Sequence Implementations in Haskell

Pedro R. Borges

Linacre College

November 1997

Qualifying dissertation submitted in support of an application for transfer to D.Phil. status.

Oxford University Computing Laboratory
Programming Research Group

Sequence Implementations in Haskell

Pedro R. Borges

Abstract

An abstract data type *sequence* is defined with the operations empty, isEmpty, cons, snoc, popFront, popRear, lenghtSeq, toList, and toSeq. A sequence with the operations lookupSeq and updateSeq is an *Indexable Sequence*. A sequence with catenation is called a *Catenable Sequence*. Some functional implementations of these abstract data types taken from the literature are described. These implementations are classified as stacks, deques, flexible arrays, and catenable lists, if they can be used as efficient implementations for each of these traditional data types. Some of them are extended to provide the operations defined for sequences. Some comments and directions for further research are also included.

The implementations are made in the functional programming language Haskell as instances of one or more of the classes `Sequence`, `IndSeq`, and `CatSeq`, with the operations defined for each type. These instances are classified by the subset of these operations that each instance supports efficiently, i.e. with at most logarithmic complexity.

Contents

List of Figures

1 Introduction

Data Structures are concrete representations of *Abstract Data Types*. An abstract data type is a set and a collection of operations that can be performed in elements of the set. The design and implementation of data structures has been, traditionally, a major area of research in computer science. Most of the research conducted in the area has been in data structures designed to be implemented in an imperative programming language, and there is an extensive literature on the topic (see for example [MT90]), including many good textbooks describing the subject (e.g. [AHU74, Sed88, CLR90]). However, many of these data structures make use of destructive updates (assignments) and multiple access-points (through the arbitrary placement of pointers). These features are not available in a functional programming language and therefore many of the "traditional" data structures are not easily implemented in this context.

A straightforward technique to implement an imperative data structure in a functional language is to simulate the behavior of its underlying model of computation (e.g. a RAM machine [AHU74]) in a functional language. This functional implementation, however, has an extra cost of $O(\log s)$ where s is the space used in the original imperative implementation [Pon88, BAG92]. A better approach is to design data structures for a functional setting, possibly adapting existing imperative data structures so that they can be implemented directly in a functional language. This approach have often yielded functional data structures with the same asymptotic complexity as that of the existent imperative data structures for the same abstract data type.

Data structures for a functional language are called *functional data structures*. Apart from the obvious interest for the functional programming community, they are, automatically, *persistent*. A data structure is persistent if it supports multiple versions. That is, if after an update operation, both the updated and the non-updated versions are still available[1]. Since imperative data structures make heavy use of destructive updates, usually only the last version of a data structure is available (i.e. they are *ephemeral*).

Persistent data structures are useful in areas such as computational geometry, text and file editing, and tree pattern matching [DSST89, DR91], therefore being of interest for programmers in general. In the context of imperative programming, some work has been done to develop persistent data structures (e.g. [DSST89, IS91]). A frequently used approach to achieve persistence is to implement a data structure in a functional style (e.g. [Mye84, Bur82]). The developments in the area of functional data structures are then of relevance for imperative programmers as well as for functional programmers.

Although in a functional language implementation persistence is guaranteed, it is common that the complexity of the operations on a data structure is not as good when it is used persistently as when it is used *single-threadedly*, i.e., when no persistent uses are actually made. It is often the case when the complexity of the data structure is amortized [Tar85] rather than worst-case, since usually the amortized analysis is no longer valid if the data structure is used persistently. Therefore, if persistence is required, it has to be considered when designing data structures for imperative as well as for functional languages.

In this work, we present many of the functional data structures that have been developed for persistent and non-persistent use. We restrict our coverage to the data types stack, queue, deque, flexible array, and catenable list. We define an abstract data type *sequence* that embeds all of them. Our sequences are similar to those considered in [PJ96] and [Tam96]. We describe various functional implementations of each of these data types in the language Haskell 1.3 [PHe96]. We use, in particular, the Hugs system [Jon96], which is a small and simple-to-use Haskell interpreter available in many platforms. The Haskell code presented should work without modifications in Haskell 1.4 [PHe97], and should also be easily translated to other modern functional languages such as ML [Pau91].

For most of the data structures described in this work, we have adapted their original implementation to the language Haskell. We have in many cases implemented operations in these data structures that had not been described in their original description, since we provide for all of them the operations that we define for sequences. These are usually a superset of the operations

[1] A data structure is sometimes called *partially* persistent if it supports queries on previous versions but not updates. If updates are possible in all versions, the data structure is said to be *fully* persistent. In this work, a persistent data structure is fully persistent in that terminology

originally described for each implementation. We also provide improved descriptions for many operations.

Haskell was chosen as the implementation language since it has become the standard lazy functional language. We use a lazy language since some of the implementations presented here only achieve the stated complexity in the presence of memoization, as offered by lazy functional languages. Therefore, if these implementations are translated to an eager language such as ML, their complexity may not be the same, at least for persistent uses [Oka96c]. When an amortized complexity is given, we do not usually give details of the analysis by which it is obtained, but instead refer the reader to the original source. These analysis usually require the use of the standard techniques for determining amortized complexity: the potential and banker methods [Tar85, CLR90], or the debit method described by Okasaki for persistent structures [Oka96b, Oka96c].

In the next section we give the definition of sequences used in this work and the operations supported by them. We define a class `Sequence` of which the implementations presented later will be made instances. We also describe how these implementations are classified. In section 3 we describe some implementations of trees that will be used in the next sections. In section 4 we describe some sequence implementations that can be used as efficient implementations for queues. The same is done in sections 5, 6, and 7, where various sequence implementations are described, that can be used as implementations of deques, flexible arrays, and catenable lists, respectively. We conclude with section 8 where we make some comments about this work and present some directions for further work.

2 Sequences

A *sequence* over the type A is a finite linear succession of zero or more elements where elements can be added or removed at either end. No operations are required over A, though if A provides an equality operation a membership test on the sequence is often defined. There is an implicit ordering on the elements of a sequence given by both the order and the end on which elements are added to the sequence. We will denote a sequence with n elements by $< x_0, \ldots, x_{n-1} >$. The operations we define on sequences are front, rear, cons, snoc, popFront, popRear and lengthSeq, defined informally for a sequence of n elements by:

$$\begin{aligned}
\mathsf{front}\ &< x_0, \ldots, x_{n-1} > = x_0 \\
\mathsf{rear}\ &< x_0, \ldots, x_{n-1} > = x_{n-1} \\
\mathsf{cons}\ y\ &< x_0, \ldots, x_{n-1} > = < y, x_0, \ldots, x_{n-1} > \\
\mathsf{snoc}\ &< x_0, \ldots, x_{n-1} >\ y = < x_0, \ldots, x_{n-1}, y > \\
\mathsf{popFront}\ &< x_0, \ldots, x_{n-1} > = < x_1, \ldots, x_{n-1} > \\
\mathsf{popRear}\ &< x_0, \ldots, x_{n-1} > = < x_0, \ldots, x_{n-2} > \\
\mathsf{lengthSeq}\ &< x_0, \ldots, x_{n-1} > = n
\end{aligned}$$

in addition to empty, which returns an empty sequence and isEmpty, which checks whether a sequence is empty or not.

A sequence described this way corresponds closely to what is usually called a *deque* in the literature on data structures. For many programs, we may want to have operations to find the n-th element of a sequence and to change it. We call a sequence with this operations an *Indexable Sequence* which provides:

$$\begin{aligned}
\mathsf{lookupSeq}\ i\ &< x_0, \ldots, x_{n-1} > = x_i \\
\mathsf{updateSeq}\ i\ y\ &< x_0, \ldots, x_{n-1} > = < x_0, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_{n-1} >
\end{aligned}$$

In addition, a catenation of sequences may be required, and we call a sequence with such operation a *Catenable Sequence*, with

$$\mathsf{cat}\ < x_0, \ldots, x_{n-1} > < y_0, \ldots, y_{m-1} > = < x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1} >$$

An alternative approach is to consider the indexing and catenation operations as part of the sequence definition, without defining the Indexable and Catenable subtypes. This is the view taken in [PJ96], where what we call Indexable-Catenable Sequences are defined as Sequences, as opposed to *Collections*, where insertion order is irrelevant and A is required to have equality, and they are both called *Bulk types*. We prefer however the definitions above, since for some of the implementations described later, indexing and catenation are not naturally implemented.

On function names. There are different approaches when deciding how to name operations that have the same meaning over different data types. For example the operation that returns the front element of a queue, a deque, a list and so on. There are at least three alternatives, each with its own pros and cons. The first alternative is to provide a totally different set of names for each type. The second is to provide the so called *qualified* names, so that the `front` function becomes, say, `Queue.front`, `Deque.front`, etc. The third alternative is to overload the names of functions so that the function name `front` can be invoked on all the suitable types. It is our opinion that the first alternative is the less convenient. The second alternative is easily implemented using the `module` construct of Haskell 1.4 [PHe97]. However, we are prevented of using this second alternative since the Hugs implementation of Haskell does not provide the `module` facilities to provide qualified names [Jon96]. We are then left with the overloading approach, which is implemented easily with the `class` mechanism.

To implement sequences in Haskell, we define the type class `Sequence` with the general operations described above and with two additional coercion operations to convert a sequence into a list and vice versa. A default implementation for toSeq is provided. For Indexable Sequences we

```
class Seq s where
    empty     :: s a
    isEmpty   :: s a -> Bool
    front     :: s a -> a
    rear      :: s a -> a
    cons      :: a -> s a -> s a
    snoc      :: s a -> a -> s a
    popFront  :: s a -> s a
    popRear   :: s a -> s a
    lengthSeq :: s a -> Int
    toList    :: s a -> [a]
    toSeq     :: [a] -> s a
    toSeq xs = foldr cons empty xs

class (Seq s) => IndSeq s where
    lookupSeq :: Int -> s a -> a
    updateSeq :: Int -> a -> s a -> s a

class (Seq s) => CatSeq s where
    cat :: s a -> s a -> s a
```

Figure 1: Class definitions for sequences.

define the class `IndSeq` and the class `CatSeq` for Catenable Sequences. The declarations for these definitions appear in figure 1.

As pointed out in [PJ96], sequences admit naturally the `map` method of the standard Haskell class `Functor` as well as the methods of the classes `Monad`, `MonadZero` and `MonadPlus` (in the case of Indexable Sequences). It is therefore valid to make `Functor` and `MonadZero` superclasses of `Seq` and `MonadPlus` a superclass of `CatSeq`. Many of the operations of these classes have a standard default definition in terms of the operations defined here for sequences. Therefore, in order to prevent our implementations to get unnecessarily long, we do not implement them in this work, except for the catenation operator when suitable.

There are a whole lot of operations that make sense for sequences, in particular many of the standard operations on lists, such as `take`, `drop`, `scan`, `split`, etc. Again, we do not implement them here.

Our sequences are, in a sense, a generalization of the traditional types stack, queue, deque, random-access list, flexible array and catenable list. Traditionally, the distinction between these abstract data types is considered to be the operations allowed for each of them. The data type queue, for example, is usually allowed only operations empty, isEmpty, front, dequeue, enqueue and sometimes length. These correspond to our (sequence) operations empty, isEmpty, front, popFront, snoc and lengthSeq. We depart from this tradition by embracing all these data types in our sequences. We classify the different implementations of sequence presented here by which operations are supported efficiently in each implementation, as opposed to which operations are allowed. We consider an operation efficient if, for a sequence of length n, the operation have complexity at most $O(\log n)$. The sequence implementations are then classified on the following groups:

Queues. Those sequence implementations where at least the following operations are efficient: front, snoc and popFront. One instance (`Queue`) is described in section 4.

Deques. Those sequence implementations where at least the following operations are efficient: front, rear, cons, snoc, popFront and popRear. We present two implementations in section 5: `DequeH` and `DequeRS`.

Flexible Arrays. Those (indexable) sequence implementations where at least the following operations are efficient: cons, snoc, popFront, popRear, lookupSeq and updateSeq. We describe two implementations in section 6: `ArrB` and `ArrK`.

Catenable Lists. Those (catenable) sequence implementations where at least the following operations are efficient: front, cons, popFront and catSeq. We describe two implementations in section 7.

Table 1 summarizes the complexities of the operations for each implementation.

	List	Queue	DequeH	DequeRS	ArrB	ArrK	CListO	CListRS*
front	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$
rear	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$
cons	$O(1)$	$O^a(1)$	$O^a(1)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O^a(1)$	$O(1)$
snoc	$O(n)$	$O^a(1)$	$O^a(1)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O^a(1)$	$O(1)$
popFront	$O(1)$	$O^a(1)$	$O^a(1)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O^a(1)$	$O(1)$
popRear	$O(n)$	$O(n)$	$O^a(1)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$
lengthSeq	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
toList	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
toSeq	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
lookupSeq	$O(n)$							
updateSeq	$O(n)$							
cat	$O(n)$	$O(n)$	$O(n)$				$O^a(1)$	$O(1)$

Table 1: Summary of complexities for the sequence implementations. Complexities presented as O^a are amortized.

In section 3 we describe some implementations of trees that are used later in some of the sequence implementations.

Most of our work can be used if a different operation-naming convention is required or if a restriction on the operations provided for each implementation is desired. Both changes can be easily done using the `module` construct since it provides detailed control on both aspects (in Haskell 1.4). The `class` mechanism is also optional.

The type of cons-lists predefined in Haskell (and in most of the functional programming languages) gives, as it is, an efficient implementation for stacks. Many programs are written with lists, even when used as a different data type, specially for rapid prototyping. With this purpose in mind, we provide Haskell declarations to make lists an instance of the sequence classes, as shown in figure 2.

```
instance Seq [] where
    empty      = []
    isEmpty    = null
    front      = head
    rear       = last
    popFront   = tail
    popRear    = init
    cons       = (:)
    snoc xs x  = xs ++ [x]
    lengthSeq  = length
    toList     = id
    toSeq      = id

instance IndSeq [] where
    lookupSeq = flip (!!)

    updateSeq 0 y (_:xs) = y:xs
    updateSeq n y (x:xs) = x : updateSeq (n-1) y xs
    updateSeq _ _ _      = error "UpdateSeq:index out of range"

instance CatSeq [] where
    cat = (++)
```

Figure 2: Instance declarations for lists.

3 Some basic trees

In this section we describe the tree types that are used in some of the sequence implementations described in the next sections. These tree types are leaf trees (type `LTree`) and Braun trees (type `BrTree`). In each case we only describe here what is needed for our sequence implementations.

3.1 Leaf trees

Leaf trees over a type A as defined here are non-empty full binary trees with elements (of A) stored in the leaves. By *full binary* we mean that every internal node has two successors. In addition, each internal node contains the number of elements in its descendants. In our implementation (in figure 3), they correspond to the type declared as:

```
data LTree a = Lf a | Bin Int (LTree a) (LTree a)
```

```
data LTree a = Lf a | Bin Int (LTree a) (LTree a)

join :: LTree a -> LTree a -> LTree a
join t1 t2 = Bin (size t1 + size t2) t1 t2

size :: LTree a -> Int
size (Lf _)      = 1
size (Bin n _ _) = n

frontier :: LTree a -> [a]
frontier t = frontierAcc t []
             where frontierAcc (Lf l)  = (l:)
                   frontierAcc (Bin _ l r) = frontierAcc l . frontierAcc r

ind :: Int -> LTree a -> a
ind 0  (Lf x) = x
ind i  (Bin s l r)
    | i < size l = ind i l
    | otherwise  = ind (i - size l) r
ind _ _ = error "ind: index out of bounds in LTree."

upd :: Int -> a -> LTree a -> LTree a
upd 0 x (Lf y) = Lf x
upd i x (Bin s l r)
    | i < size l = Bin s (upd i x l) r
    | otherwise  = Bin s l (upd (i - size l) x r)
upd _ _ _ = error "upd: non-existent position in LTree"

delLast :: LTree a -> LTree a
delLast (Bin _ l (Lf _)) = l
delLast (Bin n l r)      = Bin (n-1) l (delLast r)

delFirst :: LTree a -> LTree a
delFirst (Bin _ (Lf _) r) = r
delFirst (Bin n l      r) = Bin (n-1) (delFirst l) r

forestOfComp :: [a] -> [LTree a]
forestOfComp xs = foldr insComp [] (map Lf xs)
                  where insComp t (t1:ts)
                                | size t1 == size t = insComp (join t t1) ts
                                | otherwise         = t:t1:ts
                        insComp t [] = [t]
```

Figure 3: Implementation of `LTree`

The *size* of any subtree is the number of elements in it, easily calculated in constant time by:

```
size (Lf _) = 1
size (Bin s _ _) = s
```

When the constructor `Bin` is applied, the size field must be set appropriately. For convenience, we provide the function `join` that applies the constructor `Bin` to its arguments setting the size field:

```
join t1 t2 = Bin (size t1 + size t2) t1 t2
```

Let the *frontier* of a tree be the list of its elements in left-to-right order. We define the corresponding function in Haskell by:

```
frontier t = frontierAcc t []
             where frontierAcc (Lf l)    = (l:)
                   frontierAcc (Bin l r) = frontierAcc l . frontierAcc r
```

with time complexity proportional to the size of the tree.

The elements of a tree can be indexed according to their position in the frontier of the tree. The function ind with arguments i and t gives the i-th element of t (the first element is at position 0):

```
ind 0 (Lf x) = x
ind i (Bin s l r)
    | i < size l = ind i l
    | otherwise  = ind (i - size l) r
```

The tree upd i x t has the element x at position i and is otherwise the same as t. The function upd is implemented by:

```
upd 0 x (Lf y) = Lf x
upd i x (Bin s l r)
    | i < size l = Bin s (upd i x l) r
    | otherwise  = Bin s l (upd (i - size l) x r)
```

Both implementations take time proportional to the height of the tree.

The insertion and deletion of elements in a tree varies in general according to the properties that each tree is desired to have, such as balancing properties. However, we define very simple versions of functions for the deletion of the left and rightmost elements. We name these functions delFirst and delLast, satisfying the following equations, with x in A and t a leaf tree:

$$\begin{aligned}\mathsf{frontier}\ (\mathsf{delFirst}\ x\ t) &= \mathsf{tail}\ (\mathsf{frontier}\ t)\\ \mathsf{frontier}\ (\mathsf{delLast}\ t) &= \mathsf{init}\ (\mathsf{frontier}\ t)\end{aligned}$$

where tail l gives the list l without its first element and init l gives the list l without its last element. The functions delLast and delFirst are implemented by:

```
delLast (Bin _ l (Lf _)) = l
delLast (Bin n l r)      = Bin (n-1) l (delLast r)

delFirst (Bin _ (Lf _) r) = r
delFirst (Bin n l r)      = Bin (n-1) (delFirst l) r
```

with time complexities proportional to the height of the tree [2].

Perfect trees

A special kind of tree that is very useful in practice is one which is *perfectly size-balanced*, that is, whose every internal node has both descendants of equal size. A (full binary) leaf tree with this property is called *perfect* in [KD96] and a *complete binary leaf tree* in [Oka96b]. Among other properties, a *perfect* tree, for short, has all the leaves at the same depth and is of height r and size 2^r for some natural r, which is called its *rank*. A perfect tree is also a very regular structure in the sense that all its subtrees have the same properties, that is:

$$\text{If } \mathsf{perfect}\ t \text{ and } t' \text{ is a subtree of } t \text{ then:}\quad \mathsf{perfect}\ t' \tag{3.1}$$

[2] More precisely, proportional to the length of the right and left spine for delLast and delFirst, respectively

where $\mathsf{perfect}\ t$ holds if t is perfect.

The function forestOfPerf creates, given a list l, a list of perfect trees of strictly increasing sizes with elements drawn from l, more formally:

$$\begin{array}{rl} \mathsf{forestOfPerf}\ l = [t_0, \ldots, t_k] & \equiv \quad \text{(i)–(iii) below holds:} \\ \text{(i)} & \forall i \in \{0, \ldots, k\} : \mathsf{perfect}\ t_i \\ \text{(ii)} & \mathsf{frontier}\ t_0 +\!\!+ \ldots +\!\!+ \mathsf{frontier}\ t_k \ = \ l \\ \text{(iii)} & \forall i \in \{0, \ldots, k-1\} : \mathsf{size}\ t_i < \mathsf{size}\ t_{i+1} \end{array} \tag{3.2}$$

Note that there is a t_i of size 2^i for every non-zero digit of weight 2^i in the binary representation of $\mathsf{length}\ l$, and then the list $[t_0, \ldots, t_k]$ is well defined and its length is $\mathrm{O}(\log n)$.

The implementation of forestOfPerf converts every element of the given list `xs` into a leaf and then repeatedly invokes insPerf, which inserts each leaf in size-ascending order in its second argument, preserving the relative order of the members of the given list. The function insPerf keeps in its second argument a strictly size-increasing list of perfect trees which is built beginning with the empty list. It is always invoked with a tree whose size is at most the size of the first tree in the second argument. Therefore it preforms a join when the tree being inserted is of the same size of the first tree on the list. This creates a new perfect tree that is inserted in turn recursively, until the first tree in the list is of larger size than the tree to be inserted. The final list obtined can be seen to satisfy equation (3.2). The implementation is:

```
forestOfPerf xs = foldr insPerf [] (map Lf xs)
                  where insPerf t [] = [t]
                        insPerf t (t1:ts)
                            | size t1 == size t = insPerf (join t t1) ts
                            | otherwise         = t:t1:ts
```

Note that the cost of insPerf is proportional to the number of join's performed, one for each node of every tree. A perfect tree of size 2^k has $\mathrm{O}(2^k)$ nodes, therefore the total number of join's to create $[t_0, \ldots, t_k]$ is $\mathrm{O}(\mathsf{size}\ t_0), + \cdots +, \mathrm{O}(\mathsf{size}\ t_k)$ and is then proportional to the length of the given list.

The functions ind, upd, delFirst and delLast are described similarly in [KD96], where leaf trees are used to implement dynamic lists, flexible arrays and dynamic sets.

3.2 Braun trees

Braun trees [BR83] are node-oriented binary trees whose shape is uniquely determined by the number of elements on it, i.e., by its *size*. Every subtree of a Braun tree is also a Braun tree. They are *size-balanced*: the size of the left and right subtrees differ by at most one, in which case the left subtree is the larger one. That is:

$$\begin{array}{rl} \multicolumn{2}{l}{\text{For any Braun tree } t \text{ with left subtree } l \text{ and right subtree } r\text{:}} \\ \text{(i)} & \mathsf{size}\ l = \mathsf{size}\ r \lor \mathsf{size}\ l = \mathsf{size}\ r + 1 \\ \text{(ii)} & l \text{ and } r \text{ are Braun trees} \end{array} \tag{3.3}$$

The height of a Braun tree is always the minimum possible, and is therefore logarithmic on its size. We implement Braun trees by the type `BrTree` as:

```
data BrTree a = Nil | Node a (BrTree a) (BrTree a)
```

A Braun tree t of size n has elements at positions 0 to $n-1$. Let a_i be the element at position i in t. Then a_0 is stored in the root of t. Elements of the form a_{2i+1} are stored in the left subtree and those of the form a_{2i} with $i > 0$ are stored in the right subtree. The element at position $2i+1$ in t is at position i in the left subtree, while the element at position $2i$ in t is at position $i-1$ in the right subtree.

To obtain the list of elements of a Braun tree ordered by their positions we define the function brOrder. Note that the list of elements in the left subtree of t is $[a_1, a_3, \ldots, a_{2i+1}, \ldots]$ indexed with their positions in t. Similarly, the list of elements in the right subtree is $[a_2, a_4, \ldots, a_{2i}, \ldots]$. Then to obtain the list of elements in t we must take one element from the first list, then one from the second, and so on. This is done by the function merge. The implementation is:

```
data BrTree a = Nil | Node a (BrTree a) (BrTree a)

brOrder :: BrTree a -> [a]
brOrder Nil          = []
brOrder (Node x l r) = x : merge (brOrder l) (brOrder r)
                       where merge xs     [] = xs
                             merge []     ys = ys
                             merge (x:xs) ys = x : merge xs ys

ind :: Int -> BrTree a -> a
ind 0     (Node x _ _) = x
ind (i+1) (Node _ l r)
    | even i    = ind (i `div` 2) l
    | otherwise = ind (i `div` 2) r
ind _ _ = error "ind: index out of range in BrTree"

upd :: Int -> a -> BrTree a -> BrTree a
upd 0     x Nil          = Node x Nil Nil
upd 0     x (Node _ l r) = Node x l r
upd (i+1) x (Node y l r)
    | even i    = Node y (upd (i `div` 2) x l) r
    | otherwise = Node y l (upd (i `div` 2) x r)
upd _ _ _ = error "upd: index out of range in BrTree"

insFirst :: a -> BrTree a -> BrTree a
insFirst x Nil          = Node x Nil Nil
insFirst x (Node y l r) = Node x (insFirst y r) l

delFirst:: BrTree a -> BrTree a
delFirst (Node _ Nil Nil)          = Nil
delFirst (Node _ l@(Node y _ _) r) = Node y r (delFirst l)

insLastAt :: Int -> a -> BrTree a -> BrTree a
insLastAt n x t = upd n x t

delLastAt :: Int -> BrTree a -> BrTree a
delLastAt 0     _ = Nil
delLastAt (n+1) (Node x l r)
    | even n    = Node x (delLastAt (n `div` 2) l) r
    | otherwise = Node x l (delLastAt (n `div` 2) r)

size :: BrTree a -> Int
size Nil          = 0
size (Node _ l r) = let n = size r
                    in 1 + 2*n + diff l n
                    where diff Nil              0 = 0
                          diff (Node _ Nil Nil) 0 = 1
                          diff (Node _ l r)  (i+1)
                              | even i    = diff l (i `div` 2)
                              | otherwise = diff r (i `div` 2)

mkBrT :: [a] -> BrTree a
mkBrT = head . foldr build [Nil] . levels 1
        where build (n,xs) ts = let (ts1,ts2) = splitAt n (ts ++ repeat Nil)
                                in zipWith3 Node xs ts1 ts2
              levels n [] = []
              levels n xs = let (xs1,xs2) = splitAt n xs
                            in (n,xs1) : levels (2*n) xs2
```

Figure 4: Implementation of BrTree

```
brOrder Nil          = []
brOrder (Node x s t) = x : merge (brOrder s) (brOrder t)
                       where merge xs     [] = xs
                             merge []     ys = ys
                             merge (x:xs) ys = x : merge ys xs
```

which has complexity $O(n \log n)$ for a tree of size n. Since the shape of a Braun tree is determined by its size, brOrder is a bijection between Braun trees over A and the lists over A.

Thanks to the properties of Braun trees, the indexing operations are very simple, as well as the addition or removal of the first and last elements. The element at position 0 of a tree is obtained from its root. The $(i+1)$-th element is at position $i \operatorname{div} 2$ of the left or right subtree. It is in the left subtree if $i+1$ is odd, i.e. if i is even. Similarly, it is in the right subtree if i is odd. This gives the following code for the indexing function ind:

```
ind 0     (Node x _ _) = x
ind (i+1) (Node _ l r)
    | even i    = ind (i `div` 2) l
    | otherwise = ind (i `div` 2) r
```

and the very similar function upd, which changes the element at the given position to be `x`:

```
upd 0     x (Node _ l r) = Node x l r
upd (i+1) x (Node y l r)
    | even i    = Node y (upd (i `div` 2) x l) r
    | otherwise = Node y l (upd (i `div` 2) x r)
```

In languages with $n*k+p$-patterns, like Gofer [Jon91], ind (and similarly upd) could be expressed using those patterns (as in [Hoo92a]):

```
ind 0       (Node x _ _) = x
ind (2*k+1) (Node x l _) = ind k l
ind (2*k+2) (Node x _ r) = ind k r
```

These operations are clearly seen to have complexity $O(\log i)$ when accessing position i.

To insert and remove an element at position 0 we define the functions insFirst and delFirst, that satisfy:

$$\begin{aligned} \mathsf{brOrder}\ (\mathsf{insFirst}\ x\ t) &= [x] +\!\!+ \mathsf{brOrder}\ t \\ \mathsf{brOrder}\ (\mathsf{delFirst}\ t) &= \mathsf{init}\ (\mathsf{brOrder}\ t) \end{aligned}$$

Let $t1$ be the tree insFirst x t. Note that elements at odd positions in t are at even positions in $t1$ in the same relative order, hence the left subtree of t must be the right subtree of $t1$. Similarly, the right subtree of t must be the left one of $t1$ but must incorporate x. This should make clear the following implementation:

```
insFirst x Nil          = Node x Nil Nil
insFirst x (Node y l r) = Node x (insFirst y r) l
```

A symmetric reasoning gives the following for delFirst:

```
delFirst (Node x Nil Nil)          = Nil
delFirst (Node x l@(Node y _ _) r) = Node y r (delFirst l)
```

The complexity of these functions is proportional to the height of the tree, therefore logarithmic in the size.

Functions similar to ind,upd, insFirst and delFirst can be found in [Hoo92a] and in chapter 4 of [Pau91]. In both sources Braun Trees are described in the context of its use to implement flexible arrays.

To calculate the size of a Braun tree, we use an algorithm recently described by Okasaki [Oka97b]. The algorithm uses property (3.3)(ii): for any Braun tree, if the right subtree is of size

n, the size of the left subtree is either n or $n+1$. Discerning between sizes n or $n+1$ can be implemented more efficiently than a direct approach to calculate (any) size. To this end, a function diff is defined satisfying the following property: for any Braun tree t,

$$\mathsf{size}\ t = n + \mathsf{diff}\ t\ n, \text{ if } n \leq \mathsf{size}\ t \leq n+1 \tag{3.4}$$

That is, $\mathsf{diff}\ t\ n$ is 0 if $\mathsf{size}\ t = n$ or 1 if $\mathsf{size}\ t = n+1$.

To implement diff, the case $n = 0$ is trivial:

```
diff Nil              0 = 0
diff (Node x Nil Nil) 0 = 1
```

If the second argument is $i+1$, we analyze separately when i is even and when i is odd. If i is even, let $i = 2k$ and l and r the left and right subtrees of a Braun tree t. Then we do some calculations with the premise of (3.4):

$$\begin{array}{ll} & 2k+1 \leq \mathsf{size}\ t \leq 2k+2 \\ \equiv & \{\mathsf{size}\ t = 1 + \mathsf{size}\ l + \mathsf{size}\ r\} \\ & 2k \leq \mathsf{size}\ l + \mathsf{size}\ r \leq 2k+1 \\ \equiv & \{\text{by } (3.3)(\text{i})\} \\ & k \leq \mathsf{size}\ l \leq k+1 \wedge \mathsf{size}\ r = k \\ \Rightarrow & \{\text{by } (3.4) \text{ and } (3.3)(\text{ii})\} \\ & \mathsf{size}\ l = k + \mathsf{diff}\ l\ k \wedge \mathsf{size}\ r = k \end{array} \tag{3.5}$$

We use this result to calculate $\mathsf{diff}\ t\ (2k+1)$ from (3.4), assuming that $2k+1 \leq \mathsf{size}\ t \leq 2k+2$:

$$\begin{array}{ll} & \mathsf{size}\ t = 2k+1+\mathsf{diff}\ t\ (2k+1) \\ \equiv & \{\mathsf{size}\ t = 1 + \mathsf{size}\ l + \mathsf{size}\ r\} \\ & 1 + \mathsf{size}\ l + \mathsf{size}\ r = 2k+1+\mathsf{diff}\ t\ (2k+1) \\ \equiv & \{\text{by } (3.5)\} \\ & 1 + 2k + \mathsf{diff}\ l\ k = 2k+1+\mathsf{diff}\ t\ (2k+1) \\ \equiv & \\ & \mathsf{diff}\ t\ (2k+1) = \mathsf{diff}\ l\ k \end{array} \tag{3.6}$$

If the second argument to diff is $i+1$ with $i = 2k+1$, we calculate, as in (3.5):

$$\begin{array}{ll} & 2k+2 \leq \mathsf{size}\ t \leq 2k+3 \\ \equiv & \{\mathsf{size}\ t = 1 + \mathsf{size}\ l + \mathsf{size}\ r\} \\ & 2k+1 \leq \mathsf{size}\ l + \mathsf{size}\ r \leq 2k+2 \\ \equiv & \{\text{by } (3.3)(\text{i})\} \\ & \mathsf{size}\ l = k+1 \wedge k \leq \mathsf{size}\ r \leq k+1 \wedge \\ \Rightarrow & \{\text{by } (3.4) \text{ and } (3.3)(\text{ii})\} \\ & \mathsf{size}\ l = k+1 \wedge \mathsf{size}\ r = k + \mathsf{diff}\ r\ k \end{array} \tag{3.7}$$

And now we redo (3.6) for $\mathsf{diff}\ t\ (2k+2)$, assuming $2k+2 \leq \mathsf{size}\ t \leq 2k+3$:

$$\begin{array}{ll} & \mathsf{size}\ t = 2k+2+\mathsf{diff}\ t\ (2k+2) \\ \equiv & \{\text{omitted steps, similar to } (3.6)\} \\ & \mathsf{diff}\ t\ (2k+2) = \mathsf{diff}\ r\ k \end{array} \tag{3.8}$$

Finally, note that if $i+1 = 2k+1$ or $i+1 = 2k+2$, we have $k = i \operatorname{div} 2$ and, by (3.6) and (3.8), $\mathsf{diff}\ t\ (i+1)$ can be implemented by:

```
diff (Node _ l r) (i+1) =
    | even i    = diff l (i `div` 2)
    | otherwise = diff r (i `div` 2)
```

It is clearly seen that this implementation of `diff t n` runs in $\mathrm{O}(\log(\mathsf{size}\,t))$ time.

Now, size can be implemented using diff by:

```
size Nil          = 0
size (Node _ l r) = let n = size r
                     in 1 + 2*n + diff l n
```

The second clause follows from (3.3) and (3.4). The running time of this implementation is proportional to the total cost of the calls to diff. If the tree is of size n, diff is called with second arguments $n \operatorname{div} 2, n \operatorname{div} 4, n \operatorname{div} 8, \ldots$; and then it has a total cost of $\log(n \operatorname{div} 2) + \log(n \operatorname{div} 4) + \log(n \operatorname{div} 8) +, \ldots$; which is $\mathrm{O}(\log^2 n)$. A direct approach to implement size using $\mathsf{size}\,t = \mathsf{size}\,l + \mathsf{size}\,r$ would have time complexity $\mathrm{O}(n)$.

We can now define functions to insert and remove an element at the last position of a Braun tree. In contrast to the first position which is always at the root node, the last position is at the left or right subtree depending of the size of the tree.

Once the size of a tree is known, an algorithm similar to that used for the function ind guides the search for the last element, which is at position $n-1$ for a tree of size n. The function delLast is then implemented (for a non-`Nil t`) by:

```
delLast t = delLastAt (size t - 1) t

delLastAt 0 _ = Nil
delLastAt (n+1) (Node x l r)
    | even n    = Node x (delLastAt (n `div` 2) l) r
    | otherwise = Node x l (delLastAt (n `div` 2) r)
```

To insert an element at the last position of a tree of size n, we must find where position n should be and do the insertion. A similar implementation to delLast can be used, but we present instead an implementation adapted from [Pau91]. The key idea is that upd as defined so far, would almost do the job if called with argument n. The problem is that when it reaches the desired position, it is a `Nil` node. The problem is solved adding the following clause for `upd`:

```
upd 0 x Nil = Node x Nil Nil
```

and then insLast can be implemented by:

```
insLast x t = insLastAt (size t) x t

insLastAt n x t = upd n x t
```

Since delLast and insLast are not used in the next sections, only its versions with size are included in figure 4. These implementations of insLastAt and delLastAt run in $\mathrm{O}(\log n)$ for a tree of size n.

Finally, we describe an algorithm also published by Okasaki in [Oka97b] to create a Braun tree with elements taken from a list. What we want is a function mkBrTree defined for any list l by:

$$\mathsf{brOrder}\ (\mathsf{mkBrTree}\ l) = l$$

Note that an straightforward algorithm for mkBrTree would just repeatedly call insFirst (or insLast) with each element of the list. This approach would yield an implementation with complexity $\mathrm{O}(|\,l\,| \log |\,l\,|)$. Okasaki's approach is implemented in $\mathrm{O}(|\,l\,|)$.

First, let us present some relations between a Braun tree t over A and the list $\mathsf{brOrder}\ t$. Assume that no element of A appears in t more than once so that, given any element $x \in A$ in t, we can denote its left child as $\mathsf{left}\ x$ and its right child as $\mathsf{right}\ x$ without ambiguity. This assumption will not affect the generality of the algorithm obtained for mkBrTree , since elements $x \in A$ will only be referenced by its position in l, therefore its actual value will not matter.

Consider the root of t at depth 0, its children at depth 1 and so on. Let $k = \lfloor \log(\mathsf{size}\ t) \rfloor$, then t has elements at depths $0, \ldots, k$ and all levels except possibly k are full. There are 2^i elements at depth i with $i \in \{0, \ldots, k-1\}$. Since there are $2^k - 1$ of these elements, the remaining $\mathsf{size}\ t - (2^k - 1)$ are at depth k.

Let level i t be the list of elements at depth i in t, in left-to-right order. That is, the function level can be defined by:

$$\begin{array}{ll} \text{For any } i \in \{0,\ldots,k\}, \mathsf{level}\ i\ t \text{ satisfies (i) and (ii) below:} & (3.9) \\ \text{(i)} \quad \mathsf{brOrder}\ t = \mathsf{level}\ 0\ t +\!\!+ \mathsf{level}\ 1\ t +\!\!+ \ldots +\!\!+ \mathsf{level}\ k\ t & \\ \text{(ii)} \quad |\,\mathsf{level}\ i\ t\,| = \begin{cases} 2^i & \text{if } i \in \{0,\ldots,k-1\} \\ \mathsf{size}\ t - (2^k - 1) & \text{if } i = k \end{cases} & \end{array}$$

Then, we can state the relationship between the elements at (full) levels and those at the next level (if it has elements on it) by

$$\begin{array}{c} \text{If } \mathsf{level}\ i\ t = [a_0,\ldots,a_{2^i-1}] \wedge \mathsf{level}\ (i+1)\ t = [b_0,\ldots,b_m] \text{ then:} \\ \mathsf{left}\ a_j = b_j \text{ if } j \leq m \\ \mathsf{right}\ a_j = b_{j+2^i} \text{ if } j + 2^i \leq m \end{array} \tag{3.10}$$

This property along with definition (3.9) suggests a way of inverting brOrder to obtain an implementation for mkBrTree in two phases. First, the list of elements to be in the tree is partitioned into what would be the levels of the tree. Then, (3.10) is used to create the tree from the partition, as we will explain later.

The first step is to partition the given list into levels. This is done by the function levels, which gives the desired partition but each level i is paired with 2^i. This is the size of each level except possibly for the last one. The function levels is implemented recursively and is provided with a first argument that indicates the length of the level to be created at each call:

```
levels n [] = []
levels n xs = let (xs1,xs2) = splitAt n xs
              in (n,xs1) : levels (2*n) xs2
```

The required partition of a list l is obtained invoking `levels 1 l`. Note that this algorithm for levels has $O(|\,l\,|)$ complexity.

Now we must find how to create the tree from the levels-partition. Assume that level i and $i+1$ are both full. By (3.9) we have: level i $t = [a_0,\ldots,a_{2^i-1}]$ and level $(i+1)$ $t = [b_0,\ldots,b_{2^{i+1}-1}]$. From (3.10) we know that b_0 and b_{2^i} are respectively the left and right children of a_0, b_1 and b_{2^i+1} are the left and right children of b_1 and so on. Then, if we split level $(i+1)$ t in two halves of length 2^i, we obtain $l_1 = [b_0,\ldots,b_{2^i-1}]$ and $l_2 = [b_{2^i},\ldots,b_{2^{i+1}-i}]$. But l_1 and l_2 are respectively the lists of left and right children of the members of level i t in the correct order. It is now straightforward to define a function build that will create, for any level i t, the list ts of subtrees whose roots are the members of level i t, given the analogous ts_{i+1} obtained from level $(i+1)$ t. The list ts_{i+1} must be split in halves and the constructor `Node` is mapped over the elements of level $(i+1)$ t taking its second and third arguments from the first and second half of ts_{i+1}. This could be implemented by:

```
build (n,xs) ts = let (ts1,ts2) = splitAt n ts
                  in zipWith3 Node xs ts1 ts2
```

The parameter n is the length of the level being processed, since the function levels actually pairs each level with its length.

However, (3.10) does not directly provide a way of creating the nodes of t whose left and/or right child do not appear in brOrder t. It is easily seen that in t, any missing child in brOrder t is a `Nil` node. Therefore the previous implementation is modified to generate `Nil` nodes whenever a subtree is required and is not in the argument. The final implementation of build is:

```
build (n,xs) ts = let (ts1,ts2) = splitAt n (ts ++ repeat Nil)
                  in zipWith3 Node xs ts1 ts2
```

The function build is then folded over the partition created by levels. Note that if the list given to mkBrTree is empty, its partition is also empty and build should produce a list with one `Nil` subtree. Therefore the fold is provided with `[Nil]` for an empty list. Since a one-element list is obtained with build, this element is extracted with head:

```
mkBrTree = head . foldr build [Nil] . levels 1
```

If this implementation of mkBrTree is invoked on l, each call of `build` runs in time proportional to its parameter n, whose successive values during the fold sum up to $\mathrm{O}(|\,l\,|)$. Since levels also runs in $\mathrm{O}(|\,l\,|)$, the time complexity of mkBrTree is $\mathrm{O}(|\,l\,|)$.

The description of this algorithm for mkBrTree in [Oka97b] is less analytical and less precise but possibly more intuitive than ours, since the former is more "visual".

4 Queues

A queue should provide efficient implementations of the operations empty, isEmpty, front, popFront and snoc. In imperative programming languages, this is easily achieved by maintaining pointers to both ends of a linked list or by using a circular array [AHU83, CLR90] providing all these operations with O(1) worst-case complexity.

The standard queue implementation in functional languages has been around for some time [Gri81, Bur82] and appears in modern functional programming textbooks [Pau91, Tho96]. It is very simple, adapting the idea of pointers to both ends of a queue to a functional setting, as described below.

The trick is to represent a queue by a pair of lists f and r such that the intended queue is $f + \mathsf{reverse}\ r$. The empty queue is then a pair of empty lists. The front of the queue is either the front of f, thus accessible in constant time, or the last element of r if f is empty, requiring a traversal of r in order to access it. To avoid the latter case to occur, f is not allowed to be empty unless the whole queue is empty, i.e., the following invariant is maintained — [Bur82] and [Tho96] do not require this invariant since they combine the operations front and popFront:

$$f = [\,] \implies r = [\,] \tag{4.1}$$

With this invariant, the front element of the queue is always the front of f, so popFront just take the tail of f, possibly violating the invariant by causing f to be empty. To *normalize* the resulting queue, i.e. to restore the invariant, normQ is invoked, which brings the whole queue to f. Normalization is also called by snoc since it may cause the invariant to be violated.

```
newtype Queue a = Q([a],[a])

instance Seq Queue where
    empty = (Q([],[]))

    isEmpty (Q([],_)) = True
    isEmpty _         = False

    front (Q(x:xs,_)) = x
    front _           = error "front: queue is empty"

    rear (Q(_ ,y:ys)) = y
    rear (Q([],_   )) = error "rear: queue is empty"
    rear (Q(xs,_   )) = last xs

    popFront (Q(x:xs,ys)) = Q(normQ (xs,ys))
    popFront _            = error "popFront: queue is empty"

    popRear (Q(xs,y:ys)) = Q(xs,ys)
    popRear (Q([],_   )) = error "popRear: queue is empty"
    popRear (Q(xs,_   )) = Q(init xs,[])

    cons x (Q(xs,ys)) = Q(x:xs,ys)

    snoc (Q(xs,ys)) y = Q(normQ (xs,y:ys))

    lengthSeq (Q(xs,ys)) = length xs + length ys

    toList (Q(xs,ys)) = xs ++ reverse ys

    toSeq xs = Q(xs,[])

normQ :: ([a],[a]) -> ([a],[a])
normQ ([],r) = (reverse r,[])
normQ q      = q

instance CatSeq Queue where
    catSeq (Q(xs1,ys1)) (Q(xs2,ys2)) = Q(xs1++reverse ys1++xs2,ys2)
```

Figure 5: Implementation of Queue.

The operations described so far take constant time except when a normalization is performed, which takes time proportional to $|r|$. We can amortize this cost over the (at least) $|r|$ snoc's that were necessary to create r, thus obtaining a O(1) amortized complexity for all the operations that are required to be efficient. This argument is valid if a queue is used single-threadedly. With persistent uses, lazy evaluation helps in some cases, but is not enough to guarantee O(1) complexity for certain unbounded persistent uses (see Chapter 3 of [Oka96b]).

If persistence is required, invariant (4.1) can be strengthened to:

$$|f| \geq |r|$$

which, under lazy evaluation, gives O(1) amortized complexity. The only change required is to the operation normQ, which now must check for the new invariant, as in:

```
normQ q@(f,r)
    | length f >= length r = q
    | otherwise            = (f ++ reverse r,[])
```

It would be convenient to add two integer fields to maintain the lengths of f and r. A detailed analysis of the complexity of both implementations can be found in [Oka96a].

The additional operations are easy to implement, but popRear, lengthSeq and the coercions take time proportional to the length of the queue. If desired, it is straightforward to make a constant time lengthSeq by adding an integer to the pair of lists to maintain the length of the queue, without affecting the complexity of the other operations.

The Haskell code in figure 5 implements the type Queue as described. The type `([a],[a])` is renamed to `Q([a],[a])` since Haskell requires a type constructor to define an instance of a class [PHe96]. We also define a catenation operation in case it is desired.

A functional implementation of queues with constant-time operations even with persistent use is described in [HM81]. The constant-time is achieved by performing the reverse of the rear list incrementally in intermediate lists. The implementation obtained is however very complex, consisting of six lists and two counters. A similar technique is used by Okasaki in [Oka95b], but using a lazy list to perform the reverse incrementally. He obtained a data structure consisting of three lists, one of which is a lazy list. The deque implementations described in the next section also provide efficient implementations for queues.

5 Deques

A deque (or double-ended queue) has efficient operations to work on both ends. A simple implementation is possible that gives O(1) amortized complexity by modifying the previous implementation of `Queue` to treat both ends of the sequence symmetrically. This implementation is described in [GT86], but we prefer the presentation in a functional setting of [Hoo92b] which includes its formal derivation. A much more complicated implementation is that of [KT95] which has O(1) worst-case complexity. We now present both implementations.

A deque with indexing operations is called a *flexible array* in this work, and some of its implementations are described in section 6.

5.1 Deques as pair of lists

Hoogerwoord represents a deque by a pair (f, r), where the intended deque is $f \mathbin{+\!\!+} \mathsf{reverse}\ r$. We add a third component n to maintain the length of the deque since it is used by the normalization procedure. The invariant of the type is:

$$(f = [\,] \implies |\, r \,| \leq 1) \;\wedge\; (r = [\,] \implies |\, f \,| \leq 1) \tag{5.1}$$

which ensures that the elements at the ends of the queue are at the front of either f or r.

The implementation of `Deque` appears in figure 6. The functions cons, snoc, popFront and popRear just insert or take out the front of f or r accordingly and call normD to normalize the deque in case the invariant is violated.

The normalization is done by splitting the deque in two halves and making them the new f and r in the proper order, as in:

```
normD d@(n,[],ys)
        | n <= 1    = d
        | otherwise = (n,reverse f,r)
        where (r,f) = splitAt (n `div` 2) ys
normD d@(n,xs,[])
        | n <= 1    = d
        | otherwise = (n,f,reverse r)
        where (f,r) = splitAt (n `div` 2) xs
normD d = d
```

The cost of the normalization is proportional to the length of the deque (either $|\, f \,|$ or $|\, r \,|$) but can be amortized over the at least $|\, f \,|$ or $|\, r \,|$ cons's and snoc's that had to be invoked before, thus obtaining constant time amortized complexity. A proof of this bound using the potential method [Tar85] when used single-threadedly is done in [Hoo92b]. However, when used persistently, this argument fails since the cons's and snoc's can be re-executed by different versions of the data structure.

The coercion function LengthSeq simply returns the value of n in constant time. Finally, toList returns the intended deque as a list and toSeq puts the whole list as f and normalize the deque. Both have time complexity proportional to the length of the deque. Again, we implement a catenation operation that takes time proportional to the length of the result deque.

In addition to the sequence operations, we implement a folding function for the type `DequeH`, which is used in the implementations of catenable lists in section 7. It is implemented as a right to left fold, following the schema of the `foldr` function over lists defined in Haskell [PHe96]. Though it is defined over the type `DequeH`, it can in fact be defined on the class `Sequence`, since it only invokes sequence operations.

As for `Queue`, a generalization of (5.1) restores the constant time bound for persistent uses, but only with lazy evaluation [Oka95b]. The generalized invariant is of the form:

$$|\, f \,| \leq c \,|\, r \,| + 1 \wedge \;|\, r \,| \leq c \,|\, f \,| + 1$$

with $c \geq 2$. A detailed complexity analysis of both cases can be found in [Oka95b] and [Oka96b]. By performing the normalization in an incremental way, worst-case constant complexity may be achieved. This technique has been applied for deques in [Hoo82, CG93, Oka95b].

```
newtype DequeH a = D(Int,[a],[a])

instance Seq DequeH where
    empty = D(0,[],[])

    isEmpty (D(0,_,_)) = True
    isEmpty _        = False

    front (D(_,x:xs,_  )) = x
    front (D(_,_   ,[y])) = y
    front _               = error "front: deque is empty"

    rear (D(_,_  ,y:ys)) = y
    rear (D(_,[x],_   )) = x
    rear _               = error "rear: deque is empty"

    cons a (D(n,x,y)) = D(normD (n+1,a:x,y))

    snoc (D(n,x,y)) a = D(normD (n+1,x,a:y))

    popFront (D(_,[]   ,[_])) = empty
    popFront (D(n,x:xs,ys )) = D(normD (n-1,xs,ys))
    popFront _                = error "popFront: deque is empty"

    popRear (D(_,[_],[]  )) = empty
    popRear (D(n,xs ,y:ys)) = D(normD (n-1,xs,ys))
    popRear _               = error "popRear: deque is empty"

    lengthSeq (D(n,_,_)) = n

    toList(D(_,xs,ys)) = xs ++ reverse ys

    toSeq xs = D(normD(length xs,xs,[]))

normD d@(n,[],ys)
        | n <= 1    = d
        | otherwise = (n,reverse f,r)
        where (r,f) = splitAt (n `div` 2) ys
normD d@(n,xs,[])
        | n <= 1    = d
        | otherwise = (n,f,reverse r)
        where (f,r) = splitAt (n `div` 2) xs
normD d = d

instance CatSeq DequeH where
    cat (D(n1,xs1,ys1)) (D(n2,xs2,ys2)) = D(normD(n1+n2,xs1++reverse ys1 ++xs2,ys2))

foldrD :: (a -> b -> b) -> b -> DequeH a -> b
foldrD f e d
    | isEmpty d = e
    | otherwise = f (front d) (foldrD f e (popFront d))
```

Figure 6: Implementation of DequeH.

5.2 Recursive-Slowdown deques

Kaplan and Tarjan describe in [KT95] a technique for structuring data type representations which they call *recursive-slowdown*, adapting ideas from the so called *segmented* binary numbers implementation (see for example Chapter 3 of [Oka96b]) and from the work on [BST92]. They use this technique to implement deques and catenable deques with constant time operations even with persistent use. Their data structure is described for an imperative language (using pointers) but in an applicative style and taking care to maintain persistence, which requires some copying and bookkeeping operations not explicitly needed in a functional language implementation. In this section we describe a deque implementation that follows closely the description given by Kaplan and Tarjan.

The basic idea of the recursive-slowdown technique is to recursively decompose a data-structure —a deque in this case— into *levels*. Operations in each level are implemented with worst-case constant time complexity. Each operation in the whole data structure manipulates the top level and only occasionally manipulates further levels. We first describe an implementation that, when further levels need to be manipulated, may require more than constant time to do so. Then we describe a final version that always performs in worst-case constant time.

General description of the data structure

A deque over A, or *deq* A is either empty or is decomposed in a *prefix* over A, an (inner) deque over $A \times A$ (or *deq* $(A \times A)$), and a *suffix* over A. A prefix and a suffix over A are both deques with at most five elements. Since the length of prefixes and suffixes is limited, operations on them perform in constant time for any reasonable implementation. In particular, the implementation described in the previous section could be used. We prefer to use lists instead, to simplify our implementation and mainly to benefit from the use of pattern-matching on them. We can describe a *deq* A by:

$$\begin{array}{l} \text{A } \mathit{deq}\ A \text{ is either:} \\ \begin{cases} \mathit{empty} & \text{the empty deque} \\ (px, d, sx) & px, sx \text{ lists over } A,\ d \text{ a } \mathit{deq}\ (A \times A) \end{cases} \end{array} \tag{5.2}$$

The elements of the sequence defined by a deque are ordered as follows. First are the elements of the prefix. Then the elements of the inner deque, ordered in the same way. The first component of each pair precedes the second. Last are the elements of the suffix. The elements in a prefix or a suffix are in the order in which they are stored in each list.

A deque with this description may contain an inner deque which in turn may contain another deque and so on. This deques are enumerated from the outermost to the innermost by:

$$\begin{array}{l} \text{For } d \text{ a } \mathit{deq}\ A\text{, let:} \\ \quad d_0 = d \\ \quad d_{i+1} = d' \quad \text{if } d_i = (px_i, d', sx_i) \end{array} \tag{5.3}$$

A deque d with $m+1$ levels, i.e. with m internal deques, could be represented as a list l of its non-recursive components. That is, as a list of *ps*'s, or prefix-suffix pairs. This list could be defined by:

$$\begin{array}{l} l = [ps_0, ps_1, \ldots, ps_{m-1}, ps_m] \\ \text{with: } \; ps_i = (px_i, sx_i) \quad \text{if } d_i = (px_i, d_{i+1}, sx_i) \\ \qquad\quad ps_m = () \quad \text{if } d_m \text{ is empty } (() \text{ is the empty pair}) \end{array}$$

However, this list can not be implemented in Haskell, since the *ps*'s are not of the same type. Indeed, according to (5.2) and (5.3), the lists in ps_0 are over A, those in ps_1 contain pairs of elements of A, those on ps_2 contain pairs of pairs, and so on. In general, the prefix and suffix in ps_i with $i > 0$ are lists over $A^i = A^{i-1} \times A^{i-1}$, with ps_0 a pair of lists over $A^0 = A$. But a pair of type A^i can be easily represented by a perfect leaf tree of height i, since the subtrees of the latter are precisely perfect trees of height $i-1$. Elements of A can be represented by trees of height 0, that is, by leaves. The order of the elements of A in a pair in A^i corresponds to the left-to-right order of the leaves of the tree. Leaf trees were described in section 3.1, including perfect (leaf) trees.

Using the notation A^i as above, the list l representing a deque d can be specified by:

$$\begin{array}{l}\text{A deque } d \text{ with } m+1 \text{ levels is represented by } l = [ps_0, \ldots, ps_m] \text{ where:}\\ \text{(i)} \quad ps_i = \begin{cases} (px_i, sx_i) & \text{if } d_i = (px_i, d_{i+1}, sx_i) \\ ps_m = () & \text{if } d_m \text{ is empty} \end{cases}\\ \text{(ii)} \quad px_i, sx_i \text{ are lists over } A^i \text{ with at most 5 elements each}\end{array} \tag{5.4}$$

In this description of l, only the heights of the trees varies from level to level. The type system of Haskell does not allow us to parameterize the trees by their sizes, thus we will implement them as general leaf trees. However, the implementation will respect the previous description.

We use description (5.4) to implement these deques in Haskell by the type `DequeRS`. A member of `DequeRS` is a list of ps's, implemented by the type `PST`. The suffix and prefix of a ps are implemented as lists of leaf trees, which are implemented by the type `LTree` of section 3.1. The declarations for `DequeRS` are:

```
newtype DequeRS a = D [PS a]

type PST a = ([LTree a],[LTree a])
```

The first version of the implementation of `DequeRS` can be seen in figures 7 and 10. An additional type: `Color`, is defined. Its use is described later.

Note that the last ps of the list l as described in (5.4) is always the empty pair. This could be represented by a pair of empty lists. However, we do not actually include this pair in the implementation of l. We call a list that does not end in a pair of empty lists *regular*.

Since the list is kept regular, the empty deque is represented as `[]` instead of `[([],[])]`. Then we implement the function empty by:

```
empty = D []
```

And isEmpty by:

```
isEmpty (D pss) = null pss
```

To make clear how the elements of a deque are stored in the list l, we describe here how the function toList is implemented. If we denote by list px and list sx the elements of A stored in a px and a sx, in the order in which they appear in the deque, we can informally define toList by:

$$\begin{aligned}&\mathsf{toList}\ [(px_0, sx_0), \ldots, (px_m, sx_m)] = \\ &\qquad \mathsf{list}\ px_0 +\!\!+ \mathsf{list}\ px_1 +\!\!+ \ldots +\!\!+ \mathsf{list}\ px_m +\!\!+ \mathsf{list}\ sx_m +\!\!+ \ldots +\!\!+ \mathsf{list}\ sx_1 +\!\!+ \mathsf{list}\ sx_0\end{aligned} \tag{5.5}$$

A possible implementation of toList is to traverse-down the list adding the elements in each px and then add the elements of the sx's while traversing up. However, we prefer to implement it as a fold over l. This implementation follows the structure of the following alternative definition for toList:

$$\begin{aligned}\mathsf{toList}\ [\,] &= [\,]\\ \mathsf{toList}\ ([(px_0, sx_0)] +\!\!+ [ps_1, \ldots]) &= \mathsf{list}\ px_0 +\!\!+ \mathsf{toList}\ [ps_1, \ldots] +\!\!+ \mathsf{list}\ sx_0\end{aligned} \tag{5.6}$$

What is needed then, is a function that takes a ps_i and toList $([ps_{i+1}, \ldots])$, and gives the equivalent to the right-hand side of the second clause of (5.6). We name this function toListPS.

We also need to implement the function list to obtain the elements of A stored in a prefix or a suffix in the correct order. As mentioned before, the elements of A are stored in left-to-right order in the leaves of the trees. Therefore the list of these elements in each tree is obtained collecting the leaves of the tree in left-to-right order. But this is precisely what the function frontier of section 3.1 does. The lists of elements in each tree must then be catenated in the order in which they appear in the list. Then we have: list = concatMap . frontier, where concatMap is the Haskell function defined in the standard prelude [PHe96]. We do not explicitly implement list, but inline its definition, obtaining the following implementation:

```
toListPS (px,sx) rest = concatMap frontier px ++ rest
                        ++ concatMap frontier sx
```

The function toList on deques can then be implemented by:

```
toList (D pss) = foldr toListPS [] pss
```

Invariant of the data structure

Consider a deque d with only two levels, that is, $d = (px, empty, sx)$. This is very similar to the previous implementation of a deque as a pair of lists. The main difference here is that both px and sx may contain at most five elements, to bound the cost of the operations on them. Therefore when an element has to be inserted in d, px and sx may be at full capacity. This gives rise to another level of d, as some elements of the first level are pushed into the second one. This in turn gives rise to the second major difference respect to the previous implementation: in a $ps_i = (px_i, sx_i)$, the elements in px_i are not, in general, contiguous to the elements in sx_i, as is evident in (5.5).

Similarly, when an element has to be extracted from either px or sx in one level, the required list may be empty. An attempt to extract an element from an empty px will be called an *underflow*. An attempt to insert an element in a px of length five will be called an *overflow*. The implementation must ensure that overflows and underflows never occur. To this end, an invariant property of the data structure is maintained using a color assignment to the ps's, defined in terms of a color assignment to the px's and sx's.

A px or a sx is said to be red if its length is 0 or 5, thus signalling that an operation on it could cause an underflow or an overflow. A px or a sx is said to be yellow if an operation on it may turn it red, i.e., if its length is 1 or 4. Otherwise it is said to be green. This is the case if its length is 2 or 3. We define the function colorX that gives the color of a px or a sx by:

If a list psx is a px or a sx:

$$\mathsf{colorX}\ psx = \begin{cases} \text{red} & \text{if } \mathsf{length}\ psx = 0 \lor \mathsf{length}\ psx = 5 \\ \text{yellow} & \text{if } \mathsf{length}\ psx = 1 \lor \mathsf{length}\ psx = 4 \\ \text{green} & \text{if } \mathsf{length}\ psx = 2 \lor \mathsf{length}\ psx = 3 \end{cases} \tag{5.7}$$

The colors assigned to a prefix or suffix indicate how close they are of being in a state where an underflow or an overflow may occur. A red color signals that the next operation could cause one of these conditions. A yellow color signals that two operations could cause these conditions, while at least three operations are necessary to cause an underflow or an underflow on a green prefix/suffix. The colors are then ordered with red as the minimum and green as the maximum. They are defined in the implementation as the members of the type `Color`, which is ordered as explained by making it an instance of the class `Ord` (see figure 7).

The color assignment is extended to the prefix-suffix pairs. A ps is as close to an error condition as the closer of its components is. Therefore a $ps = (px, sx)$ is assigned the minimum of the colors of its components.

However, the last $ps = (px, sx)$ in a list l is a special case. As can be seen in (5.5), the elements in px and sx are contiguous in the deque. Therefore they can be interpreted as a single list. If px and sx are both not empty, the color of ps is assigned as in the general case. If one of px and sx is not empty, ps has the color of the one which is not empty.

We define the function colorPS that gives the color of a ps by:

In an $l = [ps_0, \ldots, ps_m]$ representing a deque:

(i) For any $i \in \{0, \ldots, m-1\}$: $\mathsf{colorPS}\ ps_i = \mathsf{min}\ (\mathsf{colorX}\ px_i)\ (\mathsf{colorX}\ sx_i)$

$$\text{(ii)}\quad \mathsf{colorPS}\ ps_m = \begin{cases} \mathsf{colorX}\ px_m & \text{if } \mathsf{isEmpty}\ sx_m \land \mathsf{not}\ (\mathsf{isEmpty}\ px_m) \\ \mathsf{colorX}\ sx_m & \text{if } \mathsf{isEmpty}\ px_m \land \mathsf{not}\ (\mathsf{isEmpty}\ sx_m) \\ \mathsf{min}\ (\mathsf{colorX}\ px_m)\ (\mathsf{colorX}\ sx_m) & \text{otherwise} \end{cases} \tag{5.8}$$

The function colorX is implemented in figure 10, directly from definition (5.7). However, we do not implement colorPS directly. We implement instead a function colorOfFirst which, given a list of

ps's, returns the color of the first ps on the list. In this way colorOfFirst can easily check whether the first ps is also the last one. The following implementation, which also appears in figure 7, can be seen to conform to (5.8):

```
colorOfFirst [([],sx)]      = colorX sx
colorOfFirst [(px,[])]      = colorX px
colorOfFirst ((px,sx):pss) = min (colorX px) (colorX sx)
```

The colors of the prefix-suffix pairs are used to impose an invariant property on l. Each red ps_i must be preceded in l by an unique green ps_j. We call this green ps_j the *safety* pair of ps_i, as defined by:

$$\mathsf{safety}\ ps_i = ps_j \quad \equiv \quad j < i \wedge \mathsf{isGreen}\ ps_j \wedge \forall k(j < k < i \quad \Longrightarrow \quad \mathsf{isYellow}\ ps_k)$$

Where isYellow and isGreen hold when its arguments are respectively yellow or green. The invariant can then be stated as:

$$\begin{array}{c}\text{For any red } ps_i \text{ in an } l = [ps_0, \ldots, ps_m] \text{ representing a deque:} \\ \exists ps_j(\mathsf{safety}\ ps_i = ps_j)\end{array} \tag{5.9}$$

In particular, this invariant implies that ps_0 is always non-red in l, or otherwise its safety pair would not exist. It also implies that no two consecutive pairs in l are both red, or otherwise the second red pair would not have a safety pair.

It is clear now that we can always insert/extract an element from ps_0 without causing any overflow/underflow. After the insertion or deletion the invariant can only be violated in two cases:

1. When ps_0 becomes red, since safety ps_0 does not exist. In this case ps_0 was yellow before the current operation, therefore ps_1 is not red since it did not have a safety pair.

2. When ps_0 becomes yellow and it was the safety pair of the first non-yellow pair ps_k in l, since now ps_k does not have a safety pair. Since ps_k has not changed and is red, ps_{k+1} is not red, or there would be two consecutive red pairs.

In both cases the invariant is violated if the first non-yellow pair in l is red — this is ps_0 and ps_k in the first and second cases above. This first non-yellow pair must be *fixed* to restore the invariant. The next pair, if it exists, is guaranteed to be non-red.

Whenever an element is inserted/extracted from the deque possibly violating the invariant, the list l is *normalized*, that is, the invariant is restored if it was indeed violated. We define the function normD to perform this normalization. To determine if a violation of the invariant has effectively occurred, the first non-yellow pair is examined and it is fixed in case it is red. A red pair is fixed by making it green, for what the next pair is used. The function normD is implemented by:

```
normD l@((px,sx):pss) = case colorOfFirst l of
                             Red    -> fix (px,sx) pss
                             Yellow -> (px,sx) : normD pss
                             Green  -> l
normD [] = []
```

In the first clause, initial yellow pairs are by-passed until the first non-yellow pair is found. If it is green, the list conforms to the invariant. If it is red, the function fix is invoked with the pair to be fixed and the rest of the list. The second clause handles the case of a list that becomes empty while looking for the first non-yellow pair. In this case there are no red pairs in l thus the invariant is not violated.

From the color assignments (5.7) and (5.8), we can derive the following property:

$$\mathsf{isRed}\ (px, sx) \quad \Longrightarrow \quad \mathsf{length}\ (px +\!\!+\, sx) = 0 \vee 5 \leq \mathsf{length}\ (px +\!\!+\, sx) \leq 10 \tag{5.10}$$

Where isRed holds if its argument is red. The combined length of px and sx is zero if both are empty. Otherwise, at least one of them is not empty and red, therefore its length is 5, while the length of the other is at most 5.

To fix a red pair (px_i, sx_i) when it is the last pair of l, we calculate the number of elements in $px_i +\!\!+\, sx_i$. If it is zero, the pair is empty and is removed from the list, so that it does not need a safety pair. If there are five or six elements in $px_i +\!\!+\, sx_i$, we can store the first 3 in px_i and the remaining 2 or 3 in sx_i, thereby making the pair green. If $px_i +\!\!+\, sx_i$ contains more than six elements we create a new empty pair and use the function makeGrPS to distribute the elements between both pairs making (px_i, sx_i) green. This case is handled by:

```
fix (px,sx) []
    | n == 0    = []
    | n <= 6    = [splitAt 3 (px ++ sx)]
    | otherwise = let (ps1,ps2) = makeGrPS (px,sx) ([],[])
                  in [ps1,ps2]
    where n = length px + length sx
```

Note that by (5.10), the value of `n` above is either zero or greater than five, but no more than ten. Since the new pair is empty, there is enough room for all the elements.

To fix a red (px_i, sx_i) that is not the last pair of l, we use the function makeGrPS as in the previous case to distribute the elements on (px_i, sx_i) and the next pair ps_{i+1}:

```
fix  ps (ps1:pss) = let (ps',ps1') = makeGrPS ps ps1
                    in ps' : regular (ps1':pss)
```

Since makeGrPS may need to extract some elements from ps_{i+1}, this could become empty and, if it is the last pair in l, l is now non-regular. The function regular, in figure 7, deletes ps_{i+1} in this case.

To make green a pair (px_i, sx_i), both its prefix and its suffix must be made green. This is done pushing or extracting some elements to or from the next pair ps_{i+1}. The prefix is made green by the function makeGrPx and the suffix by makeGrSx. The function makeGrPS is thus implemented by:

```
makeGrPS (px,sx) ps1 = let (px',ps1' ) = makeGrPx px ps1
                           (sx',ps1'') = makeGrSx sx ps1'
                       in ((px',sx') , ps1'')
```

Note that since ps_{i+1} is non-red, neither its prefix nor its suffix can be red. Therefore we can always insert an element in its prefix and in its suffix. However, ps_{i+1} may contain only one element in total. This is the case if ps_{i+1} is the last pair in l and one of its components is empty while the other has one element. In this case, by (5.8), ps_{i+1} is yellow. Since it is the last pair, it can be interpreted as a single list. Therefore makeGrPx can always extract one element from it, if necessary. But makeGrSx must consider that makeGrPx might have left ps_{i+1} empty.

To make a prefix px_i green, it must be left with 2 or 3 elements. If it contains 4 or 5 elements, the last two of them are pushed to the next pair ps_{i+1}. By (5.4) the elements of ps_{i+1} are pairs of elements of px_i. In our implementation, the elements of ps_{i+1} are trees of height $i+1$ while the elements of px_i are trees of height i. Therefore the two elements of px_i that are pushed into ps_{i+1} are made a single tree of height $i+1$ with the function join of section 3.1. Similarly, an element of ps_{i+1} is extracted if px contains 0 or 1 element. The element extracted from ps_{i+1} is a tree of the form `Bin n a1 a2` where `a1` and `a2` are trees of height i —see section 3.1— and can be made elements of px. Finally, if px contains 2 or 3 elements, it is already green. It may occur if sx_i is red. The function makeGrPx is then implemented by:

```
makeGrPx px ps@(px1,sx1)
    | n >= 4 = let (px',[a1,a2]) = splitAt (n-2) px
               in  (px' , (join a1 a2:px1,sx1))
    | n <= 1 = let Bin _ a1 a2 = headPS ps
               in (px ++ [a1,a2] , tailPS ps)
    | otherwise = (px,ps)
    where n = length px
```

An element is extracted from ps_{i+1} with the function headPS, the function tailPS is used to extract an element from it. Both functions, described later, are shown in figure 10.

A suffix sx_i is made green by makeGrSx by a procedure similar to that of makeGrPx. The only difference is that in this case ps_{i+1} might be empty. This is indicated by the function isEmptyPS. Since in this case ps_{i+1} is the last pair in l, it is considered empty only if both of its components are empty. This gives the following implementation:

```
makeGrSx sx ps@(px1,sx1)
    | n >= 4                        = let a1:a2:sx' = sx
                                      in  (sx' , (px1,sx1++[join a1 a2]))
    | n <= 1 && not (isEmptyPS ps) = let Bin _ a1 a2 = lastPS ps
                                      in (a1:a2:sx , initPS ps)
    | otherwise = (sx,ps)
    where n = length sx
          isEmptyPS ([],[]) = True
          isEmptyPS _       = False
```

Note that elements in ps_{i+1} precede the elements of sx_i in the deque. Thus elements at the beginning of sx_i are inserted at the end of ps_{i+1}. Analogously, elements at the end of ps_{i+1} are inserted at the beginning of sx_i. The functions makeGrPS, makeGrPx, makeGrSx, headPS, and lastPS are shown in figure 10, and they will remain unchanged for the final version of the implementation.

It is clear that after $ps_i = (px_i, sx_i)$ is fixed, px_i is made green by makeGrPx. However, sx_i might be left yellow by makeGrSx. This is the case when ps_{i+1} is the last pair of l and becomes empty. In this case ps_i is not made green but yellow, and ps_{i+1} will be deleted from l making ps_i the last pair of l. Since the invariant had been violated because ps_i was red, this is no longer the case. Since ps_{i+1} was the last pair of l, and ps_i was the first non-yellow pair of l, the new l contains only yellow pairs, thus the invariant is restored.

If ps_i is made green it does not need now a safety pair. If ps_{i+1} was green, it may now be yellow. This could be a problem if ps_{i+1} was the safety pair of a red pair ps_k with $k > i+1$ in l. But in this case the safety pair of ps_k is now ps_i. If ps_{i+1} was yellow, it may be red now, but ps_i, being now green, is its safety pair. Therefore if ps_i is made green the invariant is restored.

The operations described so far which are related to the color assignment and the normalization have constant time complexity except for the function normD. Since this function traverse the list l to find the first non-yellow pair, it may need to traverse the whole list. To overcome this problem, the data structure is modified for the second version of the implementation. We now describe the remaining operations for the first version.

Deque operations, first version

Some operations on the list l need to extract an element from a pair $ps_i = (px_i, sx_i)$. In general, operations related to the first element in ps_i must manipulate px_i, while those related to the last element in ps_i must manipulate sx_i. But, as mentioned before, if ps_i is the last pair in l, elements in px_i and sx_i are contiguous in the deque. Therefore ps_i may be considered a single list $px_i \mathbin{+\!\!+} sx_i$.

We could, for each extraction operation on a ps, differentiate the cases when ps is the last pair and when it is not. However, the invariant property ensures that whenever these operations are used, they may only need to interpret ps as a single list if ps is indeed the last pair. We can then implement them as if px_i and sx_i were always contiguous. If the first element of ps_i is needed it is looked up in px_i if it is not empty. If px_i is empty the element is looked up in sx_i. For instance, to return the first element in a ps, we implement the function headPS by:

```
headPS ([]   ,x:xs) = x
headPS (x:xs,_   )  = x
```

And a similar implementation is given for tailPS, that extracts the first element in ps:

```
tailPS ([]   ,y:ys) = ([],ys)
tailPS (x:xs,ys  ) = (xs,ys)
```

```
newtype DequeRS a = D [PST a]
type PST a = ([LTree a],[LTree a])
data Color = Red | Yellow | Green deriving (Eq,Ord)

instance Seq DequeRS where
    empty = D []

    isEmpty (D pss) = null pss

    front (D [])       = error "front: deque is empty"
    front (D (ps:pss)) = let Lf a = headPS ps
                         in a

    rear (D [])       = error "rear: deque is empty"
    rear (D (ps:pss)) = let Lf a = lastPS ps
                        in a

    cons a (D [])            = D ([ ([Lf a],[]) ])
    cons a (D ((px,sx):pss)) = D (normD ((Lf a:px,sx) : pss))

    snoc (D []) a            = D ([ ([],[Lf a]) ])
    snoc (D ((px,sx):pss)) a = D (normD ((px,sx++[Lf a]) : pss))

    popFront (D [])       = error "popFront: deque is empty"
    popFront (D (ps:pss)) = D (normD (tailPS ps : pss))

    popRear (D [])       = error "popRear: deque is empty"
    popRear (D (ps:pss)) = D (normD (initPS ps : pss))

    lengthSeq (D pss) = foldr accSum 0 pss
                        where accSum (px,sx) n = length px + length sx + 2*n

    toList (D pss) = foldr toListPS [] pss

normD :: [PST a] -> [PST a]
normD l@((px,sx):pss) = case colorOfFirst l of
                           Red    -> fix (px,sx) pss
                           Yellow -> (px,sx) : normD pss
                           Green  -> l
normD [] = []

fix :: PST a -> [PST a] -> [PST a]
fix (px,sx) []
    | n == 0    = []
    | n <= 6    = [splitAt 3 (px ++ sx)]
    | otherwise = let (ps1,ps2) = makeGrPS (px,sx) ([],[])
                  in [ps1,ps2]
    where n = length px + length sx
fix  ps (ps1:pss) = let (ps',ps1') = makeGrPS ps ps1
                    in ps' : regular (ps1':pss)

regular :: [PST a] -> [PST a]
regular [ ([],[]) ] = []
regular l           = l

colorOfFirst :: [PST a] -> Color
colorOfFirst [([],sx)]     = colorX sx
colorOfFirst [(px,[])]     = colorX px
colorOfFirst ((px,sx):pss) = min (colorX px) (colorX sx)
```

Figure 7: Implementation of DequeRS, first version

The functions lastPS that returns the first element in a $ps = (px, sx)$, and initPS that extracts it are implemented analogously. That is, they manipulate sx if it not empty, or px otherwise. These operations are shown in figure 10.

We now describe the top-level functions on a deque, which are shown in figure 7. The functions empty and isEmpty has already been described, as well as toList.

The function front must return the first element in the deque. This is the first element of ps_0 in the list l representing the deque, but is stored in ps_0 as a tree of height 0, i.e. as a leaf. A leaf with an element a in A is of the form `Lf a` —see section 3.1. This tree is obtained by headPS. The implementation of front is:

```
front (D (ps:pss)) = let Lf a = headPS ps
                     in a
```

The function rear is very similar, but requires the last element of ps_0, which is obtained with lastPS. The implementation is:

```
rear (D (ps:pss)) = let Lf a = lastPS ps
                    in a
```

To insert an element a of A in a deque represented by l, cons insert a tree of the form `Lf a` in px_0 if l is not empty. The invariant guarantees that ps_0 is non-red and therefore the insertion is always possible. Since the length of px_0 is changed, its color might be changed and cause a violation of the invariant. The function normD is then invoked to restore it if it is needed. If l is empty, a new empty pair is created and the tree is inserted in its prefix. Since the pair is now yellow, the invariant is maintained. This gives the following implementation:

```
cons x (D [])            = D ([ ([Lf a],[]) ])
cons x (D ((px,sx):pss)) = D (normD ((Lf a:px,sx) : pss))
```

The function snoc is implemented analogously, with the new element inserted in the suffix of px_0:

```
snoc (D []) x            = D ([ ([],[Lf a]) ])
snoc (D ((px,sx):pss)) x = D (normD ((px,sx++[Lf a]) : pss))
```

To extract the first or last element of a non-empty deque, it is extracted from ps_0 by the function headPS or by the function tailPS as required. The color of ps_0 is not red by the invariant, therefore the extraction does not cause any underflow/underflow. Since the color of px_0 might be changed, the normalization function normD is invoked. This gives the following implementations for popFront and popRear:

```
popFront (D (ps:pss)) = D (normD (tailPS ps : pss))

popRear (D (ps:pss)) = D (normD (initPS ps : pss))
```

The functions front, rear, cons, snoc, popFront, and popRear have constant time complexity except when a normalization is required.

To calculate the length of a deque represented by l, we must count the total number of leaves in all the trees stored in l. This is the number of trees of height 0 stored in l. According to (5.4), the trees stored in a ps_i are trees of height A^i. Consider a list $l = [ps_1, \dots, ps_{m-1}, ps_m]$. If ps_m contains n_m trees of height m, each of these trees contains 2 trees of height $m-1$. If there are n_{m-1} trees of height $m-1$ in ps_{m-1}, the total number of trees of height $m-1$ in both ps_{m-1} and ps_m is $n_{m-1} + 2n_m$.

In general, if a ps_i contains n_i trees of height i, and there are $n_{i+1..m}$ trees of height $i+1$ in $[ps_{i+1}, \dots, ps_m]$, the number of trees of height i in $[ps_i, \dots, ps_m]$ is $n_i + 2n_{i+1..m}$. Then, if we fold this calculation from right to left in l, we get the number of leaves in l. We implement lengthSeq by:

```
lengthSeq (D pss) = foldr accSum 0 pss
                    where accSum (px,sx) n = length px + length sx + 2*n
```

The cost of lengthSeq is proportional to the length of all the px's and sx's in l. Since the size of the trees increases exponentially from one ps to the next one, and the length of each px and sx is at most 5, the total length of the px's and sx's is logarithmic on the length of the deque. Therefore the cost of lengthSeq for a deque of length n is $O(\log n)$.

For the function toSeq, we resort to the default implementation given in the definition of the class `Sequence` (in figure 1).

Final version of the implementation

The first version of `DequeRS`, as described, does not have constant worst-case time because the function normD may need to traverse the list l. This is needed to find the first non-yellow pair in l. This problem is solved changing the structure of l. The change consists of storing contiguous yellow pairs as a single internal list in l. Then, consecutive yellow pairs can be by-passed in a single step in the traversal of l.

The elements of an l representing a deque d are now denoted as $c_{i..j}$. Each $c_{i..j}$ may be a single ps_i if it is non-yellow, or a list $[ps_i, \ldots, ps_j]$ of one or more yellow pairs. In a $c_{i..j} = ps_i$ or a $c_{i..j} = [ps_i]$, we have $i = j$.

We change the definition (5.4) of l to reflect its new structure:

$$
\begin{array}{l}
\text{A deque } d \text{ with } m+1 \text{ levels is represented by } l = [c_{0..k}, \ldots, c_{l..m}] \text{ where:} \\
\begin{array}{ll}
\text{(i)} & c_{i..j} = \begin{cases} ps_i & \text{if } \mathsf{not}\ (\mathsf{isYellow}\ ps_i) \wedge i = j \\ [ps_i, \ldots, ps_j] & \text{if } \forall k \in \{i, \ldots, j\} : \mathsf{isYellow}\ ps_k \end{cases} \\
\text{(ii)} & ps_i = \begin{cases} (px_i, sx_i) & \text{if } d_i = (px_i, d_{i+1}, sx_i) \\ ps_m = () & \text{if } d_m \text{ is empty} \end{cases} \\
\text{(iii)} & px_i, sx_i \text{ are lists over } A^i
\end{array}
\end{array}
\tag{5.11}
$$

Where the enumeration of the internal deques and the representation of the ps's is retained. The implementation of l is also kept regular, that is, $ps_m = ()$ does not actually appear in the list.

The internal lists of yellow pairs, or yellow lists for short, are always of maximal length and not empty, that is:

$$
\begin{array}{l}
\text{If } c_{i..j} = [(px_i, sx_i), \ldots, (px_j, sx_j)] \text{ in } l : \\
\quad \begin{array}{rl}
\text{(i)} & \text{if } i > 0 : \quad \mathsf{not}\ (\mathsf{isYellow}\ c_{i-1}) \\
\text{(ii)} & \text{if } c_{j+1} \text{ exists in } l : \quad \mathsf{not}\ (\mathsf{isYellow}\ c_{j+1}) \\
\text{(iii)} & j \geq i
\end{array}
\end{array}
$$

With this structure, the first non-yellow pair in l is either $c_{1..1}$, or $c_{i..i}$ with $l = [c_{0..i-1}, c_{i..i}, \ldots]$. Therefore the normalization will find the required pair in at most two steps.

The type declarations for this version have an additional type to represent the elements $c_{i..j}$ of l. If a $c_{i..i}$ is a single pair, it has the constructor `PS`. If a $c_{i..j}$ is a yellow list. it has the constructor `Y`. The declarations are:

```
newtype DequeRS a = D [C a]
data C a = PS (PST a) | Y [PST a]
type PST a = ([LTree a],[LTree a])
data Color = Red | Yellow | Green deriving (Eq,Ord)
```

We now describe how the implementation of some operations are modified to manipulate the new representation. The functions on the type `PST` do not change, and are shown in figure 10. The top-level operations are shown in figure 8 and the remaining auxiliary functions are shown in figure 9.

Two additional operations are implemented that deal directly with the modification of l. The first one, popPS, is used to extract the first prefix-suffix pair in l, whether it is yellow or not. If it is non-yellow, it is stored in l as a single $c_{i..i}$ with the constructor `PS`. If it is yellow, it is extracted from the corresponding internal list. If this internal list contains a single yellow pair, the yellow list is deleted from l. The value returned is a pair with the first ps in l and l without it. The implementation is:

```
newtype DequeRS a = D [C a]
data C a = PS (PST a) | Y [PST a]
type PST a = ([LTree a],[LTree a])
data Color = Red | Yellow | Green deriving (Eq,Ord)

instance Seq DequeRS where
    empty = D []

    isEmpty (D pss) = null pss

    front (D []) = error "front: deque is empty"
    front (D cs) = let (ps,_) = popPS cs
                       Lf a   = headPS ps
                   in a

    rear (D []) = error "rear: deque is empty"
    rear (D cs) = let (ps,_) = popPS cs
                      Lf a   = lastPS ps
                  in a

    cons a (D []) = D ( [PS ([Lf a],[])] )
    cons a (D cs) = let ((px,sx),cs1) = popPS cs
                    in D (normD (fixY (PS (Lf a:px,sx) : cs1)))

    snoc (D []) a = D ( [PS ([],[Lf a])] )
    snoc (D cs) a = let ((px,sx),cs1) = popPS cs
                    in D (normD (fixY (PS (px,sx++[Lf a]) : cs1)))

    popFront (D []) = error "popFront: deque is empty"
    popFront (D cs) = let (ps,cs1) = popPS cs
                      in D (normD (fixY (PS (tailPS ps) : cs1)))

    popRear (D []) = error "popRear: deque is empty"
    popRear (D cs) = let (ps,cs1) = popPS cs
                     in D (normD (fixY (PS (initPS ps) : cs1)))

    lengthSeq (D cs) = foldr accSum 0 cs
                       where accSum (PS (px,sx)) n = length px + length sx +
                                                      2*n
                             accSum (Y pss) n = foldr accSum n (map PS pss)

    toList (D cs) = foldr toListC [] cs
                    where toListC (PS ps) rest = toListPS ps rest
                          toListC (Y pss) rest = foldr toListPS rest pss
```

Figure 8: Implementation of DequeRS, final version

```
popPS (PS ps     :cs) = (ps,cs)
popPS (Y [ps]    :cs) = (ps,cs)
popPS (Y (ps:pss):cs) = (ps,Y pss:cs)
```

Note that it is defined only if its argument is not empty.

The second additional operation, fixY, is used each time a *ps* is inserted in l and it might be yellow. It is always inserted as a single *ps* (with the constructor PS), and fixY stores it in a yellow list if it is yellow. Since the yellow lists must be of maximal length, if the next member of l is a yellow list, the new yellow pair is inserted on it. The implementation is:

```
fixY l@[PS ps]
    | isYellow ps = [Y [ps]]
    | otherwise   = l
    where isYellow ([],sx) = colorX sx == Yellow
          isYellow (px,[]) = colorX px == Yellow
          isYellow (px,sx) = min (colorX px) (colorX sx) == Yellow
fixY l@(PS ps:cs)
    | isYellow ps = case cs of
                      (Y pss:cs1) -> (Y (ps:pss):cs1)
                      _           -> (Y [ps] :cs)
```

```
        | otherwise   = l
        where isYellow (px,sx) = min (colorX px) (colorX sx) == Yellow
```

Note that since the color assignment for a pair is different if it is the last pair or not, in each clause a local function isYellow is used, according to (5.7).

The functions empty and isEmpty have the same implementation. The functions cons, snoc, popFront, and popRear are modified in a similar way. The major change involves the extraction of the first *ps* in l. This was done before by pattern-matching in the list l. It is done now with the function popPS. The other modification involves the functions that insert a *ps* in l. It is inserted as a $c_{i..i}$ with the constructor `PS` regardless of its color, but the function fixY is invoked to change this if necessary. For instance, the function cons is now implemented as:

```
cons a (D []) = D ( [PS ([Lf a],[])] )
cons a (D cs) = let ((px,sx),cs1) = popPS cs
                in D (normD (fixY (PS (Lf a:px,sx) : cs1)))
```

The other functions are shown in figure 8.

The function lengthSeq is slightly changed. It used a local function accSum (see figure 7) to calculate the length of a single *ps*. The first argument of this function is now a $c_{i..j}$. If it is a single *ps*, it does the same as in the previous version. If its argument is a yellow list, it is recursively processed as if it were a list of single *ps*'s. The implementation is:

```
lengthSeq (D cs) = foldr accSum 0 cs
                   where accSum (PS (px,sx)) n = length px + length sx +
                                                 2*n
                         accSum (Y pss) n = foldr accSum n (map PS pss)
```

The function toList (in figure 7) was implemented by folding the function toListPS (in figure 10) over l. Since now the elements of l are of type `C`, an intermediate function toListC is defined. It invokes toListPS if its first argument is a single *ps*. If its argument is a yellow list, it invokes an internal fold over the *ps*'s in the yellow list. The implementation is:

```
toList (D cs) = foldr toListC [] cs
                where toListC (PS ps) rest = toListPS ps rest
                      toListC (Y pss) rest = foldr toListPS rest pss
```

The functions normD, regular, and fixY, have essentially the same implementation, but its type is changed to reflect the new type of l.

Finally, the function colorOfFirst now has an extra clause for when the first element of l is a yellow list. In this case, the color of the first *ps* is obviously yellow. The new implementation is:

```
colorOfFirst (Y ps:cs)        = Yellow
colorOfFirst [PS ([],sx)]     = colorX sx
colorOfFirst [PS (px,[])]     = colorX px
colorOfFirst (PS (px,sx):pss) = min (colorX px) (colorX sx)
```

```
popPS :: [C a] -> (PST a,[C a])
popPS (PS ps     :cs) = (ps,cs)
popPS (Y [ps]    :cs) = (ps,cs)
popPS (Y (ps:pss):cs) = (ps,Y pss:cs)

normD :: [C a] -> [C a]
normD l@(c:cs) = case colorOfFirst l of
                     Red    -> fix c cs
                     Yellow -> c : normD cs
                     Green  -> l
normD [] = []

fix :: C a -> [C a] -> [C a]
fix (PS (px,sx)) []
    | n == 0    = []
    | n <= 6    = [PS (splitAt 3 (px ++ sx))]
    | otherwise = let (ps',ps1) = makeGrPS (px,sx) ([],[])
                  in (PS ps' : fixY [PS ps1])
    where n = length px + length sx
fix (PS ps) cs = case makeGrPS ps ps1 of
                     (ps',([],[])) -> fixY (PS ps': regular (PS ([],[]):cs1))
                     (ps',ps1'   ) -> PS ps': fixY (PS ps1':cs1)
                 where (ps1,cs1) = popPS cs

fixY :: [C a] -> [C a]
fixY l@[PS ps]
    | isYellow ps = [Y [ps]]
    | otherwise   = l
    where isYellow ([],sx) = colorX sx == Yellow
          isYellow (px,[]) = colorX px == Yellow
          isYellow (px,sx) = min (colorX px) (colorX sx) == Yellow
fixY l@(PS ps:cs)
    | isYellow ps = case cs of
                        (Y pss:cs1) -> (Y (ps:pss):cs1)
                        _           -> (Y [ps] :cs)
    | otherwise   = l
    where isYellow (px,sx) = min (colorX px) (colorX sx) == Yellow

regular :: [C a] -> [C a]
regular [PS ([],[])] = []
regular l            = l

colorOfFirst :: [C a] -> Color
colorOfFirst (Y ps:cs)        = Yellow
colorOfFirst [PS ([],sx)]     = colorX sx
colorOfFirst [PS (px,[])]     = colorX px
colorOfFirst (PS (px,sx):pss) = min (colorX px) (colorX sx)
```

Figure 9: Auxiliary functions for DequeRS, final version

```
makeGrPS :: PST a -> PST a -> (PST a , PST a)
makeGrPS (px,sx) ps1 = let (px',ps1' ) = makeGrPx px ps1
                           (sx',ps1'') = makeGrSx sx ps1'
                       in ((px',sx') , ps1'')

makeGrPx :: [LTree a] -> PST a -> ([LTree a], PST a)
makeGrPx px ps@(px1,sx1)
    | n >= 4 = let (px',[a1,a2]) = splitAt (n-2) px
               in  (px' , (join a1 a2:px1,sx1))
    | n <= 1 = let Bin _ a1 a2 = headPS ps
               in (px ++ [a1,a2] , tailPS ps)
    | otherwise = (px,ps)
    where n = length px

makeGrSx :: [LTree a] -> PST a -> ([LTree a], PST a)
makeGrSx sx ps@(px1,sx1)
    | n >= 4                      = let a1:a2:sx' = sx
                                      in  (sx' , (px1,sx1++[join a1 a2]))
    | n <= 1 && not (isEmptyPS ps) = let Bin _ a1 a2 = lastPS ps
                                      in (a1:a2:sx , initPS ps)
    | otherwise = (sx,ps)
    where n = length sx
          isEmptyPS ([],[]) = True
          isEmptyPS _       = False

colorX :: [LTree a] -> Color
colorX px
    | n == 2 || n == 3 = Green
    | n == 1 || n == 4 = Yellow
    | otherwise        = Red
    where n = length px

toListPS :: PST a -> [a] -> [a]
toListPS (px,sx) rest = concatMap frontier px ++ rest ++ concatMap frontier sx

headPS :: PST a -> LTree a
headPS ([]  ,x:xs) = x
headPS (x:xs,_   )  = x

lastPS :: PST a -> LTree a
lastPS (xs,[]) = last xs
lastPS (_ ,ys) = last ys

tailPS :: PST a -> PST a
tailPS ([]  ,y:ys) = ([],ys)
tailPS (x:xs,ys  ) = (xs,ys)

initPS :: PST a -> PST a
initPS (xs,[]) = (init xs,[])
initPS (xs,ys) = (xs,init ys)
```

Figure 10: Operations on PST for DequeRS

6 Flexible arrays

Flexible arrays or *random-access deques*, are arrays in that they have efficient indexing operations, and are flexible in the sense that they can grow and shrink at either end, as deques. Arrays that can be extended or shortened at only one end are sometimes called *one-sided flexible arrays* or *random-access lists*, as opposed to *fully flexible arrays*. The arrays considered here are all fully flexible according to that terminology.

Arrays of fixed length are predefined in all major imperative languages supporting constant time indexing. They have also been incorporated in some functional languages (e.g. `Array` library in Haskell) with predefined implementations for efficient indexing. However, when the array is required to be flexible, the predefined implementations are not well suited, hence a user-defined implementation is used.

The simplest implementation of flexible arrays is to use plain lists, which yields linear[3] complexity (on the size of the array) for the indexing operations. It has constant time complexity for cons and popFront, and linear time for snoc and popRear. To improve the cost of the indexing operations, trees of height logarithmic on their sizes are used instead of lists. This gives logarithmic complexity for all operations. Traditional implementations of balanced trees such as AVL-trees and B-trees, can be used to represent arrays [Cha91, Mye84]. However, other forms of trees have been described for that purpose which are simpler to program and avoid the cumbersome rotations of the traditional forms.

We present two implementations of flexible arrays using the kinds of trees described in section 3. One with Braun trees, based in the work presented by Hoogerwoord in [Hoo92a]. The other with leaf trees, as described by Dielissen and Kaldewaij in [DK95]. We consider the front element of an array of length n to be at index 0, and the rear one at index $n-1$.

Another interesting data structure that supports efficient indexing for sequences is described in [Oka95a], where a random-access list is implemented as a forest of complete pre-ordered node-oriented trees. We do not include this data structure here since it does not support an efficient snoc.

6.1 Arrays as Braun trees

Braun trees and its use to represent flexible arrays are presented in [Hoo92a], where Hoogerwoord describes a simple and elegant derivation of an implementation formerly presented in [BR83]. Once Braun trees are defined, most of the operations for flexible arrays are straightforward. An implementation of Braun trees —as type `BrTree`— was presented in section 3.2, following the work in [Hoo92a] and additional operations taken from [Oka97b].

We implement a flexible array by a pair (n, t), where t is a Braun tree whose positions correspond to the positions of the array and n is its size. The size of the tree is then the length of the array. The (n, t) pair is made a new type with the constructor `A`, as usual in previous sequences, which gives the type declaration:

```
newtype ArrB a = A (Int,BrTree a)
```

The empty array is then represented by $(0, \texttt{Nil})$. The sequence operations just invoke the corresponding functions on t and keep account of n.

Our implementation can be seen in figure 11. The functions front and rear first check that the array is not empty and then obtain the element at position 1 or $n-1$ with ind. The functions cons and snoc increment n and insert the new element at the first and last position. Similarly, popFront and popRear decrement n and delete the desired element accordingly. All these operations have $O(\log n)$ worst-case complexity. The function lengthSeq just gives n in constant time.

The coercion operations are also provided by the Braun trees implementation: toList obtains the list of elements of the tree ordered by its position with brOrder; toSeq creates the desired tree with mkBrTree and sets the n field with its size. Both operations take time proportional to the length of the sequence.

Finally, the indexing operations are again provided by the implementation in section 3.2, both with complexity $O(\log i)$ when accessing position i. The function lookupSeq gets the required

[3] more precisely: $O(i)$ for accessing the position i.

```
newtype ArrB a = A (Int,BrTree a)

instance Seq ArrB where
    empty = A (0,Nil)

    isEmpty (A (0,_)) = True
    isEmpty _         = False

    front (A (0,_)) = error "front: array is empty"
    front (A (_,t)) = ind 0 t

    rear (A (0,_)) = error "rear: array is empty"
    rear (A (n,t)) = ind (n-1) t

    cons x (A (n,t)) = A (n+1,insFirst x t)

    snoc x (A (n,t)) = A (n+1,insLastAt n x t)

    popFront (A (0,_)) = error "popFront: array is empty"
    popFront (A (n,t)) = A (n-1,delFirst t)

    popRear (A (0,_)) = error "popRear: array is empty"
    popRear (A (n,t)) = A (n-1,delLastAt (n-1) t)

    lengthSeq (A (n,_)) = n

    toList (A (_,t)) = brOrder t

    toSeq xs = let t = mkBrTree xs
               in A (size t,t)

instance IndSeq ArrB where
    lookupseq i (A (_,t)) = ind i t

    updateSeq i x (A (n,t))
        | i < n     = A (n,upd i x t)
        | otherwise = error "updateSeq: non-existent index"
```

Figure 11: Implementation of `ArrB`.

element with ind. The function updateSeq checks that the given position is not greater than the last position of the array and updates it with upd. Additionally, the position to access should be checked to be non-negative. However, we leave the functions over Braun trees to detect this error.

6.2 Arrays as leaf trees

We describe here an implementation that closely follows the work presented by Dielissen and Kaldewaij in [DK95] and [KD96], using leaf trees. As with any other kind of trees, the complexity of the operations is in general proportional to the height of the trees. Though accessing a single element in a leaf tree may be more expensive —but still of logarithmic complexity— than in a Braun tree, leaf trees are more suitable to other operations not used here, such as manipulations of segments of the array. They also seem to perform better when the elements are accessed in sequential order [Oka95a].

Most of the operations needed on leaf trees are described in section 3.1. However, operations to insert an element are not described there, since they determine the shape of the tree and therefore its height.

Since our implementation of leaf trees does not consider empty trees, we represent a non-empty array as a leaf tree and the empty array as a special constructor `Empty`. The type declarations is:

```
data ArrK a = Empty | ATree (LTree a)
```

The complete implementation appears in figure 12.

Then, empty simply returns `Empty`, and front and rear check that its argument is not `Empty` and use ind to obtain the first or last element of the array. The functions popFront and popRear signal an error if its argument is `Empty`, if it is a leaf they return `Empty`, and otherwise they delete the

```
data Arr a = Empty | ATree (LTree a)

instance Seq Arr where
    empty = Empty

    isEmpty Empty = True
    isEmpty _     = False

    front (ATree t) = ind 0 t
    front Empty     = error "front: array is empty"

    rear (ATree t) = ind (size t - 1) t
    rear Empty     = error "rear: array is empty"

    cons x (ATree t) = ATree (insFirst x t)
    cons x Empty     = ATree (Lf x)

    snoc (ATree t) x = ATree (insLast x t)
    snoc Empty     x = ATree (Lf x)

    popFront (ATree (Lf _)) = Empty
    popFront (ATree t)      = ATree (delFirst t)
    popFront Empty          = error "popFront: array is empty"

    popRear (ATree (Lf _)) = Empty
    popRear (ATree t)      = ATree (delLast t)
    popRear Empty          = error "popRear: array is empty"

    lengthSeq Empty     = 0
    lengthSeq (ATree t) = size t

    toList (ATree t) = frontier t
    toList Empty     = []

    toSeq []  = Empty
    toSeq xs  = ATree ((foldl1 join . forestOfComp) xs)

insFirst :: a -> LTree a -> LTree a
insFirst x (Bin n l r) = Bin (n+1) (insRR x l) r
insFirst x t           = insRR x t

insRR :: a -> LTree a -> LTree a
insRR x (Lf y) = Bin 2 (Lf x) (Lf y)
insRR x t@(Bin n l r)
    | size l == size r = Bin (n+1) (Lf x) t
    | otherwise        = Bin (n+1) (insRR x l) r

insLast :: a -> LTree a -> LTree a
insLast x (Bin s l r) = Bin (s+1) l (insLL x r)
insLast x t           = insLL x t

insLL:: a -> LTree a -> LTree a
insLL x (Lf y) = Bin 2 (Lf y) (Lf x)
insLL x t@(Bin n l r)
    | size l == size r = Bin (n+1) t (Lf x)
    | otherwise        = Bin (n+1) l (insLL x r)

instance IndSeq Arr where
    lookupSeq i (ATree t) = ind i t
    lookupSeq i Empty     = error "lookupSeq: array is empty"

    updateSeq i x (ATree t) = ATree (upd i x t)
    updateSeq i x Empty     = error "lookupSeq: array is empty"
```

Figure 12: Implementation of `ArrK`.

first or last element of the tree. The simple implementations of functions $\mathsf{delFirst}$ and $\mathsf{delLast}$ in section 3.1 do the job. The function $\mathsf{lengthSeq}$ simply gives zero for an empty array or the size of the tree otherwise.

The coercion toList gives the empty list if the array is empty and otherwise gives the frontier of the corresponding tree. The function toSeq depends in general on the desired shape of the tree, as described later.

The trees used by Dielissen and Kaldewaij have certain properties ensuring that its height is logarithmic on its size. A tree is said to be *left-perfect* if all the left subtrees are perfect (see section 3.1). The predicate $\mathsf{leftPerfect}$ is then defined over `LTree` (with the predicate $\mathsf{perfect}$ of section 3.1) by:

$$\begin{aligned}\mathsf{leftPerfect}\ (\mathtt{Lf\ x}) &= \mathsf{true}\\ \mathsf{leftPerfect}\ (\mathtt{Bin\ n\ l\ r}) &= \mathsf{perfect}\ \mathtt{l} \wedge \mathsf{leftPerfect}\ \mathtt{r}\end{aligned}$$

A tree is *leftist* if the height of all the left subtrees is at least the height of their right siblings. Formally, the predicate $\mathsf{leftist}$ can be defined over `LTree` by:

$$\begin{aligned}\mathsf{leftist}\ (\mathtt{Lf\ x}) &= \mathsf{true}\\ \mathsf{leftist}\ (\mathtt{Bin\ n\ l\ r}) &= \mathsf{height}\ \mathtt{l} \geq \mathsf{height}\ \mathtt{r} \wedge \mathsf{leftist}\ \mathtt{l} \wedge \mathsf{leftist}\ \mathtt{r}\end{aligned}$$

where $\mathsf{height}\ t$ is the height of the tree t.

Similarly, the symmetric predicates $\mathsf{rightPerfect}$ and $\mathsf{rightist}$ can be defined as:

$$\begin{aligned}\mathsf{rightPerfect}\ (\mathtt{Lf\ x}) &= \mathsf{true}\\ \mathsf{rightPerfect}\ (\mathtt{Bin\ n\ l\ r}) &= \mathsf{perfect}\ \mathtt{r} \wedge \mathsf{rightPerfect}\ \mathtt{l}\end{aligned}$$

and:

$$\begin{aligned}\mathsf{rightist}\ (\mathtt{Lf\ x}) &= \mathsf{true}\\ \mathsf{rightist}\ (\mathtt{Bin\ n\ l\ r}) &= \mathsf{height}\ \mathtt{r} \geq \mathsf{height}\ \mathtt{l} \wedge \mathsf{rightist}\ \mathtt{l} \wedge \mathsf{rightist}\ \mathtt{r}\end{aligned}$$

With this definitions, it is clearly seen that the height of leftist and rightist trees depends on the heights of its left and right subtrees, respectively. These subtrees are perfect for left-perfect and right-perfect trees, respectively. Since the height of a perfect leaf tree t is $\log(\mathsf{size}\ t)$, we have:

$$\begin{array}{c}\text{If } \mathsf{leftist}\ (\mathtt{Bin\ n\ l\ r}) \wedge \mathsf{leftPerfect}\ (\mathtt{Bin\ n\ l\ r}) \text{ then:}\\ \mathsf{height}\ (\mathtt{Bin\ n\ l\ r}) = 1 + \log(\mathsf{size}\ \mathtt{l})\end{array} \tag{6.1a}$$

$$\begin{array}{c}\text{If } \mathsf{rightist}\ (\mathtt{Bin\ n\ l\ r}) \wedge \mathsf{rightPerfect}\ (\mathtt{Bin\ n\ l\ r}) \text{ then:}\\ \mathsf{height}\ (\mathtt{Bin\ n\ l\ r}) = 1 + \log(\mathsf{size}\ \mathtt{r})\end{array} \tag{6.1b}$$

The size of its subtrees also indicates when these trees are perfect:

$$\begin{array}{c}\text{If } \mathsf{leftist}\ (\mathtt{Bin\ n\ l\ r}) \wedge \mathsf{leftPerfect}\ (\mathtt{Bin\ n\ l\ r}) \text{ then:}\\ \mathsf{perfect}\ (\mathtt{Bin\ n\ l\ r}) \quad\equiv\quad \mathsf{size}\ \mathtt{l} = \mathsf{size}\ \mathtt{r}\end{array} \tag{6.2a}$$

$$\begin{array}{c}\text{If } \mathsf{rightist}\ (\mathtt{Bin\ n\ l\ r}) \wedge \mathsf{rightPerfect}\ (\mathtt{Bin\ n\ l\ r}) \text{ then:}\\ \mathsf{perfect}\ (\mathtt{Bin\ n\ l\ r}) \quad\equiv\quad \mathsf{size}\ \mathtt{l} = \mathsf{size}\ \mathtt{r}\end{array} \tag{6.2b}$$

Note also that a perfect tree has all the properties above:

$$\mathsf{perfect}\ t \implies \mathsf{leftPerfect}\ t \wedge \mathsf{leftist}\ t \tag{6.3a}$$

$$\mathsf{perfect}\ t \implies \mathsf{rightPerfect}\ t \wedge \mathsf{rightist}\ t \tag{6.3b}$$

Now, to make use of all these properties, a non-empty array is represented either as `Empty`, or as a leaf, or as a (leaf) tree whose top-level left subtree is a rightist right-perfect tree, and whose top-level right subtree is a leftist left-perfect tree. That is, the type `ArrK` has the following invariant:

$$\begin{array}{l}\text{An array } a \text{ is represented by:}\\ \begin{cases}\texttt{Empty} & \text{if } \mathsf{size}\ a = 0\\ \texttt{ATree (Lf x)} & \text{if } \mathsf{size}\ a = 1\\ \texttt{ATree (Bin n l r)} & \text{if } \mathsf{size}\ a = n > 1\end{cases}\\ \text{such that } \ \mathsf{rightist}\ \texttt{l} \wedge \mathsf{rightPerfect}\ \texttt{l} \wedge\\ \qquad\qquad\quad \mathsf{leftist}\ \texttt{r} \wedge \mathsf{leftPerfect}\ \texttt{r}\end{array} \tag{6.4}$$

Then, by property (6.1) the height of a tree representing an array is always logarithmic on its size. Note that the functions delFirst and delLast of section 3.1 maintain the invariant. Note also that by properties (3.1) and (6.3), any perfect tree can represent a non-empty array, according to the invariant, so we have:

$$\mathsf{perfect}\ t \implies t \text{ satisfies (6.4) for any non-empty array} \tag{6.5}$$

We now implement the functions to insert and remove an element at the ends of the array. Since both ends are treated symmetrically, we describe only those working on the front of the array. All implementations can be seen in figure 12.

To insert an element at the front of the array, it is made a leaf if the array is empty, otherwise it is inserted at the first position of the tree representing the array, with the function insFirst. Then we implement cons by:

```
cons x (ATree t) = ATree (insFirst x t)
cons x Empty     = ATree (Lf x)
```

To insert an element at the first position of the tree representing the array, we first check if the tree is not a leaf. In this case, its first position must be in its left subtree which, according to (6.4), is a rightist right-perfect tree. For this insertion we define the function insRR. If the tree is a leaf, we use insRR on the leaf. This will create a perfect tree which, by (6.5), is adequate. The implementation of insFirst is very simple:

```
insFirst x (Bin n l r) = Bin (n+1) (insRR x l) r
insFirst x t           = insRR x t
```

The implementation of insRR must maintain the invariant that its argument is a rightist right-perfect tree. If the tree is perfect, a simple solution is to make it a right subtree and the new element the left subtree. If it is not the case, we can recursively insert the new element in the left subtree without violating the invariant. To determine if the tree is perfect, we make use of property (6.2b). This gives the implementation:

```
insRR x (Lf y) = Bin 2 (Lf x) (Lf y)
insRR x t@(Bin n l r)
    | size l == size r = Bin (n+1) (Lf x) t
    | otherwise        = Bin (n+1) (insRR x l) r
```

Finally, we implement toSeq. Given a list l, if l is empty the `Empty` array is created. If l is non-empty, a leaf tree satisfying the representation invariant (6.4) must be created. This can be done in different ways since the exact shape of the tree is not determined by the invariant. As in the case of Braun trees, a straightforward approach of repeatedly inserting each element of l with insFirst or insLast would yield a $\mathrm{O}(n \log n)$ algorithm with $|\, l \,| = n$. However, using the function forestOfPerf of section 3.1 —definition (3.2)— we can implement toSeq with $\mathrm{O}(n)$ complexity for the same list.

Applying forestOfPerf gives us a size-increasing list of perfect trees, which should be made a single tree. The easiest way of doing it is to join them together from right to left. This creates a single tree whose right subtree is perfect and whose left subtree is rightist right-perfect. This tree satisfies the invariant by (6.3a). In particular, if the length of the original list is a power of 2, this will give a perfect tree, which is correct by (6.5). The implementation is:

```
toSeq []   = Empty
toSeq xs   = ATree ((foldl1 join . forestOfPerf) xs)
```

Altogether, toSeq performs a join for every node in the result tree. Since there are $O(n)$ nodes in a tree with n leaves, the complexity of toSeq is $O(n)$ for a list of length n.

Most of the functions presented here are also described in [DK95, KD96]. The exceptions are front, rear, popFront and popRear since they are special cases of some of the other functions; and empty, isEmpty,toList and lengthSeq which are not considered there. Our implementation of toSeq turned out to be a special case of an algorithm described in [Bir97] to create a tree of minimum height.

7 Catenable lists

Catenable lists —c-lists, for short— are sequences with an efficient implementation for cat. In addition, constant-time complexity operations on lists should remain so for c-lists, namely cons and popFront. Additionally, if creating a one-element sequence takes constant time, snoc is made efficient as well, since it is implemented as a catenation of the original sequence and a one-element sequence (with the new rear element).

Sequences with efficient catenation can be implemented with some kinds of trees (e.g. AVL trees [Mye84]) obtaining $O(\log n)$ complexity for all operations. In [DST94], an implementation is described which achieves constant time for the desired list operations and $O(\log\log n)$ for catenation. By recording all the operations that have been performed on the data structure, catenation can be $O(\log\log k)$ where k is the number of operations. This work is described in an imperative setting making use of a lot of bookkeeping information and may be not easily adapted to a functional setting.

We describe here an implementation which closely follows one presented by Okasaki in [Oka96a] and [Oka96b] with constant time (amortized) complexity for the operations that should be efficient in a c-list. This implementation uses n-ary trees.

Next we describe an implementation presented in [KT95] with constant time complexity for the same operations. The implementation uses the recursive-slowdown technique and requires an implementation of (non-catenable) deques. Recently, Okasaki described a variation of this data structure with constant amortized time in [Oka97a].

7.1 Catenable lists as trees

The implementation of Okasaki develops from a simple idea. Since elements at the front of a c-list should be easily accessible, a c-list is implemented as an n-ary pre-ordered tree whose elements are c-lists. When c-lists l_1 and l_2 are catenated, the front element of l_2 will become the front element of the whole only after all elements of l_1 have been removed. Therefore l_2 is made the last child of the tree representing l_1. Similarly, after the root of a tree is removed, the first child contains the first elements of the sequence, hence it is made the top-level tree.

Given this behavior of the children of a node, they are collected in a queue in [Oka96a, Oka96b]. In our implementation we provide a popRear operation for c-lists which requires the manipulation of the last child, therefore we use a deque instead.

To achieve the amortized constant time bounds, operations on the deque must also be constant time, either worst-case or amortized. We use in particular the type `DequeH` of section 5.1.

A c-list is either empty or have two components: the root element —i.e. the front— and a deque of its children, which are c-lists as well. This gives the type `CList0`:

```
data CList0 a = Empty | Cons a (DequeH (CList0 a))
```

The implementation can be seen in figure 7.1. We will make references to this type as a null pair () for `Empty` and a pair $(x, [cl_0, \ldots, cl_m])$ for a `Cons` with element x and $[\ldots]$ representing the deque of children.

To keep the deques unnecessarily long and to facilitate some operations, the following invariant for `CList0` is maintained:

$$\begin{array}{c}\text{For any } (x, [cl_0, \ldots, cl_m]) \text{ occurring at any level:}\\ cl_i \neq () \;;\; 0 \leq i \leq m\end{array} \tag{7.1}$$

As said before, cons and snoc are implemented creating a new c-list with the new element and catenating it to the given c-list:

```
cons a cl = cat (Cons a empty) cl

snoc cl a = cat cl (Cons a empty)
```

The implementation of cat makes use of the function linkCL to insert the second c-list as the last child on the first one. It also ensures that (7.1) is maintained:

```
data CList0 a = Empty | Cons a (DequeH (CList0 a))

instance Seq CList0 where
    empty = Empty

    isEmpty Empty = True
    isEmpty _     = False

    front (Cons a _) = a
    front Empty      = error "front: catenable list is empty"

    rear (Cons x d)
        | isEmpty d = x
        | otherwise = rear (rear d)
    rear Empty = error "rear: catenable list is empty"

    cons a cl = cat (Cons a empty) cl

    snoc cl a = cat cl (Cons a empty)

    popFront (Cons _ d) = foldrD cat Empty d
    popFront Empty = error "popFront: catenable list is empty"

    popRear (Cons x d)
        | isEmpty d = Empty
        | otherwise = let (initD,lastCl) = (popRear d,rear d)
                          lastCl' = popRear lastCl
                      in cat (Cons x initD) lastCl'
    popRear Empty = error "popRear: catenable list is empty"

    lengthSeq (Cons _ d) = foldrD addLength 1 d
                           where addLength cl n = n + lengthSeq cl
    lengthSeq Empty      = 0

    toList (Cons x d) = x : concatMap toList (toList d)
    toList Empty      = []

instance CatSeq CList0 where
    cat Empty cl     = cl
    cat cl    Empty = cl
    cat cl1   cl2   = linkCL cl1 cl2

linkCL :: CList0 a -> CList0 a -> CList0 a
linkCL (Cons a d) cl2 = Cons a (snoc d cl2)
```

Figure 13: Implementation of CList0.

```
cat Empty cl     = cl
cat cl    Empty = cl
cat cl1   cl2   = linkCL cl1 cl2

linkCL (Cons a d) cl2 = Cons a (snoc d cl2)
```

To remove the front of a c-list $(x, [cl_0, \ldots, cl_m])$, cl_0 is made the top-level c-list and its children must be made a single deque along with $cl_1, \ldots, cl_m$. Since a future call to popFront on cl_0 will perform the same operation, only cl_1 is inserted as child of cl_0, and $cl_2, \ldots, cl_m$ are each inserted in the respective previous children. This is accomplished catenating $cl_0, \ldots, cl_m$ from right to left, which is in fact a fold over $[cl_0, \ldots, c_m]$. The function foldrD of section 5.1 is used to implement popFront as:

```
popFront (Cons _ d) = foldrD cat Empty d
```

The remaining list operation is front, whose implementation is trivial:

```
front (Cons a _) = a
```

The functions cons and snoc are implemented in terms of the cat operation. The function popFront performs as much as $O(n)$ catenations for a c-list of n elements. However, the cost of popFront can be amortized over the cat's that have been performed directly or by a cons or snoc. The analysis of this amortization is detailed in [Oka96b], and gives amortized constant-time complexity for cons, snoc, cat, and popFront with lazy evaluation.

We now implement the remaining operations on sequences, which are not provided in [Oka96a, Oka96b], since they do not have O(1) complexity.

The rear element of a c-list $(x, [cl_0, \ldots, cl_m])$ is at the rear of cl_m. The same applies recursively to cl_m until a c-list of the form $(y, [\,])$ is reached. This gives the following implementation for rear:

```
rear (Cons x d)
    | isEmpty d = x
    | otherwise = rear (rear d)
```

whose worst-case may take as much as $O(n)$ for a c-list of length n.

To remove the rear element of $(x, [cl_0, \ldots, cl_m])$, we take cl_m out of the deque and recursively remove its rear element obtaining cl'_m. Then cl'_m must be inserted back on the deque instead of cl_m, maintaining the invariant in case cl'_m is empty. This can be done with cat. The trivial case is to remove the rear element of a c-list $(x, [\,])$, that simply produce (). This algorithm is used to implement popRear:

```
popRear (Cons x d)
    | isEmpty d = Empty
    | otherwise = let (initD,lastCl) = (popRear d,rear d)
                      lastCl' = popRear lastCl
                  in cat (Cons x initD) lastCl'
```

As for rear, this implementation has $O(n)$ worst-case complexity for a c-list of length n.

The length of a c-list (x, d) is one more than the sum of the lengths of the c-lists in d. The implementation of lengthSeq just calculates the mentioned sum with a foldrD and provides 1 as an initial value:

```
lengthSeq (Cons _ d) = foldrD addLength 1 d
                       where addLength cl n = n + lengthSeq cl
lengthSeq Empty      = 0
```

Finally, to convert (x, d) into a list, the elements of d are themselves converted into lists, and this lists are catenated, with x at the front:

```
toList (Cons x d) = x : concatMap toList (toList d)
toList Empty      = []
```

For toSeq we resort to the default implementation provided by the class Seq (see figure 1), which for this implementation has $O(n)$ complexity.

7.2 Recursive-slowdown catenable lists

Kaplan and Tarjan describe in [KT95] an implementation of what they call a *catenable stack-ended queue* or *c-steque* which supports front, cons, snoc, popFront and cat in constant time complexity. A data structure with this operations is sometimes called an *output-restricted deque*. The implementation does not support a constant time popRear, but we implement it here since it is part of our operations on sequences.

We implement this data structure in two steps. First, we describe a preliminary version in which all operations have constant time complexity except when its invariant needs to be restored. Then we describe how the preliminary version is modified to achieve constant time complexity for all the operations that should be efficient in a c-list, as well as for snoc. The preliminary version can be seen in figure 14. The complete final version is found in figures 15 and 16.

General description of the data structure

The basic idea, similar to the same author's implementation of deques (see section 5.2), is that a c-list of A, or *c-list* A, is either a deque over A, or is decomposed in a prefix over A, a suffix over A and an inner (medium) part. The prefix and suffix are deques over A. The inner part is a c-list of *c-el* A, or *c-list* (*c-el* A). A *c-el* A, or c-element of A is either a prefix over A or a pair (prefix over A, *c-list* (*c-el* A)). That is:

$$\text{A } \textit{c-list}\, A \text{ is either:} \\ \begin{cases} d\,; & d \text{ a deque over } A \\ (px, cl, sx)\,; & px, sx \text{ deques over } A,\ cl \text{ a } \textit{c-list}\,(\textit{c-el}\, A) \end{cases} \tag{7.2}$$

and

$$\text{A } \textit{c-el}\, A \text{ is either:} \\ \begin{cases} px\,; & px \text{ a deque over } A \\ (px, cl)\,; & px \text{ a deque over } A,\ cl \text{ a } \textit{c-list}\,(\textit{c-el}\, A) \end{cases} \tag{7.3}$$

The elements of the sequence defined by a c-list are ordered as follows. If a c-list is a single deque, the elements of the sequence come in the order in which they appear in the deque. If a c-list is of the form (px, cl', sx), The elements contained in px comes first, then the elements in cl' and then the elements contained in sx. In c-elements of the form (px, cl'), the elements in px precede the elements in cl'.

A c-list with this description may contain an inner c-list, which in turn may contain another c-list and so on, until a c-list made of a single deque is found. This c-lists are enumerated from the outermost to the innermost by:

$$\text{For } cl \text{ a } \textit{c-list}\, A\text{, let:} \\ \begin{aligned} cl_0 &= cl \\ cl_{i+1} &= cl' \quad \text{if } cl_i = (px_i, cl', sx_i) \end{aligned} \tag{7.4}$$

Note that cl_0 is a c-list of A, cl_1 is a c-list of c-elements of A, cl_2 is a c-list of c-elements of c-elements of A, and so on.

Then, a c-list with $m+1$ *levels* could be represented by a list of its non-recursive components $(px_0, sx_0), \ldots, (px_{m-1}, sx_{m-1}), d_m$. This list l could be defined by:

$$\begin{aligned} & l = [ps_0, \ldots, ps_{m-1}, d_m] \\ & \text{with:} \quad ps_i = (px_i, sx_i) \text{ if } cl_i = (px_i, cl_{i+1}, sx_i) \\ & \phantom{\text{with:} \quad} d_m = cl_m \end{aligned}$$

But this l is not an homogeneous list. Indeed, d_m is a single deque while the ps_i's are pairs of deques. We will define a type `C` in Haskell that comprises both cases, thus the members of l will be all of type `C`. Then we arrive at the following description:

$$\text{If } cl \text{ is a } \textit{c-list}\, A \text{ with } m+1 \text{ levels, } cl \text{ is represented by } l = [c_0, \ldots, c_m] \text{ where:} \\ \text{(i)} \quad c_i = \begin{cases} (px_i, sx_i) & \text{if } cl_i = (px_i, cl_{i+1}, sx_i) \\ d & \text{if } cl_m = d \end{cases} \\ \text{(ii)} \quad \text{if } cl_i \text{ is a } \textit{c-list}\, B \text{ then: } px_i \text{ and } sx_i \text{ or } d = cl_i \text{ are deques of } B \tag{7.5}$$

However, the type `C` is non-homogeneous, since deques in c_0 contain members of A, while the members of the deques in c_i with $i > 0$ are of the form $\textit{c-el}^i\, A$. To define `C` in Haskell, we generalize *c-el* A to include also members of A, and define it as the type `CEl`. However, the implementation will respect description (7.5), as will be seen later.

Then, in Haskell, we define the type `C` that corresponds to the members c of l. According to (7.5)(i), each c may be a prefix-suffix pair or a single deque, all of c-elements. The type `CEl` corresponds to the c-elements, which can be now of three different types, as said above. The type of a c-list is then a list over `C`. In each case the different types are wrapped in suitable constructors. The declarations are:

```
newtype CListRS a = L [C a]
data C a = Dq (DequeH (CEl a)) | PS (DequeH (CEl a),DequeH (CEl a))
data CEl a = A a | P (DequeH  (CEl a)) | PL (DequeH  (CEl a)) [C a]
```

We use the type DequeH of section 5.1 for the deques. Any implementation for deques can be used, as long as it provides constant time front, cons, snoc, popFront and popRear. If these operations for deque have worst-case complexity, the c-list operations will also have it. If their complexity is amortized so will be the complexity of the c-list operations.

An empty c-list could be represented by an empty l or by an l with a single (empty) deque. We use an empty list and forbid the occurrence of an empty single deque in l and in any internal l'. We call a list with this property *regular*, as defined by:

$$\mathsf{regular}\,[c_0,\ldots,c_n] \quad\equiv\quad c_n \neq \mathsf{empty}$$

Keeping the list regular makes it trivial to know if a c-list is empty, since it will only be the case if l is empty. Thus we can implement isEmpty by:

```
isEmpty (L cs) = null cs
```

And there is only one possible implementation for empty:

```
empty = L []
```

To see how the elements of the sequence are obtained from its implementation, we first describe the function toList that converts a c-list into a list over A. Since the elements of l are of type C, we define a function toListC that, given a c_i and the list obtained from $c_{i+1},\ldots,c_m$, creates the list containing the members of A in $c_i,c_{i+1},\ldots,c_m$. This function makes use of toListCEl, that gives the list of members of A in a c-element. The implementation of these functions is:

```
toListC (Dq d)       lrest = concatMap toListCEl (toList d) ++ lrest
toListC (PS (px,sx)) lrest = concatMap toListCEl (toList px) ++ lrest ++
                             concatMap toListCEl (toList sx)

toListCEl (A x)       = [x]
toListCEl (P px)      = concatMap toListCEl (toList px)
toListCEl (PL px cs) = concatMap toListCEl (toList px) ++
                        foldr toListC [] cs
```

And now we can implement toList for a c-list by:

```
toList (L cs) = foldr toListC [] cs
```

These functions will be slightly modified for the final version of the implementation. Its final version appears in figure 16.

Color assignment and invariant of the structure

To achieve the constant time complexity for cons, snoc, front, popFront and cat, the elements c in l are assigned one of the colors red, yellow or green. This colors are used to impose an invariant property on l. This invariant should guarantee that these operations only need to manipulate a bounded number of c_i's, starting from c_0. In this way a complete traversal of l would never be necessary for these operations. Ideally, they should only access c_0, but we will see that restoring the invariant will need some further work. To finally achieve constant time complexity, a modification of the data structure will be done for the final version of the implementation. In the discussion that follows, a single deque d is considered a prefix as well as the suffix of a $c = d$.

The colors assigned to the c's in l can be interpreted as follows. A red c_i is one that does not allow an operation to manipulate only c_i or at most c_{i+1}. Therefore a red c_i must be "fixed" before such an operation could require c_i. But fixing c_i itself should be possible by accessing only c-elements on c_{i+1}. To keep the requirements on c_{i+1} to a minimum, fixing c_i should be done by extracting only one c-element from c_{i+1}. We will see that observing the first c-element on c_i —as may be needed by front if $i = 0$— or extracting it —as will be needed by popFront if $i = 0$ or by the fixing of c_{i-1}— will require that the first c-element on $c_i, c_{i+1}, \ldots$ is stored in c_i. Then we have that a red c_i is one that needs to be fixed before it becomes c_0 or before it is needed to fix c_{i-1}.

Furthermore, we will see that all the deques stored in a c-element of the form px or (px, cl), or that are the prefix of a $c = (px, sx)$, may eventually interchange positions. Since every c must have a color assigned, all these deques must have at least the length of the prefix of a red c, since they may become such a prefix. Therefore all the deques that occur in a c-element or that are the prefix of a $c = (px, sx)$ in l are required to be at least of length r, if r is the length of the prefix of a red c.

A yellow c_i is one that will become red after a c-element is extracted from its prefix, either by popFront or by the fixing of c_{i-1}. Then we have that a c_i is yellow if its prefix contains only one element more than the prefix of a red c. That is, if y is the length of the prefix of a yellow c, then $y = r + 1$.

A green c_i is one that can not become red after an element is extracted from its prefix. Therefore, if g is the length of the prefix of a green c_i, we have $g \geq r + 2$.

So far, we have the following properties:

$$y = r + 1 \qquad g \geq r + 2 \tag{7.6}$$

$$\text{Any } px \text{ occurring anywhere in } l \text{ is of length at least } r \tag{7.7}$$

Before finding a suitable value for r (and consequently for y and g) note that fixing a red c_{i-1} must be possible extracting one c-element from the prefix of c_i. But c_i must remain with a color assigned, for what its prefix must remain of length at least r. If c_i is also red, this would imply $r - 1 \geq r$. Therefore it is not possible to fix c_{i-1} if c_i is also red. The invariant will guarantee that whenever a red c_{i-1} is to be fixed, c_i is yellow or green.

By (7.2), (7.4) and (7.5), if the prefix of c_{i-1} contains c-elements of type B, the members of the prefix of c_i are of the form *c-el* B. That is, by (7.3), its members are either of the form px or (px, cl), with px a deque of B. Therefore to fix c_{i-1}, its prefix must be combined with px to make it green. By (7.7), px is of length at least r. Then combining two px's of length r must yield a prefix of length g, i.e. $r + r \geq g$. The first px is the one to be fixed and the second is the one extracted from the next c. From $r + r \geq g$ and (7.6) we obtain $r \geq 2$. The minimum r is chosen, thus:

$$r = 2 \tag{7.8}$$

However, $c_m = d$ is a special case. If $m = 0$ and d contains only one element, the whole c-list becomes empty after an element is extracted from it. If $m > 1$ and d is empty, c_m does not actually appear in l, since it is kept regular. If $c_{m-1} = (px, sx)$ has to be fixed, it can de done combining px and sx in a single deque, thus no c-element is required from c_m. If $m > 1$ and d is not empty, a c-element may be extracted from it to fix c_{m-1}, but the new d does not have to be fixed. Therefore a single deque does not pose any problem when an operation requires it. Then, a $c_m = d$ is always considered green[4].

The color assignment can now be stated as:

$$\begin{array}{ll} \multicolumn{2}{l}{\text{In any } l = [c_0, \ldots, c_n] \text{ representing a c-list:}} \\ \text{(i)} & \text{A } c_i = (px, sx) \text{ has the color:} \\ & \quad \begin{cases} \text{red} & \text{if } \mathsf{lengthSeq}\ px = 2 \\ \text{yellow} & \text{if } \mathsf{lengthSeq}\ px = 3 \\ \text{green} & \text{if } \mathsf{lengthSeq}\ px \geq 4 \end{cases} \\ \text{(ii)} & \text{If } c_n = d, \text{ it is always green} \end{array} \tag{7.9}$$

We will use predicates isGreen, isYellow and isRed that hold when its argument c is respectively green, yellow or red. However, only isYellow and isRed are actually implemented. Moreover, since only pairs (px, sx) can be yellow or red, the implementations are defined only for those pairs, following (7.9) (see figure 14).

[4]In [KT95],a single deque is considered a suffix, and a c with no prefix is always green. Our description is equivalent.

Finally, we describe the invariant property for l. Each red c_i in l must be preceded by an unique green c_j. We call this green c_j the *safety* c of c_i, as defined by:

$$\mathsf{safety}\ c_i = c_j \quad \equiv \quad j < i \wedge \mathsf{isGreen}\ c_j \wedge \forall k (j < k < i \implies \mathsf{isYellow}\ c_k) \qquad (7.10)$$

Any red c_i in an internal l' stored in a c-element of the form (px, l') is also required to have a safety c except for the first red c in l'. Then, we state the invariant, which incorporates property (7.7), as:

Any list $l = [c_0, \ldots, c_n]$ representing a c-list has the following properties:
(i) $i \in \{0, \ldots, n\} \wedge \mathsf{isRed}\ c_i \implies \exists c_j(\mathsf{safety}\ c_i = c_j)$
(ii) Let c'_k be the first red c in any $(px, [c'_0, \ldots, c'_m])$ anywhere in l, then: (7.11)
$\forall i > k(\mathsf{isRed}\ c'_i \implies \exists c'_j(\mathsf{safety}\ c'_i = c'_j))$
(iii) The length of any px in a c_i or in a c-element anywhere in l is at least two

Note that this invariant implies that in l, c_0 is always non-red. If $c_0 = (px_0, sx_0)$ is yellow, it becomes red after a popFront extracts a c-element from px_0. This violates the invariant since now safety c_0 does not exists. If $c_0 = (px_0, sx_0)$ is green, it may become yellow after a popFront, possibly violating the invariant if the first non-yellow c_j after c_0 is red. In this case we had safety $c_j = c_0$, thus safety c_j no longer exists. In any case, the invariant is restored fixing the first non-yellow c in l. We call this procedure a *normalization* of l.

As said before, fixing a c_i requires c_{i+1} to be non-red. If the normalization needs to fix c_0, it was yellow before the current operation, therefore c_1 is not red or otherwise safety c_1 did not exist in l. If c_{i+1} is needed to fix a c_i with $i > 0$, c_i has not changed and it is red. Therefore c_{i+1} is non-red or otherwise safety c_{i+1} did not exist.

Operations, first version

The insertion operations cons and snoc insert its argument (in A) in the list l wrapped with the constructor `A`, to make it a c-element. To sort out the possible different configurations of l, the functions consL and snocL are used. This functions insert the new c-element in l accordingly. Thus cons and snoc are implemented by:

```
cons x (L cs) = L (consL (A x) cs)

snoc (L cs) x = L (snocL cs (A x))
```

If l is empty, consL and snocL create a single deque with the new c-element and make it the new l. If c_0 in l is a single deque, the new c-element is inserted on it. Otherwise the new c-element in inserted in px_0 or sx_0 accordingly. This gives the following implementation:

```
consL cel []              = [Dq (cons cel empty)]
consL cel [Dq d]          = [Dq (cons cel d)]
consL cel (PS (px,sx):cs) = PS (cons cel px,sx) : cs

snocL []              cel = [Dq (snoc empty cel)]
snocL [Dq d]          cel = [Dq (snoc d cel)]
snocL (PS (px,sx):cs) cel = PS (px,snoc sx cel) : cs
```

Both implementations only work in c_0 and perform either a cons or a snoc on deques, which are assumed to be implemented with constant time complexity. Therefore both c-list operations are also implemented with constant time complexity. Since they can not create a red c, the invariant is not violated.

To catenate two c-lists, cat defers the work to catL that receives the two l's to create the new c-list:

```
cat (L cs1) (L cs2) = L (catL cs1 cs2)
```

Given l_1 and l_2 to create a new list, we first distinguish when at least one of the lists is empty:

```
catL [] l2 = l2
catL l1 [] = l1
```

For the rest of catL we consider three major cases. First, when l_1 is made of a single deque, i.e. $l_1 = [d_1]$. The second case is when $l_2 = [d_2]$ (and l_1 is not a single deque). The third case is when none of the lists is made of a single deque. This organization differs from that used in [KT95], but gives the same result.

If $l_1 = [d_1]$, d_1 will be a prefix of the new list. If it has less than four c-elements, its c-elements are cons-ed into l_2. If it has at least four c-elements, it is made the prefix of the new list and the first suffix of l_2 will become the suffix of the new list. If $l2 = [d_2]$, a new one-element list is created with $c_0 = (d1, d2)$. Finally, if $l_2 = [(px_2, sx_2), \ldots]$, sx_2 will be the suffix of the new list and px_2 must precede the rest of the elements in l_2. This gives the following implementation:

```
catL [Dq d1] l2
    | lengthSeq d1 < 4   = foldrD consL l2 d1
    | otherwise = case l2 of
                    [Dq d2] -> [PS (d1,d2)]
                    (PS (px2,sx2):cs) -> (PS (d1,sx2) :
                                          consL (PL px2 cs) [Dq empty])
```

This case makes evident what c-elements of the form (px, l') are used for. Note that since the first prefix of the new l has at least four elements, no red c is created, thus the invariant is preserved.

In the second major case $l_1 = [(px_1, sx_1), c_{11}, \ldots]$ and $l_2 = [d_2]$. The new c-list must conform to the ordering $px_1, c_{11}, \ldots, sx_1, d_2$. Therefore px_1 can be made the prefix of the new c-list and d_2 its suffix. The c-elements in sx_1 are inserted in either $c_{11}, \ldots$ or d_2 with the function mkSxEmpty obtaining c'_{11} and d'_2.

```
catL (PS (px1,sx1):cs1) [Dq d2] = let (cs1',d2') = mkSxEmpty cs1 sx1 d2
                                  in PS (px1,d2') : cs1'
```

This case has a constant time complexity since mkSxEmpty is also implemented with constant time complexity. The length of px_1 is not decreased, therefore the invariant is preserved.

The auxiliary function mkSxEmpty is implemented by:

```
mkSxEmpty cs sx px
    | n == 0    = (cs,px)
    | n == 1    = (cs,cons (front sx) px)
    | otherwise = (snocL cs (P sx),px)
    where n = lengthSeq sx
```

Note that if the length of sx is 1, it can not be made a prefix by property (7.11)(iii).

The final major case is with $l_1 = [(px_1, sx_1), c_{11}, \ldots]$ and $l_2 = [(px_2, sx_2), c_{12}, \ldots]$. The new c-list must conform with the ordering $px_1, c_{11}, \ldots, sx_1, px_2, c_{12}, \ldots, sx_2$. This is accomplished if in the new l, c_0 is (px_1, sx_2). Then the c-elements in sx_1 are inserted in either c_{11} or px_2 with the function mkSxEmpty obtaining $c'_{11}, \ldots$ and px'_2. Then the c-element $(px'_2, [c_{12} \ldots])$ is snoc-ed into $[c'_{11}, \ldots]$. This gives the implementation:

```
catL (PS (px1,sx1):cs1) (PS (px2,sx2):cs2) =
    let (cs1',px2') = mkSxEmpty cs1 sx1 px2
    in PS (px1,sx2) : snocL cs (PL px2' cs2')
```

Again, px_1 remains the same or its length is increased, therefore it does not become red and the invariant is preserved.

The implementation of front is very simple. An auxiliary function popCEl is used to split the list in a pair consisting of its first c-element and the list without it. By (7.5)(ii), the first c-element is an $x \in A$. The implementation of front is:

```
front (L cs) = let (A x,_) = popCEl cs in x
```

```
newtype CList a = L [C a]
data C a = Dq (DequeH (El a)) | PS (DequeH (El a),DequeH (El a))
data El a = A a | P (DequeH  (El a)) | PL (DequeH  (El a)) [C a]

instance Seq CList where
    empty = L []

    isEmpty (L cs) = null cs

    cons x (L cs) = L (consL (A x) cs)

    snoc (L cs) x = L (snocL cs (A x))

    front (L []) = error "front: empty catenable list"
    front (L cs) = let (A x,_) = popCEl cs in x

    popFront (L []) = error "popFront: empty catenable list"
    popFront (L cs) = let (_,cs') = popCEl cs in L (normL cs')

instance CatSeq CList where
    cat (L cs1) (L cs2) = L (catL cs1 cs2)

consL cel []              = [Dq (cons cel empty)]
consL cel [Dq d]          = [Dq (cons cel d)]
consL cel (PS (px,sx):cs) = PS (cons cel px,sx) : cs

snocL []              cel = [Dq (snoc empty cel)]
snocL [Dq d]          cel = [Dq (snoc d cel)]
snocL (PS (px,sx):cs) cel = PS (px,snoc sx cel) : cs

catL [] l2 = l2
catL l1 [] = l1
catL [Dq d1] l2
    | lengthSeq d1 < 4   = foldrD consL l2 d1
    | otherwise          = case l2 of
                               [Dq d2] -> [PS (d1,d2)]
                               (PS (px2,sx2):cs) -> PS (d1,sx2) :
                                                    consL (PL px2 cs) [Dq empty]
catL (PS (px1,sx1):cs1) [Dq d2] = let (cs1',d2') = mkSxEmpty cs1 sx1 d2
                                  in PS (px1,d2') : cs1'
catL (PS (px1,sx1):cs1) (PS (px2,sx2):cs2) =
    let (cs1',px2') = mkSxEmpty cs1 sx1 px2
    in PS (px1,sx2) : snocL cs1' (PL px2' cs2)

normL l@(PS (px,sx):cs)
    | isYellow (px,sx) = PS (px,sx) : normL cs
    | isRed (px,sx)    = fix (px,sx) cs
    | otherwise = l
normL cs = cs

fix (px,sx) [] = [Dq (foldrD cons sx px)]
fix (px,sx) cs = case popCEl cs of
                     (P px1,cs')     -> PS (foldrD cons px1 px,sx):cs'
                     (PL px1 l',cs')-> PS (foldrD cons px1 px,sx):
                                           catL l' cs'

isRed (px,sx) = lengthSeq px == 2

isYellow (px,sx) = lengthSeq px == 3

mkSxEmpty cs sx px
    | n == 0    = (cs,px)
    | n == 1    = (cs,cons (front sx) px)
    | otherwise = (snocL cs (P sx),px)
    where n = lengthSeq sx

popCEl [Dq d]
    | lengthSeq d == 1  = (front d,[])
    | otherwise         = (front d,[Dq (popFront d)])
popCEl (PS (px,sx):cs) = (front px,PS (popFront px,sx):cs)
```

Figure 14: Implementation of `CListRS`, first version.

To extract the first c-element in l, we differentiate when the first c is a single deque or a pair (px, sx). In each case the first c-element is extracted and, if the first c is a single deque, we take care of keeping the list regular. This gives the implementation:

```
popCEl [Dq d]
    | lengthSeq d == 1  = (front d,[])
    | otherwise         = (front d,[Dq (popFront d)])
popCEl (PS (px,sx):cs) = (front px,PS (popFront px,sx):cs)
```

Both front and popCEl are clearly of constant time complexity.

The remaining efficient operation is popFront, which is easily implemented but may cause a violation of the invariant needing it to be restored. We only need to extract the first c-element of c_0, which is done by popCEl, The resulting list might need to be normalized, therefore normL is invoked. The implementation of popFront is:

```
popFront (L cs) = let (_,cs') = popCEl cs in L (normL cs')
```

This implementation performs some constant time operations and invokes normL.

As said before when the invariant was described, it can be violated if safety c_k no longer exists for c_k the first red c in l. Furthermore, it can only happen if, in the offending list, the first non-yellow c is red. The problem is solved if we look for the first non-yellow c_k in l. If it is green, the invariant was not violated. If it is red, it is fixed by fix, that receives the red pair and the rest of the list. Then normL is implemented by:

```
normL l@(PS (px,sx):cs)
    | isYellow (px,sx) = PS (px,sx) : normL cs
    | isRed (px,sx)    = fix (px,sx) cs
    | otherwise = l
normL cs = cs
```

The second clause catch a list that contains a single deque or that becomes empty when looking for the first non-yellow c.

To fix a red (px, sx) if the rest of the list is empty, the c-elements of px —two, since (px, sx) is red— are incorporated to sx. The implementation is:

```
fix (px,sx) [] = [Dq (foldrD cons sx px)]
```

The invariant is restored since the new c is green.

If the rest of the list is not empty, the first c-element on it is extracted with popCEl. By (7.5)(ii), it is a prefix c-element or a pair prefix-c-list. If it is a prefix px_1, the elements of px are incorporated to px_1 which becomes the prefix of the fixed pair. If it is a pair (px_1, l'), the former strategy is applied but in addition l' is catenated with the rest of the list. Since px has two c-elements and px_1 has at least two (by (7.11)(iii)), the fixed pair is green. This is implemented by:

```
fix (px,sx) cs = case popCEl cs of
                   (P px1,cs')     -> PS (foldrD cons px1 px,sx):cs'
                   (PL px1 l',cs')-> PS (foldrD cons px1 px,sx):
                                      catL l' cs'
```

To see that the invariant is actually restored in this case, let c_i and $[c_{i+1}, \ldots]$ be the arguments of fix. That is, c_i is the first non-yellow c in the list l to be normalized. The invariant guarantees that c_{i+1} is non-red, otherwise safety c_{i+1} did not exist in l. The c_j's with $j < i$ in the normalized list remains the same and do not violate the invariant since they are all yellow. The rest of the new list, constructed by fix, begins with a green c'_i, as said above.

If the c-element extracted from c_{i+1} is of the form px, the next member of the new list is c'_{i+1}, which is c_{i+1} with its first c-element extracted. If c'_{i+1} is red, we have that safety $c'_{i+1} = c'_i$. Therefore the invariant is restored if all the following red c's in the new list have its corresponding safety pair. But all the c_j's with $j > i + 1$ remain unchanged in the new list, therefore all those that are red have their corresponding safety pairs within themselves.

If the extracted c-element is of the form (px, l'), the rest of the new list is the catenation of l' and the rest of the list to be normalized. As said above, the latter obeys the invariant. Besides, any red c in l' apart from possibly the first —see (7.11)(ii)— has its safety pair within l'. Therefore the catenation of l' and the rest of the list to be normalized might have only its first red c without a safety pair on it. In this case, the required safety pair is c'_i.

It can be seen that once the first non-yellow c on the list to be normalized is found, the work performed has constant time complexity. But finding the first non-yellow c implies an arbitrary traversal of the list. This traversal will be avoided in the final version of the implementation, whose description follows. The first version described so far appears in figure 14.

Final version of the implementation

As said before, the implementation as described so far does not achieve constant time complexity for popFront since it may require a normalization of the data structure. The normalization traverse the list l until the first non-yellow c is found. This traversal might be as far as to include the whole list, therefore having linear time on the length of the c-list.

To overcome this problem, the structure of l is modified by grouping contiguous yellow c's into an internal list. That is, if $c_i, \ldots, c_k$ are all yellow, they are stored as a single $c_{i..j} = [c_i, \ldots, c_j]$. We denote the modified list by l', that contains elements of the form $c_{i..j}$. Each $c_{i..j}$ in l' represents either one c in l or a list of c's in l, as described by:

$$c_{i..j} = \begin{cases} c_i & \text{if } \mathsf{not}\ (\mathsf{isYellow}\ c_i) \wedge i = j \\ [c_i, \ldots, c_j] & \text{if } \forall k \in \{i, \ldots, j\} : \mathsf{isYellow}\ c_k \end{cases} \tag{7.12}$$

We also keep the lists of yellow c's of maximal length and not empty, that is:

$$\begin{array}{ll} \text{If } c_{i..j} = [(px_i, sx_i), \ldots, (px_j, sx_j)] \text{ in } l' : & \\ \quad \text{(i)} \quad \text{if } i > 0 : \quad \mathsf{not}\ (\mathsf{isYellow}\ c_{i-1}) & \\ \quad \text{(ii)} \quad \text{if } c_{j+1} \text{ exists in } l : \quad \mathsf{not}\ (\mathsf{isYellow}\ c_{j+1}) & \\ \quad \text{(iii)} \quad j \geq i & \end{array} \tag{7.13}$$

In this way, the first non-yellow c in l is either $c_{0..0}$ in l' or $c_{i..i}$ with $l' = [c_{0..i-1}, c_{i..i}, \ldots]$. Then, the traversal made by normL (now of l') may only go as far as the second element of l'.

The type declarations for the final version are essentially the same as in the first version, except that members of type `C` now correspond to the $c_{i..j}$'s. Therefore they can now also be a list of prefix-suffix pairs, which is wrapped with the constructor `Y`. The final declaration is:

```
newtype CList a = L [C a]
data C a = Dq (DequeH (El a)) | PS (DequeH (El a),DequeH (El a))
          | Y [(DequeH (El a),DequeH (El a))]
data El a = A a | P (DequeH  (El a)) | PL (DequeH  (El a)) [C a]
```

The final implementation appears in figures 15 and 16.

We now describe the changes made to the previous implementation to obtain the final version. We start by describing the normalization function normL, whose improvement was the goal of the modification of the data structure. This function looks for the first non-yellow element of l' and, if it is red, fix it with fixY. The final implementation simply bypass $c_{0..j}$ if it is a yellow list, that is:

```
normL (Y pss:cs) = (Y pss : normL cs)
normL l@(PS (px,sx):cs)
    | isRed (px,sx) = fix (px,sx) cs
    | otherwise = l
normL cs = cs
```

This function appears in figure 15. The function fixY is not changed and appears in figure 16.

We introduce a new auxiliary function popPS that extracts from l' the first prefix-suffix pair. In the first version, the list l was usually pattern-matched into three cases: l being empty, l with a single deque and l with a prefix-suffix pair as its first element. The latter case does not have a single pattern in l', since the first pair could be a stand-alone pair or the first member of a list of yellow pairs. These configurations are handled by popPS, implemented by:

```
newtype CList a = L [C a]
data C a = Dq (DequeH (El a)) | PS (DequeH (El a),DequeH (El a))
         | Y [(DequeH (El a),DequeH (El a))]
data El a = A a | P (DequeH  (El a)) | PL (DequeH  (El a)) [C a]

instance Seq CList where
    empty = L []

    isEmpty (L cs) = null cs

    cons x (L cs) = L (consL (A x) cs)

    snoc (L cs) x = L (snocL cs (A x))

    front (L [])    = error "front: empty catenable list"
    front (L cs)    = let (A x,_) = popCEl cs in x

    rear (L []) = error "rear: empty catenable list"
    rear cs     = last (toList cs)

    popFront (L []) = error "popFront: empty catenable list"
    popFront (L cs) = let (_,cs') = popCEl cs in L (normL cs')

    toSeq [] = L []
    toSeq xs = L [Dq (toSeq (map A xs))]

    toList (L [])  = []
    toList (L cs) = foldr toListC [] cs

instance CatSeq CList where
    cat (L cs1) (L cs2) = L (catL cs1 cs2)

consL :: El a -> [C a] -> [C a]
consL cel []     = [Dq (cons cel empty)]
consL cel [Dq d] = [Dq (cons cel d)]
consL cel cs     = let ((px,sx),cs') = popPS cs
                   in  fixY (PS (cons cel px,sx) : cs')

snocL :: [C a] -> El a -> [C a]
snocL []     cel = [Dq (snoc empty cel)]
snocL [Dq d] cel = [Dq (snoc d cel)]
snocL cs     cel = let ((px,sx),cs') = popPS cs
                   in fixY (PS (px,snoc sx cel) : cs')

catL :: [C a] -> [C a] -> [C a]
catL [] l2 = l2
catL l1 [] = l1
catL [Dq d1] l2
    | lengthSeq d1 < 4 = foldrD consL l2 d1
    | otherwise = case l2 of
                     [Dq d2] -> [PS (d1,d2)]
                     _       -> let ((px2,sx2),cs) = popPS l2
                                in PS (d1,sx2):consL (PL px2 cs) [Dq empty]
catL l1 [Dq d2] = let ((px1,sx1),cs1) = popPS l1
                      (cs1',d2')      = mkSxEmpty cs1 sx1 d2
                  in fixY (PS (px1,d2') : cs1')
catL l1 l2 = let ((px1,sx1),cs1) = popPS l1
                 ((px2,sx2),cs2) = popPS l2
                 (cs1',px2')     = mkSxEmpty cs1 sx1 px2
             in fixY (PS (px1,sx2) : snocL cs1' (PL px2' cs2))

normL :: [C a] -> [C a]
normL (Y pss:cs) = (Y pss : normL cs)
normL l@(PS (px,sx):cs)
    | isRed (px,sx) = fix (px,sx) cs
    | otherwise = l
normL cs = cs
```

Figure 15: Final version of CListRS: part I.

```
popPS (PS ps      : cs) = (ps,cs)
popPS (Y [ps]     : cs) = (ps,cs)
popPS (Y (ps:pss) : cs) = (ps,(Y pss:cs))
```

As can be seen, popPS returns a pair whose first component is the first pair in l' and the rest of l'. Care is taken of not leaving empty lists of yellow pairs in l'.

Another new auxiliary function is fixY. It is invoked whenever a prefix-suffix pair that could be yellow is inserted in l'. This pair is always inserted as the head of l'. If the pair is yellow, it is made a one-element yellow list or inserted in a adjacent yellow list if it exists. The implementation is:

```
fixY l@(PS (px,sx) : cs)
    | isYellow (px,sx) = case cs of
                             (Y pss:cs') -> Y ((px,sx):pss) : cs'
                             _           -> Y [(px,sx)] : cs
    | otherwise        = l
```

The function popCEl is slightly modified. It splits l' (previously l) into a pair whose first component is the first c-element in l' and l' without it. When $c_{0..i}$ is a prefix-suffix pair, popCEl extracts this pair with popPS and inserts it back in l' after extracting its first c-element. Since the pair inserted back might be yellow, fixY is invoked. The implementation is:

```
popCEl [Dq d]
    | lengthSeq d == 1 = (front d,[])
    | otherwise        = (front d,[Dq (popFront d)])
popCEl cs = let ((px,sx),cs') = popPS cs
            in (front px,fixY (PS (popFront px,sx):cs'))
```

The functions consL, snocL and catL are modified in the same way as popCEl. They all had a clause to extract the first prefix-suffix pair of l'. This was done by pattern-matching in the first version. This pattern-matching is now replaced by a `let` expression that invokes popPS, as was shown for popCEl. The new versions can be seen in figure 15.

The function toListC —described in page 42— takes a $c_{i..j}$ and the list of elements of the c-list contained in $c_{j+1..k}, \ldots$ and gives the list of c-list elements contained in $c_{i..j}, c_{j+1..k}, \ldots$. A clause is added to handle the new kind of elements of type `C`, namely the lists of yellow pairs. Each yellow list is processed as an internal list of pairs, for what the constructor `PS` is mapped to the internal list. The new clause is:

```
toListC (Y pss) lrest = foldr toListC lrest (map PS pss)
```

The whole new version appears in figure 16.

The remaining functions implemented in the first version do not change. For the final version we add the coercion functions. The function toSeq was described before. It remains the same except for the modification of toListC described above.

To convert a list in a c-list, the simplest possible c-list is created. It is either the empty c-list or a c-list represented by a list containing a single deque. The members of the c-list are made c-elements applying the constructor `A` to each one of them. The implementation is:

```
toSeq [] = L []
toSeq xs = L [Dq (toSeq (map A xs))]
```

The sequence operation rear does not have to be efficient in a c-list. Its implementation for `CListRS` is not as straightforward as it may seem at first sight. The difficulty being that the suffixes stored in prefix-suffix pairs can be empty. Therefore l' has to be traversed until a non-empty suffix (including the final single deque) is found. If all the suffixes are empty the rear element is found in the last c-element in the last prefix. In any case, if the c-element that contains the rear element is of the form px or (px, l), further manipulation of this c-element is needed. We opt for a simple though inefficient implementation:

```
rear cs = last (toList cs)
```

```
fix :: (DequeH (El a) , DequeH (El a)) -> [C a] -> [C a]
fix (px,sx) [] = [Dq (foldrD cons sx px)]
fix (px,sx) cs = case popCEl cs of
                     (P px1,cs')    -> PS (foldrD cons px1 px,sx):cs'
                     (PL px1 l',cs')-> PS (foldrD cons px1 px,sx):
                                             catL l' cs'

isRed :: (DequeH (El a) , DequeH (El a)) -> Bool
isRed (px,sx) = lengthSeq px == 2

isYellow :: (DequeH (El a) , DequeH (El a)) -> Bool
isYellow (px,sx) = lengthSeq px == 3

mkSxEmpty :: [C a] -> DequeH (El a) -> DequeH (El a) -> ([C a],DequeH (El a))
mkSxEmpty cs sx px
    | n == 0    = (cs,px)
    | n == 1    = (cs,cons (front sx) px)
    | otherwise = (snocL cs (P sx),px)
    where n = lengthSeq sx

popCEl :: [C a] -> (El a , [C a])
popCEl [Dq d]
    | lengthSeq d == 1 = (front d,[])
    | otherwise        = (front d,[Dq (popFront d)])
popCEl cs = let ((px,sx),cs') = popPS cs
            in (front px,fixY (PS (popFront px,sx):cs'))

popPS :: [C a] -> ((DequeH (El a),DequeH (El a)) , [C a])
popPS (PS ps      : cs) = (ps,cs)
popPS (Y [ps]     : cs) = (ps,cs)
popPS (Y (ps:pss) : cs) = (ps,(Y pss:cs))

fixY :: [C a] -> [C a]
fixY l@(PS (px,sx) : cs)
    | isYellow (px,sx) = case cs of
                           (Y pss:cs') -> Y ((px,sx):pss) : cs'
                           _           -> Y [(px,sx)] : cs
    | otherwise        = l

toListC :: C a -> [a] -> [a]
toListC (Dq d)       lrest = concatMap toListCEl (toList d) ++ lrest
toListC (PS (px,sx)) lrest = concatMap toListCEl (toList px) ++ lrest ++
                             concatMap toListCEl (toList sx)
toListC (Y pss)      lrest = foldr toListC lrest (map PS pss)

toListCEl :: El a -> [a]
toListCEl (A x)      = [x]
toListCEl (P px)     = concatMap toListCEl (toList px)
toListCEl (PL px cs) = concatMap toListCEl (toList px) ++ foldr toListC [] cs
```

Figure 16: Final version of CListRS: part II.

8 Comments and further work

We have described a representative sample of the published work in functional data structures. Though we have restricted our attention to sequences, the main design techniques used for functional data structures are represented in the implementations described. Five major techniques can be identified from the works in the area:

1. Basic Trees. Most of the basic kinds of trees used to implement data structures in general can be implemented in functional languages. Usually, the operations defined on this trees can be expressed more succinctly and elegantly in a functional language. Among the basic trees frequently used we can cite AVL-trees, (2,3)-trees, B-trees, weight-balanced trees, etc. Two basic kinds of trees were described in section 3. Their use to implement sequences is exemplified in the implementations of flexible arrays in section 6.

2. Sequences as pair of lists. An old technique to implement sequences in functional languages of which many variations have been described. These technique is used in the implementations of queues and deques described in sections 4 and 5.

3. Numerical representations. The representation of a natural number n in base b consists of a sequence of digits $a_0, a_1, \ldots, a_{\log_b n}$ where the digit a_i has weight b^i. This representation is analogous to the representation of a sequence as a list of trees $t_0, t_1, \ldots, t_m$ where each tree t_i contains b^i elements. A simplified version of the implementation of flexible arrays as leaf trees in section 6.2, described in [DK95], uses a single leftist left-perfect tree (see section 3.1) to represent a random-access list. This single leaf tree is obtained by the function `toSeq` of section 6.2, where a forest of perfect leaf trees is joined in a single tree. An equivalent data structure is obtained if the forest (actually a list of trees) is manipulated as such. This forest representation is an example of an implementation based in a binary numbers representation. The functions over leftist left-perfect trees of section 6.2 have an equivalent version that works on a forest instead of a single tree. We did not describe the forest implementation since it does not support efficient `snoc` and `popRear`. A description of this analogy can be found in [Oka96b], as well as a good description of the general technique. Other examples of the use of these technique are those in [Oka95a] and [Mye83], where implementations of random-access lists are described in analogy to skew-binary number systems.

4. Data-structural bootstrapping. These technique was described in Buchsbaum's thesis [Buc93]. It consists of two basic ideas: *Structural Abstraction* and *Structural Decomposition*. Structural abstraction is typically used to provide efficient catenation for a data type that do not support it. This can be done by inserting the data type as an element of its own. Examples of this technique are the implementations of catenable lists described in section 7, where whole lists are inserted into the abstracted list to achieve efficient catenation. Structural decomposition refers to the recursive decomposition of a data structure into data structures of limited size. This idea is present in the recursive-slowdown deques described in section 5.2.

5. Recursive-slowdown. This technique can be seen as uniting the three previous techniques. The two sequences implemented with this technique in [KT95] were described in sections 5.2 and 7.2. In them, a sequence is recursively decomposed in a collection of smaller data structures as in data-structural bootstrapping. Each of these smaller data structures is implemented as a pair of lists, and the general organization of the data structure with a system of colors is analogous to a redundant numerical representation where the colors red, yellow and green can be seen, respectively as the digits 2,0, and 1. This analogy is described in [KT96].

The data structures developed with these techniques provide most of the basic operations on sequences with the same asymptotic behavior as those provided by its imperative counterparts. Efficient functional implementations of priority queues have also been designed with some of these techniques (e.g. [Kin94, BO96]). Efficient explicit implementations of the abstract data type dictionary have also been described, mainly with the use of balanced trees (e.g. [Mye84]). However, efficient data structures for some well-known problems have not yet been described. Notably those related to graph algorithms —though some work have been presented in, for example, [Lau95]— and specially to the union-find problem. These are interesting areas of research.

8.1 Directions for further work

In the short term we plan to conduct research in the following two directions:

1. Improvements and extensions to existing data structures. We have obtained some preliminary results with a variation of the implementation of flexible arrays as leaf trees described in section 6.2. With a simplification of the algorithm to perform insertions in the tree a more general form of leaf trees is obtained. These new implementation seems to perform better at least in certain cases: when a flexible array is created only by repeated applications of cons and only by repeated applications of snoc. Some preliminary work have been done to measure the average depth of the leaves for arrays created by a random application of either cons and snoc. So far, the results for this new implementation have been better than those for the original data structure. More extensive measures, however, remain to be done, as well as a study of the worst-case performance of the new version. We have also sketched an extension to the random-access list implementation on [Oka95a] to support efficient insertions and deletions at the end of the list.

2. Better presentation of the recursive-slowdown structures. The description of these data structures in [KT95] have been improved here in sections 5.2 and 7.2. We plan to refine them to obtain a nice formal description of these structures, as well as an implementation in a more functional style. These could lead to uncover general principles in their design and to clarify their relation to the other design techniques. In particular, we expect it to state clearly their relation to finger search trees (see next topic).

Afterwards, we plan to study the relation between some of the data structures described here and the use of fingers in search trees [GMPR77, Kos81]. The relation of the recursive-slowdown deques of section 5.2 with finger search trees have been sketched in [KT96]. Also, the random-access list implementation as a forest of leaf tree mentioned before can be seen equivalent to the presence of a finger in a leaf tree. Consider, for example, a leftist left-perfect leaf tree. If we start at the leftmost leaf and proceed upwards trough the left spine until the root node, the successive right children of the nodes in this path are perfect leaf trees of increasing size. This sequence of perfect trees is precisely that obtained by the function forestOfPerfect of section 3.1. If we perform a search in this list, it is equivalent to a search on the leftist left-perfect tree that starts on the leftmost leaf and proceeds upwards through the left spine an then downwards in the appropriate node. This search is exactly how it would be performed if a finger were placed in the leftmost leaf and the nodes in the left spine pointed to its parent. This corresponds to a simple version of a finger search leaf tree. We plan to explore further these relations.

Additional research topics that we would like to explore are:

Monads in functional data structures The use of monads is becoming increasingly popular among the functional programming community (e.g. [HM96, MJ95, PJW93, Wad92]). One of its major uses is to incorporate some imperative features into functional programming languages. It seems reasonably to expect that these imperative features could be used in functional data structures to adapt some imperative data structures to a functional setting.

Circular programs and circular data structures Lazy functional languages naturally allow the creation of circular data structures and the design of circular programs. These features may yield more efficient algorithms for some operations in functional languages [Bir84, BY90]. Indeed, at least for certain problems, lazy languages seem to be more powerful than eager languages [BJdM97]. They are also naturally obtained with a programming technique described in [vGvdB95]. However, the circular structures described so far have a major shortcoming: they can not be easily modified once they are created [All89, KW91].

References

[AHU74] A. Aho, J. E. Hopcroft, and J. D. Ullmann. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison–Wesley, 1983.

[All89] Lloyd Allison. Circular Programs and Self-referential Structures. *Software Practice and Experience*, 19(2):99–109, February 1989.

[BAG92] Amir M. Ben-Amram and Zvi Galil. On Pointers versus Addresses. *Journal of the ACM*, 39(3):617–648, July 1992.

[Bir84] R. S. Bird. Using Circular Programs to Eliminate Multiple Traversal of Data. *Acta Informatica*, 21(3):239–250, 1984.

[Bir97] Richard S. Bird. On building trees with minimun height. *Journal of Functional Programming*, 7(4), July 1997.

[BJdM97] Richard Bird, Geraint Jones, and Oege de Moor. More haste, less speed: lazy versus eager evaluation. *Journal of Functional Programming*, 7(5), September 1997.

[BO96] Gerth Stølting Brodal and Chris Okasaki. Optimal Purely Functional Priority Queues. *Journal of Functional Programming*, 6(6):839–858, November 1996.

[BR83] W. Braun and M. Rem. A logarithmic implementation of flexible arrays. Memorandum MR83/4, Eindhoven University of Technology, 1983.

[BST92] A. L. Buchsbaum, R. Sundar, and R. E. Tarjan. Data structural bootstrapping, linear path compression, and catenable heap ordered double ended queues. In *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science*, pages 40–49, 1992.

[Buc93] Adam Louis Buchsbaum. *Data-structural Bootstrapping and catenable deques.* PhD thesis, Department of Computer Science, Princeton University, June 1993.

[Bur82] F. Warren Burton. An efficient functional implementation of FIFO queues. *Information Processing letters*, 14(5):205–206, July 1982.

[BY90] F. Warren Burton and Hsi-Kai Yang. Manipulating Multilinked Data Structures in a Pure Functional Language. *Software Practice and Experience*, 20(11):1167–1185, November 1990.

[CG93] Tyng-Ruey Chuang and Benjamin Goldberg. Real-Time Deques, Multihead Turing Machines, and Purely Functional Programming. In *FPCA '93. Proceedings of the conference on Functional Programming Languages and Computer Architecture*, pages 289–298, Copenhagen, Denmark, June 1993. ACM.

[Cha91] D. J. Challab. Implementation of Flexible Arrays Using Balanced Trees. *The Computer Journal*, 34(5):386–396, October 1991.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[DK95] Victor J. Dielissen and Anne Kaldewaij. A simple, efficient, and flexible implementation of flexible arrays. In Bernhard Moller, editor, *Mathematics of Program Construction, Conference Proceedings (MPC'95)*, number 947 in Lecture Notes in Computer Science, pages 232–241. Springer, July 1995.

[DR91] Paul F. Dietz and Rajeev Raman. Persistence, Amortization and Randomization. In *Proceedings of the second annual ACM-SIAM Symposium on Discrete Algorithms*, pages 78–88, San Francisco, CA, 1991.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38(1):86–124, Febuary 1989.

[DST94] James Driscoll, Daniel D. K. Sleator, and Robert E. Tarjan. Fully Persistent Lists with Catenation. *Journal of the ACM*, 41(5):943–959, September 1994.

[GMPR77] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A New Representation For Linear Lists. In *Conference Record of The Ninth Annual ACM Symposium on Theory of Computing*, pages 49–60, Boulder, Colorado, May 1977.

[Gri81] David Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[GT86] Hania Gajewska and Robert E. Tarjan. Deques with heap order. *Information Processing letters*, 22:197–200, 1986.

[HM81] Robert Hood and Robert Melville. Real-Time queue operations in pure Lisp. *Information Processing letters*, 13(2):50–54, November 1981.

[HM96] Graham Hutton and Erik Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computing Science, University of Nottingham, 1996.

[Hoo82] Robert Hood. *The Efficient Implementation of Very-High-Level Programming Language Constructs.* PhD thesis, Department of Computer Science, Cornell University, 1982. Also Tec. Report TR 82–503.

[Hoo92a] Rob R. Hoogerwoord. A Logarithmic Implementation of Flexible Arrays. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of Program Construction, Second International Conference Proceedings*, number 669 in Lecture Notes in Computer Science, pages 191–207, Berlin, June–July 1992. Springer-Verlag.

[Hoo92b] Rob R. Hoogerwoord. A symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513, October 1992.

[IS91] Giuseppe Italiano and Neil Sarnak. Fully Persistent Data Structures for Disjoint Set Union Problems. In F. Dehne, J. R. Sack, and N. Santoro, editors, *Algorithms and Data Structures*, volume 519 of *Lecture Notes in Computer Science*, pages 449–460, Ottawa, Canada, August 1991. 2nd Workshop, WADS'91, Springer-Verlag.

[Jon91] Mark P. Jones. *An introduction to Gofer*, 1991. Available at http://iamwww.unibe.ch/~scg/SW/Gofer.

[Jon96] Mark P. Jones. *Hugs 1.3, The Haskell User's Gofer System. User Manual*, 1996. Available at http://cs.nott.ac.uk/Department/Staff/mpj/hugs13/manual/manual.html.

[KD96] Anne Kaldewaij and Victor J. Dielissen. Leaf Trees. *Science of Computer Programming*, 26(1–3):149–165, May 1996.

[Kin94] D. J. King. Functional Binomial Queues. In K. Hammond, D.N. Turner, and P.M. Sansom, editors, *Glasgow Functional Programming Worshop*, Ayr, Scotland, 1994. Springer-Verlag.

[Kos81] S. Rao Kosaraju. Localized Search in Sorted Lists. In *Thirteenth Annual ACM ACM Symposium on Theory of Computing*, pages 62–69, Milwaukee, Wisconsin, May 1981. ACM.

[KT95] Haim Kaplan and Robert E. Tarjan. Persistent Lists with Catenation via Recursive Slow-Down. In *ACM Symposium on Theory of Computing*, pages 93–102, Las Vegas, Nevada, May 1995.

[KT96] Haim Kaplan and Robert E. Tarjan. Purely Functional Representations of Catenable Sorted Lists. In *ACM Symposium on Theory of Computing*, pages 202–211, May 1996.

[KW91] Yugo Kashiwagi and David S. Wise. Graph Algorithms in a Lazy Functional Programming Language. In *Proceedings of the 4th International Symposium on Lucid and Intensional Programming*, pages 35–46, 1991.

[Lau95] John Launchbury. Graph Algorithms with a Functional Flavour. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 308–331, Bastad, Sweden, May 1995. First International Spring School on Advanced Functional Programming Techniques, Springer.

[MJ95] Erik Meijer and Johan Jeuring. Merging Monads and Folds for Functional Programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 308–331, Bastad, Sweden, May 1995. First International Spring School on Advanced Functional Programming Techniques, Springer.

[MT90] K. Mehlhorn and A. Tsakalidis. Data Structures. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 301–341. Elsevier—MIT Press, 1990.

[Mye83] Eugene W. Myers. An Applicative Random-Access Stack. *Information Processing letters*, 17:241–248, December 1983.

[Mye84] Eugene W. Myers. Efficient Applicative Data Types. In *Eleventh Annual Symposium on Principles of Programming Languages*, pages 66–75, Salt Lake City, Utah, January 1984.

[Oka95a] Chris Okasaki. Purely Functional Random-Access Lists. In *Functional Programming Languages and Computer Architecture*, pages 86–95. ACM Press, June 1995.

[Oka95b] Chris Okasaki. Simple and Efficient Purely Functional Queues and Deques. *Journal of Functional Programming*, 5(4):583–592, October 1995.

[Oka96a] Chris Okasaki. Functional Data Structures. In *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 131–158. Second International Summer School, August 1996.

[Oka96b] Chris Okasaki. *Purely Functional Data Structures.* PhD thesis, School of Computer Science, Carnegie Mellon University, September 1996.

[Oka96c] Chris Okasaki. The Role of Lazy Evaluation in Amortized Data Structures. In *International Conference on Functional Programming*, pages 62–72, May 1996.

[Oka97a] Chris Okasaki. Catenable Double-Ended Queues. In *International Conference on Functional Programming*, pages 66–74, June 1997.

[Oka97b] Chris Okasaki. Three Algorithms on Braun Trees. *Journal of Functional Programming*, 7(6), November 1997. To appear.

[Pau91] Laurence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[PHe96] John Peterson and Kevin Hammond (editors). *Haskell 1.3. A Non-strict, Purely Functional Language*, May 1996. Available at http://www.cs.nott.ac.uk/Research/fpg/-report/haskell-report.html.

[PHe97] John Peterson and Kevin Hammond (editors). *Haskell 1.4. A Non-strict Purely Functional Language*, April 1997. Available at http://haskell.systemsz.cs.yale.edu/-onlinereport/.

[PJ96] Simon Peyton Jones. Bulk types with class. In Phil Trinder, editor, *Proceedings of the Glasgow Functional Programming Worshop*, Ullapool, Scotland, July 1996. Available at http://www.dcs.gla.ac.uk/fp/workshops/fpw96/Proceedings96.html.

[PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative Functional Programming. In *20th ACM Symposium on Theory of Computing*, pages 71–84, January 1993.

[Pon88] Carl G. Ponder. Are Applicatives Languages Inefficient? *SIGPLAN Notices*, 23(6):135–139, June 1988.

[Sed88] Robert Sedgewick. *Algorithms*. Addison-Wesley, 2nd edition, 1988.

[Tam96] Roberto Tamassia. Data Structures. *ACM Computer Surveys*, 28(1):23–26, March 1996.

[Tar85] Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

[Tho96] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.

[vGvdB95] F. A. van Gilst and P. M. van den Broek. A new programming technique for lazy functional languages. *Science of Computer Programming*, 24(1):63–81, February 1995.

[Wad92] Philip Wadler. Monads for functional programming. Lecture notes for Marktoberdorf Summer School on Program Design Calculi, Springer-Verlag, August 92.