \text{take-tl}\ \text{drop-Suc}\ \text{Suc-diff-le}\ \text{tl-drop}\ \text{linarith}\ \text{simp}\ \text{del}:\ \text{reverseN-def})\]  
\[\text{using}\ b\ \text{by}\ \text{simp}\]  
\text{next}\  
\[\text{case}\ (2\ x\ xs\ \text{added}\ \text{old}\ \text{remained})\]  
\[\text{then}\ \text{show}\ \text{?case}\ \text{by}\ \text{auto}\]  
\text{qed}\]  
\text{qed}\]  

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

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

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

\text{with}\ 1\ \text{show}\ \text{?case}\  
\[\text{by}(\text{auto}\ \text{simp}:\ \text{first-hd})\]  
\text{next}\  
\[\text{case}\ (2\ x\ xs\ \text{added}\ \text{old}\ \text{remained})\]  
\[\text{then}\ \text{show}\ \text{?case}\]
by auto

qed

lemma list-current-size: \[ 0 < \text{size \, big}; \, \text{list-current \, big = []}; \, \text{invar \, big} \] \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: \text{invar (big :: 'a state) \implies size \, big = size (step \, big)}

by (induction \, big \, rule: \, step-state.induct)(auto \, split: \, current.splits)

lemma size-empty: \[ \text{invar (big :: 'a state); \, size \, big = 0} \] \implies \text{is-empty \, big}

proof (induction \, big)
  case Reverse
  then show ?case
    by (auto \, simp: \, min-def \, Stack-Proof.list-empty \, split: \, if-splits current.splits)
next
  case Common
  then show ?case
    by (auto \, simp: \, size-empty)

qed

lemma remaining-steps-step \, [simp]: \[ \text{invar (big :: 'a state); \, remaining-steps \, big > 0} \] \implies Suc (remaining-steps (step \, big)) = remaining-steps \, big

by (induction \, big \, rule: \, step-state.induct)(auto \, split: \, current.splits)

lemma remaining-steps-step-0 \, [simp]: \[ \text{invar (big :: 'a state); \, remaining-steps \, big = 0} \] \implies remaining-steps (step \, big) = 0

by (induction \, big)(auto \, split: \, current.splits)

lemma remaining-steps-push: \text{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: \[ \text{invar \, big; \, 0 < \text{size \, big}; \, \text{pop \, big} = (x, \, \text{big}')} \] \implies remaining-steps \, big' \leq \text{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 ⇒ 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 ⇒ 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]: [[invar big; 0 < size big; pop big = (x, big')]]
  ⇒ Suc (size big') = 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]: [[invar big; 0 < size-new big; pop big = (x, big')]]
  ⇒ Suc (size-new big') = 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: [invar (big :: 'a state); 0 < size big] ⇒ 0 < size-new big
  by(induction big)(auto simp: size-size-new)

end

17 Small Proofs

theory Small-Proof
import Common-Proof Small
begin
lemma step-size [simp]: invar (small :: 'a state) ""Longrightarrow"" size (step small) = size small
by(induction small rule: step-state.induct)(auto split: current.splits)

lemma size-empty [simp]: invar (small :: 'a state) ""Longrightarrow"" size small = 0 ""Longrightarrow"" is-empty small
by(induction small)
(auto simp: Common-Proof.size-empty Stack-Proof.list-empty split: current.splits)

lemma size-push [simp]: invar small ""Longrightarrow"" size (push x small) = Suc (size small)
by(induction x small rule: push.induct) (auto split: current.splits)

lemma size-new-push [simp]: invar small ""Longrightarrow"" size-new (push x small) = Suc (size-new small)
by(induction x small rule: push.induct) (auto split: current.splits)

lemma size-pop [simp]: [invar small; 0 < size small; pop small = (x, small')]]
""Longrightarrow"" Suc (size small') = 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]: [invar small; 0 < size-new small; pop small = (x, small')]]
""Longrightarrow"" Suc (size-new small') = size-new 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: [\invar (\small :: 'a state); \emptyset < \size \small] \implies \emptyset < \size-new \small
  by(induction \small)(auto simp: size-size-new)

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

lemma step-list-common [simp]:
  [\small = \Common \common; \invar \small] \implies \list (\step \small) = \list \small
  by auto

lemma step-list-reverse2 [simp]:
  assumes \small = \(\Reverse2 \current \aux \big \new \count\)
  \invar \small
  shows \list (\step \small) = \list \small
  proof –
    have \size-not-empty: \(\emptyset < \size \big\) = \(\\neg \is-empty \big\)
      by (simp add: Stack-Proof.size-not-empty)
    have \\neg \is-empty \big
      \implies \rev (\Stack\list (\Stack\pop \big)) @ [\Stack\first \big] = \rev (\Stack\list \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: \invar (\small :: 'a state) \implies \invar (\step \small)
  proof(induction \small rule: step-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 \rev (\Stack\list \small) @ \aux =
rev (Stack.list (Stack.pop small)) @ Stack.first small # 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
    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 = Common common \implies list (push x small) = x # list small
  by auto

lemma push-list-reverse2 [simp]: small = (Reverse2 current auxS big newS count)
  \implies list (push x small) = x # list small
by (induction x current rule: Current.push.induct) auto

lemma pop-list-Reverse2 [simp]:
  small = (Reverse2 current auxS big newS count)
  \neg is-empty small;
  invar small;
  pop small = (x, small')
\implies x # list small' = list small
proof (induction current arbitrary: x rule: Current.pop.induct)
  case (1 added old remained)
  then have 0 < size old
    by (auto simp: Stack-Proof.size-not-empty)
  with 1 show ?case
    by (auto simp: rev-take Cons-nth-drop-Suc Suc-diff-le hd-drop-conv-nth)
next
case (2 x xs added old remained)
then show ?case by auto
qed

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]: \([\text{invar small}; \, 0 < \text{size small}; \, \text{Small.pop small} = (x, \text{small}')]\) \\
\implies x \# \text{list-current small'} = \text{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\text{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 Reverse2
  then show ?case
  by auto
qed
qed

lemma list-current-size [simp]: \([0 < \text{size small}; \, \text{list-current small} = []]; \, \text{invar small}\] \implies \text{False}
proof (induction small)
  case (Reverse1 current)
  then have invar current
  by (auto split: current.splits)

  with Reverse1 show ?case
  using Current-Proof.list-size
  by auto
next
  case Reverse2
  then show ?case
  by (auto split: current.splits)
next
  case Common
  then show ?case
using list-current-size by auto

qed

lemma list-Reverse2 [simp]: []
  \(0 < \text{size} (\text{Reverse2 current auxS big newS count})\);
  \(\text{invar} (\text{Reverse2 current auxS big newS count})\)
\[
\Rightarrow \quad \text{fst (Current.pop current)} \neq \text{Small.list (Reverse2 (drop-first current) auxS big newS count)} = \text{Small.list (Reverse2 current auxS big newS count)}
\]
  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 Big-Proof Small-Proof
begin

lemmas state-splits = idle.splits Common.state.splits Small.state.splits Big.state.splits
lemmas invar-steps = Big-Proof.invar-step Common-Proof.invar-step Small-Proof.invar-step

lemma invar-list-big-first:
  \(\text{invar states} \Rightarrow \text{list-big-first states} = \text{list-current-big-first states}\)
using app-rev
by (cases states) (auto split: prod.splits)

lemma step-lists [simp]: \(\text{invar states} \Rightarrow \text{lists (step states)} = \text{lists states}\)
proof (induction states rule: lists.induct)
case (1 dir currentB big auxB count currentS small auxS)
then show ?case
proof (induction
  (States dir (Reverse currentB big auxB count) (Reverse1 currentS small auxS))

  rule: step-states.induct)
case 1
then show ?case
  by (cases currentB) auto
next
case (2-1 count')
then have \(0 < \text{size big}\)
  by (cases currentB) auto

then have big-not-empty: \(\text{Stack.list big} \neq []\)
  by (simp add: Stack-Proof.size-not-empty Stack-Proof.list-empty)

with 2-1 show ?case
using
  reverseN-step[of Stack list big count' auxB]
  Stack-Proof.list-empty[symmetric, of small]
by (cases currentB)(auto simp: first-hd funpow-swap1 reverseN-step reverseN-finish simp del: reverseN-def)
qed
next
case (2-1 dir common small)
then show ?case
  using Small-Proof.step-list-reverse2[of small]
  by (auto split: Small.state.splits)
next
case (2-2 dir big current auxS big newS count)
then show ?case
  using Small-Proof.step-list-reverse2[of Reverse2 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); lists (States dir big small) = (big', small')]
  ⇒ lists (States dir big (Small.push x small)) = (big', x # small')
by (induction States dir big (Small.push \( x \) small) rule: lists.induct)
(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)
\( \iff \) list-big-first (States dir big small) = list-current-big-first (States dir big small)
using app-rev
by (auto split: current.splits)

lemma list-big-first-pop-big [simp]: []
invar (States dir big small);
0 < size big;
Big.pop big = (\( x, big' \))
\( \Rightarrow \) \( x \) # 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]: []
invar (States dir big small);
0 < size big;
Big.pop big = (\( x, big' \))
\( \Rightarrow \) \( x \) # list-current-big-first (States dir big' small) =
list-current-big-first (States dir big small)
by auto

lemma lists-big-first-pop-big: []
invar (States dir big small);
0 < size big;
Big.pop big = (\( x, big' \))
\( \Rightarrow \) 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: []
invar (States dir big small);
0 < size big;
Big.pop big = (\( x, big' \))
\( \Rightarrow \) 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 < size small;
Small.pop small = (\( x, small' \))
\( \Rightarrow \) \( x \) # list-small-first (States dir big small') = list-small-first (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);
\theta < size small;
Small.pop small = (x, small') \implies x \# list-current-small-first (States dir big small') =
  list-current-small-first (States dir big small)
by auto

lemma lists-small-first-pop-small: |
invar (States dir big small);
\theta < size small;
Small.pop small = (x, small') \implies list-small-first (States dir big small') = 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: |
invar (States dir big small);
\theta < size big;
Big.pop big = (x, big')
\[ \Rightarrow \ \text{invar} \ \text{big}' \land \ \text{invar} \ \text{small} \]
by(auto simp: Big-Proof.invar-pop)

lemma invar-pop-big-aux: [\[ 
\Rightarrow \ \text{(case \ \text{big}', \ \text{small}) \ of} \\
(\text{Reverse} \ - \ \text{big} \ - \ \text{count}, \ \text{Reverse1} \ (\text{Current} \ - \ - \ \text{old \ remained}) \ \text{small} -) \Rightarrow \\
\text{size \ big} - \ \text{count} = \ \text{remained - size \ old} \land \ \text{count} \geq \ \text{size \ small} \\
| (\text{-}, \ \text{Reverse1} - - -) \Rightarrow \ False \\
| (\text{Reverse} - - - -, \ -) \Rightarrow \ False \\
| - \Rightarrow \ True 
\]
by(auto split: Big.state.splits Small.state.splits prod.splits)

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

proof(induction small rule: Small.pop.induct)
\begin{align*}
\text{case} \ 1 \\
\text{then \ show} \ ?\text{case} \\
by(auto \ split: \ \text{Big.state.splits} \ \text{Small.state.splits} \ \text{prod.splits})
\end{align*}

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: [ ]
i

  invar (States dir big small);
  0 < size small;
  Small.pop small = (x, small')
  ] = 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) = 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)
  = 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: [invar states; step states = States dir big small] \implies invar big  
\land 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 (Reverse currentB big' auxB 0)  
   by auto  
with 1 have invar-big: invar big  
   using Big-Proof.invar-step[of Reverse 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 (Reverse 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 Reverse2 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

61
lemma invar-step-aux: invar states \implies (case step states of
   (States - (Reverse - big - count) (Reverse1 (Current - - old remained) small -)) \Rightarrow
   size big - count = remained - size old \land count \geq size small
   | (States - - (Reverse1 - - -)) \Rightarrow False
   | (States - (Reverse - - -)) \Rightarrow False
   | \Rightarrow True
 )

proof (induction states rule: step-states.induct)
  case (2-1 dir current big aux count small)
  then show \?case
  proof (cases small)
    case (Reverse1 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)
by (cases step states) (auto simp: step-invars)

lemma step-consistent [simp]:
  \[ \forall states. \invar (states :: 'a states) \Rightarrow P (step states) = P states; invar states\]
  \implies P states = P ((step ^^ n) states)
by (induction n arbitrary: states)
  (auto simp: States-Proof.invar-step funpow-swap1)

lemma step-consistent-2:
  \[ \forall states. \invar (states :: 'a states); P states \Rightarrow P (step states); invar states; P states\]
  \implies P ((step ^^ n) 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]: [invar (states :: 'a states); remaining-steps states = 0]
  \implies remaining-steps (step states) = 0
by (induction states rule: step-states.induct) (auto split: current.splits Small.state.splits)

lemma remaining-steps-0': [invar (states :: 'a states); remaining-steps states = 0]
  \implies remaining-steps ((step ^^ n) states) = 0

62
by (induction n arbitrary: states) (auto simp: invar-step funpow-swap1)

lemma remaining-steps-decline-Suc:
[invar (states :: 'a states); 0 < remaining-steps states]
⇒ Suc (remaining-steps (step states)) = remaining-steps states

proof (induction remaining-steps rule: step-states.induct)
  case 1
  then show ?case
    by (auto simp: invar-step funpow-swap1)
next
  case (2-1 - - - - small)
  then show ?case
    by (cases small) (auto split: small)
next
  case (2-2 dir big small)
  then show ?case
  proof (cases small)
    case Reverse
    then show ?case
      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 Reverse
    then show ?case by auto
next
    case Common
    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]:
\[[\text{invar (\text{states} :: \text{'a states}); \text{remaining-steps states} \leq n]} \implies \text{remaining-steps (} (\text{step} \uparrow^n) \text{ states}) = 0\]
\begin{align*}
\text{by (induction \text{n arbitrary: states})(auto simp: funpow_swap1 invar-step)}
\end{align*}

lemma remaining-steps-n-steps-plus [simp]:
\[[n \leq \text{remaining-steps states}; \text{invar \text{(states} :: \text{'a states})] \implies \text{remaining-steps (} (\text{step} \uparrow^n) \text{ states}) + n = \text{remaining-steps states}\]
\begin{align*}
\text{by (induction \text{n arbitrary: states})(auto simp: funpow_swap1 invar-step)}
\end{align*}

lemma remaining-steps-n-steps-sub [simp]: \text{invar (\text{states} :: \text{'a states}) = \implies \text{remaining-steps (} (\text{step} \uparrow^n) \text{ states}) = \text{remaining-steps states} - n\]
\begin{align*}
\text{by (induction \text{n arbitrary: states})(auto simp: funpow_swap1 invar-step)}
\end{align*}

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

lemma step-size-new-small-2 [simp]:
\text{invar states = \implies \text{size-new small (step states) = size-new small states}}
\begin{align*}
\text{by (cases states; cases step states) auto}
\end{align*}

lemma step-size-new-big [simp]:
\[[\text{invar (States dir big small); \text{step (States dir big small)} = \text{States dir' big' small'}] \implies \text{size-new big'} = \text{size-new big}\]
\begin{align*}
\text{proof (induction States dir big small rule: step-states.induct)}
\text{case 1}
\text{then show ?case}
\end{align*}
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 ⇒ size-new-big (step states) = size-new-big states
  by(cases states; cases step states) auto

lemma step-size-small [simp]:
  [invar (States dir big small); step (States dir big small) = States dir' big' small']
  ⇒ 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 ⇒ size-small (step states) = size-small states
  by(cases states; cases step states) auto

lemma step-size-big [simp]:
  [invar (States dir big small); step (States dir big small) = States dir' big' small']
  ⇒ 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 \( \forall \text{case} \)
  \( \text{by} \) (auto split: Small.state.splits current.splits)
next
case 2-3
then show \( \forall \text{case} \)
  \( \text{by} \) (auto split: current.splits Big.state.splits)
next
case 2-4
then show \( \forall \text{case} \)
  \( \text{by} \) (auto split: Big.state.splits)
qed

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

lemma step-size-ok-1: []
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 \times size-new small
  \( \implies \) size-new big' + remaining-steps (States dir' big' small') + 2 \leq 3 \times size-new small'
  \( \text{using} \) step-size-new-small step-size-new-big remaining-steps-decline
  \( \text{by} \) (smt (verit, ccfv-SIG) add commute le-trans nat-add-left-cancel-le)

lemma step-size-ok-2: []
invar (States dir big small); step (States dir big small) = States dir' big' small';
  size-new small + remaining-steps (States dir big small) + 2 \leq 3 \times size-new big
  \( \implies \) size-new small' + remaining-steps (States dir' big' small') + 2 \leq 3 \times size-new big'
  \( \text{using} \) remaining-steps-decline step-size-new-small step-size-new-big
  \( \text{by} \) (smt (verit, best) add-le-mono le-refl le-trans)

lemma step-size-ok-3: []
invar (States dir big small); step (States dir big small) = States dir' big' small';
  remaining-steps (States dir big small) + 1 \leq 4 \times size small
  \( \implies \) remaining-steps (States dir' big' small') + 1 \leq 4 \times size small'
  \( \text{using} \) remaining-steps-decline step-size-small
  \( \text{by} \) (metis Suc-cq-plus1 Suc-le-mono le-trans)

lemma step-size-ok-4: []
invar (States dir big small); step (States dir big small) = States dir' big' small';
  remaining-steps (States dir big small) + 1 \leq 4 \times size big
  \( \implies \) remaining-steps (States dir' big' small') + 1 \leq 4 \times size big'
  \( \text{using} \) remaining-steps-decline step-size-big
  \( \text{by} \) (metis (no-types, lifting) add-mono-thms-linordered-semiring(3) order.trans)
lemma step-size-ok: [invar states; size-ok states] \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'.simp)

lemma step-n-size-ok: [[invar states; size-ok states] \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]: [
invar (States dir big small);
step (States dir big (Small.push x small)) = States dir' big' small'
]\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]: [
invar (States dir big small);
step (States dir big (Small.push x small)) = States dir' big' small'
]\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]: [
invar (States dir big small);
step (States dir (Big.push x big) small) = States dir' big' small'
]\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]: [
invar (States dir big small);
step (States dir (Big.push x big) small) = States dir' big' small'
]\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]: []
invar (States dir big small);
0 < size big;
Big.pop big = (x, bigP);
step (States dir bigP small) = States dir' big' small'
] \[ Suc (size big') = size big
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-big [simp]: [
invar (States dir big small);
0 < size big; Big.pop big = (x, bigP);
step (States dir bigP small) = States dir' big' small'
] \[ Suc (size-new big') = size-new big
using
invar-pop-big[of dir big small x bigP]
Big-Proof.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]: [
invar (States dir big small);
(step \^\^ n) (States dir big small) = States dir' big' small'
] \[ size small' = size small
using step-consistent[of size-small States dir big small n]
by simp

lemma step-n-size-big [simp]:
[invar (States dir big small); (step \^\^ n) (States dir big small) = States dir' big' small']
\[ size big' = size big
using step-consistent[of size-big States dir big small n]
by simp

lemma step-n-size-new-small [simp]:
[invar (States dir big small); (step \^\^ n) (States dir big small) = States dir' big' small']
\[ size-new small' = size-new small
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 \^\^ n) (States dir big small) = States dir' big' small']
\[ size-new big' = size-new 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 ^^ n) (States dir big (Small.push x small)) = States dir' big' small'  
] \Rightarrow size small' = Suc (size 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 ^^ n) (States dir big (Small.push x small)) = States dir' big' small'  
] \Rightarrow size-new small' = Suc (size-new small)  
by (metis Small-Proof.size-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 ^^ n) (States dir (Big.push x big) small) = States dir' big' small'  
] \Rightarrow size big' = Suc (size 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 ^^ n) (States dir (Big.push x big) small) = States dir' big' small'  
] \Rightarrow size-new big' = Suc (size-new 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 ^^ n) (States dir big smallP) = States dir' big' small'  
] \Rightarrow Suc (size small') = size 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 ^^ n) (States dir big smallP) = States dir' big' small'  
] \Rightarrow Suc (size-new small') = size-new small  
by (metis (no-types, lifting) invar-states.simps)

lemma step-n-pop-size-big [simp]: [  
invar (States dir big small);  
0 < size big; Big.pop big = (x, bigP);  
(step ^^ n) (States dir bigP small) = States dir' big' small'
\[
\Rightarrow \text{Suc (size big')} = \text{size big}
\]

**using** invar-pop-big Big-Proof.size-pop step-n-size-big

**by** fastforce

**lemma** step-n-pop-size-new-big:

\[\\]

\[\text{invar (States dir big small); \quad 0 < \text{size big}; \quad \text{Big.pop big} = (x, \text{bigP}); \quad (\text{step} \wedge n) (\text{States dir bigP small}) = \text{States dir' big' small'}}\]

\[\Rightarrow \text{Suc (size-new big')} = \text{size-new big}\]

**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]:

\[\text{invar (States dir big small)} \Rightarrow \text{remaining-steps (States dir big small)} = \text{remaining-steps (States dir big (Small.push x small))}\]

**by** (induction x small rule: Small.push.induct)(auto split: current.splits)

**lemma** remaining-steps-pop-small:

\[\text{invar (States dir big small); \quad 0 < \text{size small}; \quad \text{Small.pop small} = (x, \text{smallP})}\]

\[\Rightarrow \text{remaining-steps (States dir big smallP) \leq remaining-steps (States dir big small)}\]

**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 simp remaining-steps max.coboundedI2 split: prod.splits)

**then show** ?case

**by** (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:

\[\text{invar (States dir big small); \quad 0 < \text{size big}; \quad \text{Big.pop big} = (x, \text{bigP})}\]

\[\Rightarrow \text{remaining-steps (States dir bigP small) \leq remaining-steps (States dir big small)}\]

**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

70
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 simp del: reverseN-def)

qed

lemma remaining-steps-push-big [simp]: invar (States dir big small) \[ \Rightarrow \] 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]: []
invar (States dir big small);
4 \leq \text{remaining-steps} (States dir big small);
(step~4) (States dir (Big.push x big) small) = States dir' big' small'
\[ \Rightarrow \] 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]: []
invar (States dir big small);
4 \leq \text{remaining-steps} (States dir big small);
(step~4) (States dir big (Small.push x small)) = States dir' big' small'
\[ \Rightarrow \] 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: []
invar (States dir big small);
0 < \text{size big};
Big.pop big = (x, bigP);
4 \leq \text{remaining-steps} (States dir bigP small);
(step~4) (States dir bigP small) = States dir' big' small'
\[ \Rightarrow \] remaining-steps (States dir' big' small') \leq \text{remaining-steps} (States dir big 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: []
invar (States dir big small);

\[ \begin{align*}
0 &< \text{size small}; \\
\text{Small.pop small} &= (x, \text{smallP}); \\
4 &\leq \text{remaining-steps} (\text{States dir big smallP}); \\
(\text{step}^{\wedge}4) (\text{States dir big smallP}) &= \text{States dir' big' small'} \\
\implies &\quad \text{remaining-steps} (\text{States dir' big' small'}) \leq \text{remaining-steps} (\text{States dir big small}) - 4 \\
\text{by} &\quad (\text{metis add-le-imp-le-diff invar-pop-small remaining-steps-n-steps-plus remaining-steps-pop-small}) \\
\end{align*} \]

**lemma** step-4-pop-small-size-ok-1:

\[ \begin{align*}
invar (\text{States dir big small}); \\
0 &< \text{size small}; \\
\text{Small.pop small} &= (x, \text{smallP}); \\
4 &\leq \text{remaining-steps} (\text{States dir big smallP}); \\
(\text{step}^{\wedge}4) (\text{States dir big smallP}) &= \text{States dir' big' small'}; \\
\text{remaining-steps} (\text{States dir big small}) + 1 &\leq 4 \times \text{size small} \\
\implies &\quad \text{remaining-steps} (\text{States dir' big' small'}) + 1 \leq 4 \times \text{size small'} \\
\text{by} &\quad (\text{smt (verit, ccfv-SIG) add.leE add-le-cancel-right invar-pop-small order-trans remaining-steps-pop-big-step-n-size-small remaining-steps-n-steps-plus}) \\
\end{align*} \]

**lemma** step-4-pop-big-size-ok-1:

\[ \begin{align*}
invar (\text{States dir big small}); \\
0 &< \text{size big}; \\
\text{Big.pop big} &= (x, \text{bigP}); \\
4 &\leq \text{remaining-steps} (\text{States dir bigP small}); \\
(\text{step}^{\wedge}4) (\text{States dir bigP small}) &= \text{States dir' big' small'}; \\
\text{remaining-steps} (\text{States dir big small}) + 1 &\leq 4 \times \text{size big} \\
\implies &\quad \text{remaining-steps} (\text{States dir' big' small'}) + 1 \leq 4 \times \text{size big'} \\
\text{by} &\quad (\text{smt (z3) add.commute add-leE remaining-steps-pop-big-step-n-size-big remaining-steps-n-steps-plus}) \\
\end{align*} \]

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

\[ \begin{align*}
invar (\text{States dir big small}); \\
0 &< \text{size big}; \\
\text{Big.pop big} &= (x, \text{bigP}); \\
4 &\leq \text{remaining-steps} (\text{States dir bigP small}); \\
(\text{step}^{\wedge}4) (\text{States dir bigP small}) &\geq 4 \\
(\text{step}^{\wedge}4) (\text{States dir bigP small}) &= \text{States dir' big' small'} \\
\text{remaining-steps} (\text{States dir bigP small}) &\geq 4 \\
\text{by} &\quad (\text{smt (z3) add.commute add-leE remaining-steps-pop-big-step-n-size-big remaining-steps-n-steps-plus}) \\
\end{align*} \]

\[ 72 \]
remaining-steps \((\text{States dir big small}) + 1 \leq 4 \times \text{size big}\)

shows

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

proof –

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

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

with \text{assms} show \textcolor{red}?thesis

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

qed

\textbf{lemma} \textcolor{blue}{\text{step-4-pop-small-size-ok-3}:}

\textbf{assumes}

invar \((\text{States dir big small})\)

\(0 < \text{size small}\)

\textcolor{red}{\text{Small.pop small} = (x, smallP)}

remaining-steps \((\text{States dir big smallP}) \geq 4\)

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

\textcolor{red}{\text{size-new small} + \text{remaining-steps (States dir big small)} + 2 \leq 3 \times \text{size-new big}\textcolor{red}}\)

shows

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

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

\textbf{lemma} \textcolor{blue}{\text{step-4-pop-big-size-ok-3-aux:}} \[
\begin{align*}
0 < \text{size big}; \\
4 \leq & \text{remaining-steps (States dir big small)}; \\
(size-new small + \text{remaining-steps (States dir big small)}) + 2 \leq 3 \times \text{size-new big} \\
\implies & \text{size-new small + (remaining-steps (States dir big small) - 4) + 2 \leq 3 * (size-new big - 1)}
\end{align*}
\]

by linarith

\textbf{lemma} \textcolor{blue}{\text{step-4-pop-big-size-ok-3}:}

\textbf{assumes}

invar \((\text{States dir big small})\)

\(0 < \text{size big}\)

\textcolor{red}{\text{Big.pop big} = (x, bigP)}

remaining-steps \((\text{States dir bigP small}) \geq 4\)

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

\textcolor{red}{\text{size-new small} + \text{remaining-steps (States dir big small)} + 2 \leq 3 \times \text{size-new big}\textcolor{red}}\)

shows

\textcolor{red}{\text{size-new small'} + \text{remaining-steps (States dir' big' small')} + 2 \leq 3 \times \text{size-new big'}\textcolor{red}}\)
proof -
  from assms
  have size-new small + (remaining-steps (States dir big small) - 4) + 2 ≤ 3 * (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 ≤ 3 *
    (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:
  [ 0 < size small; 4 ≤ remaining-steps (States dir big small); size-new big + remaining-steps (States dir big small) + 2 ≤ 3 * size-new small ]
  ⇒ size-new big + (remaining-steps (States dir big small) - 4) + 2 ≤ 3 *
    (size-new small - 1)
  by linarith

lemma step-4-pop-small-size-ok-4:
  assumes
    invar (States dir big small)
    0 < size small
    Small.pop small = (x, smallP)
    remaining-steps (States dir big smallP) ≥ 4
    ((step ^^ 4) (States dir big smallP)) = (States dir' big' small')
    size-new big + remaining-steps (States dir big small) + 2 ≤ 3 * size-new small
  shows
    size-new big' + remaining-steps (States dir' big' small') + 2 ≤ 3 * size-new small'
proof -
  from assms step-4-pop-small-size-ok-4-aux
  have size-new big + (remaining-steps (States dir big small) - 4) + 2 ≤ 3 *
    (size-new small - 1)
    by (smt (verit, best) add-leE le-add-diff-inverse remaining-steps-pop-small)

with assms
  have size-new big + remaining-steps (States dir' big' small') + 2 ≤ 3 * (size-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: []
  0 < size big;
  4 ≤ remaining-steps (States dir big small);
  size-new big + remaining-steps (States dir big small) + 2 ≤ 3 * size-new small
] ⇒ size-new big - 1 + (remaining-steps (States dir big small) - 4) + 2 ≤ 3 * size-new small
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-steps (States dir bigP small) ≥ 4
  ((step ^^^ 4) (States dir bigP small)) = (States dir' big' small')
  size-new big + remaining-steps (States dir big small) + 2 ≤ 3 * size-new small
shows
  size-new big' + remaining-steps (States dir' big' small') + 2 ≤ 3 * size-new small'
proof –
  from assms step-4-pop-big-size-ok-4-aux
  have (size-new big - 1) + (remaining-steps (States dir big small) - 4) + 2 ≤
    3 * size-new small
  by linarith

  with assms
  have (size-new big - 1) + remaining-steps (States dir' big' small') + 2 ≤ 3 *
    size-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: []
invar (States dir big small);
  4 ≤ remaining-steps (States dir big small);
  (step ^^^ 4) (States dir big (Small.push x small)) = States dir' big' small';
  remaining-steps (States dir big small) + 1 ≤ 4 * size small
] ⇒ remaining-steps (States dir' big' small') + 1 ≤ 4 * size small'
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: []
invar (States dir big small);
\[4 \leq \text{remaining-steps (States dir big small)};\]
\[
\text{(step}^{\sim}4\text{) (States dir (Big.push x big) small) = States dir' big' small'};\]
\[
\text{remaining-steps (States dir big small) + 1} \leq 4 \times \text{size small}\]
\[
\implies \text{remaining-steps (States dir' big' small')} + 1 \leq 4 \times \text{size small'}\]
\[
\text{by (smt (verit, ccfv-SIG) Nat.le-diff-cone2 add-leD2 invar-push-big le-add1 le-add-diff-inverse2 remaining-steps-n-steps-sub remaining-steps-push-big step-n-size-small)}\]

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

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

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

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

\textbf{lemma step-4-push-big-size-ok-3-aux:}\]
\[
4 \leq \text{remaining-steps (States dir big small)};\]
\[
\text{size-new small + remaining-steps (States dir big small) + 2} \leq 3 \times \text{size-new big}\]
\[
\implies \text{size-new small' + remaining-steps (States dir big small) - 4) + 2} \leq 3 \times \text{Suc (size-new big)}\]
\[
\text{using distrib-left dual-order.trans le-add-diff-inverse2 by force}\]
lemma step-4-push-big-size-ok-3: [ invar (States dir big small); 
  4 \leq \text{remaining-steps} \ (\text{States dir big small});
  (\text{step}^{ \leq 4} \ (\text{States dir} \ \text{Big} \ . \text{push x big}) \ \text{small}) = \text{States dir'} \ \text{big'} \ \text{small'};
  \text{size-new small} + \text{remaining-steps} \ (\text{States dir big small}) + 2 \leq 3 \ast \text{size-new big} 
] \implies \text{size-new small'} + \text{remaining-steps} \ (\text{States dir'} \ \text{big'} \ \text{small'}) + 2 \leq 3 \ast \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: [ 
  4 \leq \text{remaining-steps} \ (\text{States dir big small});
  \text{size-new big} + \text{remaining-steps} \ (\text{States dir big small}) + 2 \leq 3 \ast \text{size-new small} 
] \implies \text{size-new big} + (\text{remaining-steps} \ (\text{States dir big small}) - 4) + 2 \leq 3 \ast \text{Suc (size-new small)} 
using distrib-left dual-order.trans le-add-diff-inverse2 by force

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

lemma step-4-push-big-size-ok-4: [ invar (States dir big small); 
  4 \leq \text{remaining-steps} \ (\text{States dir big small});
  (\text{step}^{ \leq 4} \ (\text{States dir} \ \text{Big} \ . \text{push x big}) \ \text{small}) = \text{States dir'} \ \text{big'} \ \text{small'};
  \text{size-new big} + \text{remaining-steps} \ (\text{States dir big small}) + 2 \leq 3 \ast \text{size-new small} 
] \implies \text{size-new big'} + \text{remaining-steps} \ (\text{States dir'} \ \text{big'} \ \text{small'}) + 2 \leq 3 \ast \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: [ invar (States dir big small); 
  4 \leq \text{remaining-steps} \ (\text{States dir big small});
  \text{size-ok} \ (\text{States dir big small})
] = 77
\[ \implies \text{size-ok} \left( (\text{step}^{\text{4}}) \left( \text{States dir big (Small.push } x \text{ small)}\right)\right) \]

\textbf{using} \text{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)}

\textbf{lemma} \text{step-4-push-big-size-ok:} \]
\text{invar} (\text{States dir big small});
\text{4} \leq \text{remaining-steps} (\text{States dir big small});
\text{size-ok} (\text{States dir big small})
\[ \implies \text{size-ok} \left( (\text{step}^{\text{4}}) \left( \text{States dir} \left( \text{Big.push } x \text{ big) small}\right)\right)\right) \]
\textbf{using} \text{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)}

\textbf{lemma} \text{step-4-pop-small-size-ok:} \]
\text{invar} (\text{States dir big small});
\text{0} < \text{size small};
\text{Small.pop small} = (x, \text{smallP});
\text{4} \leq \text{remaining-steps} (\text{States dir big smallP});
\text{size-ok} (\text{States dir big small})
\[ \implies \text{size-ok} \left( (\text{step}^{\text{4}}) \left( \text{States dir big smallP}\right)\right) \]
\textbf{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)}

\textbf{lemma} \text{step-4-pop-big-size-ok:} \]
\text{invar} (\text{States dir big small});
\text{0} < \text{size big}; \text{Big.pop big} = (x, \text{bigP});
\text{4} \leq \text{remaining-steps} (\text{States dir bigP small});
\text{size-ok} (\text{States dir big small})
\[ \implies \text{size-ok} \left( (\text{step}^{\text{4}}) \left( \text{States dir bigP small}\right)\right) \]
\textbf{using} \text{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)}

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

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

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

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

\textbf{lemma} \text{step-size-ok':} \left[ \text{invar states; size-ok' states n} \right] \implies \text{size-ok'} \left( \text{step states} \right) n
\textbf{by (smt (verit, ccft-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: \step (States \, dir \, big \, small) = States \, dir' \, big' \, small' \implies dir = dir'
by (induction States dir big small rule: step-states.induct) auto

lemma step-n-same: (\step^{\sim n}) (States \, dir \, big \, small) = States \, dir' \, big' \, small' \implies dir = dir'
proof (induction n arbitrary: big small big' small')
  case 0
  then show \?case
    by simp
next
case (Suc \, n)
obtain big'' small'' where \step (States \, dir \, big \, small) = States \, dir' \, big'' \, small''
  by (metis states.exhaust step-same)
with Suc show \?case
  by (auto simp: funpow-swap1)
qed

lemma step-listL: invar states \implies listL (\step \, states) = listL \, states
proof (induction states rule: listL.induct)
  case (1 big small)
  then have list-small-first (States \, Left \, big \, small) =
    Small.\list-current small \@ rev (Big.\list-current big)
    by auto
  then have listL (\step (States \, Left \, big \, small)) =
    Small.\list-current small \@ rev (Big.\list-current big)
    using 1 \step-lists by fastforce
  then have listL (States \, Left \, big \, small) =
    Small.\list-current small \@ rev (Big.\list-current big)
    by (smt (verit, ccv-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.\list-current big \@ rev (Small.\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.\list-current big \@ rev (Small.\list-current small)
    using 2 \step-lists by fastforce
then have listL (step (States Right big small)) =
  Big.list-current big @ rev (Small.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 \Rightarrow 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) \Rightarrow invar ((step^^n) states)
  by (simp add: invar-step step-consistent-2)

lemma step-n-size-ok': [invar states; size-ok' states x] \Rightarrow 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: []
  invar states;
  n < remaining-steps states;
  size-ok' states (remaining-steps states - n)
] \Rightarrow size-ok ((step ^ ^ n) states)
  by (simp add: step-n-size-ok')
lemma remaining-steps-idle: invar states
implies remaining-steps states = 0 <\rightarrow> (case states of
States - (Big,Common (Common.Idle -)) (Small,Common (Common.Idle -)) \Rightarrow True
| - \Rightarrow False)
by(cases states)
(auto split: Big.state.split Small.state.split Common.state.split current.split)

lemma remaining-steps-idle':
[invar (States dir big small); remaining-steps (States dir big small) = 0]
implies \exists big-current big-idle small-current small-idle. States dir big small =
States dir
(Big.state,Common (state,Idle big-current big-idle))
(Small.state,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.split)
end

19 Dequeue Proofs

theory RealTimeDeque-Dequeue
imports Deque RealTimeDeque States-Proof
begin

lemma list-deqL' [simp]: [invar deque; listL deque \neq []; deqL' deque = (x, deque')] 
implies x \# listL deque' = 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\((\text{cases } 3 \ast \text{length-left}' \geq \text{length-right})\)
  case True
  with 4 pop-left show ?thesis
    using Idle-Proof.pop-list[of left x left']
    by auto
next
  case False
  note Start-Transformation = False
then show ?thesis
proof\((\text{cases } \text{length-left}' \geq 1)\)
  case True
    let ?big = Reverse (Current ++ 0 right (size right – Suc length-left'))
    let ?small = Reverse1 (Current ++ 0 stack-left' (Suc (2 \ast \text{length-left'})))
  stack-left' []
  let ?states = States Left ?big ?small
  from 4 Start-Transformation True invar-left' have invar: invar ?states
    by (auto simp: Let-def rev-take rev-drop)
  with 4 Start-Transformation True invar-left'
  have States.listL ?states = tl (Idle.list left) @ rev (Stack.list right)
    using pop-list-tl[of left x left']
    by (auto simp del: reverseN-def)
  with invar
  have States.listL ((\text{step} \sim 6)?states) = tl (Idle.list left) @ rev (Stack.list right)
    using step-n-listL[of ?states 6]
    by presburger
  with 4 Start-Transformation True show ?thesis
    by (auto simp: Let-def)
next
  case False
  from False Start-Transformation 4 have [simp]: size left = 1
  using size-left' size-left'-size-left by auto
  with False Start-Transformation 4 have [simp]: Idle.list left = [x]
  by (induction left)(auto simp: length-one-hd split: stack.splits)
  obtain right1 right2 where right = Stack right1 right2
    using Stack.list.cases by blast
  with False Start-Transformation 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^^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.list-current small’ = Small.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.listL ?states-new = Small.list-current small @ rev (Big.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.listL ?states-stepped =
  Small.list-current small @ rev (Big.list-current big)
  using States-Proof.step-n-listL invar listL by metis

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

then have states-listL-list-current:
  x # listL (deqL (Transforming (States Left big small))) =
  Small.list-current small @ rev (Big.list-current big)
  by auto
with 5(1) have listL (Transforming (States Left big small)) = Small.list-current small @ rev (Big.list-current big)
  by auto

with states-listL-list-current
have x # listL (deqL (Transforming (States Left big small))) = listL (Transforming (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'
let ?states-stepped = (step^^4 ?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.list-current big' = Big.list-current big
  using big-invar big-size pop by auto
then have listL:
  x ≠ States.listL ?states-new = Big.list-current big @ rev (Small.list-current small)
proof (cases States.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]:
\[
x \neq \text{States.listL } ?\text{states-stepped} = \text{Big.list-current big} @ \text{rev} (\text{Small.list-current small})
\]
\[
\text{using States-Proof.step-n-listL[of } ?\text{states-new } 4\text{] invar listL}
\]
\[
\text{by auto}
\]
\[
\text{then have listL (deqL (Transforming (States Right big small))) = States.listL}
\]
\[
\text{?states-stepped}
\]
\[
\text{by(auto simp: Let-def pop split: prod.splits direction.splits states.splits state-splits)}
\]
\[
\text{then have listL-list-current:}
\]
\[
x \neq \text{listL (deqL (Transforming (States Right big small)))} =
\]
\[
\text{Big.list-current big} @ \text{rev} (\text{Small.list-current small})
\]
\[
\text{by auto}
\]
\[
\text{with 6(1) have listL (Transforming (States Right big small)) =}
\]
\[
\text{Big.list-current big} @ \text{rev} (\text{Small.list-current small})
\]
\[
\text{using invar-list-big-first[of States Right big small] by fastforce}
\]
\[
\text{with listL-list-current have}
\]
\[
x \neq \text{listL (deqL (Transforming (States Right big small)))} =
\]
\[
\text{listL (Transforming (States Right big small))}
\]
\[
\text{by auto}
\]
\[
\text{with 6 show } ?\text{case by auto}
\]
\[
\text{qed auto}
\]

\text{lemma list-deqL \{simp\}:
}\]
\[
\{\text{invar deq; listL deq} \neq [\] \implies listL (deqL deq) = tl (listL deq)\}
\]
\[
\text{using cons-tl[of fst (deqL' deq) listL (deqL deq) listL deq]}\]
\[
\text{by(auto split: prod.splits)}
\]

\text{lemma list-firstL \{simp\}:
}\]
\[
\{\text{invar deq; listL deq} \neq [\] \implies firstL deq = hd (listL deq)\}
\]
\[
\text{using cons-hd[of fst (deqL' deq) listL (deqL deq) listL deq]}\]
\[
\text{by(auto split: prod.splits)}
\]

\text{lemma invar-deqL:}
\[
\{\text{invar deq; } \neg \text{is-empty deq} \implies \text{invar \{deqL deq\}}\}
\]
\text{proof(\text{induction deq rule: deqL’.induct})}
\text{case (4 left right length-right)
}\]
\text{then obtain } x \text{ left’ where pop-left[simp]: Idle.pop left = (x, left’)
}\]
\[
\text{by fastforce}
\]
\text{then obtain stack-left’ length-left’
}\]
\text{where left’[simp]: left’ = idle.Idle stack-left’ length-left’
}\]
\[
\text{using idle.exhaust by blast}
\]
\text{from 4 have invar-left’: invar left’ invar left
}\]
\[
\text{using Idle-Proof.invar-pop by fastforce+}
\]
have \([\text{simp}]: \text{size stack-left'} = \text{size left } - 1\)
by (metis Idle-Proof.size-pop-sub left' pop-left size-idle.simps)

have \([\text{simp}]: \text{length-left'} = \text{size left } - 1\)
using invar-left' by auto

from 4 have list: \(\text{x} \neq \text{Idle.list left'} = \text{Idle.list left}\)
using Idle-Proof.pop-list[of left x left']
by auto

show \(?\text{case}\)
proof\((\text{cases length-right} \leq 3 * \text{size left'})\)
case True
with 4 invar-left' show \(?\text{thesis}\)
  by(auto simp: Stack-Proof.size-empty[symmetric])
next
case False
note Start-Transformation = False
then show \(?\text{thesis}\)
proof\((\text{cases } 1 \leq \text{size left'})\)
case True
let \(?\text{big} = \text{Reverse} (\text{Current} [] 0 \text{right} (\text{size right } - \text{Suc length-left'}))\)
  \text{right} [] \text{(right} (\text{size right } - \text{Suc length-left'}))
let \(?\text{small} = \text{Reverse1} (\text{Current} [] 0 \text{stack-left'} (\text{Suc } (2 * \text{length-left'})))\)
stack-left' []
let \(?\text{states} = \text{States Left } ?\text{big } ?\text{small}\)

from 4 Start-Transformation True invar-left'
have invar: invar \(?\text{states}\)
by(auto simp: Let-def rev-take rev-drop)
then have invar-stepped: invar ((step^^6) ?states)
using invar-step-n by blast

from 4 Start-Transformation True
have remaining-steps: \(6 < \text{remaining-steps } ?\text{states}\)
by auto
then have remaining-steps-end: \(0 < \text{remaining-steps } ((\text{step}^^6) ?\text{states})\)
  by(simp only: remaining-steps-n-steps-sub[of ?states 6] invar)

from 4 Start-Transformation True
have size-ok': size-ok' \(?\text{states}\) (remaining-steps ?states \(- 6\))
by auto
then have size-ok: size-ok ((step^^6) ?states)
using invar remaining-steps size-ok-steps by blast

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

with False Start-Transformation 4 have [simp]: Idle.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.list.cases by blast

with False Start-Transformation 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 = (step^^4) ?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:: Common.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
  (Big.state.Common (state.Idle big-current big-idle))
  (Small.state.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]: ¬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]: \neg \text{is-empty big-idle}
  by (auto simp: Idle-Proof.size-not-empty)

have [simp]: size small-idle \leq 3 \times size big-idle
  using 5 by auto

have [simp]: size big-idle \leq 3 \times 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 = (step^^4) ?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: Common.state.split)
next
  case False

89
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
    (Big.state.Common (state.Idle big-current big-idle))
    (Small.state.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]: ¬is-empty small-idle
  by (auto simp: Idle-Proof.size-not-empty)

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

have [simp]: size small-idle ≤ 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
imports Deque RealTimeDeque States-Proof
begin

lemma list-enqL: invar deque \implies listL (enqL x deque) = x \# listL 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 x left] 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 = Reverse (Current [] 0 left' ?length-left) left' [] ?length-left
    let ?small = Reverse1 (Current [] 0 right ?length-right) right []
    let ?states = States Right ?big ?small
    let ?states-stepped = (step^^6) ?states

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

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

    then have States.listL ?states-stepped = x \# Idle.list left \@ rev (Stack.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 = (step^4) ?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.listL ?states-stepped = x # Small.list-current small @ rev (Big.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 = (step^4) ?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 7 have invar: invar ?states 
  using invar-push-big[of Right big small x]
  by auto

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

92
by(auto split: prod.split)

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

with list-invar show ?case
  using app-rev[of Small.list-current small Big.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 =⇒ 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' ≤ 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
    (Reverse (Current [] 0 left' ?length-left) left' [] ?length-left)
    (Reverse1 (Current [] 0 right ?length-right) right [])
  let ?states-stepped = (step^^6) ?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 ?\text{states}
  using Stack-Proof.size-not-empty[of right]
  by auto

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

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

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

from False show ?\text{thesis}
  using invar-stepped remaining-steps-stepped size-ok-stepped
  by(auto simp: Let-def)
  qed

next
case (6 x big small)
  let ?\text{small} = Small.push x small
  let ?\text{states} = States Left big ?\text{small}
  let ?\text{states-stepped} = (\text{step}^{\circ 4}) ?\text{states}

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

then have invar-stepped: invar ?\text{states-stepped}
  using invar-step-n by blast

show ?\text{case}
proof(cases 4 < remaining-steps ?\text{states})
  case True

  obtain big-stepped small-stepped where stepped [simp]:
  ?\text{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 ?\text{states-stepped}
  using invar remaining-steps-n-steps-sub[of ?\text{states} 4]
  by simp

from True 6(1) have size-ok: size-ok ?\text{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
(Big.state.Common (state Idle big-current big-idle))
(Small.state.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 = (step^^4) ?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
  (Big.state.Common (state.Idle big-current big-idle))
(Small.state.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 Deque RealTimeDeque States-Proof RealTimeDeque-Deque RealTimeDeque-Enqueue RealTimeDeque-Disconnect
begin

lemma swap-lists-left: invar (States Left big small) \implies States.listL (States Left big small) = rev (States.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.listL (States Right big small) = rev (States.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 (Transforming states)
  then show ?case
  apply (cases states)
  using swap-lists-left swap-lists-right
  by (metis (full-types) RealTimeDeque.listL.simps(6) direction.exhaust invar-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 7 have [simp]: firstR q = firstL (swap q)
    by (auto split: prod.splits)
  from 7 have listL (swap q) ≠ []
    by auto
  with 7 have firstL (swap q) = hd (listL (swap q))
    using invar-swap list-firstL by blast
  then show ?case
    using ⟨firstR q = firstL (swap q)⟩ 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