Real-Time Double-Ended Queue Verified (Proof Pearl)

Balazs Toth
Department of Computer Science, Technische Univerität München, Germany

Tobias Nipkow
Department of Computer Science, Technische Univerität München, Germany

Abstract

We present the first verification of the real-time doubled-ended queue by Chuang and Goldberg where all operations take constant time. The main contributions are the full system invariant, the precise definition of all abstraction functions, the structure of the proof and the main lemmas.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Theory of computation → Data structures design and analysis; Software and its engineering → Functional languages

Keywords and phrases Double-ended queue, data structures, verification, Isabelle

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.29

Funding *Tobias Nipkow*: Research partially supported by DFG Koselleck grant NI 491/16-1.

1 Introduction

Based on the work of Chuang and Goldberg [2] we implement and formally verify a double-ended queue (*deque*) in a purely functional language such that each *enqueue* and *dequeue* operation on either end takes $\mathcal{O}(1)$ time in the worst case. This is what *real-time* means. Operations on previous versions of a deque are in constant time since purely functional data structures are persistent by default.

The deque implementation by Chuang and Goldberg consists of two stacks, with each stack corresponding to one of the two ends of the deque. These two stacks are balanced at all time, meaning that the bigger stack is never more than three times bigger than the smaller stack. The *enqueue* and *dequeue* operations use the respective stacks (by pushing and popping). The deque maintains its size invariant by rebalancing the two ends. Since such a rebalancing takes time $\mathcal{O}(n)$, it distributes a constant fraction of the rebalancing steps on the *enqueue* and *dequeue* operations before the invariant can be violated again. This achieves worst-case and not just amortized constant time for each operation. We show the detailed implementation in Section 5.

Chuang and Goldberg [2, p.292] describe the main size invariant of a real-time deque and explain how this invariant is re-established via rebalancing of the two ends. But to formally verify the implementation, we need much more detailed invariants, which also capture the state during rebalancing. For example, an explicit measure of the remaining rebalancing steps is needed. We verify the implementation w.r.t. a formal specification of deques. The verification uses the interactive theorem prover Isabelle/HOL [12, 11]. The Isabelle theories are available online [14] and comprise 4400 lines of definitions and proofs. Some of the names in this paper have been modified (mostly shortened) for presentation reasons.

14th International Conference on Interactive Theorem Proving (ITP 2023).
Editors: Adam Naumowicz and René Thiemann; Article No. 29; pp. 29:1–29:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Related Work

The quest for efficient functional queues started with the two-stack implementation by Burton [1] where all operations take amortized constant time. Hood and Melville [7] show how to obtain a real-time implementation by distributing the work of moving elements from one of the two stacks to the other one over a whole sequence of *enqueue* and *dequeue* operations. A verification of Hood and Melville's queue can be found elsewhere [4, 10]. The principles of the real-time deque were already described by Hood [6], apparently unbeknownst to Chuang and Goldberg. Madhavan and Kuncak [9] automatically verified the amortized constant-time complexity of a simpler deque (see start of Section 4).

Okasaki [13] shows how to obtain simpler implementations of real-time queues and deques by relying on lazy evaluation. The resource requirements of his code were analyzed semi-automatically by Madhavan *et al.* [8].

2 Preliminaries

Isabelle types are built from type variables, e.g. $'a$, and (postfix) type constructors, e.g. $'a\ list$; the function type arrow is $\Rightarrow$. The notation $t :: \tau$ means that term t has type τ. The notation f^n, where $f :: \tau \Rightarrow \tau$, is the n-fold composition of f with itself.

Type $'a\ list$ are lists of elements of type $'a$. They come with the following vocabulary: $(\#)$ (list constructor), $(@)$ (append), $|xs|$ (length of list xs), $rev\ xs$ (reverse of xs), hd (head), tl (tail, where $tl\ [] = []$), $take\ n\ xs$ (take the first n elements of list xs), $drop\ n\ xs$ (drop the first n elements of list xs), $take\_last\ n\ xs$ (take the last n elements of list xs), and other self-explanatory notation. Type nat are the natural numbers. Pairs come with the projection functions fst and snd. Logical equivalence is written $=$ instead of $\longleftrightarrow$. A yellow background marks the code of the actual implementation of the data structure, while code without a background is just used for the verification.

3 Specification

The interface is comprised of the functions

$$
\begin{array}{ll}
empty :: 'q & is\_empty :: 'q \Rightarrow bool \\
enqL :: 'a \Rightarrow 'q \Rightarrow 'q & enqR :: 'a \Rightarrow 'q \Rightarrow 'q \\
deqL :: 'q \Rightarrow 'q & deqR :: 'q \Rightarrow 'q \\
firstL :: 'q \Rightarrow 'a & firstR :: 'q \Rightarrow 'a
\end{array}
$$

where $'q$ is the type of deques and $'a$ the type of elements. They allow enqueuing and dequeuing elements on both ends (as indicated by the L/R suffixes). We express the specification using an abstraction function $listL :: 'q \Rightarrow 'a\ list$

$$
\begin{array}{ll}
listL\ empty = [] & is\_empty\ q = (listL\ q = []) \\
listL\ (enqL\ x\ q) = x \# listL\ q & listR\ (enqR\ x\ q) = x \# listR\ q \\
listL\ q \neq [] \longrightarrow listL\ (deqL\ q) = tl\ (listL\ q) & listR\ q \neq [] \longrightarrow listR\ (deqR\ q) = tl\ (listR\ q) \\
listL\ q \neq [] \longrightarrow firstL\ q = hd\ (listL\ q) & listR\ q \neq [] \longrightarrow firstR\ q = hd\ (listR\ q)
\end{array}
$$

where $listR\ q = rev\ (listL\ q)$. The above properties express that $listL$ and $listR$ are homomorphisms from deques to lists. There is also an invariant $invar :: 'q \Rightarrow bool$ and $invar\ q$ is an additional precondition of the above equations, except for $listL\ empty = []$. All operations are required to preserve $invar$ – we do not show the corresponding propositions.

4 Abstract Description of Implementation

A deque is represented by two stacks, one for each end of the deque. Things work well as long as both stacks remain non-empty. As soon as one becomes empty and a *deq* (= *pop*) operation is to be performed, we need to move part of the other stack over to the empty side first. It can be shown that if the (bottom) half of the non-empty stack is moved (and reversed), this leads to an implementation with amortized constant-time operations.

To achieve worst-case constant-time complexity the invariant $n \geq m \geq 1 \wedge 3 * m \geq n$ is maintained where m and n are the sizes of the smaller and the bigger stacks S and B. If the length of the deque is ≤ 3, it is represented by a single list, all operations are trivially constant-time and the above invariant does not apply. We focus on the two-stack situation. The invariant can be violated by dequeuing on the smaller stack or enqueuing on the larger stack. Let S and B be the stacks after the violating operation and let $m = |S|$ and $n = |B|$. Then $3 * m < n$ and either $3 * (m + 1) \geq n$ (pop) or $3 * m \geq n - 1$ (push), i.e. $n = 3 * m + k$ where $1 \leq k \leq 3$. Thus there are P and Q such that $B = P \mathbin{@} Q$ and $|Q| = m + 1$. Now we transform S into $S \mathbin{@} \mathit{rev}\ Q$ and B into P in 5 phases. In each phase, we pop the elements off one stack and push them onto another stack, thus reversing the order.

*Big*1	Pop the top $2 * m + k - 1$ elements off B onto a new stack rP: $B = Q$ and $rP = \mathit{rev}\ P$
*Small*1	Reverse S onto a new stack rS: $rS = \mathit{rev}\ S$
*Big*2	Reverse rP onto a new stack B': $B' = P$
*Small*2	Reverse B onto a new stack S': $S' = \mathit{rev}\ Q$
*Small*3	Reverse rS onto S': $S' = S \mathbin{@} \mathit{rev}\ Q$

Now S' and B' are the new stacks. Phases *Big*1 and *Small*1 can be performed in parallel thus taking at most $2 * m + 2 + 1$ steps – the 1 is an administrative step between phases. Similarly, phase *Big*2 can be performed in parallel with phases *Small*2 followed by *Small*3, thus taking at most $2 * m + 2 + 1$ steps again. The $4 * m + 6$ steps are spread out as follows: 6 steps are performed in the violating operation and 4 steps in each subsequent *enq* and *deq*. The invariant cannot be violated again during those m operations: we start with stacks S' and B' of size $2 * m + 1$ and $2 * m + k - 1$; in the worst case $k = 1$ and all m operations are *deq*s on B'; in the end we still have $3 * (2 * m + k - 1 - m) \geq 2 * m + 1$. In fact, it takes about $4/3 * m$ *deq*s or $4 * m$ pops before the invariant can be violated again. Because rebalancing happens in parallel with enqueuing and dequeuing, the stacks are augmented with further data structures. A counter keeps track of how many elements of the original stacks are still valid – every *deq* decrements the counter. An additional list *ext* is maintained that *enq*s push to. At the end of the 5 phases, we cannot just append S' to *ext* – this would not be constant-time. Thus stacks are actually implemented as pairs of lists (which complicates push and pop a little) and phase *Small*3 returns (ext, T) where T is S' or B' above, which are real lists, not stacks.

5 Verified Implementation

A deque can be in one of the following states:

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

- A deque contains less than four elements (first four constructors), or
- it consists of two stacks representing the ends of the deque (*Idles* constructor), or
- it is in the middle of rebalancing (*Rebal* constructor).

The emptyness check is trivial:

$$\begin{aligned}&\mathit{is\_empty}\ \mathit{Empty} = \mathit{True}\\&\mathit{is\_empty}\ \_ = \mathit{False}\end{aligned}$$

Note that all code is shown on coloured background to distinguish it easily from all verification-related material.

In the following, we will show the implementation bottom-up, except for the rebalancing process, where we follow the order of the phases. There are a number of overloaded functions that are defined on multiple types:

- Functions *push* and *pop* that implement *enq* and *deq*.
- Function *step* implements the rebalancing steps.
- An invariant *invar*.
- Two abstraction functions to lists: *list* returns the list abstraction after rebalancing, *list_current* returns the list in the current, non-rebalanced state.
- Function *remaining_steps* calculates the remaining steps of a rebalancing process.

The invariant, list abstractions and *remaining_steps* are not code but key components of the verification and important contributions of our paper. Some other functions are also overloaded. For types that only have a function *list* its *size* is defined as:

$$\mathit{size}\ d = |\mathit{list}\ d|$$

If it has *list* and *list_current* then there are *size* and *size_new*:

$$\begin{aligned}&\mathit{size}\ d = \mathit{min}\ |\mathit{list\_current}\ d|\ |\mathit{list}\ d|\\&\mathit{size\_new}\ d = |\mathit{list}\ d|\end{aligned}$$

We verified the following properties for every type that have the respective functions:

$$\begin{aligned}&\mathit{list}\ (\mathit{push}\ x\ d) = x \# \mathit{list}\ d\\&\mathit{invar}\ d \longrightarrow \mathit{size}\ (\mathit{push}\ x\ d) = \mathit{size}\ d + 1\\&\mathit{invar}\ d \longrightarrow \mathit{invar}\ (\mathit{push}\ x\ d)\\&\mathit{invar}\ d \longrightarrow \mathit{remaining\_steps}\ (\mathit{push}\ x\ d) = \mathit{remaining\_steps}\ d\end{aligned}$$

$$\begin{aligned}&\mathit{invar}\ d \wedge 0 < \mathit{size}\ d \wedge \mathit{pop}\ d = (x,\ d') \longrightarrow x \# \mathit{list}\ d' = \mathit{list}\ d\\&\mathit{invar}\ d \wedge 0 < \mathit{size}\ d \wedge \mathit{pop}\ d = (x,\ d') \longrightarrow \mathit{size}\ d' = \mathit{size}\ d - 1\\&\mathit{invar}\ d \wedge \mathit{pop}\ d = (x,\ d') \longrightarrow \mathit{invar}\ d'\\&\mathit{invar}\ d \wedge \mathit{pop}\ d = (x,\ d') \longrightarrow \mathit{remaining\_steps}\ d' \le \mathit{remaining\_steps}\ d\end{aligned}$$

$$\begin{aligned}&\mathit{invar}\ d \longrightarrow \mathit{list}\ (\mathit{step}\ d) = \mathit{list}\ d\\&\mathit{invar}\ d \longrightarrow \mathit{size}\ d = \mathit{size}\ (\mathit{step}\ d)\\&\mathit{invar}\ d \longrightarrow \mathit{invar}\ (\mathit{step}\ d)\\&\mathit{invar}\ d \longrightarrow \mathit{remaining\_steps}\ (\mathit{step}\ d) = \mathit{remaining\_steps}\ d - 1\end{aligned}$$

For *list_current* and *size_new* the same properties hold as for *list* and *size*.

Our collection of datatypes is considerably more refined than those by Chuang and Goldberg because we express a number of the implicit invariants in their code explicitly on the level of types. For example, Chuang and Goldberg's type *Deque* has a constructor *LIST* :: $'a$ *list* ⇒ *Deque* that is applied only to lists of size ≤ 4. The latter is an important implicit invariant that guarantees that operations *rev* and (@), which are applied to arguments of *LIST*, execute in constant time. Our type *deque* expresses the invariant clearly via the first four constructors. As a result, our implementation is more explicit but requires more small building blocks.

5.1 Stack

The basic building block for our implementation is the type *'a stack* that serves as the ends of the deque. It actually consists of two stacks represented by lists:

datatype *'a stack* = *Stack* (*'a list*) (*'a list*)

The stack operations below use the left of the two stacks first, and resort to the right list if the left one is empty. As explained towards the end of Section 4 the right list contains elements resulting from a rebalancing, and the left list holds elements that were newly enqueued during rebalancing.

push x (*Stack left right*) = *Stack* (*x* # *left*) *right*

pop (*Stack* [] []) = *Stack* [] []
pop (*Stack* (*x* # *left*) *right*) = *Stack left right*
pop (*Stack* [] (*x* # *right*)) = *Stack* [] *right*

first (*Stack* (*x* # *left*) *right*) = *x*
first (*Stack* [] (*x* # *right*)) = *x*

is_empty (*Stack* [] []) = *True*
is_empty (*Stack* _ _) = *False*

There is no invariant but a list abstraction function:

list (*Stack left right*) = *left* @ *right*

5.2 Idle

Datatype *idle* represents an end of the deque that is not in a rebalancing process.

datatype *'a idle* = *Idle* (*'a stack*) *nat*

It contains a *stack* to which it delegates its *push* and *pop* operations. Furthermore, we will need to check the size of the end frequently, to know whether rebalancing is required. To achieve this in constant time, we keep track of the size of the *stack* and update it with every operation accordingly.

push x (*Idle stk n*) = *Idle* (*push x stk*) (n + 1)

$pop\ (Idle\ stk\ n) = (first\ stk,\ Idle\ (pop\ stk)\ (n - 1))$

The invariant *invar* $(Idle\ stk\ n) = (size\ stk = n)$ is obvious. The *list* function delegates to the corresponding *list* function on the stack; we omit showing such trivial definitions.

5.3 Current

Now we start to look into the rebalancing procedure. Type *'a current* stores information about operations that happen during rebalancing but which have not become part of the old state that is being rebalanced.

datatype *'a current = Current ('a list) nat ('a stack) nat*

Both ends of the deque contain a *current* state which contains a list of newly enqueued elements and their number. The *push* operation on a *current* state adds to the list and increases its size counter:

$push\ x\ (Current\ ext\ extn\ old\ tar) = Current\ (x \# ext)\ (extn + 1)\ old\ tar$

Additionally, *current* has a *stack* keeping track of the end's state before rebalancing. The natural number after it is the **target size** (usually denoted by *tar*) of the end after rebalancing, but without taking the *ext* component into account. The *pop* operation on *current* enables dequeuing of elements during the rebalancing: If there are newly enqueued elements, *pop* dequeues an element from the corresponding list and adjusts its size counter. Otherwise, it dequeues an element from the old state of the end and reduces the target size by one.

$pop\ (Current\ (x \# ext)\ extn\ old\ tar) = (x,\ Current\ ext\ (extn - 1)\ old\ tar)$
$pop\ (Current\ []\ extn\ old\ tar) = (first\ old,\ Current\ []\ extn\ (pop\ old)\ (tar - 1))$

The operations preserve the obvious invariant for the counter of newly enqueued elements:

$invar\ (Current\ ext\ extn\ \_\ \_) = (|ext| = extn)$

The abstraction *list* yields the list of the state before rebalancing, but modified by the intervening *push*'s and *pop*'s. *current* has next to its *size* function based on *list*, an additional function *size_new* calculating the target size at the end of rebalancing.

$list\ (Current\ ext\ \_\ old\ \_) = ext\ @\ list\ old$

$size\_new\ (Current\ \_\ extn\ \_\ tar) = extn + tar$

5.4 Rebalancing

Rebalancing transfers elements from the bigger end to the smaller one. Datatype *states* stores both ends (types *big_state* and *small_state* are explained below) together with a *direction* indicating if the transfer happens from left to right or right to left. Therefore it also indicates which end is on which side.

datatype *'a states = States direction ('a big_state) ('a small_state)*
datatype *direction = L | R*

Table 1 Rebalancing phases.

<table>
<tr><th>Big</th><th>Small</th></tr>
<tr><td>Big1 _ (P @ Q) [] |P|</td><td>Small1 _ S []</td></tr>
<tr><td>↓</td><td>↓</td></tr>
<tr><td>Big1 _ Q (rev P) 0</td><td>Small1 _ [] (rev S)</td></tr>
<tr><td>Copy _ (rev P) [] 0</td><td>Small2 _ (rev S) Q [] 0</td></tr>
<tr><td></td><td>↓</td></tr>
<tr><td></td><td>Small2 _ (rev S) [] (rev Q) |Q|</td></tr>
<tr><td>↓(Big2)</td><td>Copy _ (rev S) (rev Q) |Q|</td></tr>
<tr><td></td><td>↓ (Small3)</td></tr>
<tr><td>Copy _ [] P |P|</td><td>Copy _ [] (S @ rev Q) (|S| + |Q|)</td></tr>
</table>

The phases described in Section 4 are represented by the following constructors for the big and small end of the deque, with their corresponding behaviour w.r.t. rebalancing steps. *Big*2 and *Small*3 perform the same work and are both represented by the constructor *Copy*.

- *Big*1 :: *'a current* ⇒ *'a stack* ⇒ *'a list* ⇒ *nat* ⇒ *'a big_state*
  *Big*1 _ *S xs n* pops the top *n* elements off *S* and puts them on *xs*.
- *Small*1 :: *'a current* ⇒ *'a stack* ⇒ *'a list* ⇒ *'a small_state*
  *Small*1 _ *S xs* pops all elements off *S* and puts them on *xs*.
- *Small*2 :: *'a current* ⇒ *'a list* ⇒ *'a stack* ⇒ *'a list* ⇒ *nat* ⇒ *'a small_state*
  *Small*2 _ *xs S ys n* pops all elements off *S*, puts them on *ys*, counts them in *n* and leaves *xs* unchanged.
- *Copy* :: *'a current* ⇒ *'a list* ⇒ *'a list* ⇒ *nat* ⇒ *'a common_state*
  Copy cur xs ys n pops elements off *xs*, puts them on *ys*, and counts them, until *n* reaches the *tar* component of *cur*. Stopping before all of *xs* has been moved has the effect of performing the *deq* operations that have accumulated in *cur* during rebalancing.

Every phase contains a *current* state that deals with *enq* and *deq* operations (see Section 5.3).

Table 1 shows how each phase leads to the next at both ends of the deque. The variables are named as in Section 4. For readability we have equated stacks with lists. For simplicity the *Copy* phases assume that the copying is not cut short by a reduced *tar*. We will implement overloaded *step* functions that advance and transition the phases step-by-step.

5.4.1 Big

The bigger end of the deque goes through two phases during rebalancing, modeled with datatype *big_state* with two constructors:

```
datatype 'a big_state = Big1 ('a current) ('a stack) ('a list) nat
                      | Big2 ('a common_state)
```

Both constructors were explained at the beginning of Section 5.4. At that point we pretended that both ends of the deque have a common constructor *Copy*. Instead, constructor *Big*2 is a wrapper around a common state *common_state* (see Section 5.4.3) that both ends delegate their *push*/*pop*/*step* operations to in phases *Big*2 and *Small*3. Operations *push* and *pop* on *Big*1 are delegated to *current*. Function *step* uses *norm* for the transition from phase *Big*1 to *Big*2 which is defined in Section 5.4.3.

$push\ x\ (Big1\ cur\ big\ aux\ n) = Big1\ (push\ x\ cur)\ big\ aux\ n$
$push\ x\ (Big2\ state) = Big2\ (push\ x\ state)$

$pop\ (Big1\ cur\ big\ aux\ n) = (\mathsf{let}\ (x,\ cur) = pop\ cur\ \mathsf{in}\ (x,\ Big1\ cur\ big\ aux\ n))$
$pop\ (Big2\ state) = (\mathsf{let}\ (x,\ state) = pop\ state\ \mathsf{in}\ (x,\ Big2\ state))$

$step\ (Big1\ cur\ big\ aux\ 0) = Big2\ (norm\ (Copy\ cur\ aux\ []\ 0))$
$step\ (Big1\ cur\ big\ aux\ n) = Big1\ cur\ (pop\ big)\ (first\ big\ \#\ aux)\ (n - 1)$
$step\ (Big2\ state) = Big2\ (step\ state)$

The remaining functions on *big_state* again delegate to *common_state* in phase *Big*2. We do not show those equations.

The following invariant is preserved by *push*, *pop* and *step*:

$invar\ (Big1\ cur\ big\ aux\ n) = \mathsf{let}\ Current\ \_\ \_\ old\ tar = cur\ \mathsf{in}$
$\quad invar\ cur$
$\wedge\quad tar \leq |aux| + n$ **(1)**
$\wedge\quad n \leq size\ big$ **(2)**
$\wedge\quad take\_last\ (size\ old)\ (rev\ aux\ @\ list\ big) = list\ old$ **(3)**
$\wedge\quad take\ tar\ (rev\ (take\ n\ (list\ big))\ @\ aux) = rev\ (take\ tar\ (list\ old))$ **(4)**

(1) The target size of the end after the rebalancing (tar) is $\leq$ to the total number of elements that the phase reverses ($|aux| + n$). This needs to hold because phase *Big*1 moves to *aux* the elements that remain on this end. Only $\leq$ but not $=$ holds because of potentially dequeued elements that reduce the target size (see Section 5.3: *pop*).

(2) There must be at least as many elements as the phase wants to reverse.

(3) Undoing the progress of the phase by reversing *aux* back and appending it back to *big* reproduces the old state of the end. We account for potentially dequeued elements by dropping those from the front of the restored end.

(4) Finishing the phase by reversing and appending n more elements to *aux* gives us the elements that remain on this end in a reversed order.

In phase *Big*1, *list* finishes rebalancing and returns the final list of the end. In contrast, *list_current* returns the original list of the end.

$list\ (Big1\ (Current\ ext\ \_\ \_\ tar)\ big\ aux\ n) =$
$\quad \mathsf{let}\ a = rev\ (take\ n\ (list\ big))\ @\ aux\ \mathsf{in}\ ext\ @\ rev\ (take\ tar\ a)$

$list\_current\ (Big1\ cur\ \_\ \_\ \_) = list\ cur$

The verification also requires the number of remaining *steps* of rebalancing:

$remaining\_steps\ (Big1\ (Current\ \_\ \_\ \_\ tar)\ \_\ \_\ n) = n + tar + 1$

In phase *Big*1, n more elements need to be moved before 1 additional step transitions to phase *Big*2 which requires *tar* steps.

5.4.2 Small

As depicted in Table 1, the smaller end of the deque goes through three different phases during rebalancing. They are represented by datatype *small_state*:

$$\textbf{datatype}\ 'a\ small\_state = Small1\ ('a\ current)\ ('a\ stack)\ ('a\ list)$$
$$|\ Small2\ ('a\ current)\ ('a\ list)\ ('a\ stack)\ ('a\ list)\ nat$$
$$|\ Small3\ ('a\ common\_state)$$

Just as in *big_state*, constructor *Small3* contains the common data structure to which the phases *Big2* and *Small3* delegate their operations (see Section 5.4.3). This time we do not show any of the trivial delegating equations.

Operations *push* and *pop* are defined analogously to their *big_state* relatives:

$$push\ x\ (Small1\ cur\ small\ aux) = Small1\ (push\ x\ cur)\ small\ aux$$
$$push\ x\ (Small2\ cur\ aux\ big\ new\ n) = Small2\ (push\ x\ cur)\ aux\ big\ new\ n$$

$$pop\ (Small1\ cur\ small\ aux) = (\mathsf{let}\ (x,\ cur) = pop\ cur\ \mathsf{in}\ (x,\ Small1\ cur\ small\ aux))$$
$$pop\ (Small2\ cur\ aux\ big\ new\ n)$$
$$= (\mathsf{let}\ (x,\ cur) = pop\ cur\ \mathsf{in}\ (x,\ Small2\ cur\ aux\ big\ new\ n))$$

In phase *Small1*, *step* idles once it has emptied its stack because it needs to wait for the big end to finish phase *Big1* before both ends can transition to their next phases simultaneously (see Section 5.4.4). In phase *Small2* the stack is popped until it is empty and phase *Small3* starts:

$$step\ (Small1\ cur\ small\ aux)$$
$$= (\mathsf{if}\ is\_empty\ small\ \mathsf{then}\ Small1\ cur\ small\ aux$$
$$\quad \mathsf{else}\ Small1\ cur\ (pop\ small)\ (first\ small\ \#\ aux))$$
$$step\ (Small2\ cur\ aux\ big\ new\ n)$$
$$= (\mathsf{if}\ is\_empty\ big\ \mathsf{then}\ Small3\ (norm\ (Copy\ cur\ aux\ new\ n))$$
$$\quad \mathsf{else}\ Small2\ cur\ aux\ (pop\ big)\ (first\ big\ \#\ new)\ (n + 1))$$

The following invariant, presented phase by phase, is preserved by *push*, *pop* and *step*:

$$invar\ (Small1\ cur\ small\ aux) = \mathsf{let}\ Current\ \_\ \_\ old\ tar = cur\ \mathsf{in}$$
$$invar\ cur$$
$$\wedge\ size\ old \leq tar \qquad \textbf{(1)}$$
$$\wedge\ size\ old \leq size\ small + |aux| \qquad \textbf{(2)}$$
$$\wedge\ take\_last\ (size\ old)\ (rev\ aux\ @\ list\ small) = list\ old \qquad \textbf{(3)}$$

(1) The target size is not smaller than the original size of the end. Otherwise, rebalancing would not be successful because the smaller end would shrink further.

(2) The stack holding the original elements of the smaller end (*old*) cannot grow but potentially shrink through *pop* operations. Moreover, since phase *Small1* is reversing a copy of the original elements of the smaller size, the total number of elements it works on is $\geq$ to the size of the stack *old*.

(3) Undoing the progress of the phase by reversing *aux* back and appending it back to *small* reproduces the old state of the end. We account for potentially dequeued elements by dropping those from the front of the restored end.

$$
\begin{aligned}
&\mathit{invar}\ (\mathit{Small2}\ \mathit{cur}\ \mathit{aux}\ \mathit{big}\ \mathit{new}\ n) = \textsf{let}\ \mathit{Current}\ \_\ \_\ \mathit{old}\ \mathit{tar} = \mathit{cur}\ \textsf{in}\\
&\qquad \mathit{invar}\ \mathit{cur}\\
&\quad\wedge\quad n = |\mathit{new}| &&\textbf{(1)}\\
&\quad\wedge\quad \mathit{tar} = n + \mathit{size}\ \mathit{big} + \mathit{size}\ \mathit{old} &&\textbf{(2)}\\
&\quad\wedge\quad \mathit{size}\ \mathit{old} \le |\mathit{aux}| &&\textbf{(3)}\\
&\quad\wedge\quad \mathit{rev}\ (\mathit{take}\ (\mathit{size}\ \mathit{old})\ \mathit{aux}) = \mathit{list}\ \mathit{old} &&\textbf{(4)}
\end{aligned}
$$

(1) The phase counts its reversed elements correctly.
(2) The elements transferred from the bigger end and the original elements from the smaller end will build the new smaller end. Consequently, the sum of their elements is equal to the target size. Hereby, the number of transferred elements is split into already reversed and not yet reversed ones.
(3, 4) Next to the reversal, phase *Small*2 also holds the already reversed original state of the smaller end. Accordingly, it is equal to *old* when reversed back and accounted for the potentially dequeued elements.

Of the abstraction functions *list* and *list_current* we merely show *list* because *list_current* simply delegates to its counterpart on *current*.

$$
\begin{aligned}
&\mathit{list}\ (\mathit{Small2}\ (\mathit{Current}\ \mathit{ext}\ \_\ \_\ \mathit{tar})\ \mathit{aux}\ \mathit{big}\ \mathit{new}\ n)\\
&= \mathit{ext}\ @\ \mathit{rev}\ (\mathit{take}\ (\mathit{tar} - n - \mathit{size}\ \mathit{big})\ \mathit{aux})\ @\ \mathit{rev}\ (\mathit{list}\ \mathit{big})\ @\ \mathit{new}
\end{aligned}
$$

Function *list* is partial. It is lacking a case for phase *Small*1 because phase *Small*1 lacks the elements coming from the bigger end, so it is impossible to simulate all further steps of the rebalancing. The lacking case will be added one level higher for *States* where we also have the bigger end available (see Section 5.4.4).

For phase *Small*2, *list* finishes the reversal of the transferred elements, prepends the reversed result of phase *Small*1 while accounting for potentially dequeued elements, and prepends the potentially enqueued elements.

The smaller end does not have its own remaining steps measurement because they depend on the state of the bigger end.

5.4.3 Common

The datatype *'a common_state* is a joint representation of phases *Big*2 and *Small*3:

$$
\begin{aligned}
\textbf{datatype}\ {}'a\ \mathit{common\_state}\ &= \mathit{Copy}\ ('a\ \mathit{current})\ ('a\ \mathit{list})\ ('a\ \mathit{list})\ \mathit{nat}\\
&\mid \mathit{Idle}\ ('a\ \mathit{current})\ ('a\ \mathit{idle})
\end{aligned}
$$

Copy represents rebalancing; *Idle* signals termination of rebalancing and keeps the rebalanced state of an end in an *idle* state (see Section 5.2).

$$
\begin{aligned}
&\mathit{step}\ (\mathit{Copy}\ \mathit{cur}\ \mathit{aux}\ \mathit{new}\ n)\\
&= (\textsf{let}\ \mathit{Current}\ \mathit{ext}\ \mathit{extn}\ \mathit{old}\ \mathit{tar} = \mathit{cur}\\
&\quad\ \textsf{in}\ \mathit{norm}\\
&\qquad (\textsf{if}\ n < \mathit{tar}\ \textsf{then}\ \mathit{Copy}\ \mathit{cur}\ (\mathit{tl}\ \mathit{aux})\ (\mathit{hd}\ \mathit{aux}\ \#\ \mathit{new})\ (n + 1)\\
&\qquad\ \ \textsf{else}\ \mathit{Copy}\ \mathit{cur}\ \mathit{aux}\ \mathit{new}\ n))\\
&\mathit{step}\ (\mathit{Idle}\ \mathit{cur}\ \mathit{idle}) = \mathit{Idle}\ \mathit{cur}\ \mathit{idle}
\end{aligned}
$$

Function *norm* performs the transition back to an idle end. If *tar* has been reached, *norm* creates a new *stack* and puts the elements that arrived during rebalancing in the front and the result of rebalancing in the back and sets the size accordingly:

$$\begin{array}{l} \textit{norm}\ (\textit{Copy cur aux new n}) \\ = (\textsf{let}\ \textit{Current ext extn old tar} = \textit{cur} \\ \quad \textsf{in if}\ \textit{tar} \leq n\ \textsf{then}\ \textit{Idle cur}\ (\textit{Idle}\ (\textit{Stack ext new})\ (\textit{extn} + n)) \\ \quad\quad \textsf{else}\ \textit{Copy cur aux new n}) \end{array}$$

Both constructors also contain a *current* state on which the *push* and *pop* operations work:

$$\begin{array}{l} \textit{push}\ x\ (\textit{Copy cur aux new n}) = \textit{Copy}\ (\textit{push}\ x\ \textit{cur})\ \textit{aux new n} \\ \textit{push}\ x\ (\textit{Idle cur}\ (\textit{Idle stk n})) = \textit{Idle}\ (\textit{push}\ x\ \textit{cur})\ (\textit{Idle}\ (\textit{push}\ x\ \textit{stk})\ (n + 1)) \end{array}$$

$$\begin{array}{l} \textit{pop}\ (\textit{Copy cur aux new n}) \\ = (\textsf{let}\ (x,\ \textit{cur}) = \textit{pop cur}\ \textsf{in}\ (x,\ \textit{norm}\ (\textit{Copy cur aux new n}))) \\ \textit{pop}\ (\textit{Idle cur idle}) = (\textsf{let}\ (x,\ \textit{idle}) = \textit{pop idle}\ \textsf{in}\ (x,\ (\textit{Idle}\ (\textit{fst}\ (\textit{pop cur}))\ \textit{idle})) \end{array}$$

Both operations also update the *idle* component when the respective phase terminated. Additionally, the *pop* operation checks if it dequeued the last element of the reversal and transitions, using *norm*, to the idle phase if so.

For the phases *Big*2 and *Small*3 the invariant is the following:

$$\begin{array}{lll} \multicolumn{2}{l}{\textit{invar}\ (\textit{Copy cur aux new n}) = \textsf{let}\ \textit{Current}\ \_\ \_\ \textit{old tar} = \textit{cur}\ \textsf{in}} & \\ & \textit{invar cur} & \\ \wedge & n < \textit{tar} & \textbf{(1)} \\ \wedge & n = |\textit{new}| & \textbf{(2)} \\ \wedge & \textit{tar} \leq |\textit{aux}| + n & \textbf{(3)} \\ \wedge & \textit{take tar}\ (\textit{list old}) = \textit{take}\ (\textit{size old})\ (\textit{rev}\ (\textit{take}\ (\textit{tar} - n)\ \textit{aux})\ @\ \textit{new}) & \textbf{(4)} \end{array}$$

(1) The number of elements for which the rebalancing is finished did not yet reach the target number.
(2) n correctly holds the number of finished elements.
(3) There are enough elements left to reach the target number.
(4) When simulating the termination by reversing the missing elements, the front of the new and old end are the same.

The invariant for the idle state requires that the subcomponents satisfy their invariants and that the fronts of the old and the rebalanced ends are the same:

$$\begin{array}{l} \textit{invar}\ (\textit{Idle cur idle}) \\ = \textit{invar cur} \wedge \textit{invar idle} \wedge \textit{take}\ (\textit{size idle})\ (\textit{list cur}) = \textit{take}\ (\textit{size cur})\ (\textit{list idle}) \end{array}$$

Function *list* finishes the phases *Big*2/*Small*3 and prepends the elements that arrived during rebalancing. In the *Idle* state it delegates to *list* on *idle*.

$$\begin{array}{l} \textit{list}\ (\textit{Copy}\ (\textit{Current ext}\ \_\ \_\ \textit{tar})\ \textit{aux new n}) = \textit{ext}\ @\ \textit{rev}\ (\textit{take}\ (\textit{tar} - n)\ \textit{aux})\ @\ \textit{new} \\ \textit{list}\ (\textit{Idle}\ \_\ \textit{idle}) = \textit{list idle} \end{array}$$

The abstraction function *list_current* simply delegates to its counterpart on *current*.

Counting of the remaining steps is similarly straightforward. In phases *Big*2/*Small*3 the difference between the processed elements and the target remains; the idle state does not need any more steps.

$$\begin{array}{l} \textit{remaining\_steps}\ (\textit{Copy}\ (\textit{Current}\ \_\ \_\ \_\ \textit{tar})\ \_\ \_\ n) = \textit{tar} - n \\ \textit{remaining\_steps}\ (\textit{Idle}\ \_\ \_) = 0 \end{array}$$

5.4.4 States

Putting the two ends together into *states* completes the implementation of the rebalancing procedure. Remember that in Section 5.4.2 phase *Small*1 could not transition to *Small*2 by itself because it needs to synchronize with the end of *Big*1. The *step* function on *states* covers this case by moving from *Small*1 to *Small*2 once *Big*1 has reached 0. The other cases were already covered by the *step* functions on *big_state* and *small_state*.

$$\begin{aligned}&\mathit{step}\ (\mathit{States}\ \mathit{dir}\ (\mathit{Big1}\ \mathit{currentB}\ \mathit{big}\ \mathit{auxB}\ 0)\ (\mathit{Small1}\ \mathit{currentS}\ \_\ \mathit{auxS}))\\ &= \mathit{States}\ \mathit{dir}\ (\mathit{step}\ (\mathit{Big1}\ \mathit{currentB}\ \mathit{big}\ \mathit{auxB}\ 0))\ (\mathit{Small2}\ \mathit{currentS}\ \mathit{auxS}\ \mathit{big}\ []\ 0)\\ &\mathit{step}\ (\mathit{States}\ \mathit{dir}\ \mathit{big}\ \mathit{small}) = \mathit{States}\ \mathit{dir}\ (\mathit{step}\ \mathit{big})\ (\mathit{step}\ \mathit{small})\end{aligned}$$

The joint list abstraction *lists* returns the pair containing the lists for the two ends. It also compensates for the partiality of *list* on the smaller end: *lists* simulates the remaining steps of phase *Small*1 and performs the transition to phase *Small*2, for which *list* is already defined, to create the missing list abstraction for phase *Small*1. For the other phases it calls the respective list abstractions.

$$\begin{aligned}&\mathit{lists}\ (\mathit{States}\ \_\ (\mathit{Big1}\ \mathit{curB}\ \mathit{big}\ \mathit{auxB}\ n)\ (\mathit{Small1}\ \mathit{curS}\ \mathit{small}\ \mathit{auxS}))\\ &= (\mathit{list}\ (\mathit{Big1}\ \mathit{curB}\ \mathit{big}\ \mathit{auxB}\ n),\\ &\quad \mathit{list}\ (\mathit{Small2}\ \mathit{curS}\ (\mathit{rev}\ (\mathit{take}\ n\ (\mathit{list}\ \mathit{small}))\ @\ \mathit{auxS})\ (\mathit{pop}^n\ \mathit{big})\ []\ 0))\\ &\mathit{lists}\ (\mathit{States}\ \_\ \mathit{big}\ \mathit{small}) = (\mathit{list}\ \mathit{big},\ \mathit{list}\ \mathit{small})\end{aligned}$$

Function *lists_current* simply delegates to the big and small end:

$$\mathit{lists\_current}\ (\mathit{States}\ \_\ \mathit{big}\ \mathit{small}) = (\mathit{list\_current}\ \mathit{big},\ \mathit{list\_current}\ \mathit{small})$$

For convenience, we define

$$\mathit{list\_small\_first}\ \mathit{states} = (\mathsf{let}\ (\mathit{big},\ \mathit{small}) = \mathit{lists}\ \mathit{states}\ \mathsf{in}\ \mathit{small}\ @\ \mathit{rev}\ \mathit{big})$$

$$\begin{aligned}&\mathit{list\_current\_small\_first}\ \mathit{states}\\ &= (\mathsf{let}\ (\mathit{big},\ \mathit{small}) = \mathit{lists\_current}\ \mathit{states}\ \mathsf{in}\ \mathit{small}\ @\ \mathit{rev}\ \mathit{big})\end{aligned}$$

and analogously *list_big_first* and *list_current_big_first*.

The invariant is defined as follows:

$$\begin{aligned}&\mathit{invar}\ (\mathit{States}\ \mathit{dir}\ \mathit{big}\ \mathit{small}) =\\ &\quad \mathit{invar}\ \mathit{big} \wedge \mathit{invar}\ \mathit{small}\\ &\wedge\ \ \mathit{list\_small\_first}\ (\mathit{States}\ \mathit{dir}\ \mathit{big}\ \mathit{small})\\ &\quad = \mathit{list\_current\_small\_first}\ (\mathit{States}\ \mathit{dir}\ \mathit{big}\ \mathit{small}) && \mathbf{(1)}\\ &\wedge\ \ \mathsf{case}\ (\mathit{big},\ \mathit{small})\ \mathsf{of}\\ &\qquad (\mathit{Big1}\ \_\ \mathit{big}\ \_\ n,\ \mathit{Small1}\ (\mathit{Current}\ \_\ \_\ \mathit{old}\ \mathit{tar})\ \mathit{small}\ \_) \Rightarrow\\ &\qquad\quad \mathit{size}\ \mathit{big} - n = \mathit{tar} - \mathit{size}\ \mathit{old} \wedge \mathit{size}\ \mathit{small} \le n && \mathbf{(2,3)}\\ &\quad \mid (\mathit{Big1}\ \_\ \_\ \_\ \_,\ \_) \Rightarrow \mathit{False} && \mathbf{(4)}\\ &\quad \mid (\mathit{Big2}\ \_\ ,\ \mathit{Small1}\ \_\ \_\ \_) \Rightarrow \mathit{False} && \mathbf{(5)}\\ &\quad \mid (\_\ ,\ \_) \Rightarrow \mathit{True}\end{aligned}$$

- **(1)** Rebalancing preserves the abstract queue (as a list): the list abstraction after the end of rebalancing must be the same as the list abstraction that uses the state before rebalancing.
- **(2)** After phase *Big*1, the bigger end transfers exactly the number of elements missing on the smaller end to reach the target size.
- **(3)** Phase *Big*1 does not finish before *Small*1. This needs to hold because the smaller end transitions from phase *Small*1 to *Small*2 at the end of stage *Big*1.

(4) Phase *Big*1 can only occur together with phase *Small*1.
(5) Phase *Big*2 cannot occur together with phase *Small*1.

The case analysis in the invariant ensures that phase *Big*1 runs in parallel with phase *Small*1, and phase *Big*2 with the phases *Small*2 and *Small*3.

The overall remaining steps are the maximum of the remaining steps of both ends:

$$
\begin{array}{l}
remaining\_steps\ (States\ \_\ big\ small) = \\
\quad max \\
\quad\quad (remaining\_steps\ big) \\
\quad\quad (\textsf{case}\ small\ \textsf{of} \\
\quad\quad\quad Small1\ (Current\ \_\ \_\ \_\ tar)\ \_\ \_ \Rightarrow \textsf{let}\ Big1\ \_\ \_\ \_\ nB = big\ \textsf{in}\ nB + tar + 2 \\
\quad\quad\ \mid Small2\ (Current\ \_\ \_\ \_\ tar)\ \_\ \_\ \_\ nS \Rightarrow tar - nS + 1 \\
\quad\quad\ \mid Small3\ state \Rightarrow remaining\_steps\ state)
\end{array}
$$

We focus on the smaller end because we covered the bigger end already in Section 5.4.1. The remaining steps for the small end in phase *Small*1 cannot be calculated in isolation because they depend on the big end: Phase *Small*1 needs to wait for phase *Big*1 to finish, which are nB steps. Then it moves via *Small*2 and *Small*3 to *Idle*, counting up until the target size tar is reached. Consequently, the smaller end needs $nB + tar$ steps from phase *Small*1 to finish and 2 more steps for the transitions. In phase *Small*2, the counter is at nS already and hence $tar - nS$ steps remain, plus 1 for the last transition. The remaining steps for phase *Small*3 are already covered in Section 5.4.3.

Finally, we must ensure that the deque re-establishes the size constraints after rebalancing. therefore *size_ok* calculates, relative to the remaining steps, if the size constraints can be met: it is not allowed that one end is more than 3 times larger than the other after rebalancing. Additionally, none of the ends is allowed to be empty at the end. Herefore, it is also important that both ends have enough elements to facilitate all the *dequeue* operations that can potentially happen. Therefore, *size_ok* uses the size measurements implemented for both ends.

$$size\_ok\ states = size\_ok'\ states\ (remaining\_steps\ states)$$

$$
\begin{array}{l}
size\_ok'\ (States\ \_\ big\ small)\ steps = \\
\quad\quad size\_new\ small + steps + 2 \le 3 * size\_new\ big \\
\quad \wedge\ size\_new\ big + steps + 2 \le 3 * size\_new\ small \\
\quad \wedge\ steps + 1 \le 4 * size\ small \\
\quad \wedge\ steps + 1 \le 4 * size\ big
\end{array}
$$

Note that $(m \le k * n \wedge n \le k * m) = (max\ m\ n \le k * min\ m\ n)$, i.e. we have merely rewritten the size invariant from Section 4.

5.5 Deque

Finally, we can put together all the parts for the overall invariant:

$$
\begin{array}{l}
invar\ (Idles\ l\ r) = \\
\quad\quad invar\ l \wedge invar\ r \wedge \neg\ is\_empty\ l \wedge \neg\ is\_empty\ r \\
\quad \wedge\ size\ l \le 3 * size\ r \wedge size\ r \le 3 * size\ l \\
invar\ (Rebal\ states) = (invar\ states \wedge size\_ok\ states \wedge 0 < remaining\_steps\ states) \\
invar\ \_ = True
\end{array}
$$

In the idle state, the deque must satisfy the invariants of both ends and the size constraints between them. During rebalancing, the deque must satisfy the invariant of the rebalancing process, must ensure that it meets the size constraints after rebalancing, and *remaining_steps* must correctly predict that there are further steps needed. The other states of the deque fulfill the invariant trivially.

The overall list abstraction function *listL* (Section 3) is composed trivially from the separate states' list abstractions:

listL Empty = []
listL (*One x*) = [*x*]
listL (*Two x y*) = [*x*, *y*]
listL (*Three x y z*) = [*x*, *y*, *z*]
listL (*Idles left right*) = *list left* @ *rev* (*list right*)
listL (*Rebal states*) = *listL states*

listL (*States L big small*) = *list_small_first* (*States L big small*)
listL (*States R big small*) = *list_big_first* (*States R big small*)

5.5.1 Enqueuing

Function *enqL* enqueues one element on the left end of the deque and returns the resulting deque.

```
enqL x Empty = One x
enqL x (One y) = Two x y
enqL x (Two y z) = Three x y z
enqL x (Three a b c) = Idles (Idle (Stack [x, a] []) 2) (Idle (Stack [c, b] []) 2)
enqL x (Idles l (Idle r nR)) =
    let Idle l nL = push x l in
    if nL ≤ 3 ∗ nR then Idles (Idle l nL) (Idle r nR)
    else let nl = nl − nR − 1;
        nR = 2 ∗ nL + 1;
        big = Big1 (Current [] 0 l nL) l [] nL;
        small = Small1 (Current [] 0 r nR) r [];
        states = States R big small;
        states = step^6 states;
    in Rebal states
enqL x (Rebal (States L big small) =
    let small = push x small;
        states = step^4 (States L big small);
    in case states of
         States L (Big2 (Idle _ big)) (Small3 (Idle _ small)) ⇒ Idle small big
       | _ ⇒ Rebal states
```

```
enqL x (Rebal (States R big small) =
    let big = push x big;
        states = step^4 (States R big small);
    in case states of
         States R (Big2 (Idle _ big)) (Small3 (Idle _ small)) ⇒ Idle big small
       | _ ⇒ Rebal states
```

Function *enqL* advances the constructors *Empty*, *One* and *Two* to the next larger one. For *Three*, it transitions the deque into the idle state by placing two elements at each end.

In the idle state, it enqueues one element on the left end and checks if the size invariant between the two ends still holds. If so, it keeps the deque in the idle state. Otherwise, it initiates rebalancing in the same way as $deqL'$, but in the other direction.

If the deque is already in the rebalancing process, *enqL* enqueues the new element and advances rebalancing by 4 steps. If that finishes rebalancing, it moves back into the idle state.

Function *enqR*, the counterpart of *enqL*, swaps the two ends of the deque, calls *enqL* and swaps the ends back.

```
enqR x d = swap (enqL x (swap d))
```

5.5.2 Dequeuing

The function $deqL'$ dequeues one element from the left end of the deque and returns the dequeued element and the remaining deque. Accordingly, it implements *deqL* and *firstL* simultaneously.

```
deqL' (One x) = (x, Empty)
deqL' (Two x y) = (x, One y)
deqL' (Three x y z) = (x, Two y z)
deqL' (Idles l (Idle r nR)) =
   let (x, Idle l nL) = pop l in
   if nR ≤ 3 ∗ nL then (x, Idles (Idle l nL) (Idle r nR))
   else if 1 ≤ nL then
     let nL' = 2 ∗ nL + 1;
         nR' = nR − nL − 1;
         small = Small1 (Current [] 0 l nL') l [];
         big = Big1 (Current [] 0 r nR') r [] nR';
         states = States L big small;
         states = step^6 states;
     in (x, Rebal states)
   else case r of Stack r1 r2 ⇒ (x, small_deque r1 r2)
deqL' (Rebal (States L big small) =
   let (x, small) = pop small;
       states = step^4 (States L big small);
   in case states of
        States R (Big2 (Idle _ big)) (Small3 (Idle _ small)) ⇒ (x, Idle big small)
      | _ ⇒ (x, Rebal states)
```

```
deqL' (Rebal (States R big small) =
   let (x, big) = pop big;
       states = step^4 (States R big small);
   in case states of
        States R (Big2 (Idle _ big)) (Small3 (Idle _ small)) ⇒ (x, Idle big small)
      | _ ⇒ (x, Rebal states)
```

If the deque has less than four elements, $deqL'$ dequeues the leftmost element and transitions to the next smaller constructor (not shown).

In the idle state, $deqL'$ dequeues an element from the left end and checks if the size invariant between the two ends still holds. If so, the deque stays in the idle states. Otherwise, it checks if the left end became empty and transitions to one of the small states using *small_deque* (below) in that case. In the last case, when the left end is not empty and the size constraints are violated, it starts rebalancing. Therefore, it divides the total number of elements into two almost equal halves – the right is one larger because the total number is odd. Then, the phases *Big*1 (for the bigger, right side) and *Small*1 (for the smaller, left side) are initialized with these numbers as target sizes and the state of the respective end. Finally, $deqL'$ starts rebalancing by executing 6 steps.

When the deque is already in the rebalancing state, $deqL'$ dequeues one element from the respective end and advances the rebalancing with 4 more steps. If that finishes rebalancing, it transitions the deque back into the idle state.

$$
\begin{array}{ll}
small\_deque\ []\ [] = Empty & small\_deque\ []\ [x,\ y] = Two\ y\ x \\
small\_deque\ [x]\ [] = One\ x & small\_deque\ []\ [x,\ y,\ z] = Three\ z\ y\ x \\
small\_deque\ []\ [x] = One\ x & small\_deque\ [x,\ y,\ z]\ [] = Three\ z\ y\ x \\
small\_deque\ [x]\ [y] = Two\ y\ x & small\_deque\ [x,\ y]\ [z] = Three\ z\ y\ x \\
small\_deque\ [x,\ y]\ [] = Two\ y\ x & small\_deque\ [x]\ [y,\ z] = Three\ z\ y\ x
\end{array}
$$

Function $deqR'$, analogously to *enqR*, is reduced to $deqL'$ by swapping the ends twice. *deqR* and *firstR* are specializations of $deqR'$.

$$deqR'\ deque = (\mathsf{let}\ (x,\ deque) = deqL'\ (swap\ deque)\ \mathsf{in}\ (x,\ swap\ deque))$$

5.6 Proof

In this section we explain how the top-level properties of the specification in Section 3 are proved. This is what we proved for *enqL* and $deqL'$:

$$invar\ d \longrightarrow listL\ (enqL\ x\ d) = x\ \#\ listL\ d \qquad (*)$$
$$invar\ d \longrightarrow invar\ (enqL\ x\ d)$$

$$invar\ d \wedge listL\ d \neq [] \wedge deqL'\ d = (x,\ d') \longrightarrow x\ \#\ listL\ d' = listL\ d$$
$$invar\ d \wedge \neg\ is\_empty\ d \longrightarrow invar\ (deqL\ d)$$

The proofs are case analyses over all the defining equations of the non-recursive functions *enqL* and $deqL'$. In each case, the proof is largely by application of the verified properties for the underlying data structures (see Section 5). As an example of these top-level proofs we present one crucial case of $(*)$:

$$listL\ (enqL\ x\ (Rebal\ (States\ L\ big\ small))) = x\ \#\ listL\ (Rebal\ (States\ L\ big\ small))$$

assuming that the deque stays in the rebalancing state. We start by defining $states = States\ L\ big\ small$, $small' = push\ x\ small$ and $states' = States\ L\ big\ small'$ as shorthands. Then we can unfold the definition of *enqL*:

$$listL\ (enqL\ x\ (Rebal\ (States\ L\ big\ small))) = listL\ (step^4\ states')$$

Using the property of the *step* functions preserving list abstractions (see Section 5), we can simply ignore the four rebalancing steps:

$\ldots \quad = \mathit{listL}\ \mathit{states}'$

This enables us to unfold the definition of *listL*:

$\ldots \quad = \mathit{list\_small\_first}\ \mathit{states}'$
$\quad\quad = \mathsf{let}\ (\mathit{bs},\ \mathit{ss}') = \mathit{lists}\ \mathit{states}'\ \mathsf{in}\ \mathit{ss}'\ @\ \mathit{rev}\ \mathit{bs}$

Now, we can utilize that *push* operations prepend the new element to the list abstractions:

$\ldots \quad = \mathsf{let}\ (\mathit{bs},\ x\ \#\ \mathit{ss}) = \mathit{lists}\ \mathit{states}'\ \mathsf{in}\ (x\ \#\ \mathit{ss})\ @\ \mathit{rev}\ \mathit{bs}$
$\quad\quad = x\ \#\ (\mathsf{let}\ (\mathit{bs},\ \mathit{ss}) = \mathit{lists}\ \mathit{states}\ \mathsf{in}\ \mathit{ss}\ @\ \mathit{rev}\ \mathit{bs})$

Concluding the proof, we fold the definition of *listL* again:

$\ldots \quad = x\ \#\ \mathit{list\_small\_first}\ \mathit{states}$
$\quad\quad = x\ \#\ \mathit{listL}\ (\mathit{Rebal}\ \mathit{states})$

The required properties of *firstL* and *deqL* are simple corollaries of the above properties for *deqL′*. The dual properties of *enqR* and *deqR′* are again corollaries via these additional properties:

$\mathit{invar}\ d \longrightarrow \mathit{listR}\ (\mathit{swap}\ d) = \mathit{listL}\ d$
$\mathit{invar}\ d \longrightarrow \mathit{invar}\ (\mathit{swap}\ d)$

5.7 Complexity

All operations of our implementation take constant time because they only employ constant-time functions (arithmetic, (#), *hd*, *tl*) and are not recursive. Some of the auxiliary functions used in the verification are not constant-time but this is irrelevant. Our colour schema helps to distinguish the two worlds.

6 Conclusion

We have presented an implementation of a real-time double-ended queue and in particular the key ingredients of its verification: the abstraction functions, the invariants (incl. all auxiliary functions to define them), and the key theorems about the implementation. It would be interesting to investigate if our invariants could be simplified and if semi-automatic theorem provers like Why3 [3] could automate the proof significantly beyond the current level.

Finally note that our deque implementation is fully executable and that Isabelle can generate code in many functional languages (including Haskell and Scala) from it [5].

References

1. F. Warren Burton. An efficient functional implementation of FIFO queues. *Inf. Process. Lett.*, 14(5):205–206, 1982. `doi:10.1016/0020-0190(82)90015-1`.
2. Tyng-Ruey Chuang and Benjamin Goldberg. Real-time deques, multihead turing machines, and purely functional programming. In *Functional programming languages and computer architecture - FPCA '93*. ACM Press, 1993. `doi:10.1145/165180.165225`.
3. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *European Symposium on Programming (ESOP 2013)*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.
4. Alejandro Gómez-Londoño. Hood-Melville queue. *Archive of Formal Proofs*, January 2021. URL: `https://isa-afp.org/entries/Hood_Melville_Queue.html`.

5 Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.

6 Robert Hood. *The Efficient Implementation of Very-High-Level Programming Language Constructs*. PhD thesis, Cornell University, 1982. TR 82-503.

7 Robert Hood and Robert Melville. Real-time queue operation in pure LISP. *Inf. Process. Lett.*, 13(2):50–54, 1981. doi:10.1016/0020-0190(81)90030-2.

8 Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. Contract-based resource verification for higher-order functions with memoization. In Giuseppe Castagna and Andrew D. Gordon, editors, *Symposium on Principles of Programming Languages, POPL 2017*, pages 330–343. ACM, 2017. doi:10.1145/3009837.3009874.

9 Ravichandhran Madhavan and Viktor Kuncak. Symbolic resource bound inference for functional programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification, CAV 2014*, volume 8559 of *LNCS*, pages 762–778. Springer, 2014. doi:10.1007/978-3-319-08867-9_51.

10 Tobias Nipkow, editor. *Functional Data Structures and Algorithms. A Proof Assistant Approach.* ACM Books, Forthcoming. URL: https://functional-algorithms-verified.org/.

11 Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. URL: http://concrete-semantics.org.

12 Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

13 Chris Okasaki. Simple and efficient purely functional queues and deques. *J. Funct. Program.*, 5(4):583–592, 1995. doi:10.1017/S0956796800001489.

14 Balazs Toth and Tobias Nipkow. Real-time double-ended queue. *Archive of Formal Proofs*, June 2022. , Formal proof development. URL: https://www.isa-afp.org/entries/Real_Time_Deque.html.