Real-Time Double-Ended Queue

Balazs Toth and Tobias Nipkow
Technical University of Munich

July 1, 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

theory *Deque*
imports *Main*
begin

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

locale *Deque* =
fixes *empty* :: $'q$
fixes *enqL* :: $'a \Rightarrow 'q \Rightarrow 'q$
fixes *enqR* :: $'a \Rightarrow 'q \Rightarrow 'q$
fixes *firstL* :: $'q \Rightarrow 'a$
fixes *firstR* :: $'q \Rightarrow 'a$
fixes *deqL* :: $'q \Rightarrow 'q$
fixes *deqR* :: $'q \Rightarrow 'q$
fixes *is-empty* :: $'q \Rightarrow bool$
fixes *listL* :: $'q \Rightarrow 'a\ list$
fixes *invar* :: $'q \Rightarrow bool$

assumes *list-empty*:
$listL\ empty = []$

assumes *list-enqL*:
$invar\ q \Longrightarrow listL(enqL\ x\ q) = x\ \#\ listL\ q$
assumes *list-enqR*:
$invar\ q \Longrightarrow rev\ (listL\ (enqR\ x\ q)) = x\ \#\ rev\ (listL\ q)$
assumes *list-deqL*:
$[\![invar\ q; \neg\ listL\ q = []]\!] \Longrightarrow listL(deqL\ q) = tl(listL\ q)$
assumes *list-deqR*:
$[\![invar\ q; \neg\ rev\ (listL\ q) = []]\!] \Longrightarrow rev\ (listL\ (deqR\ q)) = tl\ (rev\ (listL\ q))$

assumes *list-firstL*:
$[\![invar\ q; \neg\ listL\ q = []]\!] \Longrightarrow firstL\ q = hd(listL\ q)$
assumes *list-firstR*:

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

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

assumes *invar-empty*:
$invar\ empty$

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

abbreviation $listR :: 'q \Rightarrow 'a\ list$ **where**
$listR\ deque \equiv rev\ (listL\ deque)$

end

end

2 Type Classes

theory *Type-Classes*
imports *Main*
begin

Overloaded functions:

class *is-empty* =
fixes $is\text{-}empty :: 'a \Rightarrow bool$

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

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

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

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

end

3 Stack

theory *Stack*
imports *Type-Classes*
begin

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

end

end

4 Current Stack

theory *Current*
imports *Stack*
begin

This data structure is composed of:

- the newly added elements to one end of a deque during the 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 *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 *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:

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

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.

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

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

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

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

definition *reverseN* **where**
[*simp*]: *reverseN n xs acc* ≡ *rev* (*take n xs*) @ *acc*

fun *list* :: *'a state* ⇒ *'a list* **where**
list (*Idle - idle*) = *Idle.list idle*
| *list* (*Copy* (*Current extra - - remained*) *old new moved*)
= *extra* @ *reverseN* (*remained* − *moved*) *old new*

fun *list-current* :: *'a state* ⇒ *'a list* **where**
list-current (*Idle current -*) = *Current.list current*
| *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 x 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*

| *remaining-steps* (*Copy* (*Current* - - - *remained*) *aux new moved*) = *remained* − *moved*

instance..
end

end

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.

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 transformation 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.

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
```

```
| invar (Reverse current big aux count) ⟷ (
   case current of Current extra added old remained ⇒
     invar current
    ∧ List.length aux ≥ remained − count

    ∧ count ≤ size big
    ∧ Stack.list old = rev (take (size old) ((rev (Stack.list big)) @ aux))
   ∧ take remained (Stack.list old) = rev (take remained (reverseN count (Stack.list
big) aux))
)

instance..
end

instantiation state ::(type) size
begin

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

instance..
end

instantiation state ::(type) remaining-steps
begin

fun remaining-steps-state :: 'a state ⇒ nat where
  remaining-steps (Common state) = remaining-steps state
| remaining-steps (Reverse (Current - - - remaining) - - count) = count + remain-
ing + 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 *step* function the originally contained elements are reversed.

Reverse2: Using the *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 *Common*. Is used to reverse the elements from the two previous phases again to get them again in the original order.

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

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

Functions:

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

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

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

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

list: List abstraction of the elements which this end will contain after the 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.

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

fun *list* :: *'a state* ⇒ *'a list* **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*

instance..
end

end

9 Combining Big and Small

theory *States*
imports *Big Small*
begin

datatype *direction* = *Left* | *Right*

datatype *'a states* = *States direction 'a Big.state 'a Small.state*

instantiation *states*::(*type*) *step*
begin

fun *step-states* :: *'a states* ⇒ *'a states* **where**
  step (*States dir* (*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

instantiation *states*::(*type*) *remaining-steps*
begin

fun *remaining-steps-states* :: *'a states* ⇒ *nat* **where**
  remaining-steps (*States - big small*) = *max*
    (*remaining-steps big*)
    (*case small of*
       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 * size of left end $\geq$ size of right end
- 3 * size of right end $\geq$ 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*#[]) [] = *Three z y x*
| *small-deque* (*x*#*y*#[]) (*z*#[]) = *Three z y x*
| *small-deque* (*x*#[]) (*y*#*z*#[]) = *Three z y x*

fun *deqL'* :: *'a deque* ⇒ *'a* ∗ *'a deque* **where**
deqL' (*One x*) = (*x*, *Empty*)

```
| deqL' (Two x y) = (x, One y)
| deqL' (Three x y z) = (x, Two y z)
| deqL' (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
        (Big.Common (Common.Idle - big))
        (Small.Common (Common.Idle - small))
       ⇒ Idle big small
   | - ⇒ Transforming states
 )

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

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*]⟧ ⟹ *Suc 0* ≤ *length xs*
by(*induction xs*) *auto*

lemma *take-last*: *xs* ≠ [] ⟹ *take 1* (*rev xs*) = [*last xs*]
by(*induction xs*)(*auto simp*: *take-last-length*)

lemma *take-hd* [*simp*]: *xs* ≠ [] ⟹ *take* (*Suc 0*) *xs* = [*hd xs*]
by(*induction xs*) *auto*

lemma *cons-tl*: *x* # *xs* = *ys* ⟹ *xs* = *tl ys*
by *auto*

lemma *cons-hd*: *x* # *xs* = *ys* ⟹ *x* = *hd ys*
by *auto*

lemma *take-hd'*: *ys* ≠ [] ⟹ *take* (*size ys*) (*x* # *xs*) = *take* (*Suc* (*size xs*)) *ys* ⟹ *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* ≠ [] ⟹ *hd xs* # *drop* (*Suc 0*) *xs* = *xs*
by(*induction xs*) *auto*

lemma *hd-drop* [*simp*]: *n* < *length xs* ⟹ *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* ∧ *0* < *y* ⟹ *take x xs* = *take y ys* ⟹ *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* ≠ [] ⟹ *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* ⟹ *0* < *y* ⟹ *Suc* (*min* (*x* − *Suc 0*) (*y* − *Suc*

0)) = *min x y*
by *auto*

lemma *rev-tl-hd*: *xs* ≠ [] ⟹ *rev* (*tl xs*) @ [*hd xs*] = *rev xs*
by (*simp add*: *rev-app-single*)

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

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

lemma *Suc-sub*: *Suc n* = *m* ⟹ *n* = *m* − *1*
by *simp*

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

end

12 Stack Proofs

theory *Stack-Proof*
imports *Stack RTD-Util*
begin

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

lemma *pop-list* [*simp*]: ¬ *is-empty stack* ⟹ *list* (*pop stack*) = *tl* (*list stack*)
by(*induction stack rule*: *pop.induct*) *auto*

lemma *first-list* [*simp*]: ¬ *is-empty stack* ⟹ *first stack* = *hd* (*list stack*)
by(*induction stack rule*: *first.induct*) *auto*

lemma *list-empty*: *list stack* = [] ⟷ *is-empty stack*
by(*induction stack rule*: *is-empty-stack.induct*) *auto*

lemma *list-not-empty*: *list stack* ≠ [] ⟷ ¬ *is-empty stack*
by(*induction stack rule*: *is-empty-stack.induct*) *auto*

lemma *list-empty-2* [*simp*]: ⟦*list stack* ≠ []; *is-empty stack*⟧ ⟹ *False*
by (*simp add*: *list-empty*)

lemma *list-not-empty-2* [*simp*]:⟦*list stack* = []; ¬ *is-empty stack*⟧ ⟹ *False*
by (*simp add*: *list-empty*)

lemma *list-empty-size*: *list stack* = [] ⟷ *size stack* = *0*
by(*induction stack*) *auto*

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

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

lemma *list-not-empty-size-2* [*simp*]:$[\![$*list stack* = []; *0* $<$ *size stack*$]\!] \Longrightarrow$ *False*
by (*simp add*: *list-empty-size*)

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

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

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

lemma *size-not-empty*: *size* (*stack* :: *'a stack*) $>$ *0* $\longleftrightarrow$ ¬ *is-empty stack*
by(*induction stack rule*: *is-empty-stack.induct*) *auto*

lemma *size-empty-2*[*simp*]: $[\![$*size* (*stack* :: *'a stack*) = *0*; ¬*is-empty stack*$]\!] \Longrightarrow$ *False*
by (*simp add*: *size-empty*)

lemma *size-not-empty-2*[*simp*]: $[\![$*0* $<$ *size* (*stack* :: *'a stack*); *is-empty stack*$]\!] \Longrightarrow$ *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*]: ¬ *is-empty stack* $\Longrightarrow$ *first stack* # *list* (*pop stack*) = *list stack*
by(*induction stack rule*: *pop.induct*) *auto*

lemma *push-not-empty* [*simp*]: $[\![$¬ *is-empty stack*; *is-empty* (*push x stack*)$]\!] \Longrightarrow$ *False*
by(*induction x stack rule*: *push.induct*) *auto*

lemma *pop-list-length* [*simp*]: ¬ *is-empty stack*
$\Longrightarrow$ *Suc* (*length* (*list* (*pop stack*))) = *length* (*list stack*)
by(*induction stack rule*: *pop.induct*) *auto*

lemma *first-take*: ¬*is-empty stack* $\Longrightarrow$ [*first stack*] = *take 1* (*Stack.list stack*)
by (*simp add*: *list-empty*)

lemma *first-take-tl* [*simp*]: *0* $<$ *size big*

$\Longrightarrow$ (*first big # take count* (*tl* (*list big*))) = *take* (*Suc count*) (*list big*)
by(*induction big rule*: *Stack.first.induct*) *auto*

lemma *first-take-pop* [*simp*]: ⟦¬*is-empty stack*; *0* < *x*⟧
$\Longrightarrow$ *first stack # take* (*x* − *Suc 0*) (*list* (*pop stack*)) = *take x* (*list stack*)
by(*induction stack rule*: *pop.induct*) (*auto simp*: *take-Cons'*)

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

lemma *first-hd*: *Stack.first stack* = *hd* (*Stack.list stack*)
by(*induction stack rule*: *first.induct*)(*auto simp*: *hd-def*)

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

lemma *pop-drop*: *list* (*pop stack*) = *drop 1* (*list stack*)
by (*simp add*: *drop-Suc*)

lemma *popN-drop* [*simp*]: *list* ((*pop* ^^ *n*) *stack*) = *drop n* (*list stack*)
by(*induction n*)(*auto simp*: *drop-Suc tl-drop*)

lemma *popN-size* [*simp*]: *size* ((*pop* ^^ *n*) *stack*) = (*size stack*) − *n*
by(*induction n*) *auto*

lemma *take-first*: ⟦*0* < *size s1*; *0* < *size s2*; *take* (*size s1*) (*list s2*) = *take* (*size s2*) (*list s1*)⟧
$\Longrightarrow$ *first s1* = *first s2*
by(*induction s1 rule*: *first.induct*; *induction s2 rule*: *first.induct*) *auto*

end

13 Idle Proofs

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

lemma *push-list* [*simp*]: *list* (*push x idle*) = *x # list idle*
by(*induction idle arbitrary*: *x*) *auto*

lemma *pop-list* [*simp*]: ⟦¬ *is-empty idle*; *pop idle* = (*x*, *idle'*)⟧ $\Longrightarrow$ *x # list idle'* = *list idle*
by(*induction idle arbitrary*: *x*)(*auto simp*: *list-not-empty*)

lemma *pop-list-tl* [*simp*]:
⟦¬ *is-empty idle*; *pop idle* = (*x*, *idle'*)⟧ $\Longrightarrow$ *x # tl* (*list idle*)) = *list idle*
by(*induction idle arbitrary*: *x*) (*auto simp*: *list-not-empty*)

lemma *pop-list-tl'* [*simp*]: ⟦*pop idle* = (*x*, *idle'*)⟧ ⟹ *list idle'* = *tl* (*list idle*)
by(*induction idle arbitrary*: *x*)(*auto simp*: *drop-Suc*)

lemma *size-push* [*simp*]: *size* (*push x idle*) = *Suc* (*size idle*)
by(*induction idle arbitrary*: *x*) *auto*

lemma *size-pop* [*simp*]: ⟦¬*is-empty idle*; *pop idle* = (*x*, *idle'*)⟧ ⟹ *Suc* (*size idle'*) = *size idle*
by(*induction idle arbitrary*: *x*)(*auto simp*: *size-not-empty*)

lemma *size-pop-sub*: ⟦*pop idle* = (*x*, *idle'*)⟧ ⟹ *size idle'* = *size idle* − *1*
by(*induction idle arbitrary*: *x*) *auto*

lemma *invar-push*: *invar idle* ⟹ *invar* (*push x idle*)
by(*induction x idle rule*: *push.induct*) *auto*

lemma *invar-pop*: ⟦¬ *is-empty idle*; *invar idle*; *pop idle* = (*x*, *idle'*)⟧ ⟹ *invar idle'*
by(*induction idle arbitrary*: *x rule*: *pop.induct*) *auto*

lemma *size-empty*: *size idle* = *0* ⟷ *is-empty* (*idle* :: *'a idle*)
by(*induction idle*)(*auto simp*: *size-empty*)

lemma *size-not-empty*: *0* < *size idle* ⟷ ¬*is-empty* (*idle* :: *'a idle*)
by(*induction idle*)(*auto simp*: *size-not-empty*)

lemma *size-empty-2* [*simp*]: ⟦¬*is-empty* (*idle* :: *'a idle*); *0* = *size idle*⟧ ⟹ *False*
by (*simp add*: *size-empty*)

lemma *size-not-empty-2* [*simp*]: ⟦*is-empty* (*idle* :: *'a idle*); *0* < *size idle*⟧ ⟹ *False*

by (*simp add*: *size-not-empty*)

lemma *list-empty*: *list idle* = [] ⟷ *is-empty idle*
by(*induction idle*)(*simp add*: *list-empty*)

lemma *list-not-empty*: *list idle* ≠ [] ⟷ ¬ *is-empty idle*
by(*induction idle*)(*simp add*: *list-not-empty*)

lemma *list-empty-2* [*simp*]: ⟦*list idle* = []; ¬*is-empty* (*idle* :: *'a idle*)⟧ ⟹ *False*
using *list-empty* **by** *blast*

lemma *list-not-empty-2* [*simp*]: ⟦*list idle* ≠ []; *is-empty* (*idle* :: *'a idle*)⟧ ⟹ *False*
using *list-not-empty* **by** *blast*

lemma *list-empty-size*: *list idle* = [] ⟷ *0* = *size idle*
by (*simp add*: *list-empty size-empty*)

lemma *list-not-empty-size*: *list idle* ≠ [] ⟷ *0* < *size idle*

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

lemma *list-empty-size-2* [*simp*]: ⟦*list idle* ≠ []; *0* = *size idle*⟧ ⟹ *False*
by (*simp add*: *list-empty size-empty*)

lemma *list-not-empty-size-2* [*simp*]: ⟦*list idle* = []; *0* < *size idle*⟧ ⟹ *False*
by (*simp add*: *list-empty-size*)

end

14 Current Proofs

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

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

lemma *pop-list*: ⟦¬ *is-empty current*; *pop current* = (*x*, *current′*)⟧
⟹ *x* # *list current′* = *list current*
by(*induction current arbitrary*: *x rule*: *pop.induct*)(*auto simp*: *list-not-empty*)

lemma *pop-list-size*: ⟦*invar current*; *0* < *size current*; *pop current* = (*x*, *current′*)⟧
⟹ *x* # *list current′* = *list current*
by(*induction current arbitrary*: *x rule*: *pop.induct*)(*auto simp*: *size-not-empty list-not-empty*)

lemma *drop-first-list* [*simp*]:
¬ *is-empty current* ⟹ *list* (*drop-first current*) = *tl* (*list current*)
by(*induction current rule*: *pop.induct*)(*auto simp*: *drop-Suc*)

lemma *pop-list-2* [*simp*]:
⟦*0* < *size current*; *invar current*⟧ ⟹ *fst* (*pop current*) # *tl* (*list current*) = *list current*
by(*induction current rule*: *pop.induct*)(*auto simp*: *size-not-empty list-not-empty*)

lemma *drop-first-list-size* [*simp*]: ⟦*invar current*; *0* < *size current*⟧
⟹ *list* (*drop-first current*) = *tl* (*list current*)
by(*induction current rule*: *pop.induct*)(*auto simp*: *drop-Suc*)

lemma *invar-push*: *invar current* ⟹ *invar* (*push x current*)
by(*induction x current rule*: *push.induct*) *auto*

lemma *invar-pop*: ⟦¬ *is-empty current*; *invar current*; *pop current* = (*x*, *current′*)⟧
⟹ *invar current′*
by(*induction current arbitrary*: *x rule*: *pop.induct*) *auto*

lemma *invar-size-pop*: ⟦*0* < *size current*; *invar current*; *pop current* = (*x*, *cur-*

rent')⟧
  ⟹ *invar current'*
  by(*induction current arbitrary*: *x rule*: *pop.induct*) *auto*

lemma *invar-size-drop-first*: ⟦*0 < size current*; *invar current*⟧ ⟹ *invar* (*drop-first current*)
  using *invar-size-pop*
  by (*metis eq-snd-iff*)

lemma *invar-drop-first*: ⟦¬ *is-empty current*; *invar current*⟧ ⟹ *invar* (*drop-first current*)
  by(*induction current rule*: *pop.induct*) *auto*

lemma *push-not-empty* [*simp*]: ⟦¬ *is-empty current*; *is-empty* (*push x current*)⟧ ⟹ *False*
  by(*induction x current rule*: *push.induct*) *auto*

lemma *size-empty*: *invar* (*current* :: *'a current*) ⟹ *size current* = *0* ⟹ *is-empty current*
  by(*induction current*)(*auto simp*: *size-empty*)

lemma *size-new-empty*: *invar* (*current* :: *'a current*) ⟹ *size-new current* = *0* ⟹ *is-empty current*
  by(*induction current*)(*auto simp*: *size-empty*)

lemma *list-not-empty* [*simp*]: ⟦*list current* = []; ¬*is-empty current*⟧ ⟹ *False*
  by(*induction current*)(*auto simp*: *list-empty*)

lemma *list-size* [*simp*]: ⟦*invar current*; *list current* = []; *0 < size current*⟧ ⟹ *False*
  by(*induction current*)(*auto simp*: *size-not-empty list-empty*)

lemma *size-new-push* [*simp*]: *invar current* ⟹ *size-new* (*push x current*) = *Suc* (*size-new current*)
  by(*induction x current rule*: *push.induct*) *auto*

lemma *size-push* [*simp*]: *size* (*push x current*) = *Suc* (*size current*)
  by(*induction x current rule*: *push.induct*) *auto*

lemma *size-new-pop* [*simp*]: ⟦*0 < size-new current*; *invar current* ⟧
  ⟹ *Suc* (*size-new* (*drop-first current*)) = *size-new current*
  by(*induction current rule*: *pop.induct*) *auto*

lemma *size-pop* [*simp*]: ⟦*0 < size current*; *invar current* ⟧
  ⟹ *Suc* (*size* (*drop-first current*)) = *size current*
  by(*induction current rule*: *pop.induct*) *auto*

lemma *size-pop-suc* [*simp*]: ⟦*0 < size current*; *invar current*; *pop current* = (*x*,

current′) ⟧
  ⟹ *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′*)
⟧
  ⟹ *size current′* = *size current* − *1*
 by(*induction current rule*: *pop.induct*) *auto*

lemma *size-drop-first-sub*: ⟦*0* < *size current*; *invar current* ⟧
  ⟹ *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* ≠ [] ⟹ *reverseN n* (*tl xs*) (*hd xs* # *acc*) = *reverseN* (*Suc n*) *xs acc*
 by (*simp add*: *take-Suc*)

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

lemma *reverseN-tl-hd*: *0* < *n* ⟹ *xs* ≠ [] ⟹ *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* ⟹ *x* = *xs* ! *n* ⟹ *x* # *reverseN n xs ys* = *reverseN* (*Suc n*) *xs ys*
 by (*simp add*: *take-Suc-conv-app-nth*)

lemma *step-list* [*simp*]: *invar common* ⟹ *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: re-
verseN-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 = [])
```

```
      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
qed

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

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

```
  pop common = (x, common')
⟧ ⟹ 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 ≤ length new)
      case True

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

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

      from 1 have 0 < remained
        by auto

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

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

      with 1 False show ?thesis
```

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

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

lemma pop-list [simp]: invar common ⟹ 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
```

```
      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
  qed
 next
  case 2
  then show ?case by auto
 qed
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*) ⟹ *size common* = *0* ⟹ *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*) ⟹ *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*)
   ⟹ *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*]: ⟦*invar* (*common* :: *'a state*); *remaining-steps common* > *0*⟧
  ⟹ *Suc* (*remaining-steps* (*step common*)) = *remaining-steps common*
  by(*induction common*)(*auto split*: *current.splits*)

lemma *remaining-steps-step-sub* [*simp*]: ⟦*invar* (*common* :: *'a state*)⟧
⟹ *remaining-steps* (*step common*) = *remaining-steps common* − *1*
  by(*induction common*)(*auto split*: *current.splits*)

lemma *remaining-steps-step-0* [*simp*]: ⟦*invar* (*common* :: *'a state*); *remaining-steps common* = *0*⟧
  ⟹ *remaining-steps* (*step common*) = *0*
  by(*induction common*)(*auto split*: *current.splits*)

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

lemma *remaining-steps-pop*: ⟦*invar common*; *0* < *size common*; *pop common* = (*x*, *common'*)⟧
  ⟹ *remaining-steps common'* ≤ *remaining-steps common*
proof(*induction common rule*: *pop.induct*)
  case (*1 current idle*)
  then show *?case*
  proof(*induction idle rule*: *Idle.pop.induct*)
    case *1*
    then show *?case*
      by(*induction current rule*: *Current.pop.induct*) *auto*
  qed
next
  case (*2 current aux new moved*)
  then show *?case*
    by(*induction current rule*: *Current.pop.induct*) *auto*
qed

lemma *size-push* [*simp*]: *invar common* ⟹ *size* (*push x common*) = *Suc* (*size common*)
  by(*induction x common rule*: *push.induct*) (*auto split*: *current.splits*)

lemma *size-new-push* [*simp*]: *invar common* ⟹ *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*]: ⟦*invar common*; *0* < *size common*; *pop common* = (*x*, *common'*)⟧
  ⟹ *Suc* (*size common'*) = *size common*
proof(*induction common rule*: *Common.pop.induct*)

case (*1 current idle*)
then show *?case*
using *size-drop-first-sub*[*of current*] *Idle-Proof.size-pop-sub*[*of idle*]
by(*auto simp*: *size-not-empty split*: *prod.splits*)
next
case (*2 current aux new moved*)
then show *?case*
by(*induction current rule*: *Current.pop.induct*) *auto*
qed

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

lemma *size-size-new*: ⟦*invar* (*common* :: *'a state*); *0* < *size common*⟧ ⟹ *0* < *size-new common*
by(*cases common*) *auto*

end

16 Big Proofs

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

lemma *step-list* [*simp*]: *invar big* ⟹ *list* (*step big*) = *list big*
proof(*induction big rule*: *step-state.induct*)
case *1*
then show *?case*
by *auto*
next
case *2*

then show *?case*
by(*auto split*: *current.splits*)
next
case *3*
then show *?case*
by(*auto simp*: *rev-take take-drop drop-Suc tl-take rev-drop split*: *current.splits*)
qed

lemma *step-list-current* [*simp*]: *invar big* ⟹ *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 < *size* (*Reverse current big aux count*);
invar (*Reverse current big aux count*)
⟧ ⟹ *first current* # *list* (*Reverse* (*drop-first current*) *big aux count*) =
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* < *size old*; *0* < *remained*; *added* = *0*; *remained* − *count* ≤ *length aux*; *count* ≤ *size big*;
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* − *count*) *aux*) @ *rev* (*take remained* (*rev* (*take count* (*Stack.list big*))))⟧
⟹ *hd* (*rev* (*take* (*size old* − *size big*) *aux*) @ *rev* (*take* (*size old*) (*rev* (*Stack.list big*)))) =
(*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.absorb2 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*]: ⟦*0* < *size big*; *invar big*; *list big* = []⟧ ⟹ *False*
proof(*induction big rule*: *list.induct*)
case *1*
then show *?case*
using *list-size* **by** *auto*
next
case *2*
then show *?case*
by (*metis list.distinct*(*1*) *list-Reverse*)
qed

lemma *pop-list* [*simp*]: ⟦*0* < *size big*; *invar big*; *Big.pop big* = (*x*, *big′*)⟧
⟹ *x* # *list big′* = *list big*
proof(*induction big arbitrary*: *x rule*: *list.induct*)
case *1*
then show *?case*
by(*auto split*: *prod.splits*)
next
case *2*
then show *?case*
by (*metis Big.pop.simps*(*2*) *list-Reverse prod.inject*)
qed

lemma *pop-list-tl*: ⟦*0* < *size big*; *invar big*; *Big.pop big* = (*x*, *big′*)⟧
⟹ *Big.list big′* = *tl* (*Big.list big*)
using *pop-list cons-tl*[*of x list big′ list big*]
by *force*

lemma *invar-step*: *invar* (*big* :: *′a state*) ⟹ *invar* (*step big*)
proof(*induction big rule*: *step-state.induct*)
case *1*
then show *?case*

```
  by(auto simp: invar-step)
next
  case (2 current big aux)

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


  with 2 have ⟦current = Current extra (length extra) old remained; remained ≤
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)⟧
    ⟹ remained ≤ size old
    by(metis length-rev length-take min.absorb-iff2 size-list-length take-append)

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

  with current 2 show ?case
    by(auto simp: 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* $\Longrightarrow$ *invar* (*push x big*)
by(*induction x big rule*: *push.induct*)(*auto simp*: *invar-push split*: *current.splits*)

lemma *invar-pop*: $[\![$
0 $<$ *size big*;
invar big;
pop big = (*x*, *big'*)
$]\!] \Longrightarrow$ *invar big'*
proof(*induction big arbitrary*: *x rule*: *pop.induct*)
case (*1 state*)
then show *?case*
by(*auto simp*: *invar-pop split*: *prod.splits*)
next
case (*2 current big aux count*)
then show *?case*
proof(*induction current rule*: *Current.pop.induct*)
case (*1 added old remained*)
have *linarith*: $\bigwedge$*x y z*. *x* − *y* ≤ *z* $\Longrightarrow$ *x* − (*Suc y*) ≤ *z*
by *linarith*

have *a*: $[\![$*remained* ≤ *count* + *length aux*; *0* < *remained*; *added* = *0*; *x* = *Stack.first old*;
big' = *Reverse* (*Current* [] *0* (*Stack.pop old*) (*remained* − *Suc 0*)) *big aux count*;
count ≤ *size big*; *Stack.list old* = *rev aux* @ *Stack.list big*;
take remained (*rev aux*) @ *take* (*remained* − *length aux*) (*Stack.list big*) =
drop (*count* + *length aux* − *remained*) (*rev aux*) @
drop (*count* − *remained*) (*take count* (*Stack.list big*));
¬ *size old* ≤ *length aux* + *size big*$]\!]$
$\Longrightarrow$ *tl* (*rev aux* @ *Stack.list big*) = *rev aux* @ *Stack.list big*
by (*metis le-refl length-append length-rev size-list-length*)

have *b*: $[\![$*remained* ≤ *length* (*reverseN count* (*Stack.list big*) *aux*); *0* < *size old*;
0 < *remained*; *added* = *0*;
x = *Stack.first old*;
big' = *Reverse* (*Current* [] *0* (*Stack.pop old*) (*remained* − *Suc 0*)) *big aux count*;
remained − *count* ≤ *length aux*; *count* ≤ *size big*;
Stack.list old =
drop (*length aux* − (*size old* − *size big*)) (*rev aux*) @
drop (*size big* − *size old*) (*Stack.list big*);
take remained (*drop* (*length aux* − (*size old* − *size big*)) (*rev aux*)) @
take (*remained* + (*length aux* − (*size old* − *size big*)) − *length aux*)
(*drop* (*size big* − *size old*) (*Stack.list big*)) =
drop (*length* (*reverseN count* (*Stack.list big*) *aux*) − *remained*)

```
      (rev (reverseN count (Stack.list big) aux))⟧
    ⟹ tl (drop (length aux − (size old − size big)) (rev aux) @
          drop (size big − size old) (Stack.list big)) =
       drop (length aux − (size old − Suc (size big))) (rev aux) @
       drop (Suc (size big) − size old) (Stack.list big)
  apply(cases size old − size big ≤ length aux; cases size old ≤ size big)
  by(auto simp: tl-drop-2 Suc-diff-le le-diff-conv le-refl a)

 from 1 have remained ≤ length (reverseN count (Stack.list big) aux)
  by(auto)

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

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

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

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

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

```
    by auto
  qed
qed

lemma list-current-size: ⟦0 < size big; list-current big = []; invar big⟧ ⟹ False
proof(induction big rule: list-current.induct)
  case 1
  then show ?case
    using list-current-size
    by simp
next
  case (2 current uu uv uw)
  then show ?case
    apply(cases current)
    by(auto simp: Stack-Proof.size-not-empty Stack-Proof.list-empty)
qed

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

lemma size-empty: ⟦invar (big :: 'a state); size big = 0⟧ ⟹ 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]: ⟦invar (big :: 'a state); remaining-steps big >
0⟧
   ⟹ 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]: ⟦invar (big :: 'a state); remaining-steps big
= 0⟧
    ⟹ remaining-steps (step big) = 0
  by(induction big)(auto split: current.splits)

lemma remaining-steps-push: invar big ⟹ remaining-steps (push x big) = re-
maining-steps big
  by(induction x big rule: push.induct)(auto split: current.splits)

lemma remaining-steps-pop: ⟦invar big; 0 < size big; pop big = (x, big')⟧
   ⟹ remaining-steps big' ≤ 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* $\Longrightarrow$ *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* $\Longrightarrow$ *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'*)⟧
$\Longrightarrow$ *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'*)⟧
$\Longrightarrow$ *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*⟧ $\Longrightarrow$ *0* < *size-new big*
by(*induction big*)(*auto simp*: *size-size-new*)

end

17 Small Proofs

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

lemma *step-size* [*simp*]: *invar* (*small* :: *'a state*) ⟹ *size* (*step small*) = *size small*
by(*induction small rule*: *step-state.induct*)(*auto split*: *current.splits*)

lemma *size-empty*: *invar* (*small* :: *'a state*) ⟹ *size small* = *0* ⟹ *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* ⟹ *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* ⟹ *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'*)⟧
⟹ *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'*)⟧
⟹ *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); 0 < size small⟧ ⟹ 0 < size-new
small
  by(induction small)(auto simp: size-size-new)

lemma step-list-current [simp]: invar small ⟹ 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⟧ ⟹ 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: (0 < size big) = (¬ is-empty big)
    by (simp add: Stack-Proof.size-not-empty)

  have ¬ is-empty big
     ⟹ 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) ⟹ 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 ⟹ list (push x
small) = x # list small
  by auto

lemma push-list-reverse2 [simp]: small = (Reverse2 current auxS big newS count)
  ⟹ 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);
  ¬is-empty small;
  invar small;
  pop small = (x, small')
⟧ ⟹ 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*]: ⟦*invar small*; *0* < *size small*; *Small.pop small* = (*x*, *small′*)⟧
  ⟹ *x* # *list-current small′* = *list-current small*
proof(*induction small arbitrary*: *x rule*: *pop.induct*)
  case (*1 state*)
  then show *?case*
    by(*auto simp*: *pop-list-current split*: *prod.splits*)
next
  case (*2 current small auxS*)
  then have *invar current*
    by(*auto split*: *current.splits*)

  with *2* **show** *?case*
    by *auto*
next
  case (*3 current auxS big newS count*)
  then show *?case*
  proof(*induction current rule*: *Current.pop.induct*)
    case (*1 added old remained*)
    then have ¬*is-empty old*
      by(*auto simp*: *Stack-Proof.size-not-empty*)

    with *1* **show** *?case*
      by(*auto simp*: *rev-take drop-Suc drop-tl*)
  next
    case *2*
    then show *?case*
      by *auto*
  qed
qed

lemma *list-current-size* [*simp*]: ⟦*0* < *size small*; *list-current small* = []; *invar small*⟧ ⟹ *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 < size (Reverse2 current auxS big newS count);
  invar (Reverse2 current auxS big newS count)
⟧ ⟹
    fst (Current.pop current) # Small.list (Reverse2 (drop-first current) auxS big
newS count) =
   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:
    invar states ⟹ list-big-first states = list-current-big-first states
  using app-rev
  by(cases states)(auto split: prod.splits)

lemma step-lists [simp]: invar states ⟹ lists (step states) = lists states
proof(induction states rule: lists.induct)
  case (1 dir currentB big auxB count currentS small auxS)
  then show ?case
  proof(induction
    (States dir (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 < size big
      by(cases currentB) auto

    then have big-not-empty: Stack.list big ≠ []
      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 re-
verseN-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*) ⟷
list-big-first (*States dir big small*) = *list-current-big-first* (*States dir big small*)
using *app-rev*
by(*auto split*: *prod.splits*)

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

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

```
 ⟹ invar big' ∧ invar small
  by(auto simp: Big-Proof.invar-pop)

lemma invar-pop-big-aux: ⟦
  invar (States dir big small);
  0 < size big;
  Big.pop big = (x, big')⟧
 ⟹ (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
     )
  by(auto split: Big.state.splits Small.state.splits prod.splits)

lemma invar-pop-big: ⟦
  invar (States dir big small);
  0 < size big;
  Big.pop big = (x, big')⟧
 ⟹ invar (States dir big' small)
  using invars-pop-big[of dir big small x big']
        lists-small-first-pop-big[of dir big small x big']
        invar-pop-big-aux[of dir big small x big']
  by auto

lemma invars-pop-small: ⟦
  invar (States dir big small);
  0 < size small;
  Small.pop small = (x, small')⟧
 ⟹  invar big  ∧ invar small'
  by(auto simp: Small-Proof.invar-pop)

lemma invar-pop-small-aux: ⟦
  invar (States dir big small);
  0 < size small;
  Small.pop small = (x, 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
     )
proof(induction small rule: Small.pop.induct)
  case 1
  then show ?case
    by(auto split: Big.state.splits Small.state.splits prod.splits)
next
```

```
  case (2 current)
  then show ?case
  proof(induction current rule: Current.pop.induct)
    case 1
    then show ?case
      by(auto split: Big.state.splits)
  next
    case 2
    then show ?case
      by(auto split: Big.state.splits)
  qed
next
  case 3
  then show ?case
    by(auto split: Big.state.splits)
qed

lemma invar-pop-small: ⟦
    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*⟧ ⟹ *invar big* ∧ *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* ⟹
list-small-first (*step states*) = *list-current-small-first* (*step states*)
using *step-lists-current step-lists invar-states.elims*(*2*)
by *fastforce*

lemma *invar-step-aux*: *invar states* $\Longrightarrow$(*case step states of*
(*States - (Reverse - big - count) (Reverse1 (Current - - old remained) small -*)) $\Rightarrow$
size big $-$ *count* $=$ *remained* $-$ *size old* $\wedge$ *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*) $\Longrightarrow$ *invar* (*step states*)
using *invar-step-aux*[*of states*] *step-lists-small-first*[*of states*]
by(*cases step states*)(*auto simp*: *step-invars*)

lemma *step-consistent* [*simp*]:
⟦$\bigwedge$*states*. *invar* (*states* :: *'a states*) $\Longrightarrow$ *P* (*step states*) = *P states*; *invar states*⟧
$\Longrightarrow$ *P states* = *P* ((*step* ^^*n*) *states*)
by(*induction n arbitrary*: *states*)
(*auto simp*: *States-Proof.invar-step funpow-swap1*)

lemma *step-consistent-2*:
⟦$\bigwedge$*states*. ⟦*invar* (*states* :: *'a states*); *P states*⟧ $\Longrightarrow$ *P* (*step states*); *invar states*; *P states*⟧
$\Longrightarrow$ *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*) $\Longrightarrow$ *size-ok' states steps*
by(*induction states steps rule*: *size-ok'.induct*) *auto*

lemma *size-ok'-decline*: *size-ok' states x* $\Longrightarrow$ *x* $\geq$ *y* $\Longrightarrow$ *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*⟧
$\Longrightarrow$ *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*⟧
$\Longrightarrow$ *remaining-steps* ((*step* ^^ *n*) *states*) = *0*

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 states rule*: *step-states.induct*)
case *1*
then show *?case*
by(*auto simp*: *max-def split*: *Big.state.splits Small.state.splits current.splits*)
next
case (*2-1 - - - - - small*)
then show *?case*
by(*cases small*)(*auto split*: *current.splits*)
next
case (*2-2 dir big small*)
then show *?case*
proof(*cases small*)
case (*Reverse2 current auxS big newS count*)
with *2-2* **show** *?thesis*
using *Stack-Proof.size-empty-2*[*of big*]
by(*cases current*) *auto*
qed *auto*
next
case (*2-3 dir big current auxS big' newS count*)
then show *?case*
proof(*induction big*)
case *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*]:
⟦*invar* (*states* :: 'a *states*); *remaining-steps states* ≤ *n*⟧
⟹ *remaining-steps* ((*step* ^^ *n*) *states*) = *0*
by(*induction n arbitrary*: *states*)(*auto simp*: *funpow-swap1 invar-step*)

lemma *remaining-steps-n-steps-plus* [*simp*]:
⟦*n* ≤ *remaining-steps states*; *invar* (*states* :: 'a *states*)⟧
⟹ *remaining-steps* ((*step* ^^ *n*) *states*) + *n* = *remaining-steps states*
by(*induction n arbitrary*: *states*)(*auto simp*: *funpow-swap1 invar-step*)

lemma *remaining-steps-n-steps-sub* [*simp*]: *invar* (*states* :: 'a *states*)
⟹ *remaining-steps* ((*step* ^^ *n*) *states*) = *remaining-steps states* − *n*
by(*induction n arbitrary*: *states*)(*auto simp*: *funpow-swap1 invar-step*)

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

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

lemma *step-size-new-big* [*simp*]:
⟦*invar* (*States dir big small*); *step* (*States dir big small*) = *States dir' big' small'*⟧
⟹ *size-new big'* = *size-new big*
proof(*induction States dir big small rule*: *step-states.induct*)
case *1*
then show *?case*

```
    by(auto split: current.splits)
next
  case 2-1
  then show ?case
    by auto
next
  case 2-2
  then show ?case
    by auto
next
  case 2-3
  then show ?case
    by(auto split: Big.state.splits)
next
  case 2-4
  then show ?case
    by(auto split: Big.state.splits)
qed

lemma step-size-new-big-2 [simp]:
 invar states ⟹ 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 ?case
    by(auto split: Small.state.splits current.splits)
next
  case 2-3
  then show ?case
    by(auto split: current.splits Big.state.splits)
next
  case 2-4
  then show ?case
    by(auto split: Big.state.splits)
qed

lemma step-size-big-2 [simp]:
 invar states ⟹ size-big (step states) = size-big states
  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 ≤ 3 * size-new small
⟧ ⟹ size-new big' + remaining-steps (States dir' big' small') + 2 ≤ 3 * size-new
small'
  using step-size-new-small step-size-new-big remaining-steps-decline
  by (smt (verit, ccfv-SIG) add.commute le-trans nat-add-left-cancel-le)

lemma step-size-ok-2: ⟦
  invar (States dir big small);
  step (States dir big small) = States dir' big' small';
  size-new small + remaining-steps (States dir big small) + 2 ≤ 3 * size-new big
⟧ ⟹ size-new small' + remaining-steps (States dir' big' small') + 2 ≤ 3 *
size-new big'
  using remaining-steps-decline step-size-new-small step-size-new-big
  by (smt (verit, best) add-le-mono le-refl le-trans)

lemma step-size-ok-3: ⟦
  invar (States dir big small);
  step (States dir big small) = States dir' big' small';
  remaining-steps (States dir big small) + 1 ≤ 4 * size small
⟧ ⟹ remaining-steps (States dir' big' small') + 1 ≤ 4 * size small'
  using remaining-steps-decline step-size-small
  by (metis Suc-eq-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 ≤ 4 * size big
⟧ ⟹ remaining-steps (States dir' big' small') + 1 ≤ 4 * size big'
  using remaining-steps-decline step-size-big
  by (metis (no-types, lifting) add-mono-thms-linordered-semiring(3) order.trans)
```

lemma *step-size-ok*: ⟦*invar states*; *size-ok states*⟧ ⟹ *size-ok* (*step states*)
using *step-size-ok-1 step-size-ok-2 step-size-ok-3 step-size-ok-4*
by (*smt* (*verit*) *invar-states.elims*(*1*) *size-ok'.elims*(*3*) *size-ok'.simps*)

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

lemma *step-n-size-small* [*simp*]: ⟦
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′*
⟧ ⟹ *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′*
⟧ ⟹ *size-new small′* = *Suc* (*size-new small*)
by (*metis Small-Proof.size-new-push invar-states.simps invar-push-small step-n-size-new-small*)

lemma *step-n-push-size-big* [*simp*]: ⟦
invar (*States dir big small*);
(*step* ^^ *n*) (*States dir* (*Big.push x big*) *small*) = *States dir′ big′ small′*
⟧ ⟹ *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′*
⟧ ⟹ *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′*
⟧ ⟹ *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′*
⟧ ⟹ *Suc* (*size-new small′*) = *size-new small*
using *invar-pop-small size-new-pop step-n-size-new-small size-size-new*
by (*metis* (*no-types*, *lifting*) *invar-states.simps*)

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

$\rrbracket \Longrightarrow Suc\ (size\ big') = size\ big$
using *invar-pop-big Big-Proof.size-pop step-n-size-big*
by *fastforce*

lemma *step-n-pop-size-new-big*: $[\![$
invar (*States dir big small*);
0 < *size big*; *Big.pop big* = (*x*, *bigP*);
(*step* ^^ *n*) (*States dir bigP small*) = *States dir' big' small'*
$\rrbracket \Longrightarrow Suc\ (size\text{-}new\ big') = size\text{-}new\ big$
using *invar-pop-big Big-Proof.size-new-pop step-n-size-new-big Big-Proof.size-size-new*
by (*metis* (*no-types*, *lifting*) *invar-states.simps*)

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

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

lemma *remaining-steps-pop-big*:
$[\![$*invar* (*States dir big small*); *0* < *size big*; *Big.pop big* = (*x*, *bigP*)$]\!]$
$\Longrightarrow$ *remaining-steps* (*States dir bigP small*) $\leq$ *remaining-steps* (*States dir big small*)
proof(*induction big rule*: *Big.pop.induct*)
case (*1 state*)
then show *?case*
proof(*induction state rule*: *Common.pop.induct*)
case (*1 current idle*)
then show *?case*
by(*cases idle*)(*auto split*: *Small.state.splits*)
next
case (*2 current aux new moved*)
then show *?case*

```
    by(induction current rule: Current.pop.induct)(auto split: Small.state.splits)
  qed
next
  case (2 current big aux count)
  then show ?case
  proof(induction current rule: Current.pop.induct)
    case 1
    then show ?case
      by(auto split: Small.state.splits current.splits)
  next
    case 2
    then show ?case
      by(auto split: Small.state.splits current.splits simp del: reverseN-def)
  qed
qed

lemma remaining-steps-push-big [simp]: invar (States dir big small)
   ⟹ 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 ≤ remaining-steps (States dir big small);
  (step^^4) (States dir (Big.push x big) small) = States dir' big' small'⟧
    ⟹ remaining-steps (States dir' big' small') = 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 ≤ 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') = 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 < size big;
  Big.pop big = (x, bigP);
  4 ≤ remaining-steps (States dir bigP small);
  (step^^4) (States dir bigP small) = States dir' big' small'
⟧ ⟹ remaining-steps (States dir' big' small') ≤ 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);
```

0 < size small;
Small.pop small = (x, smallP);
4 ≤ remaining-steps (States dir big smallP);
(step^^4) (States dir big smallP) = States dir′ big′ small′
⟧ ⟹ *remaining-steps (States dir′ big′ small′) ≤ remaining-steps (States dir big small) − 4*
by (*metis add-le-imp-le-diff invar-pop-small remaining-steps-n-steps-plus remaining-steps-pop-small*)

lemma *step-4-pop-small-size-ok-1*: ⟦
invar (States dir big small);
0 < size small;
Small.pop small = (x, smallP);
4 ≤ remaining-steps (States dir big smallP);
(step^^4) (States dir big smallP) = States dir′ big′ small′;
*remaining-steps (States dir big small) + 1 ≤ 4 * size small*
⟧ ⟹ *remaining-steps (States dir′ big′ small′) + 1 ≤ 4 * size small′*
by (*smt (verit, ccfv-SIG) add.left-commute add.right-neutral add-le-cancel-left distrib-left-numeral dual-order.trans invar-pop-small le-add-diff-inverse2 mult.right-neutral plus-1-eq-Suc remaining-steps-n-steps-sub remaining-steps-pop-small step-n-pop-size-small*)

lemma *step-4-pop-big-size-ok-1*: ⟦
invar (States dir big small);
0 < size big; *Big.pop big = (x, bigP)*;
4 ≤ remaining-steps (States dir bigP small);
(step^^4) (States dir bigP small) = States dir′ big′ small′;
*remaining-steps (States dir big small) + 1 ≤ 4 * size small*
⟧ ⟹ *remaining-steps (States dir′ big′ small′) + 1 ≤ 4 * size small′*
by (*smt (verit, ccfv-SIG) add-leE add-le-cancel-right invar-pop-big order-trans remaining-steps-pop-big step-n-size-small remaining-steps-n-steps-plus*)

lemma *step-4-pop-small-size-ok-2*: ⟦
invar (States dir big small);
0 < size small;
Small.pop small = (x, smallP);
4 ≤ remaining-steps (States dir big smallP);
(step^^4) (States dir big smallP) = States dir′ big′ small′;
*remaining-steps (States dir big small) + 1 ≤ 4 * size big*
⟧ ⟹ *remaining-steps (States dir′ big′ small′) + 1 ≤ 4 * size big′*
by (*smt (z3) add.commute add-leE invar-pop-small le-add-diff-inverse2 nat-add-left-cancel-le remaining-steps-n-steps-sub step-n-size-big remaining-steps-pop-small*)

lemma *step-4-pop-big-size-ok-2*:
assumes
invar (States dir big small)
0 < size big
Big.pop big = (x, bigP)
remaining-steps (States dir bigP small) ≥ 4
((step ^^ 4) (States dir bigP small)) = States dir′ big′ small′

remaining-steps (*States dir big small*) + *1* ≤ *4* ∗ *size big*
shows
remaining-steps (*States dir′ big′ small′*) + *1* ≤ *4* ∗ *size big′*
proof −
from *assms* **have** *remaining-steps* (*States dir bigP small*) + *1* ≤ *4* ∗ *size big*
by (*meson add-le-cancel-right order.trans remaining-steps-pop-big*)

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

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

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

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

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

big'
proof−
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 ≤ *remaining-steps* (*States dir big small*);
(*step*^^*4*) (*States dir* (*Big.push x big*) *small*) = *States dir' big' small'*;
remaining-steps (*States dir big small*) + *1* ≤ *4* ∗ *size small*
⟧ ⟹ *remaining-steps* (*States dir' big' small'*) + *1* ≤ *4* ∗ *size small'*
by (*smt* (*verit*, *ccfv-SIG*) *Nat.le-diff-conv2 add-leD2 invar-push-big le-add1 le-add-diff-inverse2 remaining-steps-n-steps-sub remaining-steps-push-big step-n-size-small*)

lemma *step-4-push-small-size-ok-2*: ⟦
invar (*States dir big small*);
4 ≤ *remaining-steps* (*States dir big small*);
(*step*^^*4*) (*States dir big* (*Small.push x small*)) = *States dir' big' small'*;
remaining-steps (*States dir big small*) + *1* ≤ *4* ∗ *size big*
⟧ ⟹ *remaining-steps* (*States dir' big' small'*) + *1* ≤ *4* ∗ *size big'*
by (*metis* (*full-types*) *Suc-diff-le Suc-eq-plus1 invar-push-small less-Suc-eq-le less-imp-diff-less step-4-remaining-steps-push-small step-n-size-big*)

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

lemma *step-4-push-small-size-ok-3-aux*: ⟦
4 ≤ *remaining-steps* (*States dir big small*);
size-new small + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new big*
⟧ ⟹ *Suc* (*size-new small*) + (*remaining-steps* (*States dir big small*) − *4*) + *2* ≤ *3* ∗ *size-new big*
using *distrib-left dual-order.trans le-add-diff-inverse2* **by** *force*

lemma *step-4-push-small-size-ok-3*: ⟦
invar (*States dir big small*);
4 ≤ *remaining-steps* (*States dir big small*);
(*step*^^*4*) (*States dir big* (*Small.push x small*)) = *States dir' big' small'*;
size-new small + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new big*
⟧ ⟹ *size-new small'* + *remaining-steps* (*States dir' big' small'*) + *2* ≤ *3* ∗ *size-new big'*
using *step-n-size-new-big step-n-push-size-new-small step-4-remaining-steps-push-small*
by (*metis invar-push-small step-4-push-small-size-ok-3-aux*)

lemma *step-4-push-big-size-ok-3-aux*: ⟦
4 ≤ *remaining-steps* (*States dir big small*);
size-new small + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new big*
⟧ ⟹ *size-new small* + (*remaining-steps* (*States dir big small*) − *4*) + *2* ≤ *3* ∗ *Suc* (*size-new big*)
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 ≤ *remaining-steps* (*States dir big small*);
(*step*^^*4*) (*States dir* (*Big.push x big*) *small*) = *States dir′ big′ small′*;
size-new small + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new big*
⟧ ⟹ *size-new small′* + *remaining-steps* (*States dir′ big′ small′*) + *2* ≤ *3* ∗ *size-new big′*
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 ≤ *remaining-steps* (*States dir big small*);
size-new big + *remaining-steps* (*States dir big small*) + *2* ≤ *3* ∗ *size-new small*
⟧ ⟹ *size-new big* + (*remaining-steps* (*States dir big small*) − *4*) + *2* ≤ *3* ∗ *Suc* (*size-new small*)
using *distrib-left dual-order.trans le-add-diff-inverse2* **by** *force*

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

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

$\llbracket \Longrightarrow$ *size-ok* ((*step*^^*4*) (*States dir big* (*Small.push x small*)))
using *step-4-push-small-size-ok-1 step-4-push-small-size-ok-2 step-4-push-small-size-ok-3 step-4-push-small-size-ok-4*
by (*smt* (*verit*) *size-ok'.elims*(*3*) *size-ok'.simps*)

lemma *step-4-push-big-size-ok*: $\llbracket$
invar (*States dir big small*);
4 $\leq$ *remaining-steps* (*States dir big small*);
size-ok (*States dir big small*)
$\rrbracket \Longrightarrow$ *size-ok* ((*step*^^*4*) (*States dir* (*Big.push x big*) *small*))
using *step-4-push-big-size-ok-1 step-4-push-big-size-ok-2 step-4-push-big-size-ok-3 step-4-push-big-size-ok-4*
by (*smt* (*verit*) *size-ok'.elims*(*3*) *size-ok'.simps*)

lemma *step-4-pop-small-size-ok*: $\llbracket$
invar (*States dir big small*);
0 $<$ *size small*;
Small.pop small = (*x*, *smallP*);
4 $\leq$ *remaining-steps* (*States dir big smallP*);
size-ok (*States dir big small*)
$\rrbracket \Longrightarrow$ *size-ok* ((*step*^^*4*) (*States dir big smallP*))
by (*smt* (*verit*) *size-ok'.elims*(*3*) *size-ok'.simps step-4-pop-small-size-ok-1 step-4-pop-small-size-ok-2 step-4-pop-small-size-ok-3 step-4-pop-small-size-ok-4*)

lemma *step-4-pop-big-size-ok*: $\llbracket$
invar (*States dir big small*);
0 $<$ *size big*; *Big.pop big* = (*x*, *bigP*);
4 $\leq$ *remaining-steps* (*States dir bigP small*);
size-ok (*States dir big small*)
$\rrbracket \Longrightarrow$ *size-ok* ((*step*^^*4*) (*States dir bigP small*))
using *step-4-pop-big-size-ok-1 step-4-pop-big-size-ok-2 step-4-pop-big-size-ok-3 step-4-pop-big-size-ok-4*
by (*smt* (*verit*) *size-ok'.elims*(*3*) *size-ok'.simps*)

lemma *size-ok-size-small*: *size-ok* (*States dir big small*) $\Longrightarrow$ *0* $<$ *size small*
by *auto*

lemma *size-ok-size-big*: *size-ok* (*States dir big small*) $\Longrightarrow$ *0* $<$ *size big*
by *auto*

lemma *size-ok-size-new-small*: *size-ok* (*States dir big small*) $\Longrightarrow$ *0* $<$ *size-new small*
by *auto*

lemma *size-ok-size-new-big*: *size-ok* (*States dir big small*) $\Longrightarrow$ *0* $<$ *size-new big*
by *auto*

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

lemma *step-same*: *step* (*States dir big small*) = *States dir′ big′ small′* ⟹ *dir* = *dir′*
by(*induction States dir big small rule*: *step-states.induct*) *auto*

lemma *step-n-same*: (*step*^^*n*) (*States dir big small*) = *States dir′ big′ small′* ⟹ *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* ⟹ *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 *list-small-first* (*step* (*States Left big small*)) =
Small.list-current small @ *rev* (*Big.list-current big*)
using *1 step-lists* **by** *fastforce*

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

with *1* **show** *?case*
by *auto*
next
case (*2 big small*)
then have *a*: *list-big-first* (*States Right big small*) =
Big.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 ⟹ 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) ⟹ invar ((step^^n) states)
  by (simp add: invar-step step-consistent-2)

lemma step-n-size-ok': ⟦invar states; size-ok' states x⟧ ⟹ 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)
⟧ ⟹ size-ok ((step ^^ n) states)
  by (simp add: step-n-size-ok')
```

lemma *remaining-steps-idle*: *invar states*
⟹ remaining-steps states = 0 ⟷ (
case states of
States - (Big.Common (Common.Idle - -)) (Small.Common (Common.Idle - -)) ⇒ True
| - ⇒ False)
by(*cases states*)
(*auto split*: *Big.state.split Small.state.split Common.state.split current.splits*)

lemma *remaining-steps-idle′*:
⟦*invar* (*States dir big small*); *remaining-steps* (*States dir big small*) = *0*⟧
⟹ ∃ *big-current big-idle small-current small-idle. States dir big small = States dir*
(*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.splits*)

end

19 Dequeue Proofs

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

lemma *list-deqL′* [*simp*]: ⟦*invar deque*; *listL deque* ≠ []; *deqL′ deque* = (*x*, *deque′*)⟧
⟹ 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(cases 3 * length-left' ≥ 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(cases length-left' ≥ 1)
      case True
      let ?big = Reverse (Current [] 0 right (size right − Suc length-left'))
                         right [] (size right − Suc length-left')
        let ?small = Reverse1 (Current [] 0 stack-left' (Suc (2 * 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 ((step^^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 states.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' small
  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 # States.listL ?states-stepped = Big.list-current big @ rev (Small.list-current small)
using *States-Proof.step-n-listL[of ?states-new 4] invar listL*
by *auto*

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

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

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

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

with *6* **show** *?case* **by** *auto*
qed *auto*

lemma *list-deqL* [*simp*]:
⟦*invar deque*; *listL deque* ≠ []⟧ ⟹ *listL (deqL deque) = tl (listL deque)*
using *cons-tl*[*of fst (deqL′ deque) listL (deqL deque) listL deque*]
by(*auto split*: *prod.splits*)

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

lemma *invar-deqL*:
⟦*invar deque*; ¬ *is-empty deque*⟧ ⟹ *invar (deqL deque)*
proof(*induction deque rule*: *deqL′.induct*)
case (*4 left right length-right*)
then obtain *x left′* **where** *pop-left*[*simp*]: *Idle.pop left = (x, left′)*
by *fastforce*

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

from *4* **have** *invar-left′*: *invar left′ invar left*
using *Idle-Proof.invar-pop* **by** *fastforce+*

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

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

  from 4 have list: x # Idle.list left' = Idle.list left
    using Idle-Proof.pop-list[of left x left']
    by auto

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

      then have invar-stepped: invar ((step^^6) ?states)
        using invar-step-n by blast

      from 4 Start-Transformation True
      have remaining-steps: 6 < remaining-steps ?states
        by auto

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

      from 4 Start-Transformation True
      have size-ok': size-ok' ?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!: Com-
mon.state.split)
  next
    case False
    then have remaining-steps-stepped: remaining-steps ?states-stepped = 0
      using invar by(auto simp del: stepped)

    then obtain small-current small-idle big-current big-idle where idle [simp]:
      States Left big-stepped small-stepped =
      States Left
          (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]: ¬is-empty big-idle
      by (auto simp: Idle-Proof.size-not-empty)

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

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

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

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

  let ?states = States Right big' small
  let ?states-stepped = (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: 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 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 ⟹ 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 left x] 5 by auto

  show ?case
  proof(cases length-left' ≤ 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]
```

```
    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 ?states
      using Stack-Proof.size-not-empty[of right]
      by auto

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

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

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

    from False show ?thesis
      using invar-stepped remaining-steps-stepped size-ok-stepped
      by(auto simp: Let-def)
  qed
next
  case (6 x big small)
  let ?small = Small.push x small
  let ?states = States Left big ?small
  let ?states-stepped = (step^^4) ?states

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

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

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

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

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

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

```
      remaining-steps-push-small[of Left big small x]
    by auto

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

  then obtain small-current small-idle big-current big-idle where idle [simp]:
   ?states-stepped =
   States Left
      (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-Dequeue RealTimeDeque-Enqueue*

begin

lemma *swap-lists-left*: *invar* (*States Left big small*) ⟹
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*) ⟹
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* ⟹ *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* ⟹ *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* ⟹ *invar* (*swap q*)
by(*induction q rule*: *swap.induct*) (*auto simp*: *lists-same split*: *prod.splits*)

lemma *listL-is-empty*: *invar deque* ⟹ *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

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